ASSAT: Computing Answer Sets of A Logic Program By SAT Solvers

Fangzhen Lin and Yuting Zhao
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
{flin,yzhao}@cs.ust.hk

March 12, 2004

Abstract

We propose a new translation from normal logic programs with constraints under the answer set semantics to propositional logic. Given a normal logic program, we show that by adding, for each loop in the program, a corresponding loop formula to the program's completion, we obtain a one-to-one correspondence between the answer sets of the program and the models of the resulting propositional theory. However, in the worst case, there may be an exponential number of loops in a logic program. To address this problem, we propose an approach that adds loop formulas a few at a time, selectively. Based on these results, we implement a system called ASSAT(X), depending on the SAT solver X used, for computing one answer set of a normal logic program with constraints. We test the system on a variety of benchmarks including the graph coloring, the blocks world planning, and Hamiltonian Circuit domains. Our experimental results show that in these domains, for the problem of generating one answer set of a normal logic program, our system has a clear edge over the state-of-art answer set programming systems Smodels and DLV.

1 Introduction

Logic programming with answer sets semantics [8] and propositional logic are closely related. It is well-known that there is a local and modular translation from clauses to logic program rules such that the models of a set of clauses and the answer sets of its corresponding logic program are in one-to-one correspondence [24, 18].

The other direction is more difficult and interesting. Niemelä [18] showed that there cannot be a modular translation from normal logic programs to sets of clauses, in the sense that for any programs P_1 and P_2, the translation of $P_1 \cup P_2$ is the union of the translations of P_1 and P_2. However, the problem becomes interesting when we drop the requirement of modularity. In the special case when all rules in a program have at most one literal in their bodies (so-called 2-literal programs), Huang *et al.* [9] showed that there is an efficient translation to sets of clauses that do not need to use any extra variables. They also observed that many of the logic programs in answer set programming applications are essentially 2-literal ones. For these programs, their experiments showed the advantage of computing answer sets using SAT solvers such as SATO over Smodels [21].

In the general case, Ben-Eliyahu and Dechter [4] gave a translation for a class of disjunctive logic programs, which includes all normal logic programs. However, one problem with their translation is that it may need to use a quadratic number of extra propositional variables. While the number of variables is not always a reliable indicator of the hardness of a SAT problem, it nonetheless exerts a heavy toll on current SAT solvers when the number gets too big (for instance, the default setting in SATO allows only a maximum of 30,000 variables). In the worst case, adding one more variable could double the search space.

In this paper we shall propose a new translation. To motivate, consider the completion of a a logic program. It is well-known that every answer set of a logic program is also a model of the completion of the program, but the converse is not true in general. Fages [7] essentially showed that if a logic program has no positive loops, then every model of its completion is also an answer set. Recently, Babovich *et al.* [2] extended Fages' result and showed that this continues to be true if the logic program is what they called "tight" on every model of its completion.

Intuitively, the completion semantics is too weak because it does not handle positive cycles properly. For instance, the program $\{p \leftarrow q, \quad q \leftarrow p\}$

has a cycle $p \to q \to p$. Its completion is $\{p \equiv q, q \equiv p\}$, which has two models. But the program has a unique answer set in which both p and q are false.

To address this problem, we define a loop in a program to be a set of atoms such that for any pair of atoms in the set, there is a positive path from one to the other in the dependency graph of the program [1]. The basic idea of our translation is then to associate a formula with each loop in a program. The formula captures the logical conditions under which the atoms in the loop can be in an answer set. For instance, for the above program, $\{p, q\}$ is a loop. The formula associated with it is $(p \vee q) \supset false$, meaning that none of them can be in any answer set of the program.

In this paper, we show that if we add these formulas for loops to the completion of a program, we obtain a propositional theory whose models are in one-to-one correspondence with the answer sets of the logic program.

The advantages of our translation over the one by Ben-Eliyahu and Dechter mentioned above are that it does not use any extra variables[1] and is intuitive and easy to understand as one can easily work it out by hand for typical "textbook" example programs.

However, one problem with our translation is that in the worst case, there may be an exponential number of loops in a logic program. To overcome this, we propose an implementation strategy that does not compute all loops of a logic program at once, but iteratively computes a few with certain properties.

Our work contributes to both the areas of answer set logic programming and propositional satisfiability. It provides a basis for an alternative implementation of answer set logic programming by leveraging on existing extensive work on SAT with a choice of variety of SAT solvers ranging from complete systematic ones to incomplete randomized ones. Indeed, our experiments on some well-known benchmarks such as graph coloring, planning, and Hamiltonian Circuit (HC) show that for the problem of generating one answer set of a logic program, our system has a clear advantage over the two popular specialized answer set generators, Smodels [18, 21] and DLV [13]. On the other hand, this work also benefits SAT in providing some hard instances: we have encountered some relatively small SAT problems (about 720 variables and 4500 clauses) that could not be solved using any of the

[1]This refers to the translation from logic prorgams to propositional theories. It is well-known that to avoid generating an exponential number of clauses, one may have to introduce some new variables when converting a propositional theory to a set of clauses. See Section 5.

SAT solvers that we tried.

This paper is organized as follows. We first introduce some basic concepts and notations used in the paper. We then define a notion of loops and their associated loop formulas, and show that a set is an answer set of a logic program iff it satisfies its completion and the set of all loop formulas. Based on this result, we propose an algorithm and implement a system called ASSAT for computing the answer sets of a logic program using SAT solvers. We then report some experimental results of running ASSAT on graph coloring, blocks world planning, and HC domains, and compare them with those using Smodels and DLV.

2 Logical preliminaries

In this paper, we consider only fully grounded finite normal logic programs that may have constraints. That is, a logic program here is a finite set consisting of rules of the form:

$$p \leftarrow p_1, ..., p_k, \mathsf{not}\, q_1, ..., \mathsf{not}\, q_m, \tag{1}$$

and constraints of the form:

$$\leftarrow p_1, ..., p_k, \mathsf{not}\, q_1, ..., \mathsf{not}\, q_m, \tag{2}$$

where $k \geq 0, m \geq 0$, and p, $p_1, ..., p_k$, $q_1, ..., q_m$ are atoms without variables. Notice that the order of literals in the body of a rule or a constraint is not important under the answer set semantics, and we have written negative literals after positive ones. In effect, this means that a body is the conjunction of a set of literals, i.e. a rule of the form (1) can also be denoted by $p \leftarrow G$, where G is the set of literals in the body.

To define the *answer sets* of a logic program with constraints, we first define the *stable models* of a logic program that does not have any constraints [8]. Given a logic program P without constraints, and a set S of atoms, the Gelfond-Lifschitz transformation of P on S, written P_S, is obtained from P as follows:

- For each negative literal $\mathsf{not}\, q$ in the body of any rule in P, if $q \notin S$, then delete this literal from this body.

- In the resulting set of rules, delete all those that still contain a negative literal in their bodies, i.e. if a rule contains a $\mathsf{not}\, q$ in its body such that $q \in S$, then delete this rule.

Clearly for any S, P_S is a set of rules without any negative literals. This means that there is a unique minimal model of P_S, which is the same as the set of atoms that can be derived from the program using resolution when rules are interpreted as implications. In the following, we shall denote this set by $Cons(P_S)$. Now a set S is a stable model [8] of P iff $S = Cons(P_S)$.

In general, given a logic program P with constraints, a set S of atoms is an answer set if it is a stable model of the program obtained by deleting all the constraints in P, and it satisfies all the constraints in P, i.e. for any constraint of the form (2) in P, either $p_i \notin S$ for some $1 \leq i \leq k$ or $q_j \in S$ for some $1 \leq j \leq m$.

In the following, given a logic program P, we denote by $atom(P)$ the set of atoms appearing in the program P. Given a logic program P, its *completion*, written $Comp(P)$, is the union of the constraints in P and the Clark completion [6] of the set of rules in P, that is, it consists of following sentences:

- For each $p \in atom(P)$, let $p \leftarrow G_1, \cdots, p \leftarrow G_n$ be all the rules about p in P, then $p \equiv G_1 \vee \cdots \vee G_n$ is in $Comp(P)$. In particular, if $n = 0$, then the equivalence is $p \equiv false$, which is equivalent to $\neg p$.

- If $\leftarrow G$ is a constraint in P, then $\neg G$ is in $Comp(P)$.

Here we have somewhat abused the notation and write the body of a rule in a formula as well. Its intended meaning is as follows: if the body G is empty, then it is understood to be *true* in a formula, otherwise, it is the conjunction of the literals in G with not replaced by $\neg$. For example, the completion of the program:

$$a \leftarrow b, c, \mathsf{not}\, d.$$
$$a \leftarrow b, \mathsf{not}\, c, \mathsf{not}\, d.$$
$$\leftarrow b, c, \mathsf{not}\, d.$$

is $\{a \equiv (b \wedge c \wedge \neg d) \vee (b \wedge \neg c \wedge \neg d), \neg b, \neg c, \neg d, \neg(b \wedge c \wedge \neg d)\}$.

In this paper, we shall identify a truth assignment with the set of atoms true in this assignment, and conversely, identify a set of atoms with the truth assignment that assigns an atom true iff it is in the set. Under this convention, it is well-known that if S is an answer set of P, then S is also a model of $Comp(P)$, but the converse is not true in general.

In this paper we shall consider how to strengthen the completion so that a set is an answer set of a logic program iff it is a model of the strengthened theory. The key concepts are loops and their associated formulas. For these, it is most convenient to define the *positive dependency graph* of a logic program.

Given a logic program P, the positive dependency graph of P, written G_P, is the following directed graph: the set of vertices is $atom(P)$, and for any two vertices p, q, there is an arc from p to q if there is a rule of the form $p \leftarrow G$ in P such that $q \in G$ (recall that we can treat the body of a rule as a set of literals). Informally, an arc from p to q means that p positively depends on q. Notice that $\mathsf{not}\, q \in G$ does not imply an arc from p to q.

Recall that a directed graph is said to be *strongly connected* if for any two vertices in the graph, there is a (directed) path from one to the other. Given a directed graph, a *strongly connected component* is a set S of vertices such that for any $u, v \in S$, there is a path from u to v, and that S is not a subset of any other such set.

3 Loops and their formulas

As we mentioned, cycles are the reason why models of a logic program's completion may not be answer sets. Consider again the program $\{a \leftarrow b.\ \ b \leftarrow a.\}$. The set $\{a, b\}$ is a "loop" in the sense that a positively depends on b by the first rule, and b on a by the second rule. In propositional logic, if there is a "loop" like this, one is free to make any assumptions about the truth values of the atoms in the "loop", as long as the constraints, in this case the sentences in the completion of the program, are satisfied. But in the answer set semantics, one cannot assume that an atom is true without a justification or a proof. In this case, since there is no way to prove that either a or b is true, so they are not true by the default negation-as-failure assumption used by the answer set semantics.

Definition 1 *A nonempty subset L of $atom(P)$ is called a loop in a logic program P if for any p, q in L, there is a path in G_P from p to q of length > 0. In other words, a non-empty subset L of $atom(P)$ is a loop in P if the* *subgraph of G_P induced by L is strongly connected. In the following, atoms in a loop are sometimes called loop atoms.*

Given a logic program P, and a loop L in it, we associate two sets of rules with it:

$$R^+(L,P) = \{p \leftarrow G \mid (p \leftarrow G) \in P, p \in L, (\exists q).q \in G \wedge q \in L\}$$
$$R^-(L,P) = \{p \leftarrow G \mid (p \leftarrow G) \in P, p \in L, \neg(\exists q).q \in G \wedge q \in L\}$$

In the following, when the program P is clear from the context, we will write $R^+(L,P)$ as $R^+(L)$, and $R^-(L,P)$ as $R^-(L)$.

It is clear that these two sets are disjoint, and every rule whose head is in L is in exactly one of the sets. Intuitively, $R^+(L)$ contains rules *in the loop*, and they give rise to arcs connecting vertices in L in P's positive dependency graph; on the other hand, $R^-(L)$ contains those rules about atoms in L that are *out of the loop*. For instance, for the program $\{a \leftarrow b.\ b \leftarrow a.\ a.\}$, the only loop is $\{a,b\}$, and for this loop, R^+ is $\{a \leftarrow b.\ b \leftarrow a.\}$, and R^- is $\{a.\}$.

Example 1 As a simple example, consider P below:

$$a \leftarrow b. \quad b \leftarrow a. \quad a \leftarrow \mathsf{not}\ c.$$
$$c \leftarrow d. \quad d \leftarrow c. \quad c \leftarrow \mathsf{not}\ a.$$

There are two loops in this program: $L_1 = \{a,b\}$ and $L_2 = \{c,d\}$. For these two loops, we have:

$$R^+(L_1) = \{a \leftarrow b.\ b \leftarrow a.\}, R^-(L_1) = \{a \leftarrow \mathsf{not}\ c.\}$$
$$R^+(L_2) = \{c \leftarrow d.\ d \leftarrow c.\}, R^-(L_2) = \{c \leftarrow \mathsf{not}\ a.\}$$

■

While L_1 and L_2 above are disjoint, this is not always the case. However, if two loops have a common element, then their union is also a loop.

For any given logic program P and any loop L in P, one can observe that $\emptyset$ is the only answer set of $R^+(L)$. Therefore an atom in the loop cannot be in any answer set unless it is derived using some other rules, i.e. those from R^-. This motivates our definition of *loop formulas.*

Definition 2 *Let P be a logic program, and L a loop in it. Let $R^-(L)$ be the following set of rules:*

$$\begin{aligned} & p_1 \leftarrow G_{11}, \cdots, p_1 \leftarrow G_{1k_1}, \\ & \vdots \\ & p_n \leftarrow G_{n1}, \cdots, p_n \leftarrow G_{nk_n}. \end{aligned}$$

Then the (loop) formula *associated with L (under P), denoted by $LF(L, P)$, or simply $LF(L)$ when P is clear from the context, is the following implication:*

$$\neg[G_{11} \vee \cdots G_{1k_1} \vee \cdots \vee G_{n1} \vee \cdots \vee G_{nk_n}] \supset \bigwedge_{p \in L} \neg p. \tag{3}$$

Example 2 Consider again the program and loops in Example 1 above. $LF(L_1)$ is $c \supset (\neg a \wedge \neg b)$, and $LF(L_2)$ is $a \supset (\neg c \wedge \neg d)$. Notice that the completion of P, $Comp(P)$, is:

$$\begin{aligned} & a \equiv \neg c \vee b, \\ & b \equiv a, \\ & c \equiv \neg a \vee d, \\ & d \equiv c, \end{aligned}$$

which has three models: $\{a, b\}$, $\{c, d\}$, and $\{a, b, c, d\}$. However if we add the above two loop formulas to $Comp(P)$, it will eliminate the last model, and the remaining two are exactly the answer sets of P. The following theorem shows that this is always the case. ■

Theorem 1 *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 $Comp(P) \cup LF$.*

Proof: We prove this for programs that do not have constraints. From this, the result for the ones with constraints follows straightforwardly.

Let A be a set of atoms that satisfies $Comp(P) \cup LF$. Let P_A be the Gelfond-Lifschitz reduction of P on A. We need to show that A is exactly the set of atoms that can be derived from P_A. We use the convention that a rule with empty body is considered to have the tautology *true* as its body.

Let S_0 be the set of rules in P_A such that both their head and body are satisfied by A:

$$S_0 = \{p \leftarrow G \mid p \leftarrow G \in P_A \text{ and both } p \text{ and } G \text{ are true in } A\}$$

Now because A is a model of $Comp(P)$, for every $p \in A$, there is a rule in S_0 whose head is p. Suppose we have S_i, then construct S_{i+1} as follows:

- If there is no loop in S_i, then let $S_{i+1} = S_i$.
- If there is a loop in S_i, suppose that $L_i = (p_1, \ldots, p_n)$ is a maximal one (in terms of subset relation), and $R^+(L_i, S_i)$ is the following set of rules:

$$\begin{aligned} r_1 &: p_1 \leftarrow G_1, \\ &\vdots \\ r_m &: p_n \leftarrow G_m. \end{aligned}$$

  where $n \leq m$. Then L_i must be a loop in P as well. Now since $A \models p_1$ and A is a model of $LF(L_i, P)$, there must be some $1 \leq k \leq n$, and a rule $p_k \leftarrow G, G'$ in $R^-(L_i, P)$ such that $A \models G \wedge G'$, where G is a set of atoms and G' a set of negative literals. By the definition of $R^-(L_i, P)$, $G \cap L_i = \emptyset$, so the reduct of this rule, $r: \quad p_k \leftarrow G$, is not equal to any of $r_1, ..., r_n$. Now let S_{i+1} be the result of deleting all those rules in $r_1, ..., r_n$ whose head is p_k.

We can show the following properties about S_i:

- For some finite n, $S_k = S_n$ for all $k > n$, and for such n, S_n does not have any loops. This is quite obvious as there are only finite number of loops in S_0, and every S_i is a subset of S_0. In the following. let S be this S_n: $S = \bigcap_{i=1,2,\ldots} S_i$.
- For any i, if there is a loop in S_i, then the rule $r: \quad p_k \leftarrow G$ used in the construction of S_{i+1} from S_i is in S_{i+1}. By the construction of S_{i+1}, we only need to show that it is in S_i. We prove this by induction. It is clear that $r \in S_0$. Suppose that $r \in S_j$ for some $j < i$, we show that $r \in S_{j+1}$, that is r cannot be deleted from S_j. Suppose otherwise, there is a maximal loop L in S_j for which $r \in R^+(L, S_j)$. Now let L' be the maximal loop in S_i used in the construction of S_{i+1}. Then since

the head of r, p_k, is an element in both L and L', $L \cup L'$ is also a loop in S_j. So $L' \subseteq L$ as L is maximal. Now since r is deleted from S_j, by the construction of S_{j+1}, all rules in $R^+(L, S_j)$ whose head is p_k are also deleted. So there are no rules in $R^+(L', S_i)$ whose head is p_k, a contradiction with the selection of r in the construction of S_{i+1}.

- For each atom $p \in A$, there is a rule in S whose head is p. We prove this by induction. We have shown this for S_0 as A is a model of $Comp(P)$. Suppose that this is true for S_i, we prove it below for S_{i+1}. If there is no loop in S_i, then this is trivially true. Suppose it has a loop. By the construction of S_{i+1}, it is the result of deleting some of the rules in S_i whose heads are identical to the head of the rule $r: \quad p_k \leftarrow G$ used in the construction. So for any atom $p \in A$ that is different from p_k, there is a rule for it in S_i by our inductive assumption, and the same rule is also in S_{i+1} by our construction. Now for this atom p_k, as we have shown above, $r \in S_{i+1}$, which is a rule for p_k.

- For every $p \in A$, $S \models p$. We prove this by contradiction using induction. Notice that we have proved the following by now: (1) for each $p \in A$, there is a rule in S whose head is p; (2) for each rule in S, both the head and the body are true in A; and (3) S does not have any loops. Assume that there exists $p_0 \in A$, s.t. $S \not\models p_0$. Construct a set of sequences of atoms in A inductively as follows. First of all, let $T_0 = \{p_0\}$. Clearly, T_0 satisfies the following properties for T:

  - T is a sequence of distinct atoms in A.
  - None of atoms in T is entailed by S.
  - If p_i and p_{i+1} are two consecutive elements in T, then p_i depends on p_{i+1} in the sense that there is a rule $p_i \leftarrow G$ in S such that $p_{i+1} \in G$.

  Suppose that we have a sequence $T_k = [p_0, ..., p_k]$ with the above properties, construct T_{k+1} as follows: Let $p_k \leftarrow G_k$ be a rule in S. Since $S \not\models p_k$, $G_k \neq \emptyset$ and $S \not\models G_k$. Thus there must exist $p_{k+1} \in G_k$, s.t. $S \not\models p_{k+1}$. Since S is loop free, and each of the atoms in T_k depends on the next one, so for any i, $0 \leq i \leq k$, $p_{k+1} \neq p_i$. Now let $T_{k+1} = [p_0, \cdots, p_k, p_{k+1}]$. One can see that T_{k+1} so constructed also satisfies the two properties about T above. But this is impossible as A is finite and we could construct T_k this way for arbitrary k.

Thus $P_A \models p$ as $S \subseteq P_A$. Now if $P_A \models p$, then it must be the case that $p \in A$ as A is a model of $Comp(P)$, thus a model of P_A taken as a set of clauses. This proves that if A is a model of $Comp(P) \cup LF$, then A is an answer set of P.

Now suppose A is an answer set of P, then clearly A is a model of $Comp(P)$. We need to show that A is also a model of LF. We prove this by contradiction. Suppose L is a loop in P, and $LF(L)$ its loop formula. Suppose $R^-(L)$ in P is:

$$\begin{aligned} r_1 :\ & p_1 \leftarrow G_1, G'_1, \\ & \cdots \\ r_n :\ & p_n \leftarrow G_n, G'_n, \end{aligned}$$

where G_i is a set of atoms and G'_i a set of negative literals. If A does not satisfy $LF(L)$, then there must be a $p \in L$ such that $p \in A$, and for each rule r_i in $R^-(L)$, $A \not\models G_i \wedge G'_i$. Now if the following set of rules is the Gelfond-Lifschitz reduct of $R^-(L)$ on A:

$$\begin{aligned} r_{a_1} :\ & p_{a_1} \leftarrow G_{a_1}, \\ & \cdots \\ r_{a_k} :\ & p_{a_k} \leftarrow G_{a_k}, \end{aligned}$$

$1 \leq a_i \leq n$, then for each $1 \leq i \leq k$, $A \not\models G_{a_i}$. Since $p \in A$, $Cons(P_A) \models p$. So there must be a sequence of rules in P_A:

$$\begin{aligned} r''_1 :\ & q_1 \leftarrow Q_1, \\ & \cdots \\ r''_u :\ & q_u \leftarrow Q_u. \end{aligned}$$

s.t. $q_u = p$, $Q_1 = \emptyset$, and for each $1 < i \leq u$, $Q_i \subseteq \{q_1, \cdots, q_{i-1}\}$. In this sequence, there must be a v, $1 \leq v \leq u$, s.t. $\{q_1, \cdots, q_{v-1}\} \cap L = \emptyset$ and $q_v \in L$. Since $Q_v \subseteq \{q_1, \cdots, q_{v-1}\}$, $Q_v \cap L = \emptyset$. So r''_v must be a reduct of a rule in $R^-(L)$, i.e. for some $1 \leq i \leq k$, $r''_v = r_{a_i}$. But this is a contradiction as A must satisfy the body of r''_v but not that of r_{a_i}. This proves that if A is an answer set of P, then A is a model of $Comp(P) \cup LF$.

■

The proof of the theorem also shows the following result:

Proposition 1 *If S is a model of Comp(P), then S is an answer set of P iff there is a $P' \subseteq P$ such that P' does not have any loops, S satisfies the body of every rule in P', and there is a rule in P' for every atom in S.*

4 Computing loops

By Theorem 1, a straightforward approach of using SAT solvers to compute the answer sets of a logic program is to first compute all loop formulas, add them to its completion, and call a SAT solver. Unfortunately this may not be practical as there may be an exponential number of loops in a logic program. For an example, consider the following program by Niemelä [18] for finding Hamiltonian cycles of a graph:

$$
\begin{aligned}
&r_1: \ hc(V1, V2) \leftarrow \ arc(V1, V2), \ \mathsf{not}\, otherroute(V1, V2).\\
&r_2: \ otherroute(V1, V2) \leftarrow \ arc(V1, V2), \ arc(V1, V3),\\
&\qquad hc(V1, V3), \ V2 \neq V3.\\
&r_3: \ otherroute(V1, V2) \leftarrow \ arc(V1, V2), \ arc(V3, V2),\\
&\qquad hc(V3, V2), \ V1 \neq V3.\\
&r_4: \ reached(V2) \leftarrow \ arc(V1, V2), \ hc(V1, V2), \ reached(V1),\\
&\qquad \mathsf{not}\, initialnode(V1).\\
&r_5: \ reached(V2) \leftarrow \ arc(V1, V2), \ hc(V1, V2), \ initialnode(V1).\\
&\qquad initialnode(0).\\
&r_6: \leftarrow \ vertex(V), \ \mathsf{not}\, reached(V).
\end{aligned}
$$

For complete graphs, rule r_4, when fully instantiated, will give rise to a loop for every set of vertices.

Thus, it seems more practical to add loop formulas one by one, selectively. This motivates the following procedure.

Procedure 1

1. Let T be $Comp(P)$.

2. Find a model M of T. If there is no such model, then terminate with failure.

3. If M is an answer set, then exit with it (go back to step 2 when more than one answer sets are needed).

4. If M is not an answer set, then find a loop L such that its loop formula Φ_L is not satisfied by M.

5. Let T be $T \cup \{\Phi_L\}$ and go back to step 2.

By Theorem 1, this procedure is sound and complete, provided a sound and complete SAT solver is used. The two key questions regarding this procedure are as follows:

1. Are SAT solvers suitable for this purpose?

2. How to find a loop such that its loop formula is not satisfied by the current model in step 4?

Question 1 is empirical. For programs without loops, experiments done by Babovich *et al.* [2] and Huang *et al.* [9] pointed to a positive answer to this question. Our experiments on logic programs, including those with loops, seem to confirm this.

Question 2 is really the key to this procedure. If a set M of atoms is a model of $Comp(P)$ but not an answer set of P, then by Theorem 1, there must be a loop whose loop formula is not satisfied by M. But how hard it is to find such a loop? As it turns out, such a loop can be found in polynomial time. The key lies in the following set:

$$M^- = M - Cons(P_M).$$

(Recall that P_M is the Gelfond-Lifschitz reduct of P on M, and $Cons(P_M)$ is the set of consequences of P_M.)

Lemma 1 *If M is a model of $Comp(P)$, then $Cons(P_M) \subseteq M$.*

Proof: We prove this by induction. Suppose $Cons(P_M)$ is not empty, then for any $p \in Cons(P_M)$, there must be a sequence of rules in P_M:

$$\begin{aligned} p_1 &\leftarrow G_1, \\ &\ldots \\ p_k &\leftarrow G_k. \end{aligned}$$

s.t. $p_k = p$, $G_1 = \emptyset$, and for $1 < i \leq k$, $G_i \subseteq \{p_1, \ldots, p_{i-1}\}$.

Let $S_0 = \{p \mid p \leftarrow . \in P_M\}$. Since $Cons(P_M) \neq \emptyset$, $S_0 \neq \emptyset$. Now for any $p \in S_0$, there must be a corresponding rule $r : p \leftarrow G^-$ in P, s.t. G^- is either empty or a set of negative literals which are satisfied by M. Since M is a model of P, $p \in M$. So we have $S_0 \subseteq M$. Suppose we have S_k, and $S_k \subseteq M$. Construct S_{k+1} as: $S_{k+1} = S_k \cup \{p \mid r' : p \leftarrow G_{ki} \ \in P_M \ and \ G_{ki} \subseteq S_k\}$. For any $p \in S_{k+1} - S_k$, let the rule about p in P_M be $r' : p \leftarrow G_{ki}$ and the one in P be $r : p \leftarrow G_{ki}, G_{ki}^-$, since $G_{ki} \subseteq S_k$ and $S_k \subseteq M$, $G_{ki} \subseteq M$. Considering that r' is the Gelfond-Lifschitz reduct of r, G_{ki}^- is satisfied by M, so M satisfies the body of r. Since M is a model of $Comp(P)$, $p \in M$. So we have $S_{k+1} \subseteq M$.

Let $S = \bigcup_{i=0,1,\ldots} S_i$, then $S = Cons(P_M)$. So we have $Cons(P_M) \subseteq M$. ∎

Lemma 2 *For any $p \in M^-$, there must be a rule $p \leftarrow G$ about p in P_M such that G contains some $q \in M^-$. Furthermore, any such rule about p in P_M must have this property, i.e. its body contains an atom in M^-.*

Proof: Since p is in M, and M satisfies $Comp(P)$, there must be a rule about p in P whose body is satisfied by M. If G is the positive literals in the body of this rule, then $p \leftarrow G$ must be in P_M. Now G must have an atom in M^-, otherwise, they must be all in $Cons(P_M)$, which means that p must be in $Cons(P_M)$ as well, a contradiction with $p \in M^-$. ∎

From this Lemma, the following proposition follows easily.

Proposition 2 *There is at least one loop L in P such that $L \subseteq M^-$. Furthermore, for any $p \in M^-$, there must be a loop $L \subseteq M^-$ such that for some $q \in L$, there is a (directed) path from p to q in G_P.*

Proof: Let $p \in M^-$, then by Lemma 2 and the construction of Δ, there must be a $q \in M^-$ such that (p, q) is an arc in Δ. Since p is any node, and Δ has only finite number of nodes, so this must lead to a cycle, thus a strongly connected component reachable from p. ∎

Definition 3 *Let P be a program, and G_P its positive dependency graph. Let M be a model of $Comp(P)$. We say that a loop L of P is a* maximal loop under M *if L satisfies the following two conditions:*

1. *$L \subseteq M^-$, and*

2. L is maximal in M^-, i.e. there is no other loop $L' \subseteq M^-$ such that $L \subset L'$.

In other words, L is a strongly connected component of the subgraph of G_P induced by M^-. A maximal loop L under M is called a terminating *one if there does not exist another maximal loop L_1 under M such that for some $p \in L$ and $q \in L_1$, there is a path from p to q such that all vertices in the path are in M^-.*

Theorem 2 *If M is a model of $Comp(P)$ but not an answer set of P, then there must be a terminating loop of P under M. Furthermore, M does not satisfy the loop formula of any of the terminating loops of P under M.*

Proof: By Proposition 2, there is a loop in M^-. Since M^- is finite, there must be a maximal loop under M. Suppose now there are no terminating loops. Since G_P is finite, there must be more than one maximal loops under M, and there will be a path from any maximal loop under M to any other maximal loop under M in the subgraph of G_P induced by M^-. This means that the union of all these maximal loops is also a maximal loop under M, a contradiction with the fact that there are more than one maximal loops. So there must be a terminating loop under M.

Now let α be a terminating loop on M. Its loop formula in P is of the form:

$$(\neg G_1 \wedge \cdots \wedge \neg G_n) \supset (\neg p_1 \wedge \cdots \wedge \neg p_k),$$

where p_i's are atoms in α, and G_i's are the bodies of rules in $R^-(\alpha, P)$. Since $\alpha \subseteq M^- \subseteq M$, to show that M does not satisfy this formula, we only need to show that for any $p \in \alpha$, and any rule $p \leftarrow G$ that is in $R^-(\alpha)$, $M \models \neg G$. Suppose $M \models G$, then $p \leftarrow G^+$ is in P_M, where G^+ is the set of positive literals in G. Now there are two cases: Case 1: G^+ does not contain any atom in M^-, in this case G^+ must be in $Cons(P_M)$, this means that p must be in $Cons(P_M)$ as well, a contradiction with our assumption that $p \in M^-$; Case 2: G^+ contains a $q \in M^-$, in this case, by Proposition 2 there must be a loop, thus a maximal loop β under M such that there is a path from q to an atom in β. This means that there is a path from p to an atom in β as well. Since $p \leftarrow G$ is not in $R^+(\alpha)$, q is not in α, so α must be different from β, a contradiction with the assumption that α is a terminating and maximal loop. ∎

Thus given a model M of $Comp(P)$, if it is not an answer set, then the problem of finding a loop whose loop formula is not satisfied by M can be reduced to the problem of finding a terminating loop under M. Notice that a terminating loop under M is a strongly connected component in the subgraph of G_P induced by M^-, and that the strongly connected components of a graph and their dependency chains can be computed in $O(n+e)$ time[22], where n is the number of nodes and e the number of arcs in the graph. So a terminating loop under M can be found in $O(m+k)$ time, where m is the size of M^-, and k the number of arcs in the subgraph of G_P induced by M^-. Normally M^- is quite small compared to the number of atoms in P. Once we have identified a loop, its loop formula can be computed in time similar to that for computing the completion of a program.

5 ASSAT(X)

ASSAT(X), where X is a SAT solver, is an implemented system based on Procedure 1 in the last section:

ASSAT(X) – X a SAT solver

1. Instantiate a given program using lparse, the grounding system of Smodels.

2. Do some simple simplifications to the instantiated program, such as deleting all rules $p \leftarrow G$ such that $p \in G$.

3. Compute the completion of the resulting program and convert it to clauses.[2]

4. **Repeat**

   (a) Find a model M of the clauses using X.

   (b) If no such M exists, then exit with failure.

   (c) Compute $M^- = M - Cons(P_M)$.

[2]When converting a program's completion and loop formulas to clauses, $O(r)$ number of extra variables may have to be used in order to avoid combinatorial explosion, where r is the number of rules. So far, we do not find this to be a problem. For instance, for graph coloring and HC problems, no extra variables are needed. Notice that the approach in [4] also needs a program's completion as the base case.

(d) If $M^- = \emptyset$, then return with M for in this case it is an answer set.

(e) Compute all maximal loops under M.

(f) For each of these loops, compute its loop formula, convert it to clauses, and add them to the clausal set.

Notice that in the procedure above, when M is not an answer set, we will add the loop formula of every maximal loop under M to the current clausal set, instead of adding just the loop formula of one of the terminating loops if we want to follow Procedure 1 strictly using Theorem 2. The procedure above has the advantage of not having to check whether a loop is terminating. This is a feasible strategy as we have found from our experiments that there are usually not many such maximal loops.

6 Some experimental results

We experimented on a variety of benchmark domains. We report some of our results here for the following three domains: graph coloring, the blocks world planning, and Hamiltonian Circuit (HC) domains. More experimental data can be found on our web site http://www.cs.ust.hk/assat.

For these three domains, we used Niemelä's [18] logic program encodings that can be downloaded from Smodels' web site [3]. Among the three domains, only HC requires adding loop formulas to program completions. The graph coloring programs are always loop-free, and while the logic programs for the blocks world planning problems have loops, Babovich *et al.* [2] showed that all models of the programs' completions are answer sets. For graph coloring and the blocks world planning, our results confirmed the findings of [9], but we did it with many more and much larger instances.

For our system, ASSAT 2.0,we tried the following SAT solvers: Chaff2 (Mar 23, 2001 version) [17], Walksat 41 [19], GRASP (Feb, 2000 version) [20], Satz 215.2 [14], and SATO 4.1 (Zhang) [25]. In our experiments, we compare the performance of our system with Smodels version 2.27 [21] and DLV (May 16, 2003 version) [13] on the problem of computing one answer set of a normal logic program. More precisely, given a logic program, we measure the performance of each system by how fast it returns the first answer set of the program or reports that the program has no answer set.

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

For Smodels and ASSAT, we used lparse 1.0.13, the grounding module of Smodels, to ground a logic program (DLV has its own built-in grounding routine). We have noticed that for both Smodels and ASSAT, their behaviors are sometimes influenced by the parameters that one calls lparse with. For our experiments, we use "lparse -d none" which seemed to optimize the performance of both Smodels and ASSAT.

Our experiments were done on Sun Ultra 5 machines with 256M memory running Solaris. The reported times are in CPU seconds as reported by Unix "time" command, and include all pre-processing and post-processing time, if any. For instance, for ASSAT, they include the time for grounding the input program, for computing the completion and converting it to clauses, for computing loop formulas, for checking if a model returned by the SAT solver is an answer set, as well as the time for the SAT solver to return a model. To make the experiments feasible, we set a 2-hour cut off limit. So in the following tables, if a cell is marked by "—", it means that the system in question did not return after it had used up 2 hours of the CPU time. Also in the following tables, if a system is not included, that means it is not competitive on the problems.

We want to emphasize here that the bulk of the experiments here were done using Niemelä's early encodings for these benchmark domains. They are not the optimal ones for Smodels, and certainly not for DLV. As one of the referees for an extended abstract of this paper that we submitted to AAAI'02 pointed out, DLV is specialized in disjunctive logic programs. There are encodings of graph coloring and HC problems in disjunctive logic programs for which DLV will run faster. The newest version of Smodels also has some special constructs such as mGn that can be used to encode the problems in a more efficient way. One can also think of some encodings that are better suited for ASSAT. It is an interesting question as how all these systems will fare with each other with each using its own "best possible"encodings. While no one knows what the best encoding for each of the systems is, nonetheless for the HC domain, we shall also compare ASSAT using Niemelä's early encoding with DLV using the disjunctive encoding found on the DLV's web page, and Smodels using an encoding that uses mGn constructs.

6.1 The blocks world planning domain

This is a domain where a planning problem in the blocks world is solved by a sequence of logic programs such that every answer set of the nth program

problem	steps	atoms	rules	Smodels	DLV	ASSAT (Chaff2)	ASSAT (Satz)	ASSAT (SATO)
bw.19	9	12202	174099	17.58	—	13.69	22.3	12.33
	10	13422	191621	57.93	150.44	15.59	32.63	14.76
bw.21	10	16216	253241	24.95	236.1	19.47	33.25	17.5
	11	17690	276387	71.86	—	22.76	49.89	21.15
bw.23	11	21026	356663	36.62	621.33	27	50.81	24.58
	12	22778	386521	2284.19	890.51	31.48	76.29	29.5
bw.25	13	28758	526631	57.47	1899.95	40.62	90.07	36.31
	14	30812	564385	—	2613.85	49.34	153.66	44.04
bw.32	17	59402	1366664	172.12	—	106.87	—	96.86
	18	62702	1442764	—	4681.02	117.84	—	114.84

Legents: bw.n – a problem with n blocks. bw.15, bw.17, and bw.19 correspond to bw-large.c, bw-large.d, and bw-large.e on Smodels' web site, respectively.

Table 1: The Blocks World Planning Domain

in the sequence corresponds to a plan of the original planning problem with exactly n steps. So if a planning problem requires n steps, then in the sequence of logic programs corresponding to the planning problem, the first $n-1$ logic programs do not have any answer sets. For details, we refer the read to Niemeä [18].

For this domain, we tested the systems on 16 planning problems, ranging from one with 15 blocks to one with 32 blocks. Table-1 contains some run time data on these instances. In the tables, *atoms* and *rules* are the number of atoms and rules, respectively, in the grounded logic program returned by lparse,[4] and *steps* is the number of steps in the plan. Two numbers are given for *steps*, the shortest step and the one immediately before. For instance, the first two rows are about *bw*.19, a planning problem in a blocks world with 19 blocks. The shortest plan for this problem needs 10 steps. The logic program that corresponds to the first row has no answer set, meaning that there is no plan with 9 steps. The one that corresponds to the second row has an answer set, meaning that there is a plan with exactly 10 steps.

As one can see, ASSAT performed very well here. Among the SAT solvers used with ASSAT, SATO and Chaff2 performed best. ASSAT(Satz) also did very well for problems with ≤ 26 blocks. After that, it suddenly degraded,

[4]As we have mentioned, DLV has its own grounding module. There is no information available about the size of the grounded program in DLV

problem	atoms	rules	colorable?	Smodels	DLV	ASSAT(Chaff2)
p100e570	801	3880	y	1.89	1919.61	1.35
p300e1760	2401	11840	y	9.85	—	2.05
p600e3554	4801	23816	y	38.49	72.46	3.08
p6000e35946	48001	239784	y	4722.56	—	144.06
p10000e10000	80001	200000	y	—	—	58.98
p10000e11000	80001	203996	y	—	—	58.09
p10000e21000	80001	244000	n	31.12	3975.25	22.70
p10000e22000	80001	247992	y	—	—	49.33

Legends: pnem – a graph with n nodes and m edges.
p100e570, p300e1760, p600e3554, p6000e35946 are the same as p100, p300, p600, p6000 on Smodels' web site, respectively.

Table 2: 4-Coloring

perhaps because the problem sizes were too big for it to handle now.

We notice that for all problems that we had tested, if a shortest plan requires n steps, then Smodels did very well in verifying that there does not exist a plan with $n-1$ steps.

6.2 The graph coloring domain

We tested the systems on over 50 randomly generated large graphs for both 3-coloring and 4-coloring problems. Table 2 is the results for some of them for 4-coloring, and Table 3 for 3-coloring. Again ASSAT(Chaff2) was the clear winner. Smodels was more competitive on 3-coloring problems. But on 4-coloring ones, it could not return within our time limit after p10000e100000, except for p10000e2100 which is not colorable. In general, we have observed that Smodels and ASSAT(Chaff2) had similar performance on graphs which are not colorable.

6.3 The Hamiltonian Circuit (HC) domain

This is the only benchmark domain that we could find which requires adding loop formulas to program completions. We thus did some extensive testing in this domain. We tested three classes of problems: randomly generated graphs, hand-coded hard graphs, and complete graphs. All these are directed graphs that do not have any arc that goes from a vertex to itself, as is usually assumed in work on HC. In this domain, we found Walksat performed

problem	atoms	rules	colorable?	Smodels	DLV	ASSAT(Chaff2)
p100e570	601	2610	n	1.16	1.53	1.21
p300e1760	1801	7980	n	1.8	6	1.72
p600e3554	3601	16062	n	2.88	20.57	2.56
p6000e35946	36001	161838	n	24.15	2084.58	16.71
p10000e10000	60001	120000	y	2949.31	—	28.98
p10000e11000	60001	122997	y	2730.38	—	28.06
p10000e21000	60001	152997	n	20.65	2191.69	15.88
p10000e22000	60001	155994	?	—	—	—
p10000e23000	60001	158991	?	—	—	—

Legends: Same as in Table 2.

Table 3: 3-Coloring

surprisingly well, sometimes even better than Chaff2. However, one problem with Walksat is that it is incomplete: when it could not find an assignment, we don't know if the clauses are satisfiable or not. To address this, we invent W-C (Walksat+Chaff2): given a SAT instance, try Walksat on it first, if it does not return an assignment, then try Chaff2 on it. Another problem with Walksat is that it is a randomized system, so its performance may vary from run to run. We address this problem by running it 10 times, and taking the average. Thus in all the tables below, the data on ASSAT(W-C) are the averages over 10 runs.

Table 4 contains some statistics on 43 randomly generated Hamiltonian graphs (those with Hamiltonian Circuits). The numbers of nodes in these graphs range from 50 to 70 and numbers of arcs from 238 to 571. For those which ran out of 2 hours of CPU time, we use 7200 seconds (2 hours) in the calculation of sum, average, and standard deviation. The full set of data is available on our ASSAT web site.

Smodels could not solve 10 of them (did not return after 2 hours of CPU time), which amounts to a 23% failure rate, DLV could not solve 19 of them (44%). It is interesting to notice that compared with the other two domains, DLV fared better here. While overall it was still not as good as Smodels, there were 4 problems which Smodels could not solve but DLV could in a few seconds. ASSAT with both Chaff2 and W-C solved all of the problems. So far we had not run into any randomly generated graph which is Hamiltonian, either DLV or Smodels could solve it, but ASSAT could not. It is interesting

problem:	Smodels	DLV	ASSAT (Chaff2)	LFs	SATs	ASSAT (W-C)	LFs	SATs
SUM	74234	148705	4426	710	716	721	863	857
average	1726.39	3458.27	102.95	16.51	16.65	17	20.09	19.94
STDEV	3060.72	3513.84	238.75	13.11	12.51	10.48	8.94	8.34
unsolved	10	19	0			0		

Legends: SATs – number of calls to the SAT solver; LFs – number of loop formulas added totally; STDEV – standard deviation;.

Table 4: HC on Randomly Generated Graphs

to notice that the average number of calls to the SAT solver and the average number of loop formulas added are very close, both in ASSAT(Chaff2) and in ASSAT(W-C), so are the STDEVs. Indeed, we have found that for randomly generated graphs, if M is not an answer set, then often M^- is a loop by itself, i,e. M^- is the only maximal loop on M. Also the cost of ASSAT(X) is directly proportional to the number of calls made to X. One reason that ASSAT(W-C) out-performed ASSAT(Chaff2) is that walksat (W-C is really walksat here because it always returned a model for this group of graphs) is a bit "luckier" than Chaff2 in returning the "right" models. Also notice that on average, each call to Chaff2 took 6.2 seconds, and W-C 1 seconds.

We have found that it was difficult to come up with randomly generated non-Hamiltonian graphs which are hard. Most of them were really easy for all the systems and occurred when the number of edges is relatively small compared to that of vertices. There were reports (e.g. [5]) that randomly generated graphs with m close to $(logn + loglogn)n/2$, where n is the number of vertices and m the edges, have a 50% chance of being Hamiltonian, and was believed to be the "phase-transition" area of HC. However, Vandegriend [23] observed that for his heuristic algorithms, these graphs were very easy. Our experiments seemed to confirm this. For instance, a randomly generated graph with 70 vertices and 230 edges would have roughly 50% chance of being Hamiltonian. But for these graphs, all the systems could solve them very quickly, in a few seconds. We are still puzzled by this. For the systems that we have tested at least, the harder instances seem to be those graphs with more arcs, thus are likely to be Hamiltonian.

We did stumble on two graphs which are not Hamiltonian, but none of the systems that we tested (Smodels, DLV, ASSAT(X)) could solve them. They are not Hamiltonian for the obvious reason that some of the vertices in them

Graph	Vertex /arc	HC ?	Smodels	DLV	ASSAT (Chaff2)	LFs	SATs	ASSAT (W-C)	LFs	SATs
2xp30	60/316	n	1.28	1.39	1.57	2	2	4.69	2.2	2
2xp30.1	60/318	y	2.07	—	205.9	254	99	732.05	369.4	149.3
2xp30.2	60/318	y	—	—	70.41	213	128	1041.6	378.1	222.6
2xp30.3	60/318	y	—	—	70.54	213	128	774.45	322.5	187.5
2xp30.4	60/318	n	—	—	—			5983.48	18	12.7
4xp20	80/392	n	1.27	1.51	1.54	5	2	4.52	4.1	2
4xp20.1	80/395	n	—	—	8.3	4	2	18.76	4.1	2
4xp20.2	80/396	y	2.43	—	39.08	248	95	471.63	333.8	126.7
4xp20.3	80/396	n	1.24	1.79	9.12	9	5	19.84	17.7	13.2

Legends: SATs – number of calls to a SAT solver; LFs – number of loop formulas added totally; 2xp30 – 2 copies of p30; 2xp30.i – 2xp30 + two new arcs; 4xp20 – 4 copies of p20; 4xp20.i – 4xp40 + 3-4 new arcs.

Table 5: Hand-Coded Graphs

do not have an arc going out. They both have 60 vertices, and one has 348 arcs and the other 358. The completions of the logic programs corresponding to them, when converted to clauses, have only about 720 variables and 4500 clauses. But none of the SAT solvers that we tested could tell us whether they are satisfiable[5].

More interesting are some hand-coded hard problems. One strategy is to take the union of several copies of a small graph, and then add some arcs that connect these components. To experiment with this strategy, we took as bases p30 (a graph with 30 vertices) and p20 (a graph with 20 vertices), both taken from Smodels' distribution. The results are shown in Table 5.

Notice that SATs and LFs for ASSAT(W-C) are in general larger than the corresponding ones for ASSAT(Chaff2) this time. It is clear that ASSAT(Chaff2) performed the best here. It is interesting to notice that some of these graphs are also very hard for specialized heuristic search algorithm. For instance, for graph 2xp30.4, the HC algorithm (no.559, written in Fortran) in ACM Collection of Algorithms did not return after running for more than 60 hours. ASSAT(Chaff2) could not solve it using lparse 1.0.13 in 2 hours. But with lparse 0.99.43, ASSAT(Chaff2) solved it in about 1.5 hours.

[5]Following the requests of some SAT researchers, we generated some more hard instances like these. They can be found on our web site, `http://www.cs.ust.hk/assat/hardsat/`

Complete Graph	Smodels	DLV	ASSAT (Chaff2)	LFs	SATs	ASSAT (W-C)	LFs	SATs
c10	1.14	1.26	1.52	2	3	1.26	3.3	4.3
c20	5.15	8.12	5.92	11	12	3.02	3.3	4.3
c30	29.48	58.69	6.24	2	3	16.43	8.9	9.9
c40	115.43	242.72	11.2	1	2	77.13	18.3	19.3
c50	414.62	850.12	21.07	1	2	399.04	54	55
c60	1091.13	2075.26	275.5	37	38	1488.31	76.8	77.8
c70	2598.65	4831.81	1170.44	94	95	2281.4	87.6	88.6
c80	5173.11	—	5905.4	249	250	4396.5	116.4	117.4
c90	—	—	208.84	4	5	2844.21	57.9	58.7
c100	—	—	2732.49	66	67	3162.76	46.875	47.125

Legends: SATs – number of calls to the SAT solvers; LFs – number of loop formulas added totally; cN – a complete graph with N vertices.

Table 6: HC on Complete Graphs

Complete graphs are of special interest for ASSAT because when instantiated on these graphs, Niemelä's logic program for HC has an exponential number of loops. So one would expect that these graphs, while trivial for heuristic search algorithms, could be hard for ASSAT. Our experiments confirmed this. But interestingly, these graphs are also very hard for Smodels and DLV. The results are given in Table 6.

Complete graphs are difficult using Niemelä's encoding also because of the sheer sizes of the programs they produce. For instance, after grounding, the complete graph with 50 nodes (c50) produces a program with about 5000 atoms and 240K rules, and needs 4.5M to store it in a file. For c60, the number of atoms is about 7K and rules about 420K.

We also compared ASSAT with an implementation[6] of Ben-Eliyahu and Dechter's translation [4]. As we mentioned earlier, their translation needs n^2 extra variables, and these extra variables seemed to exert a heavy toll on current SAT solvers. For complete graphs, it could only handle those up to 30 vertices using Chaff2. It caused Chaff2 to run into bus error after running for over 2 hours on graph 2xp30. Perhaps more importantly, while Walksat was very effective on HC problems using our translation, it was totally ineffective with their translation as it failed to find an HC on even some of the simplest graphs such as p20. We speculate that the reason could be that the extra

[6]Done by Jicheng Zhao.

Problem	Smodels (New)	Smodels	DLV (New)	DLV	ASSAT
SUM	81807	74234	122802	148705	4426
average	1902.49	1726.39	2855.87	3458.27	102.95
STDEV	3166.16	3060.72	3453.47	3513.84	238.75
unsolved	11	10	16	19	0

Smodels(New) refers to the performance of Smodels using the new encoding, similarly for DLV(New). The entries for Smodels, DLV and ASSAT(Chaff2) are the same as in Table 4.

Table 7: Statistics on Random Graphs Using Different Encodings

variables somehow confuse Walksat and make its local hill-climbing strategy ineffective.

So far we have compared the three systems: ASSAT, Smodels, and DLV, using the same encoding for each of the benchmark problems. As we mentioned earlier, an interesting question is how these systems will compare when they use different encodings. To find out, we did some experiments using the HC benchmark domain. We chose for Smodels a recent short encoding using cardinality constraints of the form mGn [7], and for DLV an encoding found on its web site that uses disjunctive rules [8]. These two programs are given in the Appendix. For ASSAT, since it currently cannot handle cardinality constraints and disjunctive rule, we continued to use Niemelä's earlier encoding.

Table 7 contains some statistics of run time data on the same set of randomly generated HC problems as in Table 4. Again, the full set of data is available on the ASSAT web site. For ease of comparison, we also repeated some of the data from Table 4. As one can see, with its new encoding, DLV performed a bit better, though overall it was still not as good as Smodels and ASSAT. Surprisingly though for Smodels, its overall performance seemed a little worse using the new encoding. For instance, earlier, there were 10 problems it could not solve within our 2-hour time limit. Now there are 11. However, a big advantage of the new encoding is that when grounded, it yields a much smaller program. This seemed to have a big impact on the HC problems on complete graphs. For instance, for the complete graph with 80 vertices, Smodels with the new encoding returned a model in 100 seconds, and for the complete graphs with 100 vertices, it returned in 280 seconds. In

[7] Downloaded from `http://www.cs.engr.uky.edu/ai/benchmark-suite/ham-cyc.sm`

[8] Downloaded from `http://www.dbai.tuwien.ac.at/proj/dlv/examples/hamcycle`

comparison, for the complete graph with 100 vertices, using the old encoding, Smodels could not return in 2 hours. This also seems to confirm that for HC problems, one of the main reasons that the trivial complete graphs are hard for all the systems is because of the sizes of the grounded programs.

7 Conclusions

We have proposed a new translation from logic programs to propositional theories. Compared with the one in [4], ours has the advantage that it does not use any extra variables. We believe it is also more intuitive and simpler, thus easier to understand. However, in the worst case, it requires computing an exponential number of loop formulas. To address this problem, we have proposed an approach that adds loop formulas a few at a time, selectively. We have implemented a system called ASSAT based on this approach, and experimented it on some benchmark domains using various SAT solvers. While we were satisfied that so far our experimental results showed a clear edge of ASSAT over Smodels and DLV, we want to emphasize that the real advantage that we can see of ASSAT over specialized answer set generators lies in its ability to make use of the best and a variety of SAT solvers as they become available. For instance, with Chaff, we were able to run much larger problems than using others like SATO, and while Chaff has been consistently good on all of the benchmark problems that we have tested, other SAT solvers, like the randomized incomplete SAT solver walksat, performed surprisingly good on HC problems.

We also want to emphasize that by no means do we take this work to imply that specialized stable model generators such as Smodels are not needed anymore. For one thing, so far we have only considered the problem of finding one answer set of a logic program. It is not clear what would happen if we want to look for all the answer sets. More importantly, we hope this work, especially our new translation of logic programs to propositional logic, will lead to a cross fertilization between SAT solvers and specialized answer set solvers that will benefit both areas.

Since this work was first published [15], it has been extended in several directions. Babovich and Maratea [3] extended it to handle so-called cardinality constraints and choice rules in their second version of CMODELS. Lee and Lifschitz [11] extended our Theorem 1 and the notion of loops and loop formulas to disjunctive logic programs, Lee [10] showed that a similar result holds for McCain and Turner's causal logic [16], and finally Lee and Lin [12] showed that the idea works for propositional circumscription as well.

Finally, ASSAT as well as the experimental results reported in this paper can be found at http://www.cs.ust.hk/assat/ .

Acknowledgements

We thank Jia-Huai You and Jicheng Zhao for many useful discussions about the topics and their comments on earlier versions of this paper. We especially thank Jicheng for suggesting to define loops as sets of atoms rather than sets of rules as we initially did, and for implementing a version of Ben-Eliyahu and Dechter's algorithm. We also thank Jia-Huai for making us aware of Chaff. Without it, we would not be able to make many claims that we made in the paper. He also suggested to try complete graphs for HC. We thank the reviewers of AAAI-2002 and this special issue of AIJ for this paper for their insightful and useful comments about earlier versions of this paper, which led to many improvements. Any remaining errors in this paper are solely the authors'.

This work was supported in part by the Research Grants Council of Hong Kong under Competitive Earmarked Research Grants HKUST6061/00E and HKUST6205/02E.

References

[1] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of deductive databases and logic programming*, pages 293–322. Morgan Kaufmann, Los Altos, 1988.

[2] Y. Babovich, E. Erdem, and V. Lifschitz. Fages' theorem and answer set programming. In *Proc. of NMR-2000*, 2000.

[3] Y. Babovich and M. Maratea. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In *Proc. of LPNMR 2004*, 2004. System description. To appear.

[4] R. Ben-Eliyahu and R. Dechter. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1996.

[5] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI–91)*, pages 331–337, 1991.

[6] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logics and Databases*, pages 293–322. Plenum Press, New York, 1978.

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

[8] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Fifth International Conference and Symposium on Logic Programming*, pages 1070–1080, 1988.

[9] G.-S. Huang, X. Jia, C.-J. Liau, and J.-H. You. Two-literal logic programs and satisfiability representation of stable models: A comparison. In *Proc. 15th Canadian Conference on AI, LNCS, Springer*, 2002.

[10] J. Lee. Nondefinite vs. definite causal theories.[9] In *Proc. 7th Int'l Conference on Logic Programming and Nonmonotonic Reasoning*, 2004. To appear.

[11] J. Lee and V. Lifschitz. Loop formulas for disjunctive logic programs[10] In *Proc. ICLP-03*, 2003. To appear.

[12] J. Lee and F. Lin. Loop formulas for circumscription. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI–2004), AAAI Press, Menlo Park, CA.*, page To appear, 2004.

[13] N. Leone et al. DLV: a disjunctive datalog system, release 2001-6-11. At `http://www.dbai.tuwien.ac.at/proj/dlv/`, 2001.

[14] C. M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI-97*, pages 366–371, 1997.

[15] F. Lin and Y. Zhao. ASSAT: Computing answer sets of a logic program by sat solvers. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI–2002), AAAI Press, Menlo Park, CA.*, pages 112–118, 2002.

[9] `http://www.cs.utexas.edu/users/appsmurf/papers/nondefinite.ps` .

[10] `http://www.cs.utexas.edu/users/appsmurf/papers/disjunctive.ps` .

[16] N. McCain and H. Turner. Causal theories of action and change. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI–97), AAAI Press, Menlo Park, CA.*, pages 460–465, 1997.

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

[18] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. and AI*, 25(3-4):241–273, 1999.

[19] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343, Seattle, 1994.

[20] J. P. M. Silva and K. A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[21] P. Simons. Smodels: a system for computing the stable models of logic programs, version 2.25. At `http://www.tcs.hut.fi/Software/smodels/`, 2000.

[22] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

[23] B. Vandegriend. *Finding Hamiltonian Cycles: Algorithms, Graphs and Performance*. MSc Thesis, Department of Computer Science, University of Alberta, 1998.

[24] J. You, R. Cartwright, and M. Li. Iterative belief revision in extended logic programs. *Theoretical Computer Science*, 170, 1996.

[25] H. Zhang and M. E. Stickel. Implementing the davis-putnam method. *Journal of Automated Reasoning*, 24(1/2):277–296, 2000.

A Appendix

A.1 A Logic Program Encoding of HC Using Cardinality Constraints

Below is the "new" encoding for HC that we used in Table 7 for Smodels. This program contains choice rules, and cardinality constrains. It was modified from the program named http://www.cs.engr.uky.edu/ai/benchmark-suite/ham-cyc.sm on the University of Kentucky ASP benchmark site[11] by renaming some atoms to be consistent with Niemelä's encoding that we used in the paper.

As usual, a graph is given by a collection of facts of the forms $vertex(X)$ and $arc(X, Y)$. One vertex v, 0 here, is chosen as the initial vertex $initialvtx(v)$.

$$
\begin{array}{l}
\{hc(X,Y)\} :- \ arc(X,Y). \\
:- \ 2\{hc(X,Y) : arc(X,Y)\}, \ vertex(Y). \\
:- \ 2\{hc(X,Y) : arc(X,Y)\}, \ vertex(X). \\
:- \ vertex(X), \ \mathsf{not}\, r(X). \\
r(Y) :- \ hc(X,Y), \ arc(X,Y), \ initialvtx(X). \\
r(Y) :- \ hc(X,Y), \ arc(X,Y), \ r(X), \ \mathsf{not}\, initialvtx(X). \\
initialvtx(0).
\end{array}
$$

A.2 A Disjunctive Logic Program Encoding of HC

Below is the disjunctive logic program encoding of the HC problem for directed graphs that we used in Table 7 for DLV. It was downloaded from DLV's web site[12]. Again, a graph is given by a collection of facts of the forms $vertex(X)$ and $arc(X, Y)$, and one vertex X, 0 here, is chosen as the initial vertex $start(X)$. In the program, $in\_hc(X, Y)$ represents the fact that $arc(X, Y)$ is in the Hamiltonian Circuit to be constructed, and $out\_hm(X, Y)$ the fact that the arc (X, Y) is not.

$$reached(X) :- \ in\_hm(\_ , X).$$

[11] http://www.cs.engr.uky.edu/ai/benchmarks.html

[12] http://www.dbai.tuwien.ac.at/proj/dlv/examples/hamcycle

$$in\_hm(X,Y) \vee out\_hm(X,Y) :- \ start(X), arc(X,Y).$$
$$in\_hm(X,Y) \vee out\_hm(X,Y) :- \ reached(X), arc(X,Y).$$
$$:- \ in\_hm(X,Y), \ in\_hm(X,Y1), Y! = Y1.$$
$$:- \ in\_hm(X,Y), \ in\_hm(X1,Y), X! = X1.$$
$$:- \ vertex(X), \ notreached(X).$$
$$start(0).$$