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

Abstract

We propose a new translation from normal logic programs with constraints under the answer set semantics to propositional logic. Given a 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. Compared with the translation by Ben-Eliyahu and Dechter, ours has the advantage that it does not use any extra variables, and is considerably simpler, thus easier to understand. However, in the worst case, it requires computing exponential number of loop formulas. To address this problem, we propose an approach that adds loop formulas a few at a time, selectively. Based on these results, we implemented a system called ASSAT(X), depending on the SAT solver X used, and tested it on a variety of benchmarks including the graph coloring, the blocks world planning, and Hamiltonian Circuit domains. The results are compared with those by smodels and dlv, and it shows a clear edge of ASSAT(X) over them in these domains.

Introduction

Logic programming with answer sets semantics (Gelfond & Lifschitz 1988) 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 (You, Cartwright, & Li 1996; Niemelä 1999).

The other direction is more difficult and interesting. Niemelä (1999) showed that there cannot be a modular translation from 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.

Ben-Eliyahu and Dechter (1996) 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 quadratic (n^2) number of extra propositions. While the number of variables is not always a reliable indicator of the hardness of a SAT problem, in the worst case, adding one more variable would double the search space.

In this paper we shall propose a new translation. It works by first associating a formula with each loop in the program, and then adding these formulas to the program's completion. The advantages of this translation are that it does not use any extra variables, and is intuitive and easy to understand as one can easily work it out by hand for typical "textbook" example programs.

Our work contributes to both the areas of answer set logic programming and propositional satisfiability. On the one hand, 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 it has a clear advantage over the two popular specialized answer set generators, smodels (Niemelä 1999; Simons 2000) and dlv (Leone et al. 2001). 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 we could not solve using any of the SAT solvers that we had 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.

Logical preliminaries

We shall consider 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:

$$p \leftarrow p_1, ..., p_k, \mathsf{not}\, q_1, ..., \mathsf{not}\, q_m, \qquad (1)$$

and constraints of the form:

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

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 a set of literals. Thus we can use set-theoretic notations to talk about it. For instance, we may write $l \in G$ to mean that l is a literal in G.

Given a logic program P with constraints, a set S of atoms is its *answer set* if it is a stable model (Gelfond & Lifschitz 1988) of the program resulted from 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 $q_i \in S$ for some $1 \leq i \leq m$.

Given a logic program P, its completion, written $Comp(P)$, is the union of the constraints in P and the Clark completion (Clark 1978) of the set of rules in P, that is, it consists of following sentences:

- For any 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 $p \equiv false$.
- 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 literals in G with not replaced by $\neg$. For example, the completion of the program:

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

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 a proposition 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 we can 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 *dependency graph* of a set of rules.

Given a set R of rules, the dependency graph of R is the following directed graph: the vertices of the graph are atoms mentioned in R, and for any two vertices p, q, there is a directed arc from p to q if there is a rule of the form $p \leftarrow G$ in R such that $q \in G$ (recall that we can treat the body of a rule as a set). Informally, an arc from p to q means that p is depended on q, Notice that 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.

Loops and their formulas

It is clear that the reason a model of a logic program's completion may not be an answer set is because of loops. For instance, the logic program $\{a \leftarrow b.\ \ b \leftarrow a.\}$ has a unique stable model $\emptyset$. But its completion $\{a \equiv b, b \equiv a\}$ has two models $\emptyset$ and $\{a, b\}$. However, loops like this cannot always be deleted. For instance, if we add a fact a to this program, then the completion of the new program will have a unique model $\{a, b\}$, which is also an answer set. Notice here that in this new program, the rule $b \leftarrow a$ in the loop is used to derive b. The key point is then that a loop cannot be used to provide a circular justification of the atoms in the loop. The rules in a loop can be used only when there is an independent justification coming from outside of the loop. This is the information that we want to capture for a loop. Formally it is most convenient to define a loop as a set of atoms.

Definition 1 *A set L of atoms is called a loop of a logic program if the subgraph of the program's dependency graph induced by L is strongly connected.*

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

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

It is clear that these two sets are disjoint and for any rule whose head is in L, it is in 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 dependency graph; on the other hand, $R^-(L)$ contains those rules about atoms in L that are *out of the loop*.

Example 1 As a simple example, consider P below:

$$a \leftarrow b.\ \ b \leftarrow a.\ \ a \leftarrow \text{not } c.$$
$$c \leftarrow d.\ \ d \leftarrow c.\ \ c \leftarrow \text{not } a.$$

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

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

■

While L_1 and L_2 above are disjoint, this is not always the case in general. However, if two loops have a common atom, 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. Suppose that we enumerate the rules in $R^-(L)$ as follows:*

$$p_1 \leftarrow G_{11}, \cdots, p_1 \leftarrow G_{1k_1},$$
$$\vdots$$
$$p_n \leftarrow G_{n1}, \cdots, p_n \leftarrow G_{nk_n},$$

then the (loop) formula associated *with L (under P) is the following implication:*

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

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

$$a \equiv \neg c \vee b, \quad b \equiv a,$$
$$c \equiv \neg a \vee d, \quad d \equiv c,$$

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 stable models 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 $Comp(P) \cup LF$.*

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 in general there are exponential number of loops in a logic program. It seems more practical to add loop formulas one by one, selectively:

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 key question is step 4: given an M that satisfies $Comp(P)$ but is not an answer set of P, how can we find a loop whose loop formula is not satisfied by M? As it turns out, this can be done efficiently. The key lies in the following set:

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

Here P_M is the Gelfond-Lifschitz reduct of the set of rules in P on M, and $Cons(P_M)$ is the set of atoms that can be derived from P_M. Notice that M is a stable model of the set of rules in P iff $M^- = \emptyset$.

Definition 3 *Let P be a program, and G_P its 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 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 in the subgraph of G_P induced by M^-.*

Notice that the set of strongly connected components of a graph can be returned in $O(n + m)$, where n is the number of vertices and m the number of arcs of the graph.

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.*

ASSAT(X)

Based on Theorems 1 and 2, we have implemented a system called ASSAT along the line of Procedure 1:

ASSAT(X) – X a SAT solver

1. Instantiate a given program using lparse, the grounding system of smodels.
2. Compute the completion of the resulting program and convert it to clauses.[1]
3. **Repeat**
   (a) Find a model M of the clauses using X.
   (b) If no such M exist, then exit with failure.
   (c) Compute $M^- = M - Cons(M)$.
   (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 not many such maximal loops.

Some experimental results

We experimented on a variety of benchmark domains. We report three here. They are graph coloring, the blocks world planning, and HC domains. For these domains, we

[1]When converting a program's completion, as well as loop formulas, to clauses, $O(r)$ number of extra variables may have to be used, where r is the number of rules. This may seem to compromise our claim earlier that our translation does not use any extra variables. But this is just a peculiarity of doing SAT using clauses. In principle, SAT does not have to be done on clauses. Practically speaking, this could be a potential problem as virtually all current SAT solvers take clauses as their input. So far, we do not find this to be a problem, though. For instance, for graph coloring and HC problems, no extra variables are needed. Notice that the approach in (Ben-Eliyahu & Dechter 1996) also needs a program's completion as the base case.

used Niemelä's (1999) logic program encodings that can be downloaded from smodels' web site.[2] 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.* (2000) showed that all models of the programs' completions are answer sets. For graph coloring and blocks world planning, our results confirm the finding of (Huang *et al.* 2002), but we did it with many more and much larger problems.

The systems tested are as follows: For specialized answer set generators: smodels version 2.25[3] and dlv (Jun 11, 2001 version); for ASSAT, we tried the following SAT solvers: Chaff2 (Mar 23, 2001 version) (Moskewicz *et al.* 2001), walksat 3.7 (Selman and Kautz), relsat 2.0 (Bayardo), satz 2.13 (Li), and sato (Zhang).[4] For smodels and ASSAT, we use lparse 0.99.43, the grounding system of smodels, to ground a logic program (dlv has its own built-in grounding routine). All 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, for smodels the time for grounding, and for ASSAT the time for grounding, computing program completions, and checking that the returned assignment is indeed an answer set. We use 2 hours as the cut off limit. So in the following tables, if an entry 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 that 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 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 certainly think of some encodings that are better suited for ASSAT. It is an interesting future work to see how all these systems fare with each other with each using its own "best possible" encodings.

The blocks world planning domain

We tested the systems on 16 large problems, ranging from the ones with 15 blocks and 8 steps to ones with 32 blocks and 18 steps. For these problems, dlv did badly, could not even solve the smallest one with 15 blocks and 8 steps within our time limit. Among the SAT solvers used with ASSAT, Chaff2 performed the best, followed by satz. Table 1 contains some run time data on five representative problems.

	steps	atoms	Smodels	ASSAT (Chaff2)	ASSAT (SATZ)
bw.19	9	12202	16.07	14.78	17.36
	10	13422	47.50	19.76	24.34
bw.21	10	16216	23.48	21.64	26.27
	11	17690	71.33	29.38	40.41
bw.23	11	21026	35.34	30.66	37.36
	12	22778	247.51	41.4	54.1
bw.25	13	28758	61.2	47.38	61.45
	14	30812	—	65.75	100.97
bw.32	17	59402	187.54	132.06	—
	18	62702	—	191.46	—

Table 1: The Blocks World Planning Domain. bw.n means that this problem has n blocks. In particular, bw.19 is the same as bw-large.e on smodels' web site.

Graph	3-Coloring			4-Coloring		
		Smodels	ASSAT (Chaff2)		Smodels	ASSAT (Chaff2)
p6e36	n	22.99	17.38	y	4714.28	309.41
p10e10	y	2929.92	33.79	y	7650.54	70.52
p10e11	y	2715.76	32.22	y	—	66.88
p10e15	y	2048.82	27.34	y	—	62.80
p10e20	y	1348.53	120.27	y	—	56.54
p10e21	n	19.69	16.49	n	29.50	24.20
p10e25	?	—	—	y	—	51.28
p10e30	?	—	—	y	—	44.74

Table 2: Graph Coloring. pnem – a graph with $n * 1000$ nodes and $m * 1000$ edges (p6e36 is the same as p6000 at smodels' web site.)

ASSAT(Chaff2) clearly was the winner here. We notice that for all problems that we had tested, if an optimal plan requires n steps, then smodels did very well in verifying that there does not exist a plan with $n - 1$ steps. But it could not return an optimal plan after bw.24. ASSAT(satz) also did very well for problems with ≤ 25 blocks. After that, it suddenly degraded, perhaps because the problem sizes were too big for it to handle now.

The graph coloring domain

We tested both 3-coloring and 4-coloring problems. We tested the systems on over 50 randomly generated large graphs. Table 2 is the results for some of them. 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 p10e10, except for p10e21 which is not colorable. In general, we have observed that smodels and ASSAT(Chaff2) had similar performance on graphs which are not colorable.

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 on it. We test three classes of problems: randomly generated graphs, hand-coded hard graphs, and complete graphs. All these are directed graphs

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

[3] The current version of smodels is 2.26 which is slightly slower than 2.25 on the problems that we have tested.

[4] All these solvers can be found on the SATLIB web page http://www.satlib.org/solvers.html, except for Chaff2, which can be found at www.ee.princeton.edu/ chaff/index.php.

	No.	Ave1	SD1	Ave2	SD2	Ave3	SD3
Smodels	33	1973	3201				
DLV	24	3623	3521				
Chaff2	43	481	1526	21	15	21	16
WC	43	330	1498	20	8	20	9

Table 3: HC on random graphs. Legends: No. – the number of problems solved; Ave1 – the average run time, with an unsolved instance counts as 2 hours; Ave2 – the average number of calls to a SAT solver; Ave3 – the average number of loop formulas added; SDi – the standard deviation on Avei.

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 surprisingly well, even better than Chaff2. However, one problem with walksat is that it is incomplete. To address this, we invent WC (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 takes the average. Thus in all the tables below, the data on ASSAT(WC) are the averages over 10 runs.

Table 3 contains some statistics on 43 randomly generated Hamiltonian graphs (those with HCs). The numbers of nodes in these graphs range from 50 to 70 and numbers of arcs from 238 to 580. Smodels could not solve 12 of them (did not return after 2 hours of CPU time), which amounts to a 28% 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 3 problems which smodels could not solve but dlv could in a few seconds. ASSAT with both Chaff2 and WC 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 to notice that Ave2 and Ave3 are very close, so are SD2 and SD3. 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(WC) out-performed ASSAT(Chaff2) is that walksat (WC 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 23 seconds, and WC 16 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 arcs is relatively small compared to that of vertices. 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

Graph	HC?	SM	ASSAT1	SAT	LF	ASSAT2
2xp30	n	1	1	2	2	2
2xp30.1	y	1	52	51	125	821
2xp30.2	y	—	51	66	120	1185
2xp30.3	y	—	51	66	120	1669
2xp30.4	n	—	5160	28	42	4047
4xp20	n	1	1	2	4	162
4xp20.1	n	—	9	2	4	9
4xp20.2	y	1	7	31	74	558
4xp20.3	n	1	15	15	19	20

Table 4: Hand-coded graphs. Legends: SM – smodels; ASSAT1 – ASSAT(Chaff2); ASSAT2 – ASSAT(WC); 2xp30 – 2 copies of p30; 2xp30.i – 2xp30 + two new arcs; 4xp20 – 4 copies of p20; 4xp20.i – 4xp40 + 3-4 new arcs; SAT – number of calls to SAT; LF – number of loop formulas added.

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 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.

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 downloaded from smodels' web site. The results are shown in Table 4. Notice that SAT No. and LF No. are not given for ASSAT(WC) in the table for lack of space. They are in general larger than the corresponding ones for ASSAT(Chaff2) as this time walksat was not as lucky as Chaff2. It is clear that ASSAT(Chaff2) was very consistent. 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.

Of special interest for ASSAT are complete graphs because for these graphs, Niemelä's logic programs for HC have 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. This seems to suggest that while smodels and dlv do not explicitly compute loops, they also have to deal with them implicitly in their search algorithms. The results are given in Table 5. Again, the performance of ASSAT(WC) (ASSAT2 in the table) was sampled over 10 runs, and because of the randomized nature of walksat, it sometime ran faster on larger problems, as happened on $c90$, the complete graph with 90 nodes.

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

Graph	SM	ASSAT1	SAT	LF	ASSAT2	SAT	LF
c40	106	230	59	58	49	12	11
c50	417	857	97	96	435	41	40
c60	1046	72	4	3	1139	60	59
c70	2508	633	28	27	640	28	27
c80	4978	5833	122	121	6157	106	105
c90	—	—			4443	60	59

Table 5: HC on complete graphs. Legends: cn – a complete graph with n vertices; err – exit abnormally; the rest are the same as in Table 4;

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.

Finally, we also compared ASSAT with an implementation[5] of Ben-Eliyahu and Dechter's translation (1996). As we mentioned earlier, theirs needs to use 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 theirs 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 variables somehow confuse walksat and make its local hill-climbing strategy ineffective.

Conclusions

We have proposed a new translation from logic programs to propositional theories. Compared with the one in (Ben-Eliyahu & Dechter 1996), 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 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 run it on many problems in some benchmark domains using various SAT solvers. While we were satisfied that so far our experimental results show 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. Besides, there are special constructs such as mGn (at least m and at most n literals in G are true) in smodels one can use to write short and efficient logic programs. It is not immediately clear how these can be encoded efficiently in SAT. 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.

Finally, ASSAT can be found at `www.cs.ust.hk/faculty/flin/assat.html`

[5]Done by Jicheng Zhao.

Acknowledgments

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 are making in the paper. He also suggested us to try complete graphs for HC.

This work was supported in part by the Research Grants Council of Hong Kong under Competitive Earmarked Research Grants HKUST6145/98E and HKUST6061/00E.

References

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

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

Clark, K. L. 1978. Negation as failure. In Gallaire, H., and Minker, J., eds., *Logics and Databases*. New York: Plenum Press. 293–322.

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

Huang, G.-S.; Jia, X.; Liau, C.-J.; and You, J.-H. 2002. Two-literal logic programs and satisfiability representation of stable models: A comparison. In *Submitted*.

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

Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: engineering an efficient SAT solver. In *Proc. 39th Design Automation Conference*. Las Vegas, June 2001.

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

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

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