

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 $A = \{a_1, \dots, a_k\}$, $not(A)$ denotes the set of literals $\{\neg a_1, \dots, \neg a_k\}$. Given a set of literal $B = \{b_1, \dots, b_k\}$, 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

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

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* $\{a_1, \dots, a_m\}$ by $pos(r)$, and its *negative body* $\{a_{m+1}, \dots, a_n\}$ 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

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

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.

Copyright © 2006, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

\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

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

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

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

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

$$\{a \leftarrow b. b \leftarrow a. a \leftarrow not\ c. c \leftarrow not\ a. c \leftarrow b.\}. \quad (3)$$

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  $C = \text{Comp}(P)$ ;
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 ( $\text{isAnswerSet}(P,M)$ ) {
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       $C = C \cup F$ ; }}

```

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 backtrack 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 SMOBELS 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 $\text{head}(r) \cap M^- \neq \emptyset$ or $\text{pos}(r) \cap M^- \neq \emptyset$ or $\text{neg}(r) \cap M^+ \neq \emptyset$;
2. $\text{head}(r) \leftarrow \text{pos}(r)$, otherwise.

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

```

Input: Logic program  $P$ 
1  $C = \text{Comp}(P)$ ;
2 while (true) {
3   if ( $\text{decide}() \neq \text{done}$ ) {
4     while (true) {
5       if( $\text{deduce}() == \text{conflict}$ ){
6          $\text{blevel} = \text{analyze\_conflicts}()$ ;
7         if ( $\text{blevel} == 0$ )
8           return false;
9         else  $\text{backtrack}(\text{blevel})$ ; }
10      else {
11         $\text{status} = \text{ASP\_deduce}()$ ;
12        if ( $\text{status} == \text{nil}$ ) break;
13        else {
14           $IC = \text{gen\_inferred\_clauses}()$ ;
15           $C = C \cup IC$ ;
16          if ( $\text{status} == \text{conflict}$ ) {
17             $\text{blevel} = \text{resolve\_ASP\_conflict}()$ ;
18            if ( $\text{blevel} == 0$ )
19              return false;
20            else  $\text{backtrack}(\text{blevel})$ ; }}}}
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 $P^{(B)} = \{a \leftarrow b, b \leftarrow a, c.\}$, $\text{Atmost}(P, B) = \{c\}$, $N = \{a, b\}$, $N \cap B^+ = \emptyset$ 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  $N = \text{Atoms}(P) - \text{Atmost}(P, B)$ ;
2 if ( $N \cap B^+ \neq \emptyset$ )  $\text{status} = \text{conflict}$ ;
3 else if ( $N \subseteq B^-$ )  $\text{status} = \text{nil}$ ;
4 else  $\text{status} = \text{implication}$ ;
5 return  $\text{status}$ ;

```

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

