Fast SAT-based Answer Set Solver *

Zhijun Lin and **Yuanlin Zhang** and **Hector Hernandez**
Computer Science Department
Texas Tech University
2500 Broadway, Lubbock, TX 79409 USA
{lin, yzhang, hector}@cs.ttu.edu

Abstract

Recent research shows that SAT (propositional satisfiability) techniques can be employed to build efficient systems to compute answer sets for logic programs. ASSAT and CMODELS are two well-known such systems. They find an answer set from the full models for the completion of the input program, which is (iteratively) augmented with loop formulas. Making use of the fact that, for non-tight programs, during the model generation, a partial assignment may be extensible to a full model but may not grow into any answer set, we propose to add answer set extensibility checking on partial assignments. Furthermore, given a partial assignment, we identify a class of loop formulas that are "active" on the assignment. These "active" formulas can be used to prune the search space. We also provide an efficient method to generate these formulas. These ideas can be implemented with a moderate modification on SAT solvers. We have developed a new answer set solver SAG on top of the SAT solver MCHAFF. Empirical studies on well-known benchmarks show that in most cases it is faster than the state-of-the-art answer set solvers, often by an order of magnitude. In the few cases when it is not the winner, it is close to the top performer, which shows its robustness.

Introduction

Logic programming with answer sets semantics (Gelfond & Lifschitz 1988), and propositional satisfiability (SAT) are closely related. It is well-known that an answer set of a logic program is also a model of its completion (Clark 1978). The converse holds for tight programs (Fages 1994). For non-tight programs, Lin and Zhao (2002) show that by adding loop formulas to the completion, one can obtain a one-to-one correspondence between the answer sets of a logic program and the models of its extended completion. Lee and Lifschitz (2003) generalize the concept of loop formula for logic programs with nested expressions. As a result, two SAT-based answer set solvers were implemented: ASSAT by Lin and Zhao (2002) and CMODELS by Lierler and Maratea (2004).

Both ASSAT and CMODELS look for answer sets of a logic program from the full models of its completion. These models are generated by a SAT solver. Observing that, for non-tight programs, during the model generation, a partial assignment may be extensible to a full model but may not grow into any answer set, we propose a new answer set solving procedure that initiates answer set extensibility checking before a full model is generated. We find that some loop formulas are responsible for the checking results, and they can be further used to prune the search space. This new approach has been proved very effective by our preliminary empirical studies.

This paper is organized as follows. First we introduce basic definitions and notations. Then we present the new procedure, followed by the results on the connection between the loop formulas and the answer set extensibility checking. Finally we describe our implementation and report the experimental results before the conclusion.

Background

Given a set of atoms $A = \{a_1, \ldots, a_k\}$, $not(A)$ denotes the set of literals $\{\neg a_1, \ldots, \neg a_k\}$. Given a set of literal $B = \{b_1, \ldots, b_k\}$, B^+ denotes the set $\{b \mid b \in B\}$ and B^- denotes the set $\{b \mid \neg b \in B\}$.

A *logic program* is a finite set of rules of the form

$$a_0 \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n \qquad (1)$$

where a_i's are atoms. We assume all programs are fully grounded.

For a rule r of type (1), we denote $\{a_0\}$ by $head(r)$, its *positive body* $\{a_1, \ldots, a_m\}$ by $pos(r)$, and its *negative body* $\{a_{m+1}, \ldots, a_n\}$ by $neg(r)$. When $head(r)$ is not empty, we sometimes abuse it to denote a_0. We use $BC(r)$ to denote the propositional formula

$$a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n.$$

In the special case when $pos(r) \cup neg(r) = \emptyset$, $BC(r) = true$.

A set of atoms M satisfies a rule r (denoted by $M \models r$), if $pos(r) \not\subseteq M$ or $neg(r) \cap M \neq \emptyset$ or $head(r) \in M$. A logic program P is said to be *positive* if $neg(r) = \emptyset$ for every $r \in P$. A set of atoms A is an answer set of a positive program P if A satisfies every rule in P and A is minimal (under set inclusion). The *reduct* of a logic program P w.r.t. a set of atoms M, written as P^M, is the positive program $\{head(r) \leftarrow pos(r) \mid r \in P, neg(r) \cap M =$

*The research leading to the results in this paper was funded in part by NASA-NNG05GP48G.

$\emptyset\}$. A set of atoms A is an answer set of a logic program P iff A is an answer set of P^A.

We use $Atoms(P)$ to denote the set of all atoms appeared in a logic program P. The $Clark\ completion$ of a logic program P, denoted by $Comp(P)$, is the set of the following propositional clauses:

- For each atom $a \in Atoms(P)$, let $R = \{r \mid r \in P,\ head(r) = a\}$.
  - If $R = \emptyset$, $(\neg a)$.
  - Otherwise, $(a \equiv \bigvee_{r \in R} BC(r))$.
- For each rule $r \in P$ such that $head(r) = \emptyset$, $(\neg BC(r))$.

The $dependency\ graph$ of a logic program P, denoted by $DG(P)$, is the directed graph (V, E) where $V = Atoms(P)$ and

$$E = \{(a, b) |\ a, b \in V, r \in P, a = head(r), b \in pos(r)\}.$$

A set of atoms $L \subseteq V$ is called a *loop* of P if there is a path between any two members of L without passing any vertex outside L. We say P is $tight$ if $DG(P)$ has no loops; Otherwise, we say it is non-tight. Given a loop L of P, a rule $r \in P$ is called an *external supporting rule* for L if $head(r) \in L$ and $pos(r) \cap L = \emptyset$. Let R be the set of all external supporting rules for L. The $loop\ formula$ associated with L is the clause

$$\bigwedge_{r \in R} \neg BC(r) \supset \bigwedge_{p \in L} \neg p. \tag{2}$$

The following theorem describes the relation between the answer set of a logic program and the model of its completion extended by loop formulas.

Theorem 1. *(Lin & Zhao 2002) Let P be a logic program, $Comp(P)$ its completion, and LF the set of loop formulas associated with the loops of P. We have that for any set of atoms, it is an answer set of P iff it is a model of $Comp(P) \cup LF$.*

Finding a model for a propositional formula falls in the domain of the propositional satisfiability (SAT) problem, which is one of the most studied problems in computer science. Most of the complete SAT solvers are based on the basic DPLL algorithm (Davis, Logemann, & Loveland 1962). Recently, it has been shown that learning techniques play a significant rule in improving the performance of SAT solvers (e.g., MCHAFF(Moskewicz *et al.* 2001)). Figure (1) illustrates a typical DPLL algorithm with learning which is adapted from (Zhang *et al.* 2001).

Given an input propositional formula, the algorithm uses a backtracking search to find a truth assignment to variables such that the formula is evaluated to true. It returns satisfiable if such assignment exists, unsatisfiable if otherwise. The function $decide$ selects a free variable and a truth assignment for it based on some heuristics. It returns $done$ if all variables has been assigned a value. The function $deduce$ prunes the search space using *unit propagation*. It returns $conflict$ if a variable need to be both $true$ and $false$. In this case, the function $analyze\_conflicts$ is invoked to analyze the reasons for the conflict and find a proper decision level (blevel) to backtrack to. These reasons are added to the input formula in form of clauses, called conflict clauses, to prevent the same conflict from happening in the future search. This process is called learning, and the conflict clauses are sometimes called learning clauses. Note these conflict clauses are redundant in terms of the correctness of the algorithm. In practice, SAT solvers discard conflict clauses deemed less useful to save space.

```
1 while (true){
2     if (decide() ≠ done) {
3         while (deduce() == conflict){
4             blevel = analyze_conflicts();
5             if (blevel == 0)
6                 return unsatisfiable;
7             else backtrack(blevel);}}
8     else return satisfiable; }
```

Figure 1: DPLL algorithm with learning

SAT-based answer set solving

From Theorem 1, we can envision a straightforward procedure to compute answer sets for a logic program P: First compute all loop formulas and add them to $Comp(P)$, then use a SAT solver to generate the models for the extended completion. However, this approach is not feasible for general programs since they may have exponential number of loops. For this reason, the existing SAT-based answer set solvers (e.g., ASSAT and CMODELS) use a "generate and test" approach, which can be summarized by the procedure illustrated in Figure(2). It first finds a model for the completion of the input program. If no such model exists there is no answer set. Otherwise, if the model is an answer set it outputs the model. If it is not an answer set then the procedure computes some loop formulas and add them to the completion. Then this process is repeated until it finds an answer set or reports no solution. When the answer set test fails, to find a new model, ASSAT has to start the underlying SAT solver from scratch (black-box approach), while CMODELS just initiates a backtrack within the SAT solver(clear-box approach). For ASSAT, the loop formulas added at the end of each iteration are critical to the correctness of the algorithm, but for CMODELS they are added as learning clauses solely to speed up the search. Therefore, CMODELS' approach reduces the time cost to find a new model and eliminates the needs of space to keep all loop formulas which can be exponential in number.

Both ASSAT and CMODELS look for answer sets from the full models for the completion, extended with some loop formulas, of the input program. We observe that during the model generation, it is possible to detect that a partial assignment is not extensible to an answer set, long before it grows into a full model. For example, consider the program

$$\{a \leftarrow b.\ b \leftarrow a.\ a \leftarrow\ not\ c.\ c \leftarrow\ not\ a.\ c \leftarrow b.\}. \tag{3}$$

Its completion is $\{(a \equiv (b \vee \neg c)), (b \equiv a), (c \equiv (\neg a \vee b))\}$. The partial assignment $\{b, c\}$ is extensible to a model

```
Input: Logic program P
1 C = Comp(P);
2 while (true){
3     Find a model M for C using a SAT solver ;
4     if (no such M exists)
5       return false;
6     else{
7         if (isAnswerSet(P,M)) {
8           Output M as an answer set;
9           return true; }
10        else {
11              Compute loop formulas F
12                 that were not satisfied by M;
13              C = C ∪ F; }}}
```

Figure 2: Existing SAT-based answer set solving procedure

$\{a, b, c\}$ for its completion, but cannot be extended to any answer set.

Based on the above idea, we develop a new SAT-based answer set solving procedure listed in Figure 3. At first, it computes the completion of the input program P (line 1). The functions $decide$, $deduce$, $analyze\_conflicts$, and $backtrack$ are the same as those in Figure 1, which are responsible for generating a model for the completion. During the model generation, if $deduce$ does not return conflict (line 10), ASP_deduce will check if the current partial assignment is extensible to an answer set of P (line 11). It returns $conflict$ if the current partial assignment cannot be extended to any answer set, $implication$ if some free variables are required by the answer set semantics to take particular truth values, or nil if no new information is obtained. If nil is returned, the procedure continues with the model generation (line 12). If $conflict$ or $implication$ is returned, the procedure then invokes the function $gen\_inferred\_clause$ (line 14) to derive clauses to explain the occurrence of the conflict or implication situation. We call these clauses $inferred\ clauses$. They are added to the completion clauses to facilitate the conflict learning (line 6) and backjumping (line 9, 20), can be removed when they are no longer relevant. In case of $conflict$, the function $resolve\_ASP\_conflict$ (line 17) uses the inferred clauses to determine the decision level to backtract to.

In the following two sections, we show how to implement ASP_deduce and how to generate the inferred clauses.

Implement ASP_deduce

Currently we use the $Atmost$ operator (Simons, Niemelä, & Soininen 2002) of SMODELS to implement the ASP_deduce function.

Given a set of literals M and a rule r, the $generalized\ reduct$ of r, denoted by $r^{(M)}$, is

1. $\emptyset$, if $head(r) \cap M^- \neq \emptyset$ or $pos(r) \cap M^- \neq \emptyset$ or $neg(r) \cap M^+ \neq \emptyset$;
2. $head(r) \leftarrow pos(r)$, otherwise.

Given a logic program P and a set of literals B, $P^{(B)}$ is defined as $\{r^{(B)} \mid r \in P\}$. $Atmost(P, B)$ is the minimal model for $P^{(B)}$.

```
Input: Logic program P
1 C = Comp(P);
2 while (true) {
3    if (decide() ≠ done) {
4       while (true) {
5          if(deduce()== conflict){
6            blevel = analyze_conflicts();
7            if (blevel==0)
8               return false;
9            else backtrack(blevel); }
10         else {
11            status = ASP_deduce();
12            if (status == nil) break;
13            else {
14               IC = gen_inferred_clauses();
15               C = C ∪ IC;
16               if (status == conflict) {
17                  blevel = resolve_ASP_conflict();
18                  if (blevel==0)
19                     return false;
20                  else backtrack(blevel); }}}}}
21    else {
22       Output answer set;
23       return true; }}
```

Figure 3: New SAT-based answer set solving procedure

Figure 4 illustrates an implementation of the ASP_deduce function.

Now consider program (3) and the partial assignment $B = \{c\}$. We have $P^{(B)} = \{a \leftarrow b.\ b \leftarrow a.\ c.\}$, $Atmost(P, B) = \{c\}$, $N = \{a, b\}$, $N \cap B^+ = \emptyset$ but $N \not\subseteq B^-$. Hence, ASP_deduce returns $implication$: a and b should be set to false.

```
Input: Logic program P, partial assignment B
1 N = Atoms(P) - Atmost(P, B);
2 if (N ∩ B+ ≠ ∅) status = conflict;
3 else if (N ⊆ B−) status = nil;
4 else  status = implication;
5 return  status;
```

Figure 4: Function ASP_deduce

Computing inferred clauses

We discover that there is a close connection between the return states of ASP_deduce and loop formulas.

Definition 2. A loop LF associated with a loop L is *active* on a partial assignment B if B satisfies the antecedent of LF and $L \cap B^- = \emptyset$.

In the new procedure, the active loop formulas are used to generate the inferred clauses when ASP_deduce returns $conflict$ or $implication$.

In the following we present our main results on ASP_deduce and active loop formulas, followed by an algorithm for the $gen\_inferred\_clauses$ function.

Proposition 3. *Given a positive program P and a set of atoms L such that, for any rule $r \in P$, if its head is in L*

then its body contains an atom of L, the minimal model for P does not contain any atom of L.

Proof. Prove by contradiction. Assume M is a minimal model for P and M contains some atom from L. Let $M' = M - L$. For every rule $r \in P$, either the body of r contains some atom of L or not. In the former case, $pos(r) \not\subseteq M'$ and therefore $M' \models r$. In the latter case $head(r) \notin L$. Since r does not have any atom from L, $M' = M - L$ and $M \models r$, we have $M' \models r$. Therefore, M' is a model for P, contradicting to the minimality of M. □

Theorem 4. *Given a logic program P and a partial assignment B, if there is a loop formula active on B, then ASP_deduce on B results in either conflict or implication.*

Proof. Let LF be a loop formula active on B, and $L = \{L_1, \ldots, L_n\}$ be the associated loop. For any rule $r \in P$ such that $head(r) \in L$ and $pos(r) \cap L = \emptyset$, $r^{(B)} = \emptyset$ because B falsifies $BC(r)$, i.e., $pos(r) \cap B^- \neq \emptyset$ or $neg(r) \cap B^+ \neq \emptyset$. Therefore, for any rule r in $P^{(B)}$, if the head of r is in L then its body must contain an atom from L. By Proposition 3, the minimal model M of $P^{(B)}$, i.e., $Atmost(P, B)$, does not contain any atom of L. Let $N = Atoms(P) - M$. $L \subseteq N$ because $L \subseteq Atoms(P)$ and $L \cap M = \emptyset$. If B^+ contains an atom of L, $N \cap B^+ \neq \emptyset$, and thus ASP_deduce returns conflict. Otherwise, $N \not\subseteq B^-$ since $L \cap B^- = \emptyset$ (by the definition of the active loop formula) and $L \subseteq N$. Hence, ASP_deduce returns implication. □

Given a directed graph $G = (V, E)$ and a set of vertices $L \subseteq V$, L is called a *terminating loop* if L is a strongly connected component of G and there is no arc from any vertex in L to any vertex outside L.

This definition of terminating loop is similar to the one in (Lin & Zhao 2002). The following proposition on graph property is needed in the proof of Theorem 6.

Proposition 5. *Given a directed graph with finite vertices, if every vertex has an outgoing arc, there is a terminating loop in the graph.*

Given a logic program P and a partial assignment B, B is *stable* w.r.t. P if for any atom $a \in Atoms(P)$, $\neg a \in B$ iff either there is no rule headed by a, or for any rule $r \in P$ headed by a, $pos(r) \cap B^- \neq \emptyset$ or $neg(r) \cap B^+ \neq \emptyset$.

Theorem 6. *Given a logic program P and a stable partial assignment B, let $M = Atoms(P) - B^- - Atmost(P, B)$. If $M \neq \emptyset$, then there is a terminating loop, in the subgraph of $DG(P^{(B)})$ induced by M, whose loop formula is active on B.*

Proof. Let $C = Atmost(P, B)$. Claim 1: For all $a \in M$, there exists a rule $r \in P^{(B)}$ such that $head(r) = a$ and $pos(r) \cap M \neq \emptyset$. Consider any $a \in M$. We know $a \notin B^-$ and $a \notin C$. Since B is stable, $a \notin B^-$ implies that there exists a rule $r \in P$ such that $head(r) = a$ and $pos(r) \cap B^- = \emptyset$ and $neg(r) \cap B^+ = \emptyset$. Clearly $r^{(B)} \neq \emptyset$ and $head(r^{(B)}) = a$. Therefore, $r^{(B)}$ is a rule in $P^{(B)}$ headed by a. Since $a \notin C$, and C is the minimal model for $P^{(B)}$, $pos(r) \not\subseteq C$, which, together with $pos(r) \cap B^- = \emptyset$, implies $pos(r) \cap M \neq \emptyset$. Claim 2: For every rule $r \in P^{(B)}$, if $head(r) \in M$ then $pos(r) \cap M \neq \emptyset$. The proof of Claim 2 is similar to that of Claim 1.

Let Q be the subgraph of $DG(P^{(B)})$ induced by M. Q is not an empty graph since $M \neq \emptyset$. Every vertex of Q has an outgoing arc in accordance with Claim 1. By Proposition 5 there is a terminating loop L in Q. Let R be the external supporting rules for L w.r.t P. For any rule $r \in R$, $pos(r) \cap L = \emptyset$. We show that $r^{(B)} = \emptyset$. Otherwise, since L is a terminating loop and $head(r^{(B)}) \in L$, $pos(r^{(B)}) \cap (M - L) = \emptyset$. Since $pos(r^{(B)}) \cap L = \emptyset$, $pos(r^{(B)}) \cap M = \emptyset$, contradicting Claim 2. Since $r^{(B)} = \emptyset$, either $pos(r) \cap B^- \neq \emptyset$ or $neg(r) \cap B^+ \neq \emptyset$. Hence $BC(r)$ is falsified by B. So the clause $\bigwedge_{(r \in R)} \neg BC(r)$ is true. Clearly $L \cap B^- = \emptyset$. Therefore, the loop formula associated with L is active on B. □

Given a program P and a stable partial assignment B, if ASP_deduce on B results in $conflict$ or $implication$, then $M \neq \emptyset$. By Theorem 6, there exist loop formulas that are active on B w.r.t P. Such loop formulas can be computed by identifying the terminating loops in the subgraph of $DG(P^{(B)})$ induced by M. Figure 5 lists the algorithm for the function $gen\_inferred\_clause$.

Input: Logic program P, assignment B
1 $M = Atoms(P) - B^- - Atmost(P, B)$;
2 Find all terminating loops from the subgraph of $DG(P^{(B)})$ induced by M;
3 Compute the loop formulas for all terminating loops;
4 Return the loop formulas as the inferred clauses.

Figure 5: Function $gen\_inferred\_clauses$

A full assignment obtained by the new procedure is a model of the completion of the input program, and no loop formulas are active on it. By Theorem 1, it corresponds to an answer set.

An interesting result for tight programs can be inferred from Theorem 6.

Corollary 7. *If a program is tight, neither* implication *nor* conflict *will be returned by ASP_deduce on a stable partial assignment.*

This result can be used to improve the performance of SMODELS on tight programs by eliminating the invocation of the $Atmost$ function.

Implementation

Based on the new procedure, we build an answer set solver SAG on top of the state-of-the-art SAT solver MCHAFF[1]. It uses LPARSE [2] to ground the input program which may include cardinality rules and choice rules (Simons, Niemelä, & Soininen 2002).

These rules can be handled following a procedure described in (Ferraris & Lifschitz 2003), which is adopted

[1] http://www.princeton.edu/~chaff

[2] http://www.tcs.hut.fi/Software/smodels/

by CMODELS. Our treatment of choice rules is equivalent to that by CMODELS, but for cardinality rules, we use a "guess and check" approach. We replace each carinality constraint in the input program by a new atom, called $constraint\ atom$. In SAG, we periodically check the consistency between the truth values of the constraint atoms and the valuations of the corresponding cardinality constraints.

In the procedure of Figure 3, ASP_deduce is invoked on every decision level. In fact, to guarantee the soundness of the procedure, it is sufficient to invoke it only on full assignments. As far as efficiency is concerned, it is ideal to invoke ASP_deduce only when it will return $implication$ or $conflict$. In practice, we can use some heuristics to decide when to execute ASP_deduce. In SAG, the ASP_deduce is invoked randomly with a probability set by the solver users, called *invocation probability*.

Experimental results

SAG is tested against SMODELS[2], CMODELS using SAT solvers MCHAFF and ZCHAFF[1], and ASSAT using MCHAFF on a variety of benchmarks of non-tight programs. We do not include DLV because it is specially designed for disjunctive programs. We do not experiment with tight programs because all SAT-based solvers behave similarly due to the fact that there is a one-to-one correspondence between the answer sets and the completion models.

The experiments are carried out on a DELL Powerage 1850 (two 3.6 GHz Xeon CPUs) with Linux 2.4.21. All solvers use LPARSE to ground the input programs and the time used by LPARSE is counted in the statistics. The systems used are LPARSE-1.0.15, SMODELS-2.28, ASSAT-2.02, CMODELS-3.50 and MCHAFF-spelt3. The results reported are the CPU time in seconds for each solver to find an answer set.

					SAG	
Graph	SM	CM	CZ	Assat	p=0.5	p=0.1
2xp30.1	0.19	24.72	**	36.15	0.29	0.61
2xp30.2	**	58.65	**	27.46	0.18	1.91
2xp30.3	**	9.84	**	17.18	0.34	0.24
2xp30.4	**	**	463.4	469.6	76.97	73.03
rand2	**	24.2	109.8	93.9	32.69	2.24
rand5	**	3.94	**	40.39	1.51	1.03
rand7	0.49	8.37	**	19.98	1.5	0.87
jbr0.1	**	**	**	**	1.95	2.7
jbr0.2	4.54	**	**	191.6	8.26	5.23
jbr0.3	13.83	**	**	484.1	3.71	8.66
jbr0.4	38.45	**	**	25.95	7.87	4.58
sim7	513.2	14.22	0.87	7.27	0.06	0.06
sim8	**	89.83	**	86.23	0.22	0.23
sim9	**	**	**	168.5	10.97	6.58

Table 1: HC problems encoded as normal programs. Legends: SM – SMODELS; CM – CMODELS(MCHAFF); CZ – CMODELS(ZCHAFF); p – invocation probability of ASP_deduce; ** – time greater than 600 seconds.

				SAG	
Graph	SM	CM	CZ	p=0.5	p=0.1
2xp30.1	0.08	35.92	498.88	0.06	0.16
2xp30.2	**	156.25	10.8	0.35	0.65
2xp30.3	**	156.33	10.86	0.39	0.65
2xp30.4	**	**	**	26.47	28.97
rand2	**	192.19	**	12.85	18.66
rand5	**	31.01	**	0.66	6.16
rand7	0.16	398.17	**	0.44	0.26
jbr0.1	0.17	**	**	2.16	0.28
jbr0.2	0.48	51.03	**	0.64	0.54
jbr0.3	0.9	21.1	**	0.24	1.42
jbr0.4	1.57	575.5	**	0.28	0.27
sim7	270.5	4.03	1.01	0.11	0.12
sim8	**	11.07	472.53	0.41	0.98
sim9	**	220.5	**	0.06	0.05

Table 2: HC problems encoded as extended programs. Legends: see table 1.

The first set of programs are those finding Hamiltonian circuit (HC) from the following graphs. Graphs 2xp30.1 – 2xp30.4 are hand-coded graphs[3]; rand2, rand5 and rand7 are random graphs[4]; jbr0.1 – jbr0.4 are random graphs which are generated using ASP benchmark generator JBenge[5], with 80 vertices and the probabilities of the existence of an edge between any two vertices ranging from 0.1 to 0.4; sim7 – sim9 are simplex graphs with levels of 7 – 9 which are also generated by JBenge. Table 1 lists the results on the normal program encoding in (Niemelä 1999). Table 2 displays the results on the extended program encoding (with cardinality constraints) from http://www.cs.engr.uky.edu/ai/benchmarks.html. ASSAT is not in Table 2 since it does not support cardinality constraints.

From Table 1 and Table 2, we can see that, in most cases, SAG is at least an order of magnitude faster than other systems. In the other cases, it is very close to the top performer. SAG demonstrates its robustness in solving various problem instances. The performance difference of SAG running with the two ASP_deduce invocation probabilities seems to be marginal, compared to the differences between SAG and the other solvers.

The second class of programs are those solving bounded LTL model checking problems[6] as described in (Heljanko & Niemelä 2003). They are encoded as extended logic programs. The results are listed in Table 3. We can see that SAG running with the ASP_deduce invocation probability of 0.1 performs better than that with 0.5, and it is the overall winner.

The last set of programs are those solving problems related to checking requirements in a deterministic automaton (http://www.fmi.uni-stuttgart.de/szs/research/projects/

[3] http://assat.cs.ust.hk/Assat-2.0/hc-2.0.html

[4] http://asparagus.cs.uni-potsdam.de/

[5] http://www.cs.uni-potsdam.de/~konczak/JBenge/index.html

[6] http://www.tcs.hut.fi/ kepa/experiments/boundsmodels/

BMC	SM	CM	CZ	SAG	
				p=0.5	p=0.1
dp10io2b11	139.51	313.59	35.47	86.9	26.69
dp10so2b8	6.94	13.32	1.04	2.7	1.42
dp12so2b9	120.82	10.84	5	16.39	7.19
dp10io2b12	128.07	17.03	error	2.59	2.52
dp10so2b9	11.54	6.07	4.79	3	1.36
dp12so2b10	308.8	11.53	5.42	3.89	0.88
dp12io2b14	**	78.12	**	75.49	49.21

Table 3: Bounded Model Checking Problems.
Legends: error – program aborted due to runtime error; See Table 1 for the others.

	SM	CM	CZ	Assat	SAG	
					p=0.5	p=0.05
mutex4	16.76	4.6	4.1	4.11	5.44	4.66
phi4	0.21	27.28	error	5.22	0.39	0.34
mutex2	0.32	0.96	0.24	3.56	0.39	0.37
mutex3	146.19	**	error	mem	**	**
phi3	3.57	27.66	4.52	57.94	6.04	4.82

Table 4: Checking requirements in a deterministic automaton.
Legends: mem - program aborted due to insufficient memory; Others are the same as in the previous tables.

synthesis/benchmarks030923.html)(Stefanescu, Esparza, & Muscholl 2003). The results are listed in Table 4, where problem mutex4 and phi4 are of type "IDFD", and mutex2, mutex3 and phi3 are of type "Morin". As we can see, SAG is comparable to the top performer for all problems except for mutex3 where SMODELS is the only solver that finish within the time limit of 600 seconds.

Conclusion

To build fast SAT-based answer set solvers, we propose to add answer set extensibility checking on partial assignments for non-tight programs. When the extensibility checking returns $implication$ or $conflict$, we propose to find loop formulas active on the current assignment and use them to guide the search. A new answer set solving procedure is presented based on these proposals. The empirical studies on well-known benchmarks show that the new approach leads to a significant performance boost to the SAT-based answer set solvers.

The SAG system and the benchmarks used in this paper can be found at http://www.cs.ttu.edu/~yzhang/sag/ .

Acknowledgements

We thank Dr. Michael Gelfond for discussions and support during this research work.

References

Clark, K. L. 1978. Negation as failure. In Gallaire, H., and Minker, J., eds., *Logic and Data Bases*. Plenum Press, NY. 293–322.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem proving. *Communications of the ACM* (5):394–397.

Fages, F. 1994. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science* 1:51–60.

Ferraris, P., and Lifschitz, V. 2003. Weight constraints as nested expressions. *CoRR* cs.AI/0312045.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In Kowalski, R. A., and Bowen, K., eds., *Proceedings of the Fifth International Conference on Logic Programming*, 1070–1080. Cambridge, Massachusetts: The MIT Press.

Heljanko, K., and Niemelä, I. 2003. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming* 3(4&5):519–550.

Lee, J., and Lifschitz, V. 2003. Loop formulas for disjunctive logic programs. In Palamidessi, C., ed., *ICLP*, volume 2916 of *Lecture Notes in Computer Science*, 451–465. Springer.

Lierler, Y., and Maratea, M. 2004. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In Lifschitz, V., and Niemelä, I., eds., *LPNMR*, volume 2923 of *Lecture Notes in Computer Science*, 346–350. Springer.

Lin, F., and Zhao, Y. 2002. ASSAT: Computing answer sets of a logic program by SAT solvers. In *AAAI/IAAI*, 112–118.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intellignece* 25:241 – 273.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1–2):181–234.

Stefanescu, A.; Esparza, J.; and Muscholl, A. 2003. Synthesis of distributed algorithms using asynchronous automata. In *CONCUR*, 27–41.

Zhang, L.; Madigan, C. F.; Moskewicz, M. W.; and Malik, S. 2001. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, 279–285.