Two-Literal Logic Programs and Satisfiability Representation of Stable Models: A Comparison

Guan-Shieng Huang,[1] Xiumei Jia,[2] Churn-Jung Liau,[3] and Jia-Huai You[2]

Abstract

Logic programming with the stable model semantics has been proposed as a constraint programming paradigm for solving constraint satisfaction and other combinatorial problems. In such a language one writes function-free logic programs with negation. Such a program is instantiated to a ground program from which the stable models are computed. In this paper, we identify a class of logic programs for which the current techniques in solving SAT problems can be adopted for the computation of stable models efficiently. These logic programs are called *2-literal programs* where each rule or constraint consists of at most two literals. Many logic programming encodings of graph-theoretic, combinatorial problems given in the literature fall into the class of 2-literal programs. We show that a 2-literal program can be translated to a SAT instance without using extra variables. We report and compare experimental results on solving a number of benchmarks by a stable model generator and by a SAT solver.

1 Introduction

The satisfiability problem (SAT) is to determine whether a set of clauses in propositional logic has a model. When a model exists, a typical SAT solver also generates some models. The close relationship between the satisfiability problem and the stable model semantics [6] has attracted some attention to the issues related to representation and search efficiency. On the one hand, since both belong to the hardest problems in NP and thus there exist a polynomial-time reduction from one to the other, it is interesting to know how the two may be differentiated from the representation point of view. On the other hand, the close relation between the implementation techniques for building SAT solvers and those for building stable model generators has prompted the questions of how these solvers may be compared on classes of programs and the problems they encode.

On the issues of representation, it is known that SAT instances can be translated to logic programs locally in linear time. Conversely, the translation from logic programs to SAT instances is much more difficult. First, there is no modular translation [12]. Second, the currently known translations require a substantial amount of extra variables, in the order of n^2 in the worse case, where n is the number of variables in the given program [2, 17]. The use of extra variables could have significant impact on search efficiency, because, since each variable (a proposition) is

[1]Department of Computer Science, National Taiwan University, Taipei, Taiwan.
[2]Department of Computing Science, University of Alberta, Edmonton, Canada
[3]Institute of Information Science, Academia Sinica, Taipei, Taiwan.

true or false in any interpretation, each additional variable could double the search space, representing a possibly exponential increase of search space.

On the issues of implementation techniques, it is known, for example, that the SMODELS system [15, 16] is implemented by adopting and improving upon effective methods for building a Davis-Putnam procedure [4], on which some of the complete SAT solvers are also based. SMODELS has been demonstrated superior to some of the well-known SAT solvers experimentally, on problems such as the Hamiltonian cycle problem, whose logic programming encoding is much more compact than its satisfiability encoding [14]. For other satisfiability problems, the current understanding is that the performance of SMODELS is comparable to the performance of efficient SAT solvers (cf. [9, 15]).

In this paper,

- we identify a class of logic programs which are reducible to SAT instances without using extra variables;
- we report some experimental results on solving a number of benchmarks by SMODELS, and by reducing these logic programs to SAT instances and solving them by an efficient SAT solver, SATO [18].

SMODELS is one of the widely used systems for computing stable models.[1] SATO is one of the most efficient SAT solvers around.[2] The choice of SATO is also due to the fact that it is designed to generate all models when a SAT instance is satisfiable.

We conducted two sets of experiments. In the first one, the benchmarks are 2-literal programs. These programs are translated to sets of clauses without using extra variables. It turns out that for these benchmarks SATO consistently outperformed SMODELS.

In the second set of experiments, we are interested in two questions. The first is whether the advantage of SATO for 2-literal programs remains for non-2-literal programs that can also be translated to SAT instances without using extra variables. The second question is whether it is worthwhile to translate such a program to a 2-literal one using a linear number of extra variables. In this set of experiments, we choose the Blocks World encoding of Niemelä [12], whose completion formula is known to characterize the stable models semantics [1], thus there is a translation without using extra variables. It turns out that SATO performed well with the "right" value for the parameter g. A wrong choice could degrade SATO's performance dramatically. (The parameter g of SATO allows one to specify the number of clauses that could be saved during the execution. The "right" value could improve search efficiency significantly.) As for a translation to a 2-literal program with a linear number of extra variables, it turns out that SATO performed badly.

The paper is organized as follows. The next section gives the background on the stable model semantics. We show in Section 3 that a number of well-known

[1] **DLV** [8] is another system designed for computing the stable models of disjunctive programs. Performance comparison with SMODELS can be found in [7, 13].

[2] For a comparison with other SAT solvers, see [11].

benchmarks given in the literature are essentially 2-literal programs. We also show how in general a 2-literal program can be translated into a SAT instance without using extra variables, and provide experimental results. Section 4 reports experiments with the Blocks World example. Section 5 comments on related work, and Section 6 concludes the paper.

2 Stable models of logic programs

In this paper, we consider (normal) logic programs which are finite sets of rules

$$a \leftarrow b_1, ..., b_m, \texttt{not}\ c_1, ..., \texttt{not}\ c_n$$

where a, b_i and c_i are atoms of the underlying propositional language L. Here atoms with a `not` in front are called *default negations*. Programs without default negations are said to be *positive*.

The stable model semantics [6] is defined in two stages. The idea is that one guesses a set of atoms, and then tests whether the atoms in the set are precisely the consequences. In the first stage, given a program P and a set of atoms M, the *reduct* of P w.r.t. M is defined as:

$$P^M = \{a \leftarrow b_1, ..., b_m \mid a \leftarrow b_1, ..., b_m, \texttt{not}\ c_1, ..., \texttt{not}\ c_n \in P \text{ and } \forall i \in [1..n], c_i \notin M\}$$

Since P^M is a positive program, its deductive closure $\{\phi \mid P^M \vdash \phi, \phi \text{ is an atom in } L\}$, is the least model of P^M. Then, M is a *stable model* of P iff M is the least model of P^M.

A *constraint* is of the form $\leftarrow\ b_1, ..., b_m, \texttt{not}\ c_1, ..., \texttt{not}\ c_n$, which can be viewed as representing a rule of the form

$$f \leftarrow b_1, ..., b_m, \texttt{not}\ c_1, ..., \texttt{not}\ c_n, \texttt{not}\ f$$

where f is a new symbol. In the sequel, a program consists of program rules as well as constraints.

3 Two-literal programs

A 2-literal program is a program where each rule or constraint consists of at most two literals. A literal in this case refers to an atom ϕ or a default negation $\texttt{not}\ \phi$.

Two-literal programs are similar syntactically to 2-SAT instances, a finite set of clauses each of which consists of two (classic) literals. Although 2-SAT is linear-time solvable, the existence problem for 2-literal programs is NP-complete.

Theorem 1 *Deciding if a 2-literal program has a stable model is NP-complete.*

A number of NP-complete problems can be expressed by a 2-literal program (each of these encodings gives a proof to the NP-hardness).

3-SAT

The 3-SAT problem is NP-complete. The same reduction as given in [12] can be used to reduce a 3-SAT instance to a 2-literal program. Let (U, C) be an instance of 3-SAT where C is the set of clauses and U the set of variables. For each clause $x_i \vee y_i \vee z_i \in C$, where x_i, y_i, and z_i are literals, put three program rules and one constraint into P:

$$\begin{aligned} c_i &\leftarrow \texttt{not}\ x_i' \\ c_i &\leftarrow \texttt{not}\ y_i' \\ c_i &\leftarrow \texttt{not}\ z_i' \\ &\leftarrow \texttt{not}\ c_i \end{aligned}$$

where c_i is a new symbol, and $x_i' = x_i{}^{\sim}$ if x_i is an atom, and x_i' is the atomic part of x_i if x_i is a negated atom (i.e. if x_i is $\neg a$ then x_i' is a); similarly for y_i' and z_i'. In addition, for each variable $x \in U$, the program P has two rules $x \leftarrow \texttt{not}\ x^{\sim}$ and $x^{\sim} \leftarrow \texttt{not}\ x$. Clearly, the resulting program is a 2-literal one. It can be verified easily that M is a stable model of P iff $M - \{c_i\} - \{x_i{}^{\sim}\}$ is a model of (U, C). □

The following programs are taken from Niemelä [12] and will be used in this paper as benchmarks.

K-colorability

The problem is, given facts about $vertex(v)$, $arc(v, u)$, and available colors $col(c)$, find an assignment of colors to vertices such that vertices connected with an arc do not have the same color.

$$\begin{aligned} &color(V, C) \leftarrow vertex(V), col(C), \texttt{not}\ othercolor(V, C) \\ &othercolor(V, C) \leftarrow vertex(V), col(C), col(D), C \neq D, color(V, D) \\ &\leftarrow arc(V, U), col(C), color(V, C), color(U, C) \end{aligned}$$

An assignment of colors to vertices is represented by the predicate $color(V, C)$: vertex V is colored by color C. The first two rules above generate candidate solutions using an auxiliary predicate $othercolor(V, C)$, while the third rule eliminates illegal ones. A stable model of the program then corresponds to a solution of K-colorability, containing instances of $color(V, C)$ for each vertex V and the color C it gets. The ground instantiation of this program can be easily reduced to a 2-literal program. Once the facts about $vertex(v)$, $arc(v, u)$, and $col(c)$ are given as true atoms, their occurrences can be removed from the body of a rule. For the second rule, we ensure in addition that C and D are instantiated to different colors so that $C \neq D$ is true and can also be removed.

For example, suppose we have three colors, $col(red)$, $col(blue)$, and $col(yellow)$. For any vertex n, the ground instantiation of the first two rules includes

$$\begin{aligned} &color(n, red) \leftarrow \texttt{not}\ othercolor(n, red) \\ &othercolor(n, red) \leftarrow color(n, blue) \\ &othercolor(n, red) \leftarrow color(n, yellow) \end{aligned}$$

For example, if $color(n, red)$ is in a stable model M, $othercolor(n, red)$ must be false in M, which in turn forces $color(n, blue)$ and $color(n, yellow)$ to be false too. □

Queens

The problem is to place n queens on an $n \times n$ board so that no queen attacks any other queens.

$$
\begin{array}{l}
q(X,Y) \leftarrow d(X), d(Y), \texttt{not}\ negq(X,Y) \\
negq(X,Y) \leftarrow d(X), d(Y), \texttt{not}\ q(X,Y) \\
\leftarrow d(X), d(Y), d(X1), q(X,Y), q(X',Y), X' \neq X \\
\leftarrow d(X), d(Y), d(Y'), q(X,Y), q(X,Y'), Y' \neq Y \\
\leftarrow d(X), d(Y), d(X'), d(Y'), q(X,Y), q(X',Y'), \\
\qquad X \neq X', \neq Y', abs(X - X') = abs(Y - Y') \\
\leftarrow d(X), \texttt{not}\ hasq(X) \\
hasq(X) \leftarrow d(X), d(Y), q(X,Y)
\end{array}
$$

Instances of $d(X)$ and $d(Y)$ are given as facts providing dimensions of the board. An instance of $q(X,Y)$ describes a legal position of a queen. Thus, in a stable model instances of $q(X,Y)$ are queens' positions on the board. The first two rules generate all candidate board positions, whereas the next three constraints remove illegal ones. The last rule and the constraint above it ensure that every queen gets a position. When this program is instantiated, the true facts of predicate $d(X)$, along with inequality and equality among absolute values are removed so that the resulting ground program is a 2-literal one. □

Pigeons

The problem is to put M pigeons into N holes so that there is at most one pigeon in a hole.

$$
\begin{array}{l}
pos(P,H) \leftarrow pigeon(P), hole(H), \texttt{not}\ negpos(P,H) \\
negpos(P,H) \leftarrow pigeon(P), hole(H), \texttt{not}\ pos(P,H) \\
\leftarrow pigeon(P), \texttt{not}\ hashole(P) \\
hashole(P) \leftarrow pigeon(P), hole(H), pos(P,H) \\
\leftarrow pigeon(P), hole(H), hole(H'), pos(P,H), pos(P,H'), H \neq H' \\
\leftarrow pigeon(P), pigeon(P'), hole(H), pos(P,H), pos(P',H), P \neq P'
\end{array}
$$

Again, given facts about $pigeon(P)$ and $hole(H)$, the first two rules generate all possible arrangements of pigeons and holes. The next constraint and rule ensure that every pigeon gets a hole, followed by the constraint that no pigeon gets more than one hole. The last constraint says that no hole holds more than one pigeon. The ground instantiation of this program yields a 2-literal program. □

It can be shown by a similar instantiation process that the program given by Niemelä in [12] to solve the **Schur** problem, and the program by Marek and Truszczyński [10] to solve the **Clique** problem are essentially 2-literal ones.

3.1 Translating two-literal programs to SAT instances

The class of 2-literal programs can be translated to a set of clauses without the need of using extra variables.

Theorem 2 *There exists a polynomial time reduction from a 2-literal program P to a set of clauses S, without using extra variables, such that M is a stable model of P iff M is a model of S.*

We prove this theorem using Dung's result on fixpoint completion [5], which is based on a mechanism of reducing a program to a quasi-program. We sketch our proof below.

A *quasi-program* is a normal logic program in which no rule has any positive body literals. That is, a rule is of the form $a \leftarrow \texttt{not}\ c_1, ..., \texttt{not}\ c_n$, where $n \geq 0$. Given program rules

$$h \leftarrow a_1, ..., a_k, \texttt{not}\ c_1, ..., \texttt{not}\ c_n$$
$$a_i \leftarrow \texttt{not}\ d_{i,1}, ..., \texttt{not}\ d_{i,m_i}, \qquad 1 \leq i \leq k$$

the first rule above can be reduced to

$$h \leftarrow \texttt{not}\ d_{1,1}, ..., \texttt{not}\ d_{1,m_1}, ..., \texttt{not}\ d_{k,1}, ..., \texttt{not}\ d_{k,m_k}, \texttt{not}\ c_1, ..., \texttt{not}\ c_n$$

A fixpoint construction is defined so that every program P can be reduced to a quasi-program P_{quasi}. Dung shows that there is a one-to-one correspondence between the stable models of P and those of P_{quasi}. However, the reduction is not a polynomial time process in the general case.

If P is a 2-literal program, then the fixpoint construction above is bounded by $O(m^2)$ where m is the number of rules in the program, since each rule is reduced at most m times.

Once we get a quasi-program, we can apply Clark's predicate completion [3]. Given a propositional program Π, the *Clark completion* of Π, denoted $Comp(\Pi)$, is the following set of formulas: for each atom $\phi \in L$,

- if ϕ does not appear as the head of any rule in Π, $\phi \leftrightarrow F \in Comp(\Pi)$ (F stands for falsity here);
- otherwise, $\phi \leftrightarrow B_1 \vee ... \vee B_n \in Comp(\Pi)$ (with default negations replaced by negative literals), if there are exactly n rules $\phi \leftarrow B_i \in \Pi$ with ϕ as the head. We write T (tautology) for B_i if B_i is empty.
- for any constraint $\leftarrow b_1, ..., b_m, \texttt{not}\ c_1, ..., \texttt{not}\ c_n$ in Π, $\neg b_1 \vee ... \vee \neg b_m \vee ... \vee c_1 ... \vee c_n$ is in $Comp(\Pi)$.

For any quasi-program P_{quasi}, it is well-known that the models of $Comp(P_{quasi})$ correspond to the stable models of P_{quasi}.

Note that for a 2-literal program P, $Comp(P_{quasi})$ can be translated to a set of clauses in a simple way. For example, suppose we have n rules with the same atom a as the head: $a \leftarrow l_1$,..., $a \leftarrow l_n$. Then, the completion formula is $a \leftrightarrow l_1 \vee ... \vee l_n$ (where the occurrences of $\texttt{not}$ are replaced by $\neg$), which is equivalent to two clauses: $a \vee \neg l_1 \vee ... \vee \neg l_n$ and $\neg a \vee l_1 \vee ... \vee l_n$. We also note that the translation of an equivalence to a set of clauses in general is not guaranteed to be a polynomial time process, since it involves converting a disjunctive normal form to a conjunctive

normal form. Finally, since this proof is independent of the size of any constraint, the restriction that a constraint has at most two literals in the given program can be removed and it will not affect the claim stated in the theorem.

3.2 Experiments with two-literal programs

We compare search efficiencies for 2-literals programs experimentally. We tested three problems: **K-colorability**, **Queens**, and **Pigeons**. For each problem, we give a table listing the search times in seconds under the setting described below. For easy comparison, following each table a chart is also provided.

The logic programs for these problems and the graph instances were taken from the SMODELS web site[3]. These programs were run in SMODELS (Version 2.2.6), and the search times reported by SMODELS recorded. The search time here is the user CPU time (according to the unix time command) that incurred in running SMODELS, excluding the time spent by LPARSE, an interface to SMODELS whose main function is to instantiate a function-free program to a ground program. For a problem that can be solved within a few seconds, the time used by LPARSE could be a significant factor. For harder problems, it becomes insignificant.

We used the same encodings for translation. First, a logic program was grounded using LPARSE (Version 1.0.6), and the ground program was then translated to a SAT instance. The translation was implemented by a Prolog program which generates the appropriate input format for SATO. We then ran the translated SAT instance in SATO (Version 3.2.1) with the default setting of the parameter g,[4] and recorded the search time reported by SATO. In the tables these search times are given under the header SATO-Translation. In SATO the search time plus the "build time" is the user CPU time of the unix time command. The build time is usually a small fraction of the entire user CPU time. Note that, since we are interested only in search efficiency, neither grounding by LPARSE nor the translation to a SAT instance was considered part of the execution.

SATO comes with a number of benchmarks encoded as SAT instances, including **Queens** and **Pigeons**. We also ran these SAT benchmarks for comparison. In the tables, they are reported under the header SATO-Benchmark.

All of the experiments were performed on a Linux machine with a Pentium IV 1.5 GHz processor and 1 Gb main memory. When a solution exists, the recorded time is for the generation of the first solution. "Too long" in an entry means that no answer was returned within 2 hours time. Currently, SATO can accommodate at most 30,000 variables. "Too many variables" in an entry means that the SAT instance is over that limit. In testing **Pigeons** we used the data where the number of pigeons was one more than the number of holes, so that the problem had no solution. For colorability, the test results were generated with 3 colors.

[3] at http://saturn.tcs.hut.fi/Software/smodels/

[4] The default is $g = 10$. It was incorrectly said to be $g = 20$ in the user manual, and clarified via an email exchange with the author.

Pigeons	SATO-Benchmark	SATO-Translation	SMODELS
6	0.009	0.001	0.019
7	0.036	0.015	0.695
8	1.826	0.101	0.884
9	1.863	0.836	7.729
10	20.166	8.425	76.123
11	359.615	100.500	844.255
12	4482.400	1284.010	10563.450

Table 1: Search times for Pigeons

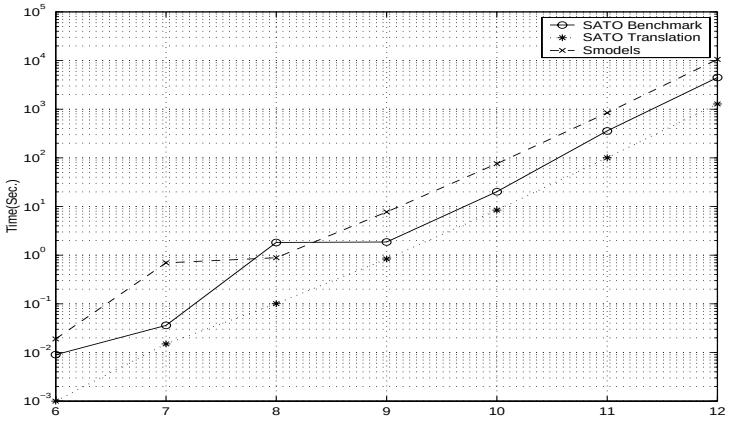

Figure 1: Pigeons

Queens	SATO-Benchmark	SATO-Translation	SMODELS
8	0.001	0.001	0.023
10	0.001	0.001	0.073
13	0.001	0.005	0.348
15	0.001	0.001	1.672
16	0.001	0.005	4.450
17	0.006	0.016	15.115
18	0.008	0.009	25.146
19	0.006	0.016	11.750
20	0.008	0.018	153.730
21	0.010	0.011	251.836
25	0.016	0.094	too long

Table 2: Search times for Queens

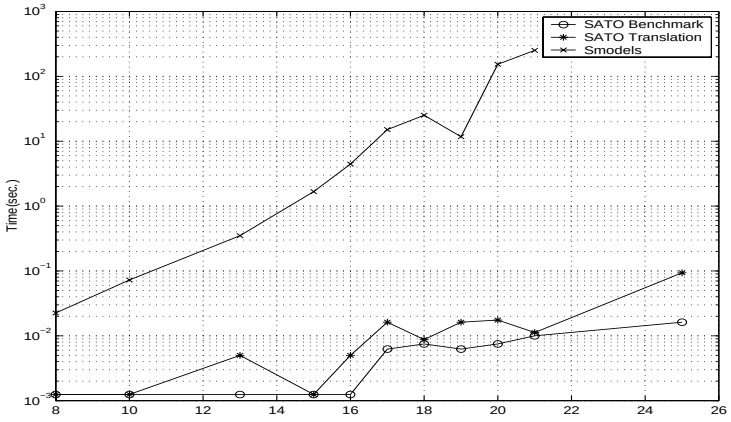

Figure 2: Queens

Nodes	SATO-Translation	SMODELS
20	0.00	0.01
25	0.00	0.02
29	0.00	0.02
100	0.01	0.03
300	0.01	0.10
600	0.02	0.25
1000	0.04	0.435
6000	too many variables	3.05

Table 3: Search times for 3-colorability

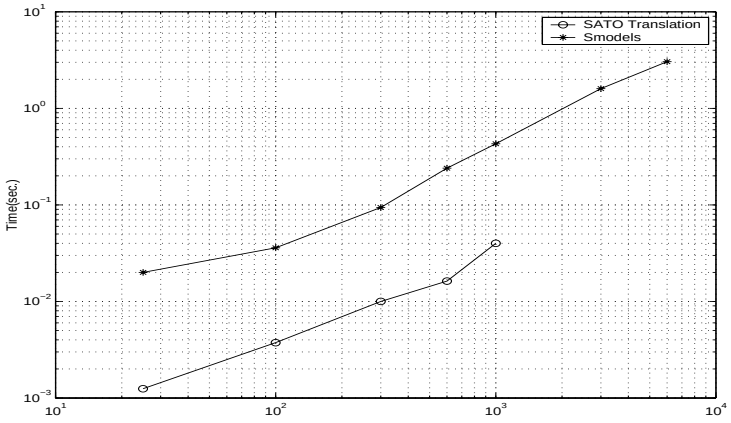

Figure 3: 3-colorability

4 Experiments with a non-2-literal program

A logic programming encoding of the Blocks World is given by Niemelä [12]. It is shown in [1] that the completion formula of the program characterizes the stable model semantics.

We give two tables showing different kinds of experimental results. The setting under which the experiments were conducted is the same as before, except that we experimented with different values of the parameter g in the case of SATO. It turns out that this is important.

In the first table, the data files large.c, large.d, and large.e (which specify the blocks, their initial configuration and the target configuration) were taken from Niemelä's web page. For each instance, two cases were tested, the case with the smaller number of steps had no solution whereas the one with the next larger number had a solution. For large.c for instance, 7 steps cannot move from the given configuration to the target one while 8 steps can. We created the data file large.f with 21 blocks which requires 11 steps to solve. This instance turns out to be important in illustrating the changing behavior of SATO. The header SATO (g=20) means that the SAT instance translated from the logic program by completion was run by setting the parameter g to 20. The last column, with the header SATO-2 (g=20), reports the experiments for the approach where a non-2-literal rule was first translated to a number of 2-literal rules using at most one extra variable, as follows:

> If r is $h \leftarrow l_1, \ldots, l_m$ with $m \geq 2$, then translate r into $h \leftarrow \texttt{not}\ x_r$, and, for $1 \leq k \leq m$, $x_r \leftarrow \texttt{not}\ l_k$ if l_k is an atom ϕ and $x_r \leftarrow l_k$ if l_k is $\texttt{not}\ \phi$.

Essentially, this breaks rule r into several 2-literal rules by representing the body of the rule by a named proposition. It can be shown that the class of programs that can be faithfully translated this way includes the class of programs whose completion characterizes the stable model semantics. Since the latter is the case for the Blocks World program of Niemelä, this translation preserves the stable model semantics. Then, the resulting 2-literal program was translated to a SAT instance, and run by SATO. As the reader can see, even with a linear number of extra variables, the performance became unacceptable very quickly.

In the second table, we illustrate the performance variations of SATO on the two larger instances. It turns out that with the default value of g no answer was returned in 2 hours; and with $g = 90$, SATO was extremely efficient.

5 Related Work

Babovich et al. [1] also reported experiments with SMODELS and SATO for the Blocks World problem. They timed their experiments using the unix time command. For SMODELS this included the time used by LPARSE. They also used an older version of SATO where no parameter g was provided. In addition, we used a much faster machine. To a large extent, our experimental results for $g = 20$ are

Problem	Blocks	Steps	SMODELS	SATO(g=20)	SATO-2(g=20)
large.c	15	7	0.58	0.38	1.17
		8	4.65	0.78	1.97
large.d	16	8	1.26	1.09	7.50
		9	10.08	0.98	11.26
large.e	17	9	1.99	1.58	11.84
		10	17.15	6.17	71.49
large.f	21	10	2.62	307.07	too long
		11	55.57	29.59	too long

Table 4: Search times for Blocks World

Problem	Blocks	Steps	SATO (g=10)	SATO (g=20)	SATO (g=90)
large.e	17	9	2.41	1.58	2.08
		10	761.93	6.17	3.34
large.f	21	10	55.88	307.07	3.87
		11	too long	29.59	8.70

Table 5: Comparison with different values of g

comparable to theirs. However, the creation of the instance large.f in our case revealed the changing behavior of SATO. In contrast, SMODELS behaved gracefully and proportionally.

Given a program P, if its completion formula characterizes the stable model semantics, then obviously P can be translated to a SAT instance without using extra variables. Note that this in general doesn't guarantee that the resulting SAT instance is of the polynomial size, since it involves converting a disjunctive normal form to a conjunctive normal form.

There are problems whose "natural" solutions are 2-literal programs but their completions may not characterize the stable models semantics. One of these problems is the reachability problem (known to be NL-complete, i.e. nondeterministic log-space complete): given a graph $G = (V, E)$ and two vertices s and t, determine whether there is a path from s to t. The problem can be solved by the following program:

$$reached(V) \leftarrow arc(U, V), reached(U)$$
$$reached(V) \leftarrow arc(s, V)$$
$$reached(t) \leftarrow \texttt{not}\ reached(t)$$

The ground instances of the first rule become 2-literal rules after true instances of $arc(U, V)$ are removed. Now consider the digraph with the set of vertices $V = \{s, t, u\}$ and the set of paths $E = \{(t, u), (u, t)\}$. Though the problem has no stable models, its completion formula has a model.

6 Conclusion

A main result of this paper is the discovery of the class of 2-literal programs for which an efficient SAT encoding exists and requires no extra variables. Our experimental results indicate that the SAT translations of these programs can be solved efficiently by a competent SAT solver. Whether to use extra variables or not in a translation could be significant, as extra variables may increase search space exponentially. Our experimental results also suggest that even with a linear number of extra variables, the performance can be degraded significantly. We also reported the changing behavior of SATO for some larger Blocks World instances, which were not known previously. This seems to suggest that the advantage of SATO on search efficiency is not at all obvious for non-2-literal programs in general, even if some of these programs can be translated to SAT instances without using extra variables.

References

[1] Y. Babovich, E. Erdem, and V. Lifschitz. Fage's theorem and answer set programming. In *Proc. Int'l Workshop on Non-Monotonic Reasoning*, 2000.

[2] R. Ben-Eliyahu and R. Dechter. Propositional semantics for disjunctive logic programs. *Annals of Math. and Artificial Intelligence*, 14:53–87, 1994.

[3] K.L. Clark. Negation as failure. *Logics and Databases*, pages 293–322, 1978.

[4] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[5] P. Dung. A fixpoint approach to declarative semantics of logic programs. In *Proc. North American Conf. on Logic Programming*, pages 604–625, 1989.

[6] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. 5th ICLP*, pages 1070–1080. MIT Press, 1988.

[7] T. Janhunen, I. Niemelä, P. Simons, and J. You. Unfolding partiality and disjunctions in stable model semantics. In *Proc. KR 2000*, pages 411–424. Morgan Kaufmann, April 2000.

[8] N. Leone et al. DLV: a disjunctive datalog system, release 2000-10-15. At `http://www.dbai.tuwien.ac.at/proj/dlv/`, 2000.

[9] V. Lifschitz. Answer set programming. In K.R. Apt et al., editor, *The Logic Programming Paradigm: A 25-Year Perspective*, pages 357–371. Springer, 1999.

[10] V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K.R. Apt et al., editor, *The Logic Programming Paradigm: A 25-Year Perspective*, pages 375–398. Springer, 1999.

[11] M. Moskecicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Enginerring an efficient sat solver. In *Proc. 38th ACM Design Automation Conference*, pages 530–535, June 2000.

[12] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Math. and Artificial Intelligence*, 25(3-4):241–273, 1999.

[13] I. Niemelä and P. Simons. *Extending the Smodels system with cardinality and weight constraints*, pages 491–521. Kluwer Academic Publishers, 2000.

[14] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[15] P. Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Helsinki, Finland, 2000.

[16] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*. To appear.

[17] M. Truszczyński. Computing large and small stable models. In *Proc. ICLP*, pages 169–183. MIT Press, 1999.

[18] H. Zhang. Sato: an efficient propositional prover. In *Proc. CADE*, pages 272–275, 1997.