J Autom Reasoning (2006) 36:345–377
DOI 10.1007/s10817-006-9033-2

Answer Set Programming Based on Propositional Satisfiability

Enrico Giunchiglia · Yuliya Lierler · Marco Maratea

Published online: 29 September 2006

Abstract Answer Set Programming (ASP) emerged in the late 1990s as a new logic programming paradigm that has been successfully applied in various application domains. Also motivated by the availability of efficient solvers for propositional satisfiability (SAT), various reductions from logic programs to SAT were introduced. All these reductions, however, are limited to a subclass of logic programs or introduce new variables or may produce exponentially bigger propositional formulas. In this paper, we present a SAT-based procedure, called ASP-SAT, that (1) deals with any (nondisjunctive) logic program, (2) works on a propositional formula without additional variables (except for those possibly introduced by the clause form transformation), and (3) is guaranteed to work in polynomial space. From a theoretical perspective, we prove soundness and completeness of ASP-SAT. From a practical perspective, we have (1) implemented ASP-SAT in Cmodels, (2) extended the basic procedures in order to incorporate the most popular SAT reasoning strategies, and (3) conducted an extensive comparative analysis involving other state-of-the-art answer set solvers. The experimental analysis shows that our solver is competitive with the other solvers we considered and that the reasoning strategies that work best on 'small but hard' problems are ineffective on 'big but easy' problems and *vice versa*.

Key words answer set programming · propositional satisfiability.

E. Giunchiglia · M. Maratea (✉)
STAR-Lab, DIST, University of Genova, viale Francesco Causa, 13-16145 Genova, Italy
e-mail: maratea@mat.unical.it

E. Giunchiglia
e-mail: enrico@dist.unige.it

Y. Lierler
Institut für Informatik, Erlangen-Nürnberg-Universität, Haberstr. 2, Erlangen, Germany
e-mail: yuliya@informatik.uni-erlangen.de

M. Maratea
Department of Mathematics, University of Calabria, viale Pietro Bucci,
Cubo 31b-87036 Rende, CS, Italy

1. Introduction

Answer Set Programming (ASP) emerged in the late 1990s as a new logic programming paradigm [52, 54] and has been successfully applied in various domains, including space shuttle control [56], planning [45], and the design and implementation of query-answering systems [4]. Syntactically, ASP programs look like Prolog programs, but they are treated by rather different computational mechanisms. Indeed, ASP systems such as CMODELS [42], SMODELS [61], SMODELS$_{cc}$ [64], DLV [39], and ASSAT [47, 49] interpret logic programs by means of the answer set semantics [21, 22]. The goal is to find the 'models' (called answer sets) of the program, and not to evaluate whether a query is true, as in standard Prolog systems. The ASP approach is thus similar to propositional satisfiability checking, where propositional formulas encode the problem and models of the formula correspond to the solutions of the problem.

Propositional satisfiability (SAT) is one of the most intensely studied fields in artificial intelligence and computer science. Various procedures that can deal with thousands of variables are now available (see, e.g., [37]). Also motivated by the availability of efficient SAT solvers (such as SATZ [40] and MCHAFF [53]), various reductions from logic programs to SAT were introduced. The most popular of such reductions is Clark's completion [8]. Fages [18] showed that if a logic program is 'tight,' then its answer sets are in one-to-one correspondence with the models of its Clark's completion. From a theoretical point of view, Fages' result was then generalized to include programs with infinitely many rules [43], programs tight 'on their completion models' [3], programs with nested expressions in the bodies of the rules [16], and disjunctive programs [38]. From a practical point of view, computation of answer sets for tight programs via Clark's completion and SAT solving was first implemented in CMODELS and has been shown to be effective on many classes of problems. Still, these results do not apply to the whole class of logic programs. In general, it is well known that each answer set corresponds to a model of its completion, but the converse is in general not true [51].

Ben-Eliyahu and Dechter [7] gave a translation from a class of disjunctive logic programs to SAT: Their translation may need $O(n^2)$ new variables and $O(n^3)$ new clauses, where n is the number of atoms in the logic program. Lin and Zhao [46] introduced a translation that needs the introduction of $O(n^2 + m)$ new variables and $O(n \times m)$ new clauses, where m is the number of rules in the logic program. Janhunen [33] presented an optimized encoding that is subquadratic in both size and number of atoms. Lin and Zhao [49] report that the grounding of a program corresponding to the computation of a Hamiltonian path in a complete graph with 50 nodes produces a program with 5,000 atoms and 240,000 rules, and in a complete graph of 60 nodes produces a program with 7,000 atoms and 420,000 rules. For problems like these, the number of variables or clauses in the resulting formula may become prohibitive.

The only reduction to SAT that does not need extra variables was proposed by Lin and Zhao [47, 49]. The drawback of this reduction is that it may blow up in space,; that is, the resulting number of clauses can be exponential. This is not by chance. A recent result by Lifschitz and Razborov [44] shows that – assuming $P \not\subseteq NC^1/poly$, a conjecture from computational complexity theory widely believed to be true – whenever we try to translate a logic program to a set of clauses

- either we have to introduce new variables,
- or an exponential blowup may occur.

Despite the potential exponential blowup, the system ASSAT based on such a reduction outperforms state-of-the-art ASP systems like SMODELS and DLV on many interesting problems.

In this paper we present a procedure, called ASP-SAT, that

1. deals with any (not necessarily tight) logic program,
2. works on a propositional formula without additional variables (except for those possibly introduced by the clause form transformation), and
3. is guaranteed to work in polynomial space.

From a theoretical perspective, we prove the soundness and completeness of ASP-SAT. We also show how to extend this basic procedure in order to compute all answer sets still working in polynomial space.

From a practical perspective, we have implemented ASP-SAT in CMODELS. We call the resulting system CMODELS2. Given the SAT-based nature of our procedure, we were able to implement – with a relatively small effort – several search strategies and heuristics that have been shown effective in the SAT literature. Then, we experimentally analyzed which combinations of reasoning strategies work best on which problems. In particular:

– We implemented various "look-ahead" strategies (used while descending the search tree), "look-back" strategies (used for recovering from a failure in the search tree), and "heuristics" (used for selecting the next literal to branch on).
– We considered CMODELS2 with various combinations of strategies, and other state-of-the-art systems such as SMODELS, SMODELS$_{cc}$, ASSAT, and DLV.
– We conducted an extensive experimental analysis, involving all the above-mentioned versions of CMODELS2 and systems, and a wide variety of tight and non-tight programs, ranging from "small", randomly generated programs with a few hundred atoms, up to "large" programs with tens of thousands variables.

Our experimental results show that the look-back (resp. look-ahead) version of CMODELS2 has a clear edge over the other state-of-the-art systems that we considered on large (resp. small randomly generated) problems. The look-back version of CMODELS2 is very competitive also on the other nonrandom, non-large programs that we considered.

If we focus on the performances of the various versions of CMODELS2, the experimental results also point out several facts:

1. On the small, randomly generated problems, "look-ahead solvers" (featuring a rather sophisticated look-ahead based on "failed literal," a simple look-back strategy – essentially backtracking – and a heuristic based on the information gleaned during the look-ahead phase) are best.
2. On the large problems,"look-back solvers" (featuring a simple but efficient look-ahead strategy – essentially unit-propagation with two-literal watching – a rather sophisticated look-back based on "learning" and a constant time heuristic based on the information gleaned during the look-back phase) are best.
3. Adding a powerful look-back (resp. look-ahead) to a look-ahead (resp. look-back) solver does not lead to better performances if the resulting solver is run on the small (resp. large) problems that we considered.

In the terminology in [29], our comparison is "fair" because all the reasoning strategies are realized on a common platform, and thus the experimental evaluation is not biased by the differences due to the quality of the implementation, and is "significant" because CMODELS2 implements state-of-the-art look-ahead/look-back strategies and heuristics. We believe that these results have important consequences both for developers and for people interested in benchmarking ASP systems. For instance, our results say that we can hardly expect to develop a solver with the best performance in all problem categories. As a consequence,

- developers should focus on specific classes of benchmarks (e.g., on randomly generated programs), and
- benchmarking should take into account whether solvers have been designed for specific classes of programs. Indeed, it hardly makes sense to run a solver designed for random (resp. large) programs on large (resp. random) programs.

The paper is structured as follows. In Section 2 we introduce the definitions, terminology, and results at the basis of our work. Then, in Section 3 we present ASP-SAT in its basic backtracking version, and we prove its soundness and completeness. We also discuss in detail what needs to be done in order to implement ASP-SAT on top of a SAT solver with learning. In Section 4 we show how we implemented ASP-SAT in CMODELS. Section 5 contains the experimental, comparative evaluation. We end the paper with the conclusions and future work in Section 6.

A preliminary version of this paper is [27]. This paper contains also results presented in [25, 26].

2. Formal Background

2.1. Syntax of Logic Programs

A *rule* is an expression of the form

$$p_0 \leftarrow p_1, \ldots, p_k, \mathit{not}\ p_{k+1}, \ldots, \mathit{not}\ p_m, \mathit{not}\,\mathit{not}\ p_{m+1}, \ldots, \mathit{not}\,\mathit{not}\ p_n \tag{1}$$

$(0 \leq k \leq m \leq n)$, where p_0 is an atom or the symbol $\bot$ ($\bot$ is the logical symbol standing for the empty disjunction, i.e., *False*); $p_1, p_2, \ldots, p_n$ are atoms; the symbol *not* is the "negation" as failure operator; p_0 is the *head* of the rule; and the expression at the right of the arrow is the *body*. The intuitive meaning of a rule (1) is that p_0 is in the solution whenever the body is satisfied.

A (*nondisjunctive logic*) *program* is a finite set of rules.

If the head of a rule is $\bot$, we call the rule a *constraint*. If a rule (1) contains an expression of the form *not not* p_i, then the rule is called *nested*; otherwise the rule is *nonnested* or *basic*. If a logic program Π contains at least one nested rule, Π is a *nested* program; otherwise it is *nonnested* or *basic*. For instance, the program

$$\begin{array}{l} p \leftarrow \mathit{not}\ \mathit{not}\ p \\ q \leftarrow \mathit{not}\ p. \end{array} \tag{2}$$

is nested, while

$$\begin{array}{l} p \leftarrow p \\ q \leftarrow \mathit{not}\ p. \end{array} \tag{3}$$

is nonnested or basic.

2.2. Answer Sets for Logic Programs

To give the definition of an answer set, we consider first the special case in which the program Π does not contain the negation as failure operator *not* (i.e., for each rule (1) in Π, $n = m = k$). Let Π be such a program, and let X be a set of atoms. We say that X is *closed* under Π if for every rule (1) in Π, $p_0 \in X$ whenever $\{p_1, p_2, \ldots, p_k\} \subseteq X$. In the $n = m = k$ hypothesis, Π has only one *answer set*, and it is the smallest set of atoms closed under Π. Computing such an answer set can be done in linear time, via the Dowling–Gallier procedure [13], or via unit propagation (assuming the symbol "$\leftarrow$" is understood as the standard material implication, and "," as conjunction).

Now consider an arbitrary program Π. Let X be a set of atoms. A rule

$$p_0 \leftarrow p_1, \ldots, p_k$$

belongs to the *reduct* Π^X *of* Π *with respect to* X if and only if there is a rule (1) in Π with $X \cap \{p_{k+1}, \ldots, p_m\} = \emptyset$ and $\{p_{m+1}, \ldots, p_n\} \subseteq X$. Π^X is a program without negation as failure. We say that a subset X of the atoms in Π is an *answer set* for Π if X is an answer set for Π^X [21, 38].

As an example, let Π be the program (2), and consider the set of atoms $\{p\}$. The reduct $\Pi^{\{p\}}$ is

$$p \leftarrow . \tag{4}$$

The set $\{p\}$ is the smallest set closed under (4), and hence it is also an answer set of the program Π. If we consider the set of atoms $\{p, q\}$, the reduct $\Pi^{\{p,q\}}$ is again (4). The set $\{p, q\}$ is not the smallest set closed under (4), and hence it is not an answer set of the program Π.

Determining the existence of an answer set for a program Π is an NP-complete problem. Indeed, checking whether a set of atoms X is an answer set of Π can be done in linear time by first computing the reduct Π^X and then computing the answer set of Π^X. NP-hardness can be easily proven by using standard reductions of the SAT problem into logic programs under answer set semantics; see, for example, [32].

2.3. Completion

Consider a program Π. For an atom p_0 the completion $Comp(\Pi, p_0)$ of Π relative to p_0 is the formula

$$p_0 \equiv \bigvee (p_1 \wedge \cdots \wedge p_k \wedge \neg p_{k+1} \wedge \cdots \wedge \neg p_m \wedge p_{m+1} \wedge \cdots \wedge p_n),$$

where the disjunction extends over all rules (1) in Π with head p_0. The *completion* $Comp(\Pi)$ of Π consists of the formulas

$$\bigvee_{i=1}^{k} \neg p_i \vee \bigvee_{i=k+1}^{m} p_i \vee \bigvee_{i=m+1}^{n} \neg p_i,$$

one for each rule (1) whose head is $\perp$, and of the formulas $Comp(\Pi, p_0)$ for each atom p_0 in Π [8, 50]. For instance, the completion of the program (2) consists of the formulas

$$\begin{aligned} p &\equiv p \\ q &\equiv \neg p, \end{aligned} \tag{5}$$

and (5) is also the completion of the program (3).

The following theorem, due to Marek and Subrahmanian [51] for basic programs and generalized in [16] to nested programs, relates the answer sets of a program to the models of its completion. In the following, we say that a set of atoms X *satisfies* (or is a *model* of) a set of formulas Γ if Γ is satisfied by the interpretation that assigns *True* to an atom p if and only if $p \in X$.

THEOREM 1. *Let* Π *be a program. If* X *is an answer set of* Π, *then* X *satisfies the completion of* Π.

The set of atoms $\{p, q\}$ does not satisfy the completion (5) of (2) (resp. (3)), and thus it is not an answer set of (2) (resp. (3)).

2.4. Tight Programs

Theorem 1 can be strengthened in the case of tight programs. A program Π is *tight* if its dependency graph is acyclic. The *dependency graph* of a program Π is the directed graph G such that

– the nodes of G are the atoms in Π, and
– for every rule (1) in Π, G has an edge from p_0 to each atom in $\{p_1, \ldots, p_k\}$.

The following theorem has been proved by Fages [18] for basic programs, and it has been generalized by Erdem and Lifschitz [16] to nested programs.

THEOREM 2. *Let* Π *be a tight program and* X *a set of atoms.* X *is an answer set for* Π *iff* X *satisfies the completion of* Π.

Program (2) is tight, while program (3) is non-tight. Hence, according to the above theorem, the answer sets of (2) coincide with the models of (5) (and thus can be computed with SAT solvers).

2.5. Loop Formulas

Theorem 1 states that if X is an answer set of program Π, then X satisfies $Comp(\Pi)$. Theorem 2 says that the converse is also true if the program is tight. If the program is non-tight, Lin and Zhao [47, 49] proved that to have the identity mapping between the answer sets of a basic program Π and the models of its completion, we have to consider the loop formulas of Π. Lee and Lifschitz [38] extended the concept of loop formulas to nested programs and proved that the same result holds with the extended definition. To formally state this last result, we need the following definitions.

A *loop* of Π is a nonempty set L of atoms such that for each pair p, p' of atoms in L there exists a path of nonzero length from p to p' in the dependency graph of Π whose intermediate nodes belong to L.

Given a loop L, we define $R(L)$ to be the set of formulas

$$(p_1 \wedge \cdots \wedge p_k \wedge \neg p_{k+1} \wedge \cdots \wedge \neg p_m \wedge p_{m+1} \wedge \cdots \wedge p_n)$$

for all rules (1) in Π, with $p_0 \in L$ and $\{p_1, \ldots, p_k\} \cap L = \emptyset$. The *loop formula associated with* L is

$$\bigvee L \supset \bigvee R(L), \tag{6}$$

where $\bigvee L$ denotes the disjunction of the atoms in L, and similarly for $\bigvee R(L)$.

THEOREM 3. *Let* Π *be a program. Let* $Comp(\Pi)$ *be the completion of* Π. *Let* $LF(\Pi)$ *be the set of all the loop formulas associated with the loops of* Π. *For each set of atoms* X, X *is an answer set of* Π iff X *is a model of* $Comp(\Pi) \cup LF(\Pi)$.

Consider the non-tight program (3). Its completion is (5). The only loop of the program is $\{p\}$ and the loop formula associated with $\{p\}$ is

$$p \supset \bot,$$

that is equivalent to $\neg p$. Thus, the answer sets of (3) are the set of atoms that satisfy (5) and also $\neg p$.

3. SAT-based Answer Set Solvers

3.1. Previous Approaches

CMODELS [42] is an answer set solver based on SAT which has evolved over the years and that, in its current version, incorporates the procedure described in this paper and in its predecessor [27]. The version of CMODELS prior to [27] is restricted to tight programs. Given a tight program Π, CMODELS

1. computes the completion $Comp(\Pi)$ of the program, and
2. calls a SAT solver to find the models of $Comp(\Pi)$ (corresponding to the answer sets of the input program). Before invoking the SAT solver, it may be necessary to convert the formulas in $Comp(\Pi)$ to a set of clauses, as required by most SAT solvers. A *clause* is a disjunction of literals, and a *literal* is an atom or the negation of an atom.

The advantage of this method is that it uses SAT solvers as black boxes. On the other hand, it is restricted to tight programs.

Theorem 3 lays the foundation for extending this method to non-tight programs.

Consider a program Π. To determine whether Π has an answer set, one possibility is to

1. compute the completion and the loop formulas of Π, that is, the set $\Gamma = Comp(\Pi) \cup LF(\Pi)$ of formulas, and then
2. invoke a SAT solver to determine the models of (the clause conversion of the formulas in) Γ.

This is an “eager”[1] approach, which may work well in practice in some domains, but the resulting propositional formula may be exponentially bigger than the input program.

ASSAT [47, 49] is a SAT-based system for basic programs that takes an alternative approach. Indeed, ASSAT adds loop formulas on demand. That is, ASSAT

1. Computes $\Gamma = Comp(\Pi)$.
2. Finds a model X of Γ by using a SAT solver (before this, it may be necessary to convert Γ to a set of clauses). If no such model exists, then the input program does not have answer sets, and the procedure terminates, returning *False*.
3. Checks whether X is an answer set. As we have already said in Section 2.2, this process can be done in linear time in the size of Π. If X is an answer set, then the procedure terminates with returning *True*. Otherwise, ASSAT
   a) finds at least one loop formula that is not satisfied by Xm adds it to Γ (as described in Section 4, this step can be done in linear time in the size of Π), and
   b) goes back to step 2.

Lin and Zhao [47, 49] showed that ASSAT can often outperform rival systems. However, ASSAT has the following two main drawbacks

1. ASSAT is not guaranteed to work in polynomial space. Lifschitz and Razborov [44] showed that there are programs Π for which $LF(\Pi)$ contains exponentially many formulas (unless $P \not\subseteq NC^1/poly$), each of which cannot be derived from the others and $Comp(\Pi)$. For these programs Π:
   - If Π has an answer set, then ASSAT performance on Π depends on how lucky the system is in generating the right model first. In the best case it generates an answer set first. In the worst case it blows up in space.
   - If Π has no answer set, then ASSAT blows up in space. In fact, adding and keeping already added loop formulas is essential to guarantee that the SAT solver does not return an already computed model, and thus to guarantee ASSAT termination.
2. Considering two successive calls to the SAT solver, the computation done for finding the first model is completely discarded, that is, not reused by the SAT solver in the second call. Thus, some branches of the search tree may get computed many times.

There are two ways in ASSAT for computing all answer sets of a program Π:

1. Compute $Comp(\Pi) \cup LF(\Pi)$ and then call a SAT enumerator, specifically a SAT solver able to return all the models of a propositional formula, for example, MCHAFF [53]; or

[1] The terminology is borrowed from the one used in decision procedures for separation logic, where “eager” approaches compile the input formula into an equisatisfiable propositional one; see, for example, [36].

2. In order to avoid the generation of the same model X, once an answer set X is found, modify ASSAT procedure in step 3 by
   a) adding to Γ one or more clauses ensuring that the same answer set X is not recomputed, and
   b) going back to step 2.
   For nested programs, the obvious clause to add to Γ is
   $$\bigvee_{A \in X} \neg A \vee \bigvee_{A \notin X} A. \tag{7}$$
   For basic programs (i.e., of the kind that ASSAT considers), we can take advantage of the fact that the following *anti-chain property* holds: If X is an answer set, no strict subset or superset of X is an answer set. For these programs it is thus sufficient to add to Γ one or both of the clauses
   $$\bigvee_{A \in X} \neg A, \qquad \bigvee_{A \notin X} A \tag{8}$$
   in order to ensure that the same answer set is not recomputed. The advantage of adding (8) instead of (7) is that each clause in (8) entails (7) and thus it prunes more search space.

The first approach is unfeasible if there are (exponentially) many loop formulas. The second approach is unfeasible also when there are many answer sets.

3.2. ASP-SAT with Backtracking

The above drawbacks can be eliminated if we do not use a SAT solver as a black box. Instead, we can take advantage of the fact that all the state-of-the-art complete SAT solvers are based on the Davis-Logemann-Loveland procedure[10]. The basic observation is that the Davis-Logemann-Loveland procedure can easily work as a SAT enumerator.

Thus, given a program Π, we may first compute the completion of Π, and then

– *generate* the models of $Comp(\Pi)$, and
– *test* whether the generated models are answer sets of Π.

We call the resulting procedure ASP-SAT. It is represented – in its simple backtracking version – in Figure 1. In the figure,

1. Given a set of formulas Γ, CNF(Γ) returns a set of clauses – possibly with newly introduced propositional variables – such that, for any interpretation μ in the extended language, the following two properties hold:
   a) if μ satisfies CNF(Γ), then the restriction of μ to the language of Γ satisfies Γ, and
   b) if μ satisfies Γ, then there exists an interpretation in the language of CNF(Γ) that (1) extends μ, and (2) satisfies CNF(Γ).

   Example of such a conversion are the "classical conversion" (which given a formula in negative normal form recursively distributes conjunctions over disjunctions) and the conversions based on "renaming," such as those described in [63, 57, 58].

Figure 1 The SAT-based ASP-SAT procedure for Answer Set Programming.

```
function ASP-SAT($\Pi$)
  return DLL(CNF($Comp(\Pi)$), $\emptyset$, $\Pi$);

function DLL($\Gamma$, $S$, $\Pi$)
  if ($\Gamma = \emptyset$) then return $test(S, \Pi)$;
  if ($\emptyset \in \Gamma$) then return $False$;
  if ($\{l\} \in \Gamma$) then return DLL($assign(l, \Gamma)$, $S \cup \{l\}$, $\Pi$);
  $p :=$ an atom occurring in $\Gamma$;
  return DLL($assign(p, \Gamma)$, $S \cup \{p\}$, $\Pi$) or
            DLL($assign(\neg p, \Gamma)$, $S \cup \{\neg p\}$, $\Pi$).
```

2. l denotes a literal, and Γ a set of clauses.
3. S is an *assignment*, that is, a consistent set of literals.
4. Given an atom p, $assign(p, \Gamma)$ is the set of clauses obtained from Γ by removing the clauses to which p belongs and by removing $\neg p$ from the other clauses in Γ. $assign(\neg p, \Gamma)$ is defined similarly.

A key feature of ASP-SAT is that it is based on DLL, which, considering its pseudocode in the figure, is almost identical to the Davis-Logemann-Loveland procedure: The only difference is that, when the empty set of clauses is generated, DLL invokes the function $test(S, \Pi)$ instead of just returning *True*. ASP-SAT thus follows a "lazy" approach to the computation of answer sets based on SAT,[2] where, intuitively speaking, the goal of the function $test(S, \Pi)$ is to return *True* if the assignment S corresponds to at least one answer set of Π, and *False* otherwise. However, the function $test(S, \Pi)$ deserves further comment. Assume P is the set of atoms in the program Π. When the function $test(S, \Pi)$ is invoked, its argument S is such that $S \cap P$ satisfies the completion of Π and is thus a candidate for being an answer set. However, S may not be a *total* assignment; that is, it is possible that for some atom $p \in P$, neither p nor $\neg p$ are in S. If p is one such atom, also $(S \cap P) \cup \{p\}$ satisfies the completion of Π and is thus another candidate for being an answer set. In general, an assignment S can potentially correspond to exponentially many sets of atoms satisfying the completion of Π, and each of them is a superset of the atoms in $S \cap P$. However, if Π is a basic program, none of these strict supersets is an answer set of Π, as established by the following proposition.

PROPOSITION 4. *Let Π be a basic program. Let X be a set of atoms satisfying $Comp(\Pi)$. If $X \subset X'$, then X' is not an answer set of* Π.

[2] The terminology is again borrowed from that used in decision procedures for separation logic, where "lazy" approaches abstract the input formula into a propositional one and refine the propositional model if it does not correspond to a model of the original formula; see, for example, [1, 2, 5, 11]. More recently it was shown [55] that better performances can be obtained by using a lazy approach in which the assignment is extended on the basis of the semantics of the original formula in separation logic. In our setting, this would correspond to assign some atoms – not entailed by the current assignment and the completion of the input program – but entailed by the current assignment, the completion of the input program and the set of loop formulas. Whether this can lead to better performances is still an open research issue.

Proof. We are given that X satisfies $Comp(\Pi)$. From completion construction, it follows that X is closed under Π^X. Since $X \subset X'$ and Π is basic, $\Pi^{X'} \subseteq \Pi^X$. Hence X is closed under $\Pi^{X'}$, and thus X' is not the smallest set closed under $\Pi^{X'}$. □

Thus, according to the above proposition, if Π is basic, *test*(S, Π) has just to check whether $S \cap P$ is an answer set of Π. Any set of atoms extending $S \cap P$ is not an answer set.

We are now ready to state our main theorem in the case of basic programs.

THEOREM 5 (Soundness and completeness for basic programs). *Let* Π *be a basic program in the set* P *of atoms. Let test*(S, Π) *be a function returning True if* $S \cap P$ *is an answer set of* Π, *and False otherwise.* ASP-SAT(Π) *returns True if* Π *has an answer set, and False otherwise.*

Proof. Soundness is trivial. For completeness, assume that ASP-SAT(Π) returns *False*. Let P be the set of atoms in Π. Let Γ be the set of assignments S that have been checked, that is, such that *test* (S, Π) has been invoked. The fact that Π has no answer sets follows from the following properties

1. The formula

$$\bigvee_{S\in\Gamma}\left(\bigwedge_{p:p\in S,\,p\in P} p \wedge \bigwedge_{p:\neg p\in S,\,p\in P} \neg p\right)$$

   is logically equivalent to the completion $Comp(\Pi)$ of Π (Proposition 5 in [24], restated as Lemma 4 in [2]).
2. The set of answer sets of Π is a subset of $\{S \cap P : S \in \Gamma\}$ (easy consequence of Theorem 1 and Proposition 4). □

Proposition 4 does not hold for arbitrary programs. In general, given a nested program Π, it is possible that two sets X and X' of atoms are such that

– X satisfies the completion of Π but is not an answer set of Π, and
– X' is a superset of X and is an answer set of Π.

This is illustrated by the following program:

$$\begin{aligned} p_1 &\leftarrow not\, not\, p_1 \\ p_2 &\leftarrow p_1 \\ p_2 &\leftarrow p_2. \end{aligned} \tag{9}$$

The completion of the program is $\{p_1 \equiv p_1, p_2 \equiv (p_1 \vee p_2\}$. The set of atoms $\{p_2\}$ satisfies the completion but is not answer set. The set of atoms $\{p_1, p_2\}$ is a superset of $\{p_2\}$ and is also an answer set of (9).

Thus, in the general case, whenever *test*(S, Π) is invoked, every set X of atoms that is

1. a superset of $S \cap P$, and
2. a subset of $\{p : \neg p \notin S, p \in P\}$

has to be checked to see whether it is an answer set of Π.

THEOREM 6 (Soundness and completeness for arbitrary programs). *Let* Π *be a program in the atoms* P. *Let test*(S, Π) *be a function returning True if there exists* *a set X with $S \cap P \subseteq X \subseteq \{p : \neg p \notin S, p \in P\}$ that is an answer set of* Π, *and False otherwise.* ASP-SAT(Π) *returns True if* Π *has an answer set, and False otherwise.*

Proof. The proof is analogous to the one of Theorem 5; the only difference is that, assuming

- P is the set of atoms in Π,
- Γ is the set of assignments S that have been checked, that is, such that *test* (S, Π) has been invoked,

the set of answer sets of Π is a subset of

$$\{X : \exists S \in \Gamma . S \cap P \subseteq X \subseteq \{p : \neg p \notin S, p \in P\}\},$$

as established by Theorem 1. □

3.3. ASP-SAT with Learning

The ASP-SAT procedure in the previous subsection is based on DLL, which is very similar to the standard Davis–Logemann–Loveland procedure with simple chronological backtracking. It is thus not infrequent for ASP-SAT to explore a possibly large subtree whose leaves are all dead ends because of some bad choices performed way up in the search tree. In SAT, the standard solution to this problem is to backjump over the choices that do not belong to the "reason" for the failure. Intuitively, if S is an assignment that falsifies the input set Γ of clauses, then a *reason* R for S is a subset of the literals in S such that any assignment extending R falsifies Γ. (We say that a set S of literals *falsifies* a set of formulas Γ if $S \cup \Gamma$ is inconsistent). Reasons are initialized as soon as a failure is generated, and are updated while backtracking. Many of the current state-of-the-art SAT procedures feature such backjumping mechanism and extend it with learning: Under certain conditions, a reason R is converted into the clause $(\bigvee_{p \in R} \neg p \vee \bigvee_{\neg p \in R} p)$, which is then learned, that is, added to the input set of clauses as additional constraint. Since exponentially many distinct reasons can be computed, suitable criteria are also used in order to forget (i.e., remove) clauses corresponding to reasons, thus maintaining the SAT solver in polynomial space.

It is beyond the goals of this paper to describe how learning is incorporated in the Davis–Logemann–Loveland procedure; see, for example, [12] for a high-level description of learning including soundness and completeness statements of the resulting procedure, [6, 60, 65] and more detailed descriptions of different learning mechanisms. For our purposes, it suffices to say that a SAT solver with learning can still be used as an underlying procedure for ASP-SAT. The only difference with respect to the procedure in Figure 1 is in the *test* procedure. In fact, as we outlined above, whenever we have a failure, we also must have a corresponding reason. In our case, if *test*(S, Π) returns *False*, it also has to return a subset R of the atoms in S such that for any total assignment S' extending R and not falsifying the completion of Π, the set of atoms in S' is guaranteed to be not an answer set of Π. One such set R is S. However, in order to maximize the effects of the backjumping and learning mechanisms in the SAT solver, it is important that R be as small as possible. In the case of a basic program, one smaller such set is the set of atoms in S (see Proposition 4). However, it is possible to take advantage of loop formulas, and – in practice – return reasons that are often less than 1% of the size of S.

To illustrate how loop formulas can help for computing small reasons, let us consider a call to *test*(S, Π). Let P be the set of atoms in Π. We assume that S does not correspond to any answer set of Π; otherwise *test*(S, Π) just has to return *True*, and the computation of a reason does not make sense.

For simplicity, assume that S is a total assignment. The idea is to find a loop formula F that is falsified by S, and return a subset S' of S necessary to falsify F. Since every answer set of Π has to satisfy all the loop formulas of Π, the set of atoms in any superset of S' is guaranteed to be not an answer set of Π. Important is the fact that determining such a set S' can be done efficiently, namely, in linear time in the size of Π, as detailed in the next section.

If S is not total but Π is basic, then – thanks to Proposition 4 – we can consider simply the total assignment $S \cup \{\neg p : p \in P, p \notin S\}$.

Now assume that Π is nested and that S is not total. Assume for simplicity that there is only one atom $p \in P$ such that neither p nor $\neg p$ is in S. Let $S_1 = S \cup \{p\}$ and $S_2 = S \cup \{\neg p\}$. Both S_1 and S_2 are total. Furthermore, $S_1 \cap P$ and $S_2 \cap P$ are not answer sets, and we can compute $S'_1 \subseteq S_1$ and $S'_2 \subseteq S_2$, each falsifying a loop formula of Π as in the previous case. If $p \notin S'_1$ (resp. $\neg p \notin S'_2$), then S'_1 (resp. S'_2) is also a subset of S and can be returned. If $p \in S'_1$ and $\neg p \in S'_2$, we can safely return $S'' = S'_1 \cup S'_2 \setminus \{p, \neg p\}$: $S'' \subseteq S$ and no set extending S'' can correspond to an answer set. The above procedure can be easily extended to the case in which there is more than one atom $p \in P$ with $\{p, \neg p\} \cap S = \emptyset$.

Notice that S may be a nontotal assignment because in ASP-SAT *test*(S, Π) is invoked whenever the input set of clauses is empty. Indeed, many SAT solvers – including MCHAFF – have a different termination condition for *True*: *True* is returned whenever either p or $\neg p$ is in S, for each atom p in the input set of clauses Γ. Assuming that all the atoms in Π occur also in Γ, the above termination condition for *True* ensures that S is total.

We remark that in order to guarantee the termination of our procedure ASP-SAT(Π), it is not necessary to store the reasons returned by *test*(S, Π): On the other hand, learning (a polynomial amount of) reasons can improve performances of the procedure. Consider in fact the program Π_k consisting of the rules

$$p_i \leftarrow p_{i+1} \qquad\qquad p_{i+1} \leftarrow p_i$$

where $i \in \{0, 2, \ldots, 2k-2\}$, and of the constraint

$$\bot \leftarrow not\ p_0, not\ p_1, \ldots, not\ p_{2k-1}.$$

Π_k has no answer set, while $Comp(\Pi_k)$ has $2^k - 1$ models. Assuming CNF($Comp(\Pi_k)$) consists of the clauses

$$\neg p_i \vee p_{i+1} \qquad\qquad \neg p_{i+1} \vee p_i \tag{10}$$

($i \in \{0, 2, \ldots, 2k-2\}$), and

$$p_0 \vee p_1 \vee \ldots \vee p_{2k-1},$$

the following facts hold (in this paragraph, for simplicity, we assume that the clauses corresponding to the reasons returned by *test*(S, Π_k) are learned and never forgotten):

– A naive implementation of *test*(S, Π_k), which returns S as reason for its failure, will cause the generation and rejection of exponentially many sets of atoms, one for each set of atoms satisfying the completion of Π_k;
– Since Π_k is basic, *test*(S, Π_k) may return the set of atoms in S as reason for its failure. Depending on the order in which the assignments are generated and then tested, different things can happen, ranging between the following two extreme cases:

  1. In the best case, the assignments containing exactly one pair $\{p_i, p_{i+i}\}$ (i even) are generated (and then rejected) first. In this case, the clause $(\neg p_i \vee \neg p_{i+1})$ is learned, and, together with (10), this implies that any other assignment generated afterwards will contain both $\neg p_i$ and $\neg p_{i+1}$. After the k sets with two positive atoms are generated, the resulting set of clauses is inconsistent, and no more assignments are generated.
  2. In the worst case, the assignments containing a maximum number of positive atoms in P are generated (and then rejected) first. The first assignment that will be generated is $\{p_0, p_1, \ldots, p_{2k-1}\}$, and the corresponding learned clause is $\neg p_0 \vee \neg p_1 \vee \ldots \vee \neg p_{2k-1}$. It is easy to see that exponentially many assignments will be generated before determining the nonexistence of answer sets.

– An implementation of *test*(S, Π) that returns a subset of S falsifying one of the loop formulas is guaranteed to test k assignments. The reason is that Π_k has k loops, $\{p_i, p_{i+1}\}$, with i even. Given a loop $\{p_i, p_{i+1}\}$, its loop formula is $(p_i \vee p_{i+1}) \supset \bot$, corresponding to

$$(\neg p_i \wedge \neg p_{i+1}). \qquad (11)$$

  Given a call to *test*(S, Π), (1) a loop formula of the form (11) falsified by S is computed; (2) the two possible subsets of S falsifying (11) are computed, namely, $\{p_i\}$ and $\{p_{i+1}\}$; (3) one of them is returned as reason; (4) assuming $\{p_i\}$ is the returned reason, the clause $\{\neg p_i\}$ is learned; and (5) after backtracking/backjumping, unit propagation immediately assigns both p_i and p_{i+1} to *False*.
  After k calls to the *test*(S, Π) procedure, the resulting set of clauses is unsatisfiable.

3.4. Computational Properties of ASP-SAT

From a computational perspective, the ASP-SAT procedure in Figure 1 has the following features:

1. It performs the search on *Comp*(Π) and thus does not introduce any extra variables except for those possibly needed by the clause form transformation.
2. It is guaranteed to work in polynomial space.
3. It can deal with both tight and non-tight programs. In the case of tight programs, for each call to *test*(S, Π), the set of atoms of Π, which are also in S, is guaranteed to be an answer set of Π, and thus ASP-SAT behaves as a standard SAT solver running on CNF(*Comp*(Π)).

If the underlying SAT solver uses learning, then all the above features still hold (assuming that the SAT solver itself works in polynomial space).

Compared to the version of CMODELS prior to [27], ASP-SAT is not restricted to work on tight programs.

Compared to ASSAT, ASP-SAT is guaranteed to work in polynomial space and has also the following advantages:

- It is easily modifiable to return all the answer sets. Assuming the solver is based on backtracking, the only action needed is to modify $test(S, \Pi)$ in order to

  1. print the set of atoms determined to be an answer set, and
  2. return *False*.

  Assuming the solver is based on learning, $test(S, \Pi)$ has to

  1. print the set of atoms determined to be answer sets, and
  2. return *False* and a reason $R \subseteq S$ such that each assignment extending R corresponds to already computed answer sets. If Π is a basic program, then the anti-chain property holds for Π. As a consequence, the set of atoms in $S \cap P$ is one such a reason, and the subset of S consisting of the negation of the atoms in P is another possibility. If Π is a nested program, the set S itself has to be returned.

- No computation is ever repeated. When $test(S, \Pi)$ fails, instead of restarting the search from scratch as done in ASSAT, the computation is restarted from the same point in the search tree where $test(S, \Pi)$ was called. The search then continues from this point following the depth-first search schema of the algorithm.

On the other hand, an advantage of ASSAT over ASP-SAT is that the SAT solver is used as a black box without any need of even minor modifications.

Compared to other state-of-the-art answer set solvers such as SMODELS, SMODELS$_{cc}$, and DLV, ASP-SAT has the advantage of being SAT-based, and thus it can leverage on the great amount of knowledge available in SAT. For instance, we are not aware of any non-SAT-based answer set solver using the analogous of two-literal watching data structures for efficiently pruning the search tree while descending it.

4. Implementation in CMODELS

4.1. Integration in CMODELS

We have integrated our implementation of ASP-SAT in CMODELS. CMODELS2 is the name that we use for the resulting system.

The input language of CMODELS2 is a grounded logic program that can be generated by the front-end LPARSE [62] and is the same as the input language of SMODELS, SMODELS$_{cc}$, and ASSAT. The input may thus contain basic rules as well as choice, cardinality, and weight constraint rules [62, Sections 5.3, 5.4]. A *choice* rule has the form

$$\{p_{01}, \ldots, p_{0j}\} \leftarrow p_1, \ldots, p_k, not\ p_{k+1}, \ldots, not\ p_m,$$

where each p with a subscript is an atom. The intuitive meaning of a choice rule is that any atom contained in $\{p_{01}, \ldots, p_{0j}\}$ may or may not belong to the solution whenever the body is satisfied. A *weight constraint* rule is an expression of the form

$$p_0 \leftarrow L\{p_1 = w_1, \ldots, p_k = w_k, not\, p_{k+1} = w_{k+1}, \ldots, not\, p_m = w_m\}U$$

where $L, U, w_1, \ldots w_m$ are integers and each p_i $(i = 0, \ldots, m)$ is an atom. The intuitive meaning of such a rule is that p_0 is in the solution if the sum of the weights of the satisfied literals in the body of the rule is between L and U. A *cardinality constraint* rule is a weight constraint rule in which all the integers in $\{w_1, \ldots, w_m\}$ are equal to 1.

It is beyond the scope of this paper to describe the semantics of programs with these rules in detail; see, for example, [61]. For our goals, it is sufficient to say that in CMODELS2 weight constraint and choice rules are eliminated by introducing auxiliary atoms and nested rules as described in [19, 45].

Traditionally, CMODELS was restricted to find answer sets for tight programs, by the following steps (see [42] for more details):

1. Simplification of the input LPARSE program, performing operations similar to those involved in SMODELS.
2. Elimination of choice and weight constraints rules in favor of nested rules.
3. Verification that the resulting program (possibly with nested rules) is tight.
4. Construction of the program's completion, conversion to a set of clauses, and call to a SAT solver. The clause conversion takes linear time and introduces up to m new atoms, where m is the number of rules in the program.

In CMODELS2, step 3 is not needed anymore (and is no longer performed) because a tight program can be considered as a particular case of a non-tight one in which each call to $test(S, \Pi)$ succeeds.

4.2. ASP-SAT Implementation

ASP-SAT is implemented on top of the SIMO system [28]. SIMO is a MCHAFF-like SAT solver and thus features unit propagation based on a two-literal watching data structure, 1-UIP learning and VSIDS heuristics (see [53] for a description of these techniques). However, it does not feature the low-level optimizations of MCHAFF, and thus it is on average within a factor of 3 slower than MCHAFF. We have used SIMO because is the system we know better, and thus we were able to integrate relatively easily the other search strategies and heuristics used for the experimental analysis.

With reference to Figure 1, to use SIMO as a search engine in an ASP solver, we had to modify it in order to

1. call $test(S, \Pi)$ whenever *True* was returned and
2. guarantee that each set S of literals in $test(S, \Pi)$ is total.

Considering the second task, SIMO – like all the MCHAFF-based SAT solvers – returns *True* when all the atoms in the input set of clauses are assigned and no empty clause has been generated. However, SIMO input set of clauses may not contain all the atoms in the input program. Indeed, as a preliminary step and before the search starts, SIMO (and many other SAT solvers as well) preprocesses the input set of clauses and

1. eliminates tautological clauses (i.e., clauses with both an atom and its negation as disjuncts), and
2. assigns *pure* literals; that is, each atom p is assigned to *True* if $\neg p$ does not belong to any clause in the input formula, and similarly for $\neg p$.

These operations are not harmful in SAT solving. However, if the SAT solver is used – as in our case – as basis for an answer set solver, both operations may lead to incorrect results. Consider, in fact, the program

$$\begin{aligned} p_1 &\leftarrow \mathit{not}\,\mathit{not}\,p_1 \\ p_2 &\leftarrow p_1 \\ p_2 &\leftarrow p_2 \\ \bot &\leftarrow \mathit{not}\,p_1,\ \mathit{not}\,p_2 \end{aligned}$$

which has $\{p_1, p_2\}$ as an answer set. The completion of the program is $\{p_1 \equiv p_1, p_2 \equiv (p_1 \vee p_2), p_1 \vee p_2\}$. After the straightforward translation to a set of clauses, and after the elimination of the tautological clauses,

1. only two clauses are left, $(\neg p_1 \vee p_2)$ and $(p_1 \vee p_2)$, and
2. after p_2 is assigned during the preprocessing, the empty set of clauses is generated.

The empty assignment is returned and is checked to see whether it is an answer set. Since it is not, *False* would be incorrectly returned. In order to avoid such undesired behavior, SIMO preprocessing has been modified to keep tautological clauses and to not assign pure literals.

To evaluate the impact of different search strategies and heuristics in solving answer set programs, we have enhanced SIMO with search strategies and heuristics other than those implemented by MCHAFF. In particular, we implemented the following:

– Failed-literal detection: Before branching, for each unassigned atom p, p is assigned to *True* and then unit propagation is called again. If a contradiction is found, p is said to be a *failed literal*, $\neg p$ can be safely assigned, and unit-propagation is again performed. Otherwise, $\neg p$ is checked following the same procedure implemented for p.
– Standard backtracking: Learning is disabled, and recovery from failure is performed by chronologically backtracking to the latest assigned branching literal.
– The unit heuristic, based on the failed-literal detection technique: Given an unassigned atom p, while doing failed-literal detection on p, we count the number $u(p)$ of unit propagations caused, and then we select the atom with maximum $1024 \times u(p) \times u(\neg p) + u(p) + u(\neg p)$. The atom is assigned to *True* first.

The above search strategies and heuristics are not novel; they are standard techniques in the SAT field and are implemented by many state-of-the-art SAT solvers. Indeed, current state-of-the-art SAT solvers can be divided in two main categories:

– "Look-ahead" solvers, featuring a rather sophisticated look-ahead based on "failed literal," a simple look-back (essentially backtracking), and a heuristic based on the information gleaned during the look-ahead phase. These solvers are best for dealing with "small but relatively difficult" randomly generated *k*-cnf formulas.[3] A solver in this category is SATZ [40].
– "Look-back" solvers, featuring a simple but efficient look-ahead (essentially unit propagation with two-literal watching), a rather sophisticated look-back based on "1-UIP learning," and a constant time heuristic based on the information gleaned during the look-back phase. These solvers are best for dealing with "large but relatively easy" instances, typically encoding nonrandom problems. A solver in this category is MCHAFF [53].

By combining SIMO original reasoning strategies with those newly implemented, we can obtain both a MCHAFF-like and a SATZ-like SAT solver, and consequently, a "look-back" answer set solver, and a "look-ahead" answer set solver. Our goal is to confirm the expectations that

– on randomly generated problems, look-ahead solvers are best, while
– on large problems, look-back solvers are best

also in answer set programming. Given that all the different search strategies are implemented, combined, and analyzed on a common platform, our results are not biased by differences in the quality of the underlying implementations.

4.3. Implementation of $test(S, \Pi)$

Consider a call to $test(S, \Pi)$, that is, such that S is a total assignment not falsifying the completion of Π. Let X be the set of atoms in S and in Π.

The primary goal of $test(S, \Pi)$ is to

1. verify if X is an answer set of Π, and
2. compute a subset R of S to be used as reason if the SAT solver uses learning.

In our implementation, the computation of the reason involves looking for a loop formula of Π that is falsified by S. To describe the procedure, we use the following terminology. In a graph, a loop L is *maximal* if it is a strongly connected component; it is also *terminating* (using standard definition) if there is no other maximal loop L' with a path from L to L'.

Assuming learning is enabled, $test(S, \Pi)$ consists of the following steps:

1. Compute the reduct Π^X of Π with respect to X;
2. Compute the answer set X' of Π^X in linear time via the Dowling-Gallier procedure [13];

[3] The terminology "small but relatively difficult" and "large but relatively easy" refers to the number of atoms and is used to convey the basic intuitions about the instances. To get a more precise idea in SAT, consider that in the SAT2003 competition, instances in the random and industrial categories had, on average, 442 and 42,703 atoms, respectively [37].

3. If $S' = X \setminus X'$ is empty, then return *True*: X is an answer set of Π (X' is by construction guaranteed to be a subset of X). Otherwise, go to Step 4.
4. Considering the dependency graph of Π restricted to the nodes in S', a terminating maximal loop L is computed, and the corresponding loop formula F is determined. X does not satisfy F: This result has been established in [47] for basic programs, and it has been generalized to include nested programs in [41].
5. F has the form (6), and since X is a superset of L, X does not satisfy each of the formulas in $R(L)$. Since each formula G in $R(L)$ is a conjunction of literals, G is traversed looking for a literal whose complementary belongs to S. This literal is added to the returned reason and the whole procedure is iterated till all the formulas in $R(L)$ are analyzed.

Each of the above steps takes at most linear time in the size of the program. The above described procedure for computing a maximal terminating loop falsified by S is the same as the one described in [49], generalized to handle also nested programs. The key difference between our approach and Lin and Zhao's is that they add the whole loop formula to the input set of clauses and then call again the SAT solver from scratch. Here, the loop formula is used only to find a (small) subset of S to be used as reason. As we already said, our procedure is guaranteed to be sound, complete, and working in polynomial space even assuming the entire set S is returned (thus, without making any use of loop formulas).

If learning is disabled (as in CMODELS2 version with backtracking), step 3 in the above description of *test*(S, Π) is modified in order to return *True* if $X \setminus X'$ is empty, and *False* otherwise.

5. Experimental Results

5.1. Solvers, Benchmarks, and Setting

To evaluate the effectiveness of our approach, we comparatively tested CMODELS2 against other state-of-the-art systems on a variety of benchmarks. The systems we considered were SMODELS version 2.27, SMODELS$_{cc}$ version 1.08, ASSAT version 2.00, and DLV release of 2005-02-23.[4] We remark that while SMODELS, SMODELS$_{cc}$, ASSAT and CMODELS2 use LPARSE as a preprocessor and thus can be run on the same input files, DLV does not. Hence, DLV has been run only on a few benchmarks. Analogously, ASSAT can deal only with basic programs and thus has not been run on some instances. We also note that DLV is a system specifically designed for disjunctive logic programs; thus, very different results can be obtained depending on the specific encoding being used.

Considering CMODELS2, we have the possibility to combine different look-ahead/look-back search strategies and heuristics. To keep track of which combination we are using, we will refer to a combination of search strategies and heuristics using an acronym where the first, second, and third letter denote the look-ahead,

[4] See http://www.tcs.hut.fi/Software/smodels/, http://www.nku.edu/~wardj1/Research/smodels_cc.html, http://assat.cs.ust.hk/, http://www.dbai.tuwien.ac.at/proj/dlv/

look-back, and heuristic used, respectively. We considered four combinations of reasoning strategies:

1. ulv: our default answer set solver, incorporating a MCHAFF-like look-back SAT solver, with standard $\underline{\text{U}}$nit propagation, backtracking enhanced with $\underline{\text{L}}$earning, and $\underline{\text{V}}$SIDS heuristic.
2. fbu: a standard SATZ-like look-ahead solver, with unit propagation enhanced with $\underline{\text{F}}$ailed literal detection, standard $\underline{\text{B}}$acktracking, and the $\underline{\text{U}}$nit heuristic.
3. flv: an hybrid solver, featuring unit propagation enhanced with $\underline{\text{F}}$ailed literal detection, backtracking enhanced with $\underline{\text{L}}$earning, and the $\underline{\text{V}}$SIDS heuristic.
4. flu: another hybrid solver, featuring unit propagation enhanced with $\underline{\text{F}}$ailed literal detection, backtracking enhanced with $\underline{\text{L}}$earning, and the $\underline{\text{U}}$nit heuristic.

We considered only these four combinations of reasoning strategies and heuristics because they are the most significant and the other possible combinations do not even make sense: the VSIDS heuristic requires "learning" in order to be significant, while the unit heuristic requires failed-literal. The two solvers that we expect to perform best on randomly generated programs and on large programs are fbu and ulv, respectively. Assuming that the expectations are met, the performances of the two hybrid solvers are of interest in order to

– determine whether adding a powerful look-back (resp. look-ahead) to a look-ahead (resp. look-back) solver leads to better performances on randomly generated (resp. large) programs; and
– get an indication of which combination of reasoning strategy is the most promising on nonrandomly generated and non-large programs.

All the solvers were run in their plain (optimal) configuration unless suggested by the authors. For examples, SMODELS$_{cc}$ has been run with option "-nolookahead" (look-ahead turned off) as explicitly suggested by the authors in the SMODELS$_{cc}$'s home page. For ASSAT, we had to increase its internal limit on the number of atoms in the (grounded) logic program (variable C_MAXATOM).

Our test set of benchmarks includes both tight and non-tight, both randomly generated and nonrandomly generated programs. Each benchmark belongs to a class of publicly available programs that have been used before in the literature or to a class of benchmarks for which a generator is available. In this latter case, we may have generated bigger instances than those reported in the literature. In order to validate our expectations, we divide the benchmarks in three categories: (*i*) randomly generated programs, (2) "large" programs with more than (approximately) 10,000 atoms, and (*iii*) other problems not falling in the previous categories. We say that a program is basic when each rule has the form (1) where $n = m$, and nonbasic when a program contains choice rules or weight constraints. Recall that choice and weight constraint rules are eliminated with the help of auxiliary atoms and nested rules of the form (1).

The results of the solvers on the most difficult instances of each class are given by means of tables, as is customary in the answer set literature. In the tables,

1. The first column is a progressive number.
2. The second column is the ratio between number of rules and number of atoms for random problems and the name of the benchmark if it is a nonrandomly generated program.

3. The third column contains the number of atoms (#VAR) after grounding. For nonrandom problems, a "+" to the right of the number indicates that the instance has answer sets.
4. The remaining columns each indicate the performance of an individual solver.

For each row, the best result is in bold, and the results within a factor of 2 from the best are underlined.

All the tests were run on a Pentium IV PC, with a 2.8 GHz processor, 1,024 MB RAM, running Linux. For SMODELS, SMODELS$_{cc}$, ASSAT, and CMODELS2, the time taken by LPARSE is not counted.[5] Further, each system was stopped after 3,600 s of CPU time on nonrandom problems and 600 s on random problems, or when it exceeded all the available memory. In the tables, these cases are denoted with "TIME" and "MEM," respectively. Otherwise, the tables report the CPU times in seconds needed by each solver to solve the problem. Some of the results presented here were also presented in [25–27]. All the experiments have been relaunched, however, thus explaining the minor differences in the results, especially with [27], where the experiments were conducted on a Pentium IV PC, with 1.8 GHz processor, 512 MB RAM DDR 266 MHz, running Linux.

5.2. Randomly Generated Programs

Table I shows the results for "small" programs, randomly generated according to two methodologies:

1. Problems (1)–(10) are translations of randomly generated k-SAT instances. A k-SAT instance consists of L distinct clauses, where each clause is generated by randomly selecting k different atoms and negating each with probability 0.5. The number of distinct possible atoms in a k-SAT instance is a priori fixed and denoted with N. Then, each k-SAT instance F is converted to a program as follows
   - If $C = (l_1 \vee \ldots \vee l_k)$, we define $sat2tlp\,(C)$ to be the rule $\bot \leftarrow not\,l_1, \ldots, not\,l_k$, where $not\,l_i$ is p if $l_i = \neg p$ and is $not\;p$ if l_i is the atom p;
   - Then, if F is a k-SAT instance, the *translation of* F, is

   $$\cup_{C\in F} sat2tlp(C) \cup \cup_{p\in P}\{p \leftarrow not p',\ p' \leftarrow not p\},$$

   where, for each atom $p \in P$, p' is a new atom associated to p. These benchmarks are tight and have been used in [17, 61, 64].

2. Lines (11)–(20) correspond to programs randomly generated according to the methodology proposed in [48]. Given a set P with N atoms and a positive number k, a randomly generated rule has
   a) the head, which is randomly selected from P, and
   b) the body consisting of $k-1$ different atoms, each randomly selected from P and negated with probability 0.5.

[5] Adding the times of LPARSE would not change the picture for DLV when compared to CMODELS2 and other systems.

Table I Performances on randomly generated logic programs.

	PB	#VAR	SMODELS	SMODELS$_{cc}$	ASSAT	DLV	ulv	flv	flu	fbu
1	4	300	1.2	7.23	0.85	2.55	**0.59**	0.8	1.5	1.37
2	4.5	300	**39.97**	TIME	TIME	130.49	TIME	TIME	115.29	40.38
3	5	300	**7.57**	149.37	TIME	26.78	456.22	538.89	17.64	11.32
4	5.5	300	**2.26**	33.12	94.78	7.37	72.83	53.26	4.42	3.59
5	6	300	**1.05**	12.72	22.5	3.26	24.73	21.89	1.83	1.63
6	4	350	4.11	12.6	13.4	49.3	**2.2**	5.74	11.48	8.85
7	4.5	350	**318.1**	TIME	TIME	TIME	TIME	TIME	TIME	384.66
8	5	350	**44.2**	TIME	TIME	147.16	TIME	TIME	134.34	54.07
9	5.5	350	**12.66**	252.11	TIME	32.07	TIME	506.08	20.37	13.61
10	6	350	**3.37**	37.99	174.61	8.76	95.61	104.36	6.05	4.86
11	4	200	3.3	2.02	2.44	32.39	5.34	3.32	1.93	**1.75**
12	4.5	200	6.84	**1.7**	3.28	83.63	6.15	5.82	2.09	1.93
13	5	200	22.8	**2.5**	8.21	82.97	9.82	9.02	3.88	3.33
14	5.5	200	9.42	**1.76**	4.14	39.47	7.5	6.38	2.97	2.85
15	6	200	8.12	**0.85**	1.4	23.93	3.24	2.95	1.25	1.53
16	4	300	298.67	73.64	234.09	TIME	265.43	218.48	41.97	**31.05**
17	4.5	300	TIME	TIME	TIME	TIME	TIME	TIME	190.73	**135.11**
18	5	300	TIME	412.69	TIME	TIME	TIME	TIME	136.67	**99.75**
19	5.5	300	TIME	233.72	TIME	TIME	TIME	TIME	129.29	**78.63**
20	6	300	TIME	191.62	TIME	TIME	TIME	TIME	107.34	**65.83**

Problems (1)–(10) are tight programs being the translation of 3-SAT benchmarks. Problems (11)–(20) are randomly generated logic programs using Lin and Zhao's methodology.

A randomly generated program with L rules consists of L randomly generated distinct rules. In general these randomly generated programs are non-tight.

Both categories of problems have been generated with $k = 3$ and L varying from $0.5 \times N$ to $12 \times N$ with step 0.5. N has been fixed to 300 and 350 for the instances being the translation of k-SAT problems, and to 200 and 300 for the instances generated according to Lin and Zhao's methodology.

For each ratio L/N (indicated in the column "PB"), we generated 10 instances. The table presents the median results for the most difficult five ratios (the other being quite easily solved by all the systems).

On these benchmarks fbu has the overall best performances: it is almost always the fastest system or within a factor of 2 from the fastest. SMODELS is faster than fbu in the median case when considering the translation of k-SAT instances. On these benchmarks, however, SMODELS times out on two programs when $N = 300$, while fbu times out only on one program.[6] The good performance of SMODELS on these benchmarks is not surprising given that SMODELS also implements failed literal detection, together with a heuristic similar to our unit heuristic. However, considering the programs generated according to Lin and Zhao's methodology, we see that

[6] Increasing N to 400, we get the same picture: SMODELS is faster than fbu in the median case, but it times out on 11 programs, while fbu times out on 10. We decided not to show the results for $N = 400$ because most of the other solvers time out also in the median case for most of the ratios L/N.

SMODELS is not competitive with fbu, which (together with flu) scales much better than all the rival systems.

Considering CMODELS2's combinations, fbu is the fastest (confirming expectations), but flu also performs quite well. Coupling these facts with the bad performance of flv, we find that the unit heuristic is very effective on these benchmarks and makes learning useless.

5.3. Large Programs

Table II shows the results when considering large (i.e., with approximately 10,000 or more atoms) programs. As in the previous subsection, the table is divided in two parts:

1. Programs (21)–(26) are tight: In particular (21)–(23) and (24)–(26) encode respectively blocks world planning and four-colorability problems in a graph with V vertexes. V is the number in the label "4cV" in column PB. All the tight programs but bw*e9 have answer sets and are available at the SMODELS Web site.
2. Programs (27)–(39) are non-tight. In particular, we consider Hamiltonian circuit problems on complete graphs, using both the basic encoding of Niemela [54] [programs (27)–(31)] and the nonbasic encoding [programs (32)–(36)] from http://www.cs.engr.uky.edu/ai/benchmark-suite/ham-cyc.sm. The remaining three programs in the table are related to the problem of checking requirements in a deterministic automaton and are described in [9]. The first of these three programs is the biggest instance in the suite of the "IDFD" problems, while the other two programs belong to the "Morin" suite.

Table II Performances on large programs.

	PB	#VAR	SMODELS	SMODELS$_{cc}$	ASSAT	DLV	ulv	flv	flu	fbu
21	bw*d9	9956+	6.76	7.63	1.72		**1.02**	5.84	2.69	2.75
22	bw*e9	12260	4.3	4.51	4.22		**0.98**	1.91	1.92	1.93
23	bw*e10	13482+	11.15	12.43	2.66		**1.29**	7.51	5.03	4.95
24	4c1000	14955+	22.28	4.95	0.6		**0.48**	37.86	15.41	15.23
25	4c3000	44961+	202.84	1143.13	**2.19**		8.86	369.27	144.12	142.83
26	4c6000	89951+	856.13	TIME	**14.85**		99.50	TIME	583.55	578.98
27	np60c	10742+	242.61	30.81	84.87	361.80	**2.83**	1611.32	44.12	44.11
28	np70c	14632+	557.08	55.31	520.80	798.96	**4.69**	TIME	97.44	97.87
29	np80c	19122+	1001.88	90.59	53.25	1587.60	**7.2**	TIME	195.08	190.49
30	np90c	24212+	2064.61	144.72	1416.24	2807.84	**10.42**	TIME	364.54	357.92
31	np100c	29902+	3573.19	215.37	TIME	TIME	**14.23**	TIME	610.2	608.96
32	np60c	10683+	7.05	3.82			**3.55**	340.86	8.03	7.82
33	np70c	14563+	15.67	**5.92**			10.54	782.69	15.39	14.92
34	np80c	19043+	32.29	**9.01**			15.05	1538.86	23.63	25.94
35	np90c	24123+	53.21	**14.13**			32.19	2918.82	38.75	50.08
36	np100c	29803+	83.11	**14.95**			34.18	TIME	59.15	62.64
37	mutex4	14698+	14.14	5.35	0.54	367.89	**0.46**	28.29	28.3	28.26
38	mutex3	278074+	163.94	**110.27**	MEM		TIME	TIME	TIME	TIME
39	phi3	16930+	3.23	3.04	53.28		**1.43**	55.62	12.15	TIME

Problems (21)–(26) are tight. Problems (27)–(39) are non-tight.

Table III Performances on nonrandom, non-large programs.

	PB	#VAR	SMODELS	SMODELS$_{cc}$	ASSAT	DLV	ulv	flv	flu	fbu
40	d*12*i*9	1186	368	435.48			**223.93**	290.15	353.53	TIME
41	k*i*29	3199	990.95	**20.88**			415.54	204.87	44.14	589.45
42	k*s*29	3169	909.46	**16.89**			353.69	1028.77	59.99	TIME
43	m*3*i*10	1933+	10.98	**1.65**			16.23	32.23	26.71	16.55
44	m*4*i*12	3475+	1132.16	**3.82**			1063.15	867.49	TIME	3229.09
45	m*4*s*8	1586+	89.26	**1.3**			17.02	27.59	421.30	327.55
46	q*i*17	2201	517.64	**53.71**			1539.96	505.15	259.05	816.26
47	e*3*i*15	7832+	35.58	77.02			479.28	TIME	7.15	**6.87**
48	e*4*i*13	6447	221.18	56.21			87.63	567.27	20.02	**19.41**
49	schur1.4-43	736+	**0.43**	0.95	0.67	590.57	1.4	2.07	0.82	0.88
50	schur1.4-44	753+	**0.44**	91.25	1.07	TIME	5.97	5.62	92.63	43.01
51	schur1.4-45	770	571.17	1110.68	434.93	TIME	229.04	417.34	244.35	**116.51**
52	schur2.4-43	564+	**0.33**	0.56			1.27	1.04	0.4	0.38
53	schur2.4-44	577+	82.72	47.78			6.14	**2.8**	47.99	18.93
54	schur2.4-45	590	578.73	672.86			226.69	392.78	148.39	**63.2**
55	15puz.18	5945+	17.55	6.94	1.06	141.68	**0.98**	2.9	9.85	9.24

Table III Continued.

	PB	#VAR	SMODELS	SMODELS$_{CC}$	ASSAT	DLV	ulv	flv	flu	fbu
56	15puz.19	6258+	20.94	7.14	3.61	208.41	**1.35**	2.93	11.65	10.76
57	15puz.20	6571	70.27	8.22	4.59	TIME	**1.28**	10.22	64.54	82.68
58	pige.9.10	210	44.77	65.91	**1.1**		1.26	4.33	1259.84	32.06
59	pige.10.11	253	484.63	1029.38	23.83		**12.41**	55.46	TIME	339.06
60	pige.51.50	5252+	106.79	24.29	2.49		**1.63**	221.33	6.85	7.26
61	8 i-1	2329	7.48	7.17	0.86		**0.49**	0.85	0.84	0.81
62	11 i-1	4760	36.18	35.53	3.15		**1.64**	4.92	2.47	2.44
63	8 i	2627+	17.35	9.30	0.98		**0.63**	1.27	0.89	0.88
64	11 i	5301+	37.71	43.90	3.59		**2.16**	15.55	6.07	5.79
65	8 i+1	2925+	12.08	15.17	**1.09**		1.34	4.31	1.34	1.37
66	11 i+1	5842+	54.30	62.39	3.9		**2.49**	24.27	22.01	19.71
67	8 i-1	1897	0.53	0.66			**0.15**	0.29	0.27	0.27
68	11 i-1	3812	1.6	1.96			**0.39**	1.71	0.75	0.7
69	8 i	2132+	0.76	0.8			**0.22**	0.42	0.27	0.3
70	11 i	4233+	1.85	2.57			**0.52**	6.76	1.9	1.88
71	8 i+1	2367+	1.8	1.05			0.68	1.65	**0.47**	0.49
72	11 i+1	4654+	2.5	4.12			**0.6**	10.42	5.26	5.21
73	d*10*i*12	1488+	132.72	**2.25**			488.76	1212.89	152.8	TIME
74	d*10*s*9	1140+	9.75	**3.11**			6.38	19.31	87.64	TIME
75	d*12*s*10	1511+	296.45	**1.1**			53.2	165.9	733.9	TIME
76	d*8*i*10	1003+	1.76	2.42			12.28	25.03	**1.21**	11.86
77	d*8*s*8	819+	0.73	**0.14**			0.47	3.73	2.38	1221.53

Benchmarks (40)–(60) are tight, while the others are non-tight.

Overall, the picture that emerges is that ulv is the fastest system. Even though SMODELS is the only system that never times out, it is far slower than ulv (and other systems as well) on many problems. The good performance of ulv is particularly remarkable given that the test suite contains Hamiltonian circuit problems and these benchmarks have exponentially many loops. Thus, one would expect these problems to be difficult for ASSAT, but also for all CMODELS2 versions in the case it will generate and then reject (exponentially) many candidate answer sets. This is not the case, however, at least for ulv. The table also shows an instance on which ASSAT blows up in memory. As a matter of fact, ASSAT exceeds all the available memory also on other instances, not shown here because all the other systems time out on them.

Considering the different CMODELS2 versions – beside the fact that ulv is the best version – by comparing ulv and flv we see that adding failed-literal usually causes a significant degradation in the performances. These results match expectations. Indeed, ulv (and also ASSAT) uses a MCHAFF-like solver and performs a few operations at each (branching) node. For (very) large programs, even a linear-time (in the number of atoms) operation can be prohibitive if performed at each branching node. Interesting, considering flv, flu, and fbu, we see that the last system almost always performs better than the second and that the second is better than the first. On these benchmarks, adding learning to a look-ahead solver does not help. However, the gap between fbu and flu is not big. Thus, adding learning to fbu does not help but does not hurt too much. We believe that this situation is due to the lazy data structures used by all the CMODELS2 versions, which are fundamental to keeping low the burden of managing learned clauses.

5.4. Nonrandom, Nonlarge Programs

Table III contains the results on nonrandom, non-large logic programs. In more detail:[7]

1. Benchmarks (40)–(48) and (73)–(77) are respectively tight and non-tight bounded model-checking (BMC) problems of asynchronous concurrent systems, as described in [31]. These problems are about proving properties in a given number of steps, represented as the last number in the instance name.
2. Benchmarks (49)–(54) are about the Schur numbers problem, expressed as basic (49)–(51) and nonbasic (52)–(54) programs. The label “schurX.K-N” refers to a problem where, given a positive integer n, the set of integers N defined as N $= \{1, 2, \ldots, n\}$ has to be partitioned into K bins such that each bin is sum-free; that is, for each Z∈N and Y∈N (1) Z and Z+Z are in different bins, and (2) if Z and Y are in the same bin, then Z+Y is in a different bin. We denote with X=1 the basic encoding and with X=2 the nonbasic encoding.
3. Benchmarks (55)–(57) are programs encoding the 15-puzzle problem. In a label “15puz.M,” M denoted the number of moves in which the final configuration has to be reached. The initial configuration is not fixed and varies from program to program.

[7] Benchmarks (40)–(48), (73)–(77), and the generator are available at http://www.tcs.hut.fi/~kepa/experiments/boundsmodels/. Benchmarks (49)–(57) are available at the ASPARAGUS Web page http://asparagus.cs.uni-potsdam.de/ Benchmarks (58)–(60) belong to the SMODELS test suite and are publicly available at http://www.tcs.hut.fi/Software/smodels/tests/, encoding by Niemela [54].

4. Benchmarks (58)–(60) are tight programs encoding the pigeons problems. In a label "pige.$h.p$," h denotes the number of holes and p the number of pigeons.
5. Benchmarks (61)–(72) are blocks world planning problems encoded as basic programs in lines (61)–(66) and as nonbasic programs in lines (67)–(72)), the formulations due to Erdem [15]. In the tables, in the column PB the "8" or "11" represents the number of blocks, while an "i" (standing for "number of steps") means that the instance corresponds to the problem of finding a plan in "i" steps, where "i" is the minimum integer for which a plan exists. Thus, the instances with "i" and "$i+1$" in the label admit at least one answer set, while those with "$i-1$" do not have answer sets. Technically speaking, these programs are non-tight. However, these problems are "tight on their completion models" [3]: If Π is one such program, each model of the completion of Π is guaranteed to be also an answer set of Π.

For these benchmarks, results are mixed. On BMC problems, SMODELS$_{cc}$ has the best performance overall, while on the other benchmarks ulv has the best performance overall. Most interesting, no version of CMODELS2 dominates the others on the BMC problems. Given this fact and the good performance of SMODELS$_{cc}$ on BMC instances, we believe that on nonrandom, non-large problems the "overall best" solver is somewhere between ulv and fbu, that is, that it can be obtained by adding a little bit of failed-literal detection to ulv. This can be done in several ways, for example, by checking whether a literal is failed only if it belongs to a pool of "most promising" literals (as, e.g., is done by SATZ) or by checking all the literals but not at each branching node. All of this is subject of future research.

We note that, overall, flu is better than flv. This result can be explained by the bad interaction between failed-literal and VSIDS. For nonrandom, large formulas, this phenomenon was already shown to hold in SAT [28].

5.5. CMODELS2 and the Other Systems

Given the results of the experimental analysis, we now summarize what we believe to be the advantages and disadvantages of each system we considered, from both a theoretical and a practical point of view, when compared to CMODELS2.

SMODELS [61]. SMODELS is a system for nondisjunctive answer set programming. Its algorithm has been inspired by Davis-Logemann-Loveland procedure, and incorporates powerful pruning techniques.

SMODELS is also the basic engine for the solver for disjunctive logic programming called GNT [34, 35]. A key feature of SMODELS is that it is a native system, that is, it works directly on the input logic program. Because of this, it can take advantage of the structure of the program, for example, by keeping more compact representations of the rules than CMODELS2, which compiles down everything to a set of clauses. However, it does not incorporate some of the most recent advances, for example, learning. The experimental results for SMODELS are still positive overall, being among the best solvers in all the categories of problems we considered.

SMODELS$_{cc}$ [64]. SMODELS$_{cc}$ is SMODELS enhanced with clause-learning [53] and a BERKMIN-like heuristic [30]. SMODELS$_{cc}$ inherits from SMODELS its compact data-structures for rules. Because of such compactness, howver, the incorporation of learning in SMODELS required the construction of an implication graph, and this operation turned out to be relatively complex and costly when compared to the analogous construction in a SAT solver. Indeed, SMODELS$_{cc}$ cannot deal with programs containing weight constraint rules, and this also witnesses the difficulty of implementing learning on top of SMODELS compact data structures for such rules. On the other hand, learning comes for free with our approach. Further, with relatively little additional programming effort, our procedure can be based on the latest SAT tools. We used our tool SIMO to validate the viability and effectiveness of the approach and obtained a solver with, for example, learning and unit propagation based on lazy data structures using a two-literal watching schema. Modifying SMODELS or SMODELS$_{cc}$ in order to use lazy data structures would require a rewriting of significant portions of the solvers. From a practical point of view, SMODELS$_{cc}$ is quite effective, especially on some classes of non-tight programs.

DLV [39]. DLV is the state-of-the-art system in disjunctive logic programming, with techniques especially tailored for this class of programs. Also DLV is a native system and its algorithm is based on the Davis-Logemann-Loveland procedure.

Since it can deal with the more expressive class of disjunctive programs, however, it needs a co-NP check to test whether a candidate model is indeed an answer set. The check is performed only if needed: In the case of nondisjunctive programs (the ones this paper faces), it is not applied.

DLV has same peculiarities. During the computation, it uses a four-valued interpretations for atoms. The truth values considered are *True*, *False*, "undefined," and "must be true"; a "must be true" atom is like an atom assigned to *True*, but it is missing a "supporting" rule that must be determined later. Moreover, DLV heuristic is guided by a preselected list of literals (PT-literals) with the aim of maintaining the candidate model as minimal as possible. The key strength of DLV is that it can deal with disjunctive logic programs. On the restricted class of nondisjunctive logic programs, however, its performance is not impressive, at least on the benchmarks that we considered and with the encodings that we used.

ASSAT [47, 49]. ASSAT has been the first ASP SAT-based system non-restricted to tight programs. The SAT solver is used as a black box, and thus ASSAT inherits all the optimizations implemented in it. ASSAT uses MCHAFF [53] as SAT solver. As we have seen, ASSAT is quite effective especially on nonrandom programs. From a theoretical perspective, the main drawback of ASSAT is that it is not guaranteed to work in polynomial space. This fact also emerges in some of the benchmarks that we considered and for which ASSAT exhausted all the available memory. From a practical point of view, ASSAT is limited to basic programs and cannot handle choice, cardinality, and weight constraint rules.

CMODELS2. CMODELS2 is a SAT-based system designed after ASSAT in order to solve its theoretical drawbacks. CMODELS2 incorporates various solvers. Our default choice for randomly generated programs is fbu, and our default choice for nonrandom programs is ulv. Experimental analysis showed that on random problems fbu has the best performance overall of all the solvers that we considered, and the same holds for ulv when considering large problems. On the other benchmarks, ulv is competitive with the best of the other solvers. These results show the effectiveness of our SAT-based approach. The results are particularly remarkable given that our two solvers implement relatively simple SAT strategies, compared to the ones now available, some of which already incorporated by various answer set solvers. For instance, ulv uses the MCHAFF heuristic, while the Berkmin heuristic (used by SMODELS$_{cc}$) is considered to be better. In fbu each not-yet-assigned literal is checked to see whether it is failed, and these checks are performed before each branching: SMODELS and SMODELS$_{cc}$ implement the correspondent strategy of failed-literal, but they check only a subset of the unassigned literals (and the unchecked are guaranteed to be not failed). We expect that the incorporation of the Berkmin heuristic and SMODELS failed-literal detection strategy in ulv and fbu, respectively, will lead to further improvements in performance on the respective application domains.

6. Conclusions and Future Work

We have presented a SAT-based procedure that (1) can deal with any logic program (2) works on a propositional formula without additional variables except for those introduced during the clause form conversion, and (3) is guaranteed to work in polynomial space. Furthermore, ASP-SAT can be easily modified in order to compute all answer sets (still working in polynomial space). We have shown how to implement ASP-SAT on top of current state-of-the-art solvers with or without learning. The experimental evaluation shows that

1. CMODELS2 is competitive with other state-of-the-art systems; and
2. depending on the type of program, different search strategies are best.

This evaluation suggests that future development of answer set solvers should be done by focusing on certain classes of problems. In our analysis we identified two classes of programs–random and large programs–that need completely different strategies. This also implies that benchmarking should be done by considering the application domain they have been developed for. This reflects what is nowadays a standard in the SAT competition, where there is a track for solvers designed for random problems and a separate track for solvers designed for large, industrial benchmarks. Solvers get designed and specialized for one track, and indeed the top performers in one track behave very badly in the other.

There are several directions in which this work can be improved.

First, CMODELS2 can be improved as a solver for nondisjunctive programs. This can be done by improving the SAT solving part, DLL, or the checking procedure, *test*.

As anticipated in the previous section, we believe that DLL performances can be improved by implementing better failed-literal detection strategies or heuristics. About the heuristics – besides those derived from the SAT literature as BERKMIN's – we believe that it is possible to design heuristics tailored for answer set solving. One such heuristic assigns atoms to *False* while branching. Intuitively, we would like to generate assignments with as many atoms as possible assigned to *False*, thus going through minimality. A first, simple implementation of this heuristic produces dramatic speedups on some domains (for instance, ulv is able to solve all the non-tight problems in Table II in a few seconds, including the mutex3 instance, i.e., the only instance on which ulv times out), but it seems to interact badly with learning in some other domains. Another possibility is to incorporate another SAT solver with the latest advancements, for example, MINISAT [14], the winner of the last SAT competition.

With regard to the checking procedure *test*, recently Gebser and Schaub [20] introduced the notion of "active elementary loop with respect to an assignment S," and they showed that the corresponding loop formula is falsified by S, like the formula associated to a maximal terminating loop. One crucial difference between an active elementary loop and a maximal terminating one is that no subloop of an active elementary loop is also falsified by S. A maximal terminating loop, on the other hand, is not always an active elementary loop of the program. It is still an open question whether the use of active elementary loops in SAT-based procedures such as CMODELS2 or ASSAT improves their performance.

Another direction of work is to extend CMODELS2 ideas in order to deal with disjunctive logic programming where, as for DLV, the co-NP check involves the use of a SAT solver. A preliminary implementation and analysis are encouraging [41], but more work has to be done in order to improve the overall efficiency of the solver.

Acknowledgments We are grateful to Paolo Ferraris, Nicola Leone, Vladimir Lifschitz, and the anonymous reviewers of this paper for their helpful comments or discussions on the subject of the paper; to Esra Erdem and Keijo Heljanko for providing us with the benchmarks; and to Francesco Calimeri for his support on DLV. This work is partially supported by MIUR (Italian Ministry of Education, University and Research), and Texas Higher Education Coordinating Board under Grant 003658-0322-2001.

References

1. Armando, A., Castellini, C., Giunchiglia, E.: SAT-based procedures for temporal reasoning. In: Lecture Notes in Computer Science, vol. 1809, pp. 97–108, (1999)
2. Armando, A., Castellini, C., Giunchiglia, E., Maratea, M.: The SAT-based approach to separation logic. J. Autom. Reason. To appear (2005)
3. Babovich, Y., Erdem, E., Lifschitz, V.: 'Fages' theorem and answer set programming. In: Proc. NMR, (2000)
4. Baral, C., Gelfond, M., Scherl, R.: 'Using answer set programming to answer complex queries. In: Workshop on Pragmatics of Question Answering at HLT-NAAC2004, (2004)
5. Barrett, C.W., Dill, D.L., Stump, A.: Checking satisfiability of first-order formulas by incremental Translation to SAT. In: Brinksma, E., Larsen, K.G. (eds.) 14th International Conference on Computer Aided Verification (CAV), vol. 2404 of Lecture Notes in Computer Science, pp. 236–249. Springer, Berlin Heidelberg New York, (2002)
6. Bayardo, R.J. Jr, Schrag, R.C.: Using CSP look-back techniques to solve real-world SAT instances. In: Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97). Menlo Park, California, pp. 203–208. AAAI (1997)
7. Ben-Eliyahu, R., Dechter, R.: Propositional semantics for disjunctive logic programs. Ann. Math. Artif. Intell. **12**, 53–87 (1996)
8. Clark, K.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Plenum, New York (1978)
9. Ştefănescu, A., Esparza, J., Muscholl, A.: Synthesis of distributed algorithms using asynchronous automata. In: Proc. CONCUR'03, vol. 2761, pp. 27–41. Springer (2003)
10. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem proving. Commun. ACM **5**(7), 394–397 (1962)
11. de Moura, L., Rueß, H., Sorea, S.: Lazy theorem proving for bounded model checking over infinite domains. In: Voronkov, A. (ed.) Automated Deduction – CADE-18, vol. 2392 of Lecture Notes in Computer Science, pp. 438–455. Springer (2002)

12. Dixon, H.E., Ginsberg, M.L., Luks, E.M., Parkes, A.J.: Generalizing Boolean satisfiability II: Theory. J. Artif. Intell. Res. (JAIR) **22**, 481–534 (2004)
13. Dowling, W., Gallier, J.: Linear-time algorithms for testing the satisfiability of propositional Horn formulae. J. Log. Program. **3**, 267–284 (1984)
14. Eén, N., Sörensson, N.: An extensible SAT-solver'. In: Theory and Applications of Satisfiability Testing, 6th International Conference, *SAT* 2003. Santa Margherita Ligure, Italy, May 5–8, 2003 Selected Revised Papers, pp. 502–518, (2003)
15. Erdem, E.: Theory and applications of answer set programming. PhD thesis, University of Texas at Austin, (2002)
16. Erdem, E., Lifschitz, V.: 'Fages' theorem for programs with nested expressions. In: Proc. International Conference on Logic Programming, pp. 242–254, (2001)
17. Faber, W., Leone, N., Pfeifer, G.: Experimenting with heuristics for answer set programming. In: IJCAI, pp. 635–640, (2001)
18. Fages, F.: Consistency of Clark's completion and existence of stable models. J. Methods Logic Comput. Sci. **1**, 51–60 (1994)
19. Ferraris, P., Lifschitz, V.: Weight constraints as nested expressions. Theory and Practice of Logic Programming **5**, 45–74 (2005)
20. Gebser, M., Schaub, T.: Loops: Relevant or redundant?. In: Proceedings of 8th International Conference on Logic Programming and Nonmonotonic Reasoning, pp. 53–65. Springer (2005)
21. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Logic Programming: Proceedings of the Fifth Int'l Conf. and Symp., pp. 1070–1080, (1988)
22. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Gener. Comput. **9**, 365–385 (1991)
23. Gent, I., Maaren, H.V., Walsh, T. (eds.) SAT2000. Highlights of Satisfiability Research in the Year 2000. IOS (2000)
24. Giunchiglia, E., Giunchiglia, F., Tacchella, A.: SAT-based decision procedures for classical modal logics. J. Autom. Reason. **28**, 143–171 (2002). Reprinted in [23]
25. Giunchiglia, E., Maratea, M.: On the relation between SAT and ASP procedures (or, between smodels and cmodels). In: Proceedings of the 21th International Conference on Logic Programming (ICLP), pp. 37–51. Springer (2005a)
26. Giunchiglia, E., Maratea, M.: Evaluating search strategies and heuristics for efficient answer set programming. In: Advanced in Artificial Intelligence: Conference of the Italian Association for Artificial Intelligence, AI*IA '05, Milan, Italy, September 20–23, 2005, Proceedings, pp. 37–51. Springer (2005b)
27. Giunchiglia, E., Maratea, M., Lierler, Y.: SAT-based answer set programming. In: Proc. 19th National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California. AAAI, The MIT (2004)
28. Giunchiglia, E., Maratea, M., Tacchella, A.: In: Effectiveness of look-ahead techniques in a modern SAT solver. In: 9th International Conference on Principles and Practice of Constraint Programming (CP-03), pp. 842–846, (2003)
29. Giunchiglia, E., Maratea, M., Tacchella, A., Zambonin, D.: Evaluating search heuristics and optimization techniques in propositional satisfiability. In: Automated Reasoning, First International Joint Conference (IJCAR), vol. 2083 of Lecture Notes in Computer Science, pp. 347–363. Springer (2001)
30. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT solver. In: Proc. of the Design, Automation and Test in Europe Conference and Exposition 2003, pp. 142–149. IEEE Computer Society (2003)
31. Heljanko, K., Niemelä, I.: Bounded LTL model checking with stable models. Theory and Practice of Logic Programming **3**(4&5), 519–550 (2003). Also available as (CoRR: arXiv:cs.LO/0305040)
32. Janhunen, T.: Translatability and intranslatability results for certain classes of logic programs'. Series A: Research report 82, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland (2003)
33. Janhunen, T.: Representing normal programs with clauses. In: Proc. of 16th European Conference on Artificial Intelligence, ECAI 2004, pp. 358–362. IOS (2004)
34. Janhunen, T., Niemelä, I.: GnT – A solver for disjunctive logic programming. In: Proc. of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), pp. 331–335. Springer (2004)
35. Janhunen, T., Niemelä, I., Seipel, D., Simons, P., You, J.-H.: Unfolding partiality and disjuntion in stable model semantics. Accepted to the ACM Transaction on Computational Logic (2005)

36. Lahiri, S.K., Seshia, S.A., Bryant, R.E.: Modeling and verification of out-of-order microprocessors in UCLID. In: Formal Methods in Computer-Aided Design, 4th International Conference, FMCAD 2002, Portland, Oregon, November 6–8, 2002, Proceedings, pp. 142–159, (2002)
37. Le Berre, D., Simon, L.: The essentials of the SAT'03 Competition'. In: Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5–8, 2003 Selected Revised Papers, vol. 2919 of LNCS, (2003)
38. Lee, J., Lifschitz, V.: Loop formulas for disjunctive logic programs. In: Proc. ICLP-03, (2003)
39. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. Accepted to ACM Transaction on Computational Logic (ToCL), (2005)
40. Li, C.M., Anbulagan: Heuristics based on unit propagation for satisfiability problems. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97). San Francisco, pp. 366–371, Morgan Kaufmann, (1997)
41. Lierler, Y.: Disjunctive answer set programming via satisfiability. In: Answer Set Programming, vol. 142 of CEUR Workshop Proceedings, (2005)
42. Lierler, Y., Lifschitz, V.: Computing answer sets using program completion. Available at http://www.cs.utexas.edu/users/tag/cmodels.html, 2003
43. Lifschitz, V.: Foundations of logic programming. In: Brewka, G. (ed.) Principles of Knowledge Representation. CSLI, pp. 69–128, (1996)
44. Lifschitz, V., Razborov, A.: Why are there so many loop formulas? ACM Transactions on Computational Logic. To appear (2004)
45. Lifschitz, V., Tang, L.R., Turner, H.: Nested expressions in logic programs. Ann. Math. Artif. Intell. **25**, 369–389 (1999)
46. Lin, F., Zhao, J.: On tight logic programs and yet another translation from normal logic programs to propositional logic. In: Proc. IJCAI, (2003a)
47. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT Solvers. In: Proc. 18th National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI-02). Menlo Park, California, pp. 112–118. AAAI (2002)
48. Lin, F., Zhao, Y.: Answer set programming phase transition: A study on randomly generated programs. In: Proc. ICLP, (2003b)
49. Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by SAT solvers. Artif. Intell. **157**(1–2), 115–137 (2004)
50. Lloyd, J., Topor, R.: Making Prolog more expressive. J. Log. Program. **3**, 225–240 (1984)
51. Marek, V., Subrahmanian, V.: The relationship between logic program semantics and non-monotonic reasoning. In: Levi, G., Martelli, M. (eds.) Logic Programming: Proceedings of the 6th Int'l Conf., pp. 600–617, (1989)
52. Marek, V., Truszczynski, M.: Stable models as an alternative programming paradigm. In: The Logic Programming Paradigm: a 25 Years perspective, Lecture Notes in Computer Science. Springer (1999)
53. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. 38th Design Automation Conference (DAC'01), pp. 530–535, (2001)
54. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Ann. Math. Artif. Intell. **25**, 241–273 (1999)
55. Nieuwenhuis, R., Oliveras, A.: DPLL(T) with exhaustive theory propagation and its application to difference logic. In: Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6–10, 2005, Proceedings, pp. 321–334, (2005)
56. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., Barry, M.: An A-Prolog decision support system for the space shuttle. In: Working Notes of the AAAI Spring Symposium on Answer Set Programming, (2001)
57. Plaisted, D., Greenbaum, S.: A structure-preserving clause form translation. J. Symbol. Comput. **2**, 293–304 (1986)
58. Sheridan, D.: The Optimality of a Fast CNF Conversion and its use with SAT. In: Proceedings of SAT, International Conference on Theory and Applications of Satisfiability Testing, Vancouver (Canada), (2004)
59. Siekmann, J., Wrightson, G. (eds.) Automation of Reasoning: Classical Papers in Computational Logic 1967–1970, Vol. 1–2. Springer (1983)
60. Silva, J.P.M., Sakallah, K.A.: GRASP – A New Search Algorithm for Satisfiability. Technical report, University of Michigan, (1996)

61. Simons, P., Niemelä, I., Timo, S.: Extending and implementing the stable model semantics. Artif. Intell. **138**(1–2), 181–234 (2002)
62. Syrjanen, T.: Lparse Manual. http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz, 2003
63. Tseitin, G.: On the Complexity of Proofs in Propositional Logics. Semin. Math. **8** (1970**)**. Reprinted in [59].
64. Ward, J., Schlipf, J.S.: Answer set programming with clause learning. In: Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, Florida, January 6–8, 2004, Proceedings, pp. 302–313 (2004)
65. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in a Boolean satisfiability solver. In: International Conference on Computer-Aided Design (IC-CAD'01), pp. 279–285, (2001)