Answer Set Programming with Clause Learning

Jeffrey Ward[1] and John S. Schlipf[2]

[1] The Ohio State University
jward@ececs.uc.edu
[2] University of Cincinnati, USA
john.schlipf@uc.edu

Abstract. A conflict clause represents a backtracking solver's analysis of why a conflict occurred. This analysis can be used to further prune the search space and to direct the search heuristic. The use of such clauses has been very important in improving the efficiency of satisfiability (SAT) solvers over the past few years, especially on structured problems coming from applications. We describe how we have adapted conflict clause techniques for use in the answer set solver Smodels. We experimentally compare the resulting program to the original Smodels program. We also compare to ASSAT and Cmodels, which take a different approach to adding clauses to constrain an answer set search.

1 Introduction

Recent years have seen the development of several stable model/answer set solvers. Smodels [Sim00,NSS00,Nie99] and DLV [EFLP00] are commonly used. (DLV implements more, namely disjunctive logic programming. However, it also serves as an effective stable model search engine.) These have demonstrated the feasibility of answer set programming as a paradigm for applications.

These programs have built upon the earlier propositional CNF satisfiability (SAT) solvers. But as the technology of the answer set programming systems has improved, the SAT solvers have gone on to implement new techniques, noticeably conflict clause usage (also known as "lemmas"), a variety of new search heuristics (which are frequently based on conflict clauses), and new highly efficient data structures. Key to many of these applications seems to be that some variables, and some combinations of variables, are far more important than others. More recent SAT solvers such as GRASP [MS99], SATO [Zheng97], rel_sat [BS97], Chaff [MMZZM01], BerkMin [GN02], and SIMO [GMT03], through creating and processing conflict clauses, often "learn" important information for the search.

The Cmodels–1 solver [Cmod] addresses this problem by piggy-backing an answer set solver onto a SAT solver. It handles a class of logic programs called *tight* [Fages94, BEL00], in which the stable models are just the models of Clark's completion of the program [Clark78] — and the completion is a classical logic problem. So Cmodels–1, after some preprocessing, passes a completion to a SAT solver, such as Chaff. Our concern in this paper is with solvers which are not limited to tight programs, so we do not discuss Cmodels–1 further here.[1]

[1] A different tool, by East and Truszczyński, takes a sort of middle ground between Cmodels–1 and ASSAT, but we shall not discuss it here either.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 302–313, 2004.
© Springer-Verlag Berlin Heidelberg 2004

ASSAT [LZ02,ASSAT] drops the restriction to tight programs by an iterative process of calling a SAT solver, evaluating whether the model produced is stable, and, if not, adding further specifications ("loop formulas") to the problem to avoid this failure of stability. Recently, Cmodels–2 [BM03] adapts the technique of ASSAT, but extends its application to extended and disjunctive rules, and makes available alternative loop formula definitions.

Here we present an answer set programming tool (for non-disjunctive answer set programs), *Smodels*$_{cc}$ (Smodels with conflict clauses), that deals with the new technologies in SAT solvers in a different way. Instead of calling a fast SAT solver to perform the search, it incorporates some of the techniques of modern SAT solvers into a variant of Smodels. Like ASSAT, Smodels$_{cc}$ is intended to deal with non-tight, as well as tight, logic programs. It turns out that it is often much faster than the ASSAT approach on non-tight programs, since it incorporates the unfounded set calculation directly into the search engine, thus allowing tighter pruning of the search tree.

2 Background

Propositional CNF-SAT solvers. A basic *Davis-Putnam-Loveland-Logeman* $(DPLL)$ SAT solver [DLL62], given a set C of clauses, performs a backtracking search to find a model for C. We sketch it below as a recursive function, passed a set of literals representing a partial truth assignment. Initially, the partial assignment is empty. A *unit clause* is a not-yet-satisfied clause containing only one unassigned literal (which is thus forced); searching regularly for unit clauses and immediately inferring those literals, called *unit-propagation*, is almost universally done. In practice, literals forced by unit propagation are put into an "inference queue" until the data structures (not shown here) are updated; a contradiction is always revealed by inferring contradictory literals.

```
srch4ModlExtndng (partlAssgn)
   while there exists a unit clause c
      let unitLit be the remaining literal in c
      partlAssgn := partlAssgn union {unitLit}
      if any clause has been falsified, return (* backtrack *)
   if partlAssgn is total (contains each variable or its negation)
      output ''SAT'', output partlAssgn, and halt program
   else
      pick ltrl to guess next (i.e., branch on) - by a heuristic
      srch4ModlExtndng(partlAssgn union {ltrl})
      srch4ModlExtndng(partlAssgn union {not ltrl})
   if  partlAssgn is empty (* back at top level *) output ''unSAT''
```

Recent solvers add *conflict clause learning*. When Chaff infers a contradiction, it finds a *relatively* small $\{\lambda_1, \lambda_2, \ldots, \lambda_m\} \subseteq$ *partlAssgn* leading to that contradiction and, functionally, adds to C the *conflict clause* (a.k.a., *lemma*) $cc = (\neg\lambda_1 \vee \neg\lambda_2 \vee \cdots \vee \neg\lambda_m)$; cc is always a resolvent of clauses in C. It then backtracks (*backjumps*) to whatever level in the search tree unit propagation was executed after the *second to the last* of $\lambda_1, \lambda_2, \ldots, \lambda_m$ was added and simply restarts its search from this point. Since cc has been added to C, once all but one of the λ_i's are ever inferred again, the final $\neg\lambda_j$ will

be inferred, *before further search*, by unit propagation — converting the DPLL search tree to a DAG.[2] Some solvers also use the conflict clauses in their heuristics to choose literals to branch on (*i.e.*, guess next). An oversimplification is that Chaff branches on the unassigned literal occurring in the most conflict clauses.

Since so many conflict clauses are generated, systems must deal with storage overflow. Also, searching a huge store of conflict clauses for unit-clauses is very time-consuming. Chaff and BerkMin periodically completely restart their searches, keeping only those conflict clauses that, by some heuristic, seem likely to be useful. The cache of conflict clauses is then garbage collected and recompacted (improving data locality). Smodels$_{cc}$ also restarts, but it discards clauses continuously throughout the search and does no recompaction. Currently, Smodels$_{cc}$ keeps the 5000 most recently generated conflict clauses and all conflict clauses with no more than 50 literals.

Answer Set Solvers. In the absence of disjunctive rules (as with Smodels), the heart of an answer set solver is a search for stable models for normal logic programs. Currently the most frequently cited are Smodels and DLV. The code for Smodels$_{cc}$ is a modification of the code for Smodels, which is open source software. Accordingly, Smodels$_{cc}$ is also open source.[3]

Smodels replaces the simple unit propagation inference rule of DPLL SAT solvers with a set of inference rules based upon an inflationary variant of the wellfounded semantics; oversimplifying, we shall refer to this as closing under the wellfounded semantics. After closing under the wellfounded semantics, Smodels computes a *lookahead* on each unassigned variable x — called a *unit lookahead*: Assuming that x is *true* (*resp.*, *false*), it computes the wellfounded extension. If that gives a contradiction, Smodels infers that x is false (*resp.*, true). If it infers both, it backtracks; otherwise, it branches on (next guesses) a literal λ maximizing the inferences obtained by looking ahead on λ and $\neg\lambda$.

Smodels does not use conflict clauses or restarts.

Reducing Answer Sets to CNF-SAT. As noted above, we restrict attention to answer set solvers which can handle non-tight programs, even though solvers restricted to tight programs may be highly useful. (Indeed many frequently cited "benchmarks," such as graph coloring, naturally translate to tight programs.)

ASSAT and Cmodels-2 are general purpose Answer Set solvers that call SAT solvers to do most of their work. Given a program P, they pass the program completion $\overline{P}$ to a SAT solver. If $\overline{P}$ has no model, P has no stable model, and ASSAT and Cmodels-2 report "no."

A set U of atoms is *unfounded* over a partial truth assignment A if, for every rule $a \leftarrow b_1, \ldots, b_k, \mathit{not}\ c_1, \ldots, \mathit{not}\ c_m$ of P with $a \in A$, either (i) some $\neg b_i$ or $c_j \in A$ or (ii) some $b_i \in U$. If U is unfounded, the stable and well-founded semantics infer $\{\neg a : a \in U\}$, as a form of negation as failure. The program completion achieves only part (i) above, inferring fewer negative literals.

[2] Since the sets of conflict clauses may be huge, traversing all of C to do unit propagation is prohibitively slow. The best current method to do this seems to be Chaff's "optimized Boolean Constraint Propagation" [MMZZM01]. We use that also in our program.

[3] Smodels$_{cc}$, and the benchmarks used in this paper, can be obtained at http://www.ececs.uc.edu/~ schlipf/.

If the SAT solver returns a model, ASSAT and Cmodels-2 check whether all atoms in all unfounded sets are assigned false. If so, they return the model. If not, they create "loop formulas," which exclude the model, and feed them back into the SAT solver — repeating until a stable model is found or the search fails. The alternation between the SAT-solver phase and the unfounded-set-check phase may be inefficient, since the SAT solver may spend a great deal of time searching a branch which immediately could be pruned away by detecting an unfounded set.

An important advantage of ASSAT and Cmodels-2 is a sort of modularity: they are free to adapt to whatever SAT solver proves to be experimentally best for their applications. Smodels$_{cc}$ merges the "classical inference" part with the unfounded set check and thus sacrifices this modularity.

3 Conflict Clause Generation

Crucial to many modern SAT solvers is creating, storing, and looking up conflict clauses. Zhang *et al.* [ZMMM01] studies several different strategies for generating conflict clauses, implementing each of them in the zChaff variant of Chaff. We describe here their most effective strategy (the *1UIP* strategy) and how we adapted it for Smodels$_{cc}$.

When a solver such as Chaff detects a contradiction, it does a "critical path analysis" to choose a conflict clause. Because Chaff's inferences are derived through unit propagation only, if x and $\neg x$ are both inferred, they must have been inferred *since the latest choice made by the brancher*. Suppose that λ_0 was the last choice of the brancher. Chaff reconstructs the sequence of inferences used to infer each truth assignment since that last choice, representing it with a digraph, called the *implication graph*. The nodes are literals. There is an edge from literal λ_1 to λ_2 if a clause $\{\lambda_2, \neg\lambda_1, \ldots\}$ was used to infer λ_2. See Fig. 1.

So there is at least one directed path in the implication graph from λ_0 to x, and at least one from λ_0 to $\neg x$. A node on all these directed paths is called a *unique implication point* (*UIP*). Literal λ_0 itself is a UIP. All UIPs must lie on a single path from λ_0. Pick the UIP λ' farthest from λ_0 (i.e., closest to the contradiction). The derivation of the contradictory x and $\neg x$ now can be broken into (i) a derivation of λ' from λ_0 plus (ii) a derivation of the contradiction from λ'. By choice of λ', clauses involved in part (ii) contain only λ' plus some literals $\kappa_1, \ldots, \kappa_m$ that had been guessed or derived before Chaff branched on λ_0 — from a point higher up the search stack. The new conflict clause is $\neg\kappa_1 \vee \neg\kappa_2 \vee \cdots \vee \neg\kappa_m \vee \neg\lambda'$; for the example in Fig. 1, that lemma is $\neg\kappa_1 \vee \neg\kappa_2 \vee \neg\kappa_3 \vee \neg\kappa_4 \vee \neg\kappa_5 \vee \neg\kappa_{10} \vee \neg\kappa_{11} \vee \neg\kappa_{12} \vee \neg\lambda'$.[4]

At this point, Chaff does not simply backtrack on the last choice assignment. Rather, it "backjumps" to the level in the search tree where the last κ_i was guessed or inferred and restarts the search there with the new conflict clause in the cache. It will infer $\neg\lambda'$ at that level before going on with the search; this will keep it from exactly retracing its previous chain of guesses and inferences.

In a stable model solver, such as Smodels, contradictory literals need not be inferred in the same level of the backtrack search: an atom x may be inferred by forward or

[4] It is also possible to store multiple conflict clauses per contradiction. Following [ZMMM01], we create only one conflict clause for each contradiction.

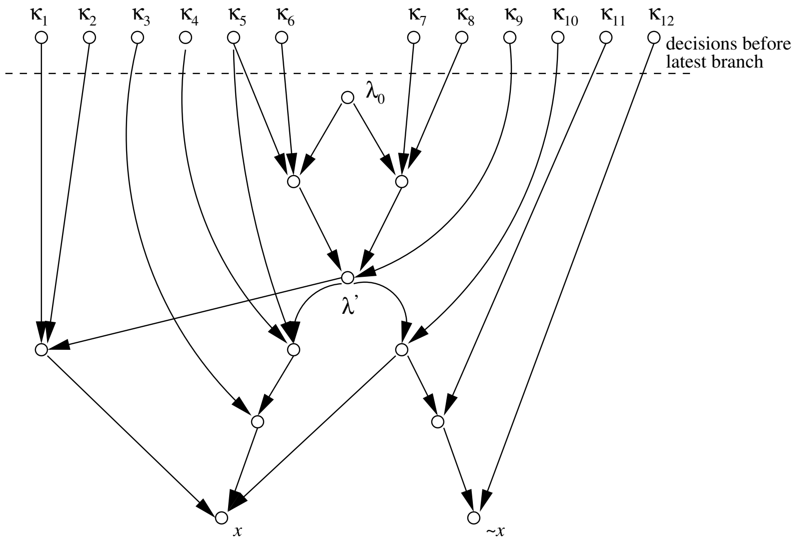

Fig. 1. An implication graph for a contradiction

contrapositive reasoning at one level, while x may appear in an unfounded set earlier or later. Thus in Smodels$_{cc}$, the construction above is altered a little. The *later* of x, $\neg x$ to be guessed or inferred is called the *conflict literal*, and, for the purposes of finding a UIP, an extra edge is added into the digraph, from the earlier literal in the conflicting pair to the conflict literal. Now a UIP is defined to be a literal, other than the conflict literal itself, appearing on every path from the guessed literal λ_0 to the conflict literal. Identifying the UIPs in Smodels$_{cc}$ is complicated by the fact that the implication graph may have cycles (in the case of inferences based upon unfounded sets). Nonetheless, identifying the UIPs may be accomplished in $O(|E|)$ time, where E is the set of edges incident to vertices between the choice literal and the conflict literal. Otherwise, Smodels$_{cc}$ constructs its conflict clauses from the implication graph as described above.

Smodels$_{cc}$ uses Smodels' five different inference rules (below), four corresponding to unit propagation on the completion of the program, and the fifth an unfounded set rule. For each of these inference rules we describe below how we add corresponding edges to the implication graph. Compared to construction of the implication graph in a DPLL-based SAT solver, implication graph construction in Smodels$_{cc}$ is relatively complex and costly.

Forward inference. *If all the subgoals in a rule*

$$a \leftarrow b_1, \ldots, b_k, not\ c_1, \ldots, not\ c_m$$

are true in the current assignment, infer a*.* Add edges from all b_i's and $\neg c_j$'s to a in the implication graph.

Kripke-Kleene negation (*all rules canceled*): *If every rule with head* a *has at least one subgoal negated in the current truth assignment, infer* $\neg a$*.* For each rule $a \leftarrow$

$b_1, \ldots, b_k, not\ c_1, \ldots, not\ c_m$ with head a, determine the cancelling assignment, $\neg b_i$ or c_j, which was guessed or inferred first (at the earliest level of the backtracking search), and add an edge from that assignment to $\neg a$ in the implication graph.

Contraposition for true heads. *If atom a is true in the current truth assignment, and if every rule with head a* except one *has at least one subgoal that is false in the current truth assignment, infer all the subgoals of that remaining rule to be true.* For example, suppose the only rules with a in their heads are $a \leftarrow b, c, not\ d$; $a \leftarrow e, f$; and $a \leftarrow not\ g, h$; and that the current truth assignment contains $a, d, \neg e$. Then $\neg g, h$ will be inferred. Add edges from each of $a, d, \neg e$ to each of $\neg g, h$ into the implication graph.

Contraposition for false heads. *If an atom a is false in the current truth assignment and some rule $a \leftarrow \lambda_1, \ldots, \lambda_k$ has every λ_i except one true in the current truth assignment, infer that last λ_i to be false.* For example, suppose the rule is $a \leftarrow b, c, not\ d$ and that $\neg a, b, c$ are in the truth assignment. Infer d, and add edges from each of $\neg a, b, c$ to d.

Wellfounded negation (Smodels' *at most*): *Temporarily removing all satisfied and undefined negative subgoals in all rules of the program yields a Horn program. Compute its least model M; the set of atoms* false *in M is unfounded; set these atoms to false in the current partial assignment.* (This is the logic; Smodels has a more complicated but faster calculation.)

For example, suppose that P contains the rules $a \leftarrow b$; $b \leftarrow c$; $c \leftarrow a$; and $a \leftarrow d$; and that these are the only rules with a, b, or c in their heads. Suppose also that $\neg d$ is in the current partial truth assignment. Then infer $\neg a$, $\neg b$, and $\neg c$. Add edges from $\neg d$ to $\neg a$, from $\neg a$ to $\neg c$, from $\neg c$ to $\neg b$, and from $\neg b$ to $\neg a$. Note that here the implication graph contains a cycle.

The algorithm for determining the edges for wellfounded negation is similar to the one for Kripke-Kleene negation: If an atom a has been detected to be unfounded then it will be the case that Smodels has found a reason to cancel every rule R with a at the head. As in Kripke-Kleene negation, add an edge from the earliest reason for the cancellation to the literal $\neg a$. In particular, if the earliest reason is that a set U of atoms mentioned positively in the body of R have become simultaneously unfounded (unsupported) with a, then the edge will have as its source node $\neg x$ where x is the member of U which was first permanently removed from Smodels' set of supported atoms during the current unfounded set check.

4 Search Heuristics

As noted earlier, Smodel's search heuristic is based on its unit lookaheads. Chaff weights its literals based upon how many of the conflict clauses they occur in[5] and always branches on an unassigned literal of maximum weight; thus Chaff can be thought of as "learning key literals."

The heuristic used in Smodels$_{cc}$ is modeled after the one in BerkMin. It works as follows: Each variable x has an "activity count", $ac(x)$, which counts the number of

[5] It also lets the weights from older clauses decay with time.

times that either x or $\neg x$ has been involved in producing a conflict (i.e., has appeared in a conflict clause or has appeared in an implication graph along a path from a conflict node to a variable in the corresponding conflict clause). We choose the branching literal x_0 or $\neg x_0$ such that $ac(x_0)$ is maximized, with a restriction: If there are unsatisfied conflict clauses in the cache, then x_0 must be chosen from the most recently constructed unsatisfied conflict clause. Branch on x_0 or $\neg x_0$, whichever has appeared in more conflict clauses.[6]

5 Experimental Results

We performed experiments to compare the performance of Smodels (version 2.26), ASSAT (version 1.50), Cmodels-2 (version 1.04), and Smodels$_{cc}$. Our experiments were run on 1533 Mhz Athlon XP processors with 1 GB of main memory, using Linux. With ASSAT, we used the -s 2 command line option, which seemed to give marginally better results than the default or the -s 1 settings. For Cmodels, we used the default settings, which produced the best results on tight problems. However, on Hamiltonian cycle problems, we used Cmodels' -si setting, which produced significant performance improvements in that domain. The SAT solver used by ASSAT was Chaff2 (version spelt3) [Chaff2], which is ASSAT's default solver. Cmodels used the zChaff SAT solver (version 2003.7.1) for tight problems and SIMO (version 3.0) [SIMO] for Hamiltonian cycle problems, as dictated by the command line settings. Note that ASSAT and Cmodels benefit from conflict clauses in our tests, because conflict clauses are heavily incorporated into Chaff and SIMO. In each of the tables below (except for the DLX benchmarks), the run times given represent the median and maximum number of user time seconds taken to solve each of eleven (11) randomly generated problem instances. Each search process was aborted ("timed out") after 3600 seconds. Runtimes reported include the time to execute Lparse, the default grounder for Smodels, ASSAT, Cmodels, and Smodels$_{cc}$.

We concentrated on three problem domains:

Boolean satisfiability. Our tests in this domain include some randomly generated 3-SAT problems and 16 of the "DLX" circuit verification benchmarks from Miroslav Velev [VB99]. [7] In each case, the problem is provided as a CNF-SAT problem, which we have converted to an answer set program. We do not expect to outperform ASSAT or

[6] We also tested a version of Smodels$_{cc}$ which used lookaheads for the search heuristic (Smodels' original default heuristic). This version frequently had somewhat smaller search trees than did Smodels$_{cc}$ with the BerkMin-like heuristic (possibly because of the extra pruning afforded when contradictions are found during lookaheads). It solved a substantial number of our experimental test problems much faster than did the original Smodels, showing that much of the power of conflict clauses comes from the backjumping and unit propagation-based pruning which they afford (i.e., not only from how they affect the search heuristic). However, because the BerkMin-like heuristic is so much less expensive to compute than the lookahead-based heuristic, we found that it provided substantially better overall performance in our experiments. Thus, all of the timings for Smodels$_{cc}$ in our Experimental Results section were obtained using the BerkMin-like heuristic.

[7] The benchmarks which we used were from the Superscalar Suite 1.0 (SSS.1.0), available at http://www.ece.cmu.edu/˜ mvelev. The eight satisfiable instances which we tested were

Cmodels on these examples since the logic programs are tight. Trying the "dlx" problems was an attempt to show that conflict clauses can be particularly helpful on non-uniform, "real world" data.

Median and maximum seconds on 11 random 3-SAT problems

Data set		Smodels		Smodels$_{cc}$		ASSAT		Cmodels		#Sat
vars	clauses	median	max	median	max	median	max	median	max	
250	1025	16.5	112.2	9.0	891.0	3.3	552.2	2.5	746.0	10
250	1050	86.1	207.2	1056.8	2136.4	334.3	2321.1	777.5	1761.2	6
250	1075	133.4	376.6	576.4	1 abort	801.1	2 abort	883.9	1 abort	3
250	1100	74.4	169.8	432.8	2110.8	329.9	1 abort	360.0	2216.0	1

Median and maximum seconds on 8 DLX benchmark problems

Data set	Smodels		Smodels$_{cc}$		ASSAT		Cmodels		#Sat
	median	max	median	max	median	max	median	max	
8 satisfiable	>3600	8 abort	11.6	17.9	2.6	4.4	2.9	10.8	8
8 unsatisfiable	>3600	7 abort	15.9	41.4	6.8	18.2	8.6	14.4	0

Graph k-coloring problems. We study these because they are standard in the literature. Since the program is tight, much of the sophistication of answer set programming is not needed. Nevertheless, it is important that the solver be reasonably efficient on such problems.

We first generated problems on uniform, random graphs with 400 to 500 vertices. We also generated "clumpy" problems in this domain by making graphs of 100 clumps with 100 nodes each (for a total of 10,000 nodes per graph). For the first set of clumpy graphs we set our density parameter $d = 150$, which means that we randomly placed 150 edges in each clump and 150 edges between clumps. This gave us graphs with a total of $100 \times 150 + 150 = 15{,}150$ edges each. We then increased d to 170 and 190, obtaining graphs with 17,170 and 19,190 edges, respectively.

As with Boolean satisfiability, we expected ASSAT and Cmodels to outperform Smodels$_{cc}$ since the program is tight. We expected Smodels to be faster than Smodels$_{cc}$ on fairly "uniform" graphs, and Smodels$_{cc}$ to be faster than Smodels on "non-uniform" graphs.

Hamiltonian cycle problems. Among common current benchmarks, these may be the "gold standard" for answer set solvers since the problem description is not tight. We used directed graphs in these experiments.

Here we considered three reductions to answer set programming. The first, a standard reduction frequently used in benchmarking, was taken from [Nie99].[8] The second reduction was a "tight on its models" reduction used with ASSAT in [LZ03]. The third reduction is the modification below of the first:

dlx2_cc_bug01.cnf, ..., dlx2_cc_bug08.cnf The eight unsatisfiable instances were dlx1_c.cnf, dlx2_aa.cnf, dlx2_ca.cnf, dlx2_cc.cnf, dlx2_cl.cnf, dlx2_cs.cnf, dlx2_la.cnf, and dlx2_sa.cnf.

[8] Except that the extended rules were replaced with choice constructions since ASSAT and Smodels$_{cc}$ do not currently support the extended rules. They will be added to Smodels$_{cc}$ soon.

Median and max secs on 11 random 3-coloring problems, uniform distribution of edges

Data set		Smodels		Smodels_{cc}		ASSAT		Cmodels		#Sat
vertices	edges	median	max	median	max	median	max	median	max	
400	900	3.4	163.0	1.9	197.2	0.3	88.3	0.3	88.2	10
400	950	99.5	511.8	89.8	1704.4	58.3	837.7	24.6	837.6	2
400	1000	20.7	134.3	16.0	68.4	3.7	14.3	4.0	13.1	0
500	1100	0.8	4.2	1.4	6.4	0.2	2.4	0.3	2.2	11
500	1150	196.0	356.6	697.9	2 abort	44.4	2 abort	188.3	2 abort	10
500	1200	2753.9	5 abort	>3600	8 abort	>3600	8 abort	>3600	8 abort	0-5

Median and max secs on 11 random 3-coloring problems, clumpy distribution of edges

Data set		Smodels		Smodels_{cc}		ASSAT		Cmodels		#Sat
vertices	edges	median	max	median	max	median	max	median	max	
10000	15150	256.1	1 abort	21.3	50.5	8.0	9.7	8.4	10.1	10
10000	17170	201.6	3 abort	19.2	21.7	8.5	11.4	10.1	11.8	7
10000	19190	>3600	8 abort	8.5	223.4	3.2	12.9	4.2	11.7	2

```
hc(X,Y) :- not not_hc(X,Y), edge(X,Y).
not_hc(X,Y) :- not hc(X,Y), edge(X,Y).
:- hc(X1,Y), hc(X2,Y), edge(X1,Y), edge(X2,Y), vtx(Y), X1 != X2.
:- hc(X,Y1), hc(X,Y2), edge(X,Y1), edge(X,Y2), vtx(X), Y1 != Y2.
:- vtx(X), not r(X).
r(Y) :- hc(X,Y), edge(X,Y), initialvtx(X).
r(Y) :- hc(X,Y), edge(X,Y), r(X), not initialvtx(X).
outgoing(V) :- edge(V,U), hc(V,U).     % added in 3rd reduction
:- vtx(V), not outgoing(V).            % added in 3rd reduction
```

The two lines which we added to obtain the third reduction state explicitly that, in a Hamiltonian cycle, every node must have an outgoing edge. This fact is implicit in the reduction from [Nie99]. However, stating it explicitly helps the solvers prune their search spaces. We note that these two lines were included in the "tight on its models" reduction from [LZ03].

In our experiments the third reduction was always faster than the first two, so we used it in all experiments reported here.[9] In all of these experiments we enforced the restriction that every vertex in every graph must have an in-degree and an out-degree ≥ 1 to avoid trivial examples.

We nonetheless found it difficult to generate hard Hamiltonian cycle problems using a random, uniform distribution of edges. For instance, our randomly generated problems with 6000 nodes were in all cases too large to run under ASSAT, which was understandable because the ground instantiations produced by Lparse were around 12MB in size.

[9] This performance improvement was very pronounced with Smodels and Smodels_{cc}. Using ASSAT the runtimes were only slightly better with the third reduction than with the second, but on larger problems, under the second reduction, the groundings provided by Lparse were too large for any of the solvers to handle. For example, on a randomly generated graph with 1000 nodes and 3500 edges, Lparse generated a ground instantiation of about 123MB under the second reduction. By contrast, the ground instantiation produced for the third reduction was 1.09MB, and the problem was easily solved by all of the solvers.

Yet these problems were not particularly difficult to solve, at least for Smodels$_{cc}$. Unsatisfiable instances taken from this distribution were generally solved by each of the solvers with no backtracks.

We sought to produce some hard Hamiltonian cycle problems that were of reasonable size and had a less uniform (more "clumpy") distribution of edges. (It is commonly believed that less uniform distributions are in fact common in actual applications.) For these experiments we randomly generated "clumpy" graphs which would be forced to have some, but relatively few, Hamiltonian cycles each.

For this purpose we generate a random mn-node "clumpy" graph G as follows: Let n be the number of clumps in the graph and let m be the number of nodes in each clump. First generate an n-node graph A as the "master graph," specifying how the clumps are to be connected; each vertex of A will correspond to a clump in the final graph. Add random edges to A until A has a Hamiltonian cycle.

Now generate the clump C corresponding to a vertex v of A. Let $x = indegree(v)$ and $y = outdegree(v)$. C has m nodes; select x nodes to be "in-nodes" and y different nodes to be "out-nodes"; increase C to $x + y$ nodes if $x + y > m$. Add random edges to clump C until there are Hamiltonian paths from each in-node of C to each out-node of C. (Thus C will have at least xy Hamiltonian paths.)

Finally, for every edge (v_i, v_j) in the master graph A, insert an edge from an out-node of C_i to an in-node of C_j. Every in-node in every clump is to have exactly one incoming edge from another clump. Likewise, every out-node in every clump is to have exactly one outgoing edge to another clump. G will have at least one Hamiltonian cycle.

Median and max secs on 11 Hamiltonian cycle problems, uniform distribution of edges

Data set		Smodels		Smodels$_{cc}$		ASSAT		Cmodels		#Sat
vertices	edges	median	max	median	max	median	max	median	max	
1000	4000	1.0	1.1	0.9	1.1	0.9	1.1	1.0	1.1	0
1000	4500	1.2	132.0	1.2	3.6	1.1	971.3	1.3	308.0	4
1000	5000	172.7	201.0	3.6	4.8	52.6	1064.1	6.8	438.4	6
1000	5500	244.6	269.5	4.5	6.9	275.5	1440.6	141.1	175.7	10
6000	30000	8.4	3 abort	8.7	64.9	...	...	8.3	2 abort	3
6000	33000	>3600	9 abort	57.9	77.7	...	...	>3600	9 abort	9
6000	36000	>3600	7 abort	64.3	81.2	...	...	1963.4	3 abort	7
6000	39000	>3600	10 abort	78.4	109.9	...	...	>3600	7 abort	10

Median and max secs on 11 Hamiltonian cycle problems, clumpy distribution of edges

Data set		Smodels		Smodels$_{cc}$		ASSAT		Cmodels		#Sat
# of clumps	vertices / clump	median	max	median	max	median	max	median	max	
10	10	0.4	1 abort	0.2	0.5	0.8	1.4	0.5	0.9	11
12	12	620.9	4 abort	0.7	1.1	2.6	10.8	2.6	4.5	11
14	14	>3600	10 abort	1.3	3.9	108.8	194.1	11.4	83.4	11
16	16	>3600	9 abort	6.5	40.5	263.0	608.8	24.8	100.6	11
18	18	>3600	11 abort	42.7	353.4	>3600	6 abort	133.7	2063.5	11

6 Conclusions

We believe that this study has confirmed the following two points:

Adding conflict clauses to answer set search will significantly increase speed on non-uniform problems. Experience has shown that conflict clause learning has immensely speeded up SAT solvers on "real world" data. We adapted conflict clause analysis to the answer set programming domain, notably by finding a reasonable way to represent inference by wellfounded negation. We tested this on some "real world" tight problems plus some randomly generated non-uniform problems. Smodels$_{cc}$ was consistently faster than Smodels, confirming our prediction. Interestingly, for uniform Hamiltonian Cycle problems, Smodels$_{cc}$ was also much faster than Smodels.

For non-tight programs, separating the classical analysis of the completion from the unfounded set check, as in ASSAT and Cmodels, is less efficient than merging the two processes into a single search. The obvious explanation seems to be that, with ASSAT or Cmodels, the SAT solver spends a great deal of time on parts of the search tree that an unfounded set check could eliminate early.

A significant advantage of ASSAT and Cmodels is that they can incorporate the latest, highly optimized SAT solvers with relatively little additional programming effort.

For future work, a useful middle ground between their approach and that of Smodels$_{cc}$ might be to run a state-of-the-art SAT solver on the program completion, but modify the SAT solver so that it includes an unfounded set check during the search.

Acknowledgement. This research was partially supported by U.S. DoD grant MDA 904-02-C-1162.

References

[BEL00] Babovich, Y., E. Erdem, and V. Lifschitz. Fage's Theorem and Answer Set Programming. *Proc. Int'l Workshop on Non-Monotonic Reasoning*, 2000.

[BM03] Babovich, Y. and M. Maratea. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. Available from *http://www.cs.utexas.edu/users/tag/cmodels.html*.

[BS97] Bayardo, R. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. *Proc. of the Int'l Conf. on Automate Deduction*, 1997.

[Clark78] Clark, K. Negation as failure. In Herve Gallaire and Jack Minker, eds., *Logic and Data Bases*, 293–322, Plenum Press, 1978.

[DLL62] Davis, M., G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communications of the Association of Computing Machinery* **5** (1962) 394–397.

[EFLP00] Eiter, T., W. Faber, N. Leone, G. Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, ed., *Logic-Based Artificial Intelligence* 2000, 79–103.

[Fages94] F. Fages. Consistency of Clark's completion and existence of stable models. *J. Methods of Logic in Computer Science* 1 (1994) 51–60.

[GMT03] Giunchiglia, E., M. Maratea, and A. Tacchella. Look-Ahead vs. Look-Back techniques in a modern SAT solver. SAT2003, May 5–8 2003. Portofino, Italy.

[GN02] Goldberg, E. and Y. Novikov. BerkMin: a Fast and Robust Sat–Solver. *Design Automation and Test in Europe (DATE)* 2002, 142–149.

[LZ03] Lin, F. and Z. Jicheng. On Tight Logic Programs and Yet Another Translation from Normal Logic Programs to Propositional Logic. *Proc. of IJCAI-03*. To appear, 2003.

[LZ02] Lin, F. and Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Proc. of the 18th Nat'l. Conf. on Artificial Intelligence (AAAI-2002)*.

[MS99] Marques-Silva, J.P. and K.A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. on Computers* 48(5), 1999, 506–521.

[MMZZM01] Moskewicz, M., C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. *Proc. of the 38th Design Automation Conf. (DAC'01)*, 2001.

[NSS00] Niemela, I., P. Simons, and T. Syrjanen. Smodels: a system for answer set programming. *Proc. of the 8th Int'l Workshop on Non-Monotonic Reasoning* 2000.

[Nie99] Niemela, I. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25(3–4) 1999, 241–273.

[Sim00] Simons, P. Extending and Implementing the Stable Model Semantics. PhD dissertation, *Helsinki University of Technology* 2000.

[VB99] Velev, M., and R. Bryant. Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic. *Correct Hardware Design and Verification Methods (CHARME'99)* 1999.

[Zheng97] Zhang, H. SATO: An Efficient Propositional Prover. *Proc. of Int'l Conf. on Automated Deduction* 1997.

[ZMMM01] Zhang, L., C. Madigan, M. Moskewicz, and S. Malik.: Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. *Proc. of ICCAD 2001*.

[ASSAT] ASSAT code available from *http://assat.cs.ust.hk/*.

[Cmod] Cmodels code available from *http://www.cs.utexas.edu/users/tag/cmodels.html*.

[Chaff2] Chaff2 code available from *http://www.ee.princeton.edu/ chaff/index1.html*.

[SIMO] Simo code is included in the Cmodels-2 distribution. See also *http://www.mrg.dist.unige.it/ sim/simo/*.