2006

Experiments with SAT-based Answer Set Programming

Enrico Giunchiglia
Universita di Genova

Yuliya Lierler
University of Nebraska at Omaha, ylierler@unomaha.edu

Marco Maratea
Universtia di Genova

A. Tacchella
Universita di Genova

Experiments with SAT-based Answer Set Programming

E. Giunchiglia[1], Yu. Lierler[2], M. Maratea[1,3], and A. Tacchella[1]

[1] STAR-Lab, DIST, University of Genova
viale Francesco Causa, 13 — 16145 Genova, Italy
{enrico,marco,tac}@dist.unige.it

[2] Department of Computer Science
University of Texas at Austin, Austin, TX, USA
yuliya@cs.utexas.edu

[3] Department of Mathematics, University of Calabria
viale Pietro Bucci, Cubo 31b — 87036 Rende (CS), Italy
maratea@mat.unical.it

Abstract. Answer Set Programming (ASP) emerged in the late 1990s as a new logic programming paradigm which has been successfully applied in various application domains. Propositional satisfiability (SAT) is one of the most studied problems in Computer Science. ASP and SAT are closely related: Recent works have studied their relation, and efficient SAT-based ASP solvers (like ASSAT and CMODELS) exist.
In this paper we report about (*i*) the extension of the basic procedures in CMODELS in order to incorporate the most popular SAT reasoning strategies, and (*ii*) an extensive comparative analysis involving also other state-of-the-art answer set solvers. The experimental analysis points out, besides the fact that CMODELS is highly competitive, that the reasoning strategies that work best on "small but hard" problems are ineffective on "big but easy" problems and vice-versa.

1 Introduction

Answer Set Programming (ASP) emerged in the late 1990s as a new logic programming paradigm [32, 34], and has been successfully applied in various domains including space shuttle control [35], planning [26], and the design and implementation of query answering systems [2]. Syntactically, ASP programs look like Prolog programs, but they are treated by rather different computational mechanisms. Indeed, ASP systems like CMODELS [24], SMODELS [36], SMODELS$_{cc}$ [37], DLV [21], and ASSAT [27, 29] interpret logic programs via the answer set semantics [11, 12]. The goal is to find the "models" (called answer sets) of the program, and not to evaluate whether a query is true or not, as in standard Prolog systems. The ASP approach is thus similar to propositional satisfiability checking, where propositional formulas encode the problem and models of the formula correspond to the solutions of the problem.

Propositional satisfiability (SAT) is one of the most intensely studied fields in Artificial Intelligence and Computer Science. Various procedures that can deal with thousands of variables are now available (see, e.g., [19]). Also motivated by the availability of efficient SAT solvers (such as SATZ [22] and MCHAFF [33]), various reductions from logic programs to SAT were introduced in the past. The most popular of such reductions is Clark's completion [3]. Fages ([10]) showed that if a logic program is "tight" then its answer sets are in one-to-one correspondence with the models of its Clark's completion. From a theoretical point of view, Fages' result was then generalized to include programs with infinitely many rules [25], programs tight "on their completion models" [1], programs with nested expressions in the bodies of the rules [8], and disjunctive programs [20]. From a practical point of view, computation of answer sets via Clark's completion and SAT solving has been first implemented in CMODELS, and has been also shown to be effective on many classes of problems. Then, using "loop formulas" [20], it was possible to enable SAT-based answer set solvers, like ASSAT and CMODELS2 (CMODELS ver. 2), to work also on the class of "non-tight" logic programs.

In [27, 29, 15] ASSAT and CMODELS2 proved to be very competitive w.r.t rival systems on many classes of problems, with CMODELS2 having some advantages over ASSAT, like it (*i*) works on a propositional formula without additional variables (except for those possibly introduced by the clause form transformation), and (*ii*) is guaranteed to work in polynomial space.

Given the SAT-based nature of our procedure, CMODELS2, we have been able to implement —with a relatively small effort— several search strategies and heuristics which have been shown effective in the SAT literature. Then, we experimentally analyze which combinations of reasoning strategies work best on which problems. In particular,

- We implemented various "look-ahead" strategies; "look-back" strategies; and "heuristics".
- We considered CMODELS2 with various combinations of strategies, and other state-of-the-art systems like SMODELS, SMODELS$_{cc}$, ASSAT, and DLV.
- We conducted an extensive experimental analysis, involving all the above mentioned versions of CMODELS2 and systems, and a variety of tight and non tight programs, ranging from "small" randomly generated programs with a few hundred atoms, up to "large" programs with tens of thousands variables.

Our experimental results show that the look-back (resp. look-ahead) version of CMODELS2 has a clear edge over the other state-of-the-art systems that we considered on large (resp. small randomly generated) problems. The look-back version of CMODELS2 is very competitive also on the other non random, non large programs that we considered.

If we focus on the performances of the various versions of CMODELS2, the experimental results also point out that:

1. On the small randomly generated problems, "look-ahead solvers" (featuring a rather sophisticated look-ahead based on "failed literal", a simple look-back strategy –essentially backtracking– and a heuristic based on the information gleaned during the look-ahead phase) are best.

2. On the large problems, "look-back solvers" (featuring a simple but efficient look-ahead strategy –essentially unit-propagation with 2 literal watching–, a rather sophisticated look-back based on "learning" and a constant time heuristic based on the information gleaned during the look-back phase) are best.
3. Adding a powerful look-back (resp. look-ahead) to a look-ahead (resp. look-back) solver does not lead to better performances if the resulting solver is run on the small (resp. large) problems that we considered.

Using the terminology in [17], our comparison is "fair" because all the reasoning strategies are realized on a common platform and thus the experimental evaluation is not biased by the differences due to the quality of the implementation, and is "significant" because CMODELS2 implements current state-of-the-art look-ahead/look-back strategies and heuristics. We believe that these results have important consequences both for developers and also for people interested in benchmarking ASP systems. For instance, our results say that we can hardly expect to develop a solver with the best performances on all the categories of problems. As a consequence,

- developers should focus on specific classes of benchmarks (e.g., on randomly generated programs), and
- benchmarking should take into account whether solvers have been designed for specific classes of programs: Indeed, it hardly makes sense to run a solver designed for random (resp. large) programs on large (resp. random) programs.

2 Formal Background

Syntax of Logic Programs. A *rule* is an expression of the form

$$p_0 \leftarrow p_1, \ldots, p_k, not\ p_{k+1}, \ldots, not\ p_m, not\ not\ p_{m+1}, \ldots, not\ not\ p_n \qquad (1)$$

($0 \leq k \leq m \leq n$) where p_0 is an atom or the symbol $\bot$ ($\bot$ is the logical symbol standing for the empty disjunction, i.e., *False*), $p_1, p_2, \ldots, p_n$ are atoms, and the symbol *not* is the "negation" as failure operator. p_0 is the *head* of the rule, and the expression at the right of the arrow is the *body*. The intuitive meaning of a rule (1) is that p_0 is in the solution whenever the body is satisfied.

A *(non disjunctive logic) program* is a finite set of rules.

If the head of a rule is $\bot$, we call the rule a *constraint*. If a rule (1) contains an expression of the form *not not* p_i, then the rule is called *nested*, otherwise the rule is *non nested* or *basic*. If a logic program Π contains at least one nested rule, Π is a *nested* program, otherwise is *non nested* or *basic*.

Answer Sets for Logic Programs. In order to give the definition we consider first the case in which the program Π does not contain the negation as failure operator *not* (i.e. for each rule (1) in Π, $n = m = k$). Let Π be such a program and let X be a set of atoms. We say that X is *closed* under Π if for every rule (1) in Π, $p_0 \in X$ whenever $\{p_1, p_2, \ldots, p_k\} \subseteq X$. In the $n = m = k$ hypothesis,

Π has only one *answer set*, and it is the smallest set of atoms closed under Π. Computing such an answer set can be done in linear time, via the Dowling-Gallier procedure [5], or via unit-propagation (assuming the symbol "$\leftarrow$" is understood as the standard material implication, and "," as conjunction). Now consider an arbitrary program Π. Let X be a set of atoms. A rule

$$p_0 \leftarrow p_1, \ldots, p_k$$

belongs to the *reduct* Π^X *of* Π *with respect to* X if and only if there is a rule (1) in Π with $X \cap \{p_{k+1}, \ldots, p_m\} = \emptyset$ and $\{p_{m+1}, \ldots, p_n\} \subseteq X$. Π^X is a program without negation as failure. We say that a subset X of the atoms in Π is an *answer set* for Π if X is an answer set for Π^X [11, 20]. Determining the existence of an answer set for a program Π is an NP-complete problem. Indeed, checking if a set of atoms X is an answer set of Π can be done in linear time by first computing the reduct Π^X and then computing the answer set of Π^X.

Completion. Consider a program Π. The completion $Comp(\Pi)$ of a program Π is a propositional formula defined starting from Π; see [3, 30] for details.

The following theorem, due to Marek and Subrahmanian ([31]) for basic programs and generalized in [8] to nested programs, relates the answer sets of a program to the models of its completion. In the following, we say that a set of atoms X *satisfies* (or is a *model* of) a set of formulas Γ if Γ is satisfied by the interpretation which assigns *True* to an atom p if and only if $p \in X$.

Theorem 1. *Let Π be a program. If X is an answer set of Π, then X satisfies the completion of Π.*

Tight Programs. Theorem 1 can be strengthened in the case of tight programs. A program Π is *tight* if its dependency graph is acyclic. The *dependency graph* of a program Π is the directed graph G such that

- the nodes of G are the atoms in Π, and
- for every rule (1) in Π, G has an edge from p_0 to each atom in $\{p_1, \ldots, p_k\}$.

The following Theorem has been proved by Fages ([10]) for basic programs, and it has been generalized by Erdem and Lifschitz ([8]) to nested programs.

Theorem 2. *Let Π be a tight program and X a set of atoms. X is an answer set for Π iff X satisfies the completion of Π.*

Loop Formulas. Theorem 1 states that if X is an answer set of program Π then X satisfies $Comp(\Pi)$. Theorem 2 says that the converse is also true if the program is tight. If the program is non tight, Lin and Zhao ([27, 29]) proved that to have the identity mapping between the answer sets of a basic program Π and the models of its completion, we have to consider the loop formulas of Π. Lee and Lifschitz ([20]) extended the concept of loop formulas to nested programs and proved that the same result holds with the extended definition. The interested reader can reference to the mentioned papers; here we only report the main result

Theorem 3. *Let Π be a program. Let $Comp(\Pi)$ be the completion of Π. Let $LF(\Pi)$ be the set of all the loop formulas associated with the loops of Π. For each set of atoms X, X is an answer set of Π iff X is a model of $Comp(\Pi) \cup LF(\Pi)$.*

3 The Cmodels2 system

As we already said, CMODELS is a system for tight logic programs. In [15] we have presented a procedure, called ASP-SAT, that extend CMODELS in order to work also with non-tight logic programs. ASP-SAT is based on the DLL algorithm: It work on (the clausification of) the completion of a program Π, and check if a model corresponds to an answer set. CMODELS2 is the name that we use for the resulting system; refer to the above mentioned paper for more details.

Cmodels2 implementation. CMODELS2 is implemented on top of the SIMO system [16]. SIMO is a MCHAFF-like SAT solver and thus features unit-propagation based on a two-literal watching data structure, 1-UIP learning and VSIDS heuristics (see [33] for a description of these techniques). However, it does not feature the low level optimizations of MCHAFF, and thus it is on average within a factor of 3 slower than MCHAFF. We have used SIMO because is the system we know better, and this allowed us to a relatively easy integration of the other search strategies and heuristics used for the experimental analysis.

Some modifications have to be made if we want to use a SAT solver as basis for an answer set solver. This is because SIMO (and many other SAT solvers as well) pre-processes the input set of clauses and

1. eliminates tautological clauses (i.e., clauses with both an atom and its negation as disjuncts),
2. assigns *pure literals*, i.e., each atom p is assigned to *True* if $\neg p$ does not belong to any clause in the input formula, and similarly for $\neg p$.

These operations are not harmful in SAT solving. However, if the SAT solver is used —as in our case— as basis for an answer set solver, both operations may lead to incorrect results. In order to avoid undesired behaviors, SIMO preprocessing has been modified in order to keep tautological clauses, and to not assign pure literals. In order to evaluate the impact of different search strategies and heuristics in solving answer set programs, we have enhanced SIMO with search strategies and heuristics other than those implemented by MCHAFF. In particular, we implemented:

- Failed-literal detection: Before branching, for each unassigned atom p, p is assigned to *True* and then unit-propagation is called again: If a contradiction is found, p is said to be a *failed literal*, $\neg p$ can be safely assigned, and unit-propagation is again performed. Otherwise, $\neg p$ is checked following the same procedure implemented for p.
- Standard backtracking: Learning is disabled, and recovery from failure is performed by chronologically backtracking to the latest assigned branching literal.

– The unit heuristic, based on the failed-literal detection technique. Given an unassigned atom p, while doing failed-literal on p we count the number $u(p)$ of unit-propagation caused, and then we select the atom with maximum $1024 \times u(p) \times u(\neg p) + u(p) + u(\neg p)$. The atom is assigned to *True* first.

The above search strategies and heuristics are not novel: They are standard techniques in the SAT field. Indeed, current state-of-the-art SAT solvers can be divided in two main categories:

– "look-ahead" solvers, featuring a rather sophisticated look-ahead based on "failed literal", a simple look-back (essentially backtracking) and a heuristic based on the information gleaned during the look-ahead phase. These solvers are best for dealing with "small but relatively difficult" randomly generated k-cnf formulas. A solver in this category is SATZ [22].
– "look-back" solvers, featuring a simple look-ahead (essentially unit-propagation with 2 literal watching), a rather sophisticated look-back based on "1-UIP learning" and a constant time heuristic based on the information gleaned during the look-back phase. These solvers are best for dealing with "large but relatively easy" instances, typically encoding non random problems. A solver in this category is MCHAFF [33].

[1] By combining SIMO original reasoning strategies with those newly implemented, we can obtain both a MCHAFF-like and a SATZ-like SAT solver, and consequently, a "look-back" answer set solver, and a "look-ahead" answer set solver. Our goal is to confirm the expectations that, also in answer set programming

– on randomly generated problems, look-ahead solvers are best, while
– on large problems, look-back solvers are best

4 Experimental Results

Solvers, benchmarks and setting. In order to evaluate the effectiveness of our approach, we comparatively tested CMODELS2 against other state-of-the-art systems on a variety of benchmarks. The systems we considered are SMODELS version 2.27, SMODELS$_{cc}$ version 1.08, ASSAT version 2.00, DLV release of 2005-02-23.[2] It worths remarking that while SMODELS, SMODELS$_{cc}$, ASSAT and CMODELS2 use LPARSE as preprocessor, and thus can be run on the same input files, DLV does not. This explains why DLV has been run only on a few benchmarks. Analogously, ASSAT can only deal with basic programs and thus it has not been run on some instances. Finally, for DLV we mention that it is a system specifically designed

[1] The terminology "small but relatively difficult" and "large but relatively easy" refer to the number of atoms and are used to convey the basic intuitions about the instances. To get an idea for SAT, in the SAT2003 competition, instances in the random and industrial categories had, on average, 442 and 42703 atoms respectively [19].

[2] See `http://www.tcs.hut.fi/Software/smodels/`, `http://www.nku.edu/~wardj1/Research/smodels_cc.html`, `http://assat.cs.ust.hk/`, `http://www.dbai.tuwien.ac.at/proj/dlv/`

for disjunctive logic programs, and that very different results can be obtained depending on the specific encoding being used.

Considering CMODELS2, we have the possibility to combine different look-ahead/look-back search strategies and heuristics. In order to keep track of which combination we are using, we will refer to a combination of search strategies and heuristics using an acronym where the first, second and third letter denote the look-ahead, look-back and heuristic used, respectively. We considered 4 combination of reasoning strategies

1. ulv: our default answer set solver, incorporating a MCHAFF-like look-back SAT solver, with standard Unit propagation, backtracking enhanced with Learning, and VSIDS heuristic.
2. fbu: a standard SATZ-like look-ahead solver, with unit propagation enhanced with Failed literal detection, standard Backtracking, and the Unit heuristic.
3. flv: an hybrid solver, featuring unit propagation enhanced with Failed literal detection, backtracking enhanced with Learning, and the VSIDS heuristic.
4. flu: an hybrid solver, featuring unit propagation enhanced with Failed literal detection, backtracking enhanced with Learning, and the Unit heuristic.

We considered only these 4 combinations of reasoning strategies and heuristics because, besides of being the most significant, the other possible combinations do not make even sense: VSIDS heuristic requires "learning" in order to be significant, while unit heuristic requires failed-literal. fbu and ulv are the two solvers that we expect to perform best on randomly generated programs and on large programs respectively. Assuming that the expectations are met, the performances of the two hybrid solvers are of interest in order to: (*i*) determine whether adding a powerful look-back (resp. look-ahead) to a look-ahead (resp. look-back) solver leads to better performances on randomly generated (resp. large) programs; and (*ii*) get indications about which combination of reasoning strategy is the most promising on non randomly generated and non large programs.

All the solvers where run in their plain (optimal) configuration unless suggested by the authors. For examples, SMODELS$_{cc}$ has been run with option "-nolookahead" (look-ahead turned off) as explicitly suggested by the authors in the SMODELS$_{cc}$'s home page. For ASSAT, we had to increase its internal limit on the number of atoms in the (grounded) logic program (variable C_MAXATOM).

About the benchmarks, our test-set includes both tight and non tight, both randomly generated and non randomly generated programs. Each benchmark belongs to a class of publicly available programs which have been used before in the literature, or to a class of benchmarks for which a generator is available. In this last case, we may have generated bigger instances than those reported in the literature. In order to validate our expectations, we divide the benchmarks in three categories, being (*i*) randomly generated programs, (*ii*) "large" programs with more than (approximately) 10000 atoms, and (*iii*) other problems not falling in the previous categories. A program is non basic when a program contains choice rules or weight constraints. Recall that choice and weight constraint rules are eliminated with the help of auxiliary atoms and nested rules of the form (1).

The results of the solvers on the most difficult instances of each class is given by means of tables, as it is customary in the answer set literature. In the tables,

(i) the first column is a progressive number; (ii) the second column is the ratio between number of rules and number of atoms for random problems, and the name of the benchmark in case it is a non randomly generated program; (iii) the third column contains the number of atoms (#VAR) after grounding. For non random problems, a "+" to the right of the number indicates that the instance has answer sets; and (iv) the remaining columns are one per solver, and they indicate its performances. For each row, the best result is in bold, and the results within a factor of 2 from the best are underlined.

Finally, all the tests were run on a Pentium IV PC, with 2.8GHz processor, 1024MB RAM, running Linux. For SMODELS, SMODELS$_{cc}$, ASSAT and CMODELS2, the time taken by LPARSE is not counted.[3] Further, each system was stopped after 3600 seconds of CPU time on non random problems, and 600 seconds on random problems, or when it exceeded all the available memory. In the tables, these cases are denoted with "TIME" and "MEM" respectively. Otherwise, the tables report the CPU times in seconds needed by each solver to solve the problem. Some of the results here presented have also been presented in [15, 13, 14]: All the experiments have been relaunched. This justifies the minor differences in the results, especially with [15], where the experiments were conducted on a Pentium IV PC, with 1.8GHz processor, 512MB RAM DDR 266MHz, running Linux.

Randomly generated programs. Table 1 shows the results for "small" programs, randomly generated according to two different methodologies:

1. Problems (1)-(10) are translation of randomly generated k-SAT instances. A k-SAT instance consists of L distinct clauses, where each clause is generated by randomly selecting k different atoms and negating each with probability 0.5. The number of distinct atoms in a k-SAT instance is a priori fixed and denoted with N. Each k-SAT instance F is converted to a program as follows
   - if $C = (l_1 \vee \ldots \vee l_k)$, we define *sat2tlp*($C$) to be the rule $\bot \leftarrow not\ l_1, \ldots, not\ l_k$ where $not\ l_i$ is p if $l_i = \neg p$ and is $not\ p$ if l_i is the atom p;
   - Then, if F is a k-SAT instance, the *translation of* F, is

   $$\cup_{C \in F} sat2tlp(C) \cup \cup_{p \in P} \{p \leftarrow not\ p', p' \leftarrow not\ p\}$$

   where, for each atom $p \in P$, p' is a new atom associated to p. These benchmarks are tight, and have been used in [9, 36, 37].
2. Lines (11)-(20) correspond to programs randomly generated according to the methodology proposed in [28]. Given a set P with N atoms and a positive number k, a randomly generated rule has
   (a) the head which is randomly selected from P, and
   (b) the body consisting of $k-1$ different atoms, each randomly selected from P and negated with probability 0.5.

   A randomly generated program with L rules consists of L randomly generated distinct rules. In general these randomly generated programs are non tight.

[3] Adding the times of LPARSE would not change the picture for DLV when compared to CMODELS2 and other systems.

	PB	#VAR	SMODELS	SMODELS$_{cc}$	ASSAT	DLV	ulv	flv	flu	fbu
1	4	300	1.2	7.23	0.85	2.55	**0.59**	0.8	1.5	1.37
2	4.5	300	**39.97**	TIME	TIME	130.49	TIME	TIME	115.29	40.38
3	5	300	**7.57**	149.37	TIME	26.78	456.22	538.89	17.64	11.32
4	5.5	300	**2.26**	33.12	94.78	7.37	72.83	53.26	4.42	3.59
5	6	300	**1.05**	12.72	22.5	3.26	24.73	21.89	1.83	1.63
6	4	350	4.11	12.6	13.4	49.3	**2.2**	5.74	11.48	8.85
7	4.5	350	**318.1**	TIME	TIME	TIME	TIME	TIME	TIME	384.66
8	5	350	**44.2**	TIME	TIME	147.16	TIME	TIME	134.34	54.07
9	5.5	350	**12.66**	252.11	TIME	32.07	TIME	506.08	20.37	13.61
10	6	350	**3.37**	37.99	174.61	8.76	95.61	104.36	6.05	4.86
11	4	200	3.3	2.02	2.44	32.39	5.34	3.32	1.93	**1.75**
12	4.5	200	6.84	**1.7**	3.28	83.63	6.15	5.82	2.09	1.93
13	5	200	22.8	**2.5**	8.21	82.97	9.82	9.02	3.88	3.33
14	5.5	200	9.42	**1.76**	4.14	39.47	7.5	6.38	2.97	2.85
15	6	200	8.12	**0.85**	1.4	23.93	3.24	2.95	1.25	1.53
16	4	300	298.67	73.64	234.09	TIME	265.43	218.48	41.97	**31.05**
17	4.5	300	TIME	TIME	TIME	TIME	TIME	TIME	190.73	**135.11**
18	5	300	TIME	412.69	TIME	TIME	TIME	TIME	136.67	**99.75**
19	5.5	300	TIME	233.72	TIME	TIME	TIME	TIME	129.29	**78.63**
20	6	300	TIME	191.62	TIME	TIME	TIME	TIME	107.34	**65.83**

Table 1. Performances on randomly generated logic programs. Problems (1)-(10) are tight programs being the translation of 3-SAT benchmarks. Problems (11)-(20) are randomly generated logic programs using Lin and Zhao's methodology.

Both categories of problems have been generated with $k = 3$ and L varying from $0.5 \times N$ to $12 \times N$ with step 0.5. N has been fixed to 300 and 350 for the instances being the translation of k-SAT problems, and to 200 and 300 for the instances generated according to Lin and Zhao's methodology.

For each ratio L/N (indicated in the column "PB"), we generated 10 instances, and the table presents the median results for the most difficult 5 ratios (the other being quite easily solved by all the systems).

On these benchmarks fbu has the overall best performances: it is almost always the fastest system or within a factor of 2 from the fastest. SMODELS is faster than fbu in the median case when considering the translation of k-SAT instances. However, on these benchmarks, SMODELS times out on 2 programs when $N = 300$, while fbu times out only on 1 program.[4] SMODELS' good performances on these benchmarks are not surprising given that also SMODELS implements failed literal detection, together with a heuristic similar to our unit heuristic. However, considering the programs generated according to Lin and Zhao's methodology,

[4] Increasing N to 400 we get the same picture: SMODELS is faster than fbu in the median case, but it times out on 11 programs, while fbu times out on 10. We decided not to show the results for $N = 400$ because most of the other solvers times out also in the median case for most of the ratios L/N.

	PB	#VAR	SMODELS	SMODELS$_{cc}$	ASSAT	DLV	ulv	flv	flu	fbu
21	bw*d9	9956+	6.76	7.63	1.72		**1.02**	5.84	2.69	2.75
22	bw*e9	12260	4.3	4.51	4.22		**0.98**	1.91	1.92	1.93
23	bw*e10	13482+	11.15	12.43	2.66		**1.29**	7.51	5.03	4.95
24	4c1000	14955+	22.28	4.95	0.6		**0.48**	37.86	15.41	15.23
25	4c3000	44961+	202.84	1143.13	**2.19**		8.86	369.27	144.12	142.83
26	4c6000	89951+	856.13	TIME	**14.85**		99.50	TIME	583.55	578.98
27	np60c	10742+	242.61	30.81	84.87	361.80	**2.83**	1611.32	44.12	44.11
28	np70c	14632+	557.08	55.31	520.80	798.96	**4.69**	TIME	97.44	97.87
29	np80c	19122+	1001.88	90.59	53.25	1587.60	**7.2**	TIME	195.08	190.49
30	np90c	24212+	2064.61	144.72	1416.24	2807.84	**10.42**	TIME	364.54	357.92
31	np100c	29902+	3573.19	215.37	TIME	TIME	**14.23**	TIME	610.2	608.96
32	np60c	10683+	7.05	3.82			**3.55**	340.86	8.03	7.82
33	np70c	14563+	15.67	**5.92**			10.54	782.69	15.39	14.92
34	np80c	19043+	32.29	**9.01**			15.05	1538.86	23.63	25.94
35	np90c	24123+	53.21	**14.13**			32.19	2918.82	38.75	50.08
36	np100c	29803+	83.11	**14.95**			34.18	TIME	59.15	62.64
37	mutex4	14698+	14.14	5.35	0.54	367.89	**0.46**	28.29	28.3	28.26
38	mutex3	278074+	163.94	**110.27**	MEM		TIME	TIME	TIME	TIME
39	phi3	16930+	3.23	3.04	53.28		**1.43**	55.62	12.15	TIME

Table 2. Performances on large programs. Problems (21)-(26) are tight. Problems (27)-(39) are non tight.

we see that SMODELS is not competitive with fbu which (together with flu) scales much better than all the rival systems.

Considering CMODELS2's combinations, fbu is the fastest (confirming expectations), but also flu performs quite well. Coupling these facts with the bad performances of flv, it emerges that the unit heuristic is very effective on these benchmarks and makes learning useless.

Large programs. Table 2 shows the results when considering large (i.e., with approximately 10.000 or more atoms) programs. As in the previous subsection, the table is divided in two parts:

1. Programs (21)-(26) are tight: In particular (21)-(23) and (24)-(26) encode respectively blocks world planning and 4-colorability problems in a graph with V vertexes. V is the number in the label "4cV" in column PB. All the tight programs but bw*e9 have answer sets and are available at SMODELS' web site.
2. Programs (27)-(39) are non tight. In particular, we consider Hamiltonian circuit problems on complete graphs, using both the basic encoding of Niemela ([34]) (programs (27)-(31)), and the non basic encoding (programs (32)-(36)) from `http://www.cs.engr.uky.edu/ai/benchmark-suite/ham-cyc.sm`. The remaining 3 programs in the table are related to the problem of checking requirements in a deterministic automaton and are described in [4]. The first of

these 3 programs is the biggest instance in the suite of the "IDFD" problems, while the other two programs belong to the "Morin" suite.

Overall, the picture that emerges is that ulv is the fastest system: Even though SMODELS is the only system that never times out, it is far slower than ulv (and other systems as well) on many problems. The good performances of ulv are particularly remarkable given that the test suite contains Hamiltonian circuit problems, and these benchmarks have exponentially many loops. Thus, one would expect these problems to be difficult for ASSAT, but also for all CMODELS2 versions in the case it will generate and then reject (exponentially) many candidate answer sets. As it can be observed, this is not the case, at least for ulv. On these benchmarks, for ASSAT and ulv, the answer set is found after the test of relatively few candidate models. The gap in performances can be explained with the different treat of loop formulas: ASSAT adds the entire loop formula when a candidate model is not an answer set, and this leads to a considerably higher number of clauses to be managed. Finally, the table also shows an instance on which ASSAT blows up in memory: As a matter of facts, ASSAT exceeds all the available memory also on other instances, here not shown because all the other systems time out on them.

Considering the different CMODELS2 versions —beside the fact that ulv is the best version— by comparing ulv and flv we see that adding failed-literal usually causes a significant degradation in the performances. These results match the expectations. Indeed, ulv (and also ASSAT) uses a MCHAFF-like solver and performs a few operations at each (branching) node: For (very) large programs, even a linear-time (in the number of atoms) operation can be prohibitive if performed at each branching node. Interestingly, considering flv, flu and fbu we see that it is almost always the case that the last system performs better than the second, and that the second is better than the first. On these benchmarks, adding learning to a look-ahead solver does not help. However, the gap between fbu and flu is not big: Adding learning to fbu does not help, but does not hurt too much. We believe that this is due to the lazy data structures used by the CMODELS2 versions, which are fundamental to keep low the burden of managing learned clauses.

Non random, non large programs. Table 3 contains the results on non random, non large logic programs. In more details,[5]

1. Benchmarks (40)-(48) and (73)-(77) are respectively tight and non tight bounded model checking (BMC) problems of asynchronous concurrent systems, as described in [18]. These problems are about proving properties in a given number of steps, represented as the last number in the instance name.
2. Benchmarks (49)-(54) are about the Schur numbers problem, expressed as basic (49)-(51) and non basic (52)-(54) programs respectively. The label "schurX.K-N" refers to a problem where, given a positive integer n, the set of integers N

[5] Benchmarks (40)-(48), (73)-(77) and the generator are available at `http://www.tcs.hut.fi/~kepa/experiments/boundsmodels/`. Benchmarks (49)-(57) are available at the ASPARAGUS web page `http://asparagus.cs.uni-potsdam.de/`. Benchmarks (58)-(60) belong to the SMODELS test suite and are publicly available at `http://www.tcs.hut.fi/Software/smodels/tests/`, encoding by Niemela ([34]).

	PB	#VAR	SMODELS	SMODELS$_{cc}$	ASSAT	DLV	ulv	flv	flu	fbu
40	d*12*i*9	1186	368	435.48			**223.93**	290.15	353.53	TIME
41	k*i*29	3199	990.95	**20.88**			415.54	204.87	44.14	589.45
42	k*s*29	3169	909.46	**16.89**			353.69	1028.77	59.99	TIME
43	m*3*i*10	1933+	10.98	**1.65**			16.23	32.23	26.71	16.55
44	m*4*i*12	3475+	1132.16	**3.82**			1063.15	867.49	TIME	3229.09
45	m*4*s*8	1586+	89.26	**1.3**			17.02	27.59	421.30	327.55
46	q*i*17	2201	517.64	**53.71**			1539.96	505.15	259.05	816.26
47	e*3*i*15	7832+	35.58	77.02			479.28	TIME	7.15	**6.87**
48	e*4*i*13	6447	221.18	56.21			87.63	567.27	20.02	**19.41**
49	schur1.4-43	736+	**0.43**	0.95	0.67	590.57	1.4	2.07	0.82	0.88
50	schur1.4-44	753+	**0.44**	91.25	1.07	TIME	5.97	5.62	92.63	43.01
51	schur1.4-45	770	571.17	1110.68	434.93	TIME	229.04	417.34	244.35	**116.51**
52	schur2.4-43	564+	**0.33**	0.56			1.27	1.04	0.4	0.38
53	schur2.4-44	577+	82.72	47.78			6.14	**2.8**	47.99	18.93
54	schur2.4-45	590	578.73	672.86			226.69	392.78	148.39	**63.2**
55	15puz.18	5945+	17.55	6.94	1.06	141.68	**0.98**	2.9	9.85	9.24
56	15puz.19	6258+	20.94	7.14	3.61	208.41	**1.35**	2.93	11.65	10.76
57	15puz.20	6571	70.27	8.22	4.59	TIME	**1.28**	10.22	64.54	82.68
58	pige.9.10	210	44.77	65.91	**1.1**		1.26	4.33	1259.84	32.06
59	pige.10.11	253	484.63	1029.38	23.83		**12.41**	55.46	TIME	339.06
60	pige.51.50	5252+	106.79	24.29	2.49		**1.63**	221.33	6.85	7.26
61	8 i-1	2329	7.48	7.17	0.86		**0.49**	0.85	0.84	0.81
62	11 i-1	4760	36.18	35.53	3.15		**1.64**	4.92	2.47	2.44
63	8 i	2627+	17.35	9.30	0.98		**0.63**	1.27	0.89	0.88
64	11 i	5301+	37.71	43.90	3.59		**2.16**	15.55	6.07	5.79
65	8 i+1	2925+	12.08	15.17	**1.09**		1.34	4.31	1.34	1.37
66	11 i+1	5842+	54.30	62.39	3.9		**2.49**	24.27	22.01	19.71
67	8 i-1	1897	0.53	0.66			**0.15**	0.29	0.27	0.27
68	11 i-1	3812	1.6	1.96			**0.39**	1.71	0.75	0.7
69	8 i	2132+	0.76	0.8			**0.22**	0.42	0.27	0.3
70	11 i	4233+	1.85	2.57			**0.52**	6.76	1.9	1.88
71	8 i+1	2367+	1.8	1.05			0.68	1.65	**0.47**	0.49
72	11 i+1	4654+	2.5	4.12			**0.6**	10.42	5.26	5.21
73	d*10*i*12	1488+	132.72	**2.25**			488.76	1212.89	152.8	TIME
74	d*10*s*9	1140+	9.75	**3.11**			6.38	19.31	87.64	TIME
75	d*12*s*10	1511+	296.45	**1.1**			53.2	165.9	733.9	TIME
76	d*8*i*10	1003+	1.76	2.42			12.28	25.03	**1.21**	11.86
77	d*8*s*8	819+	0.73	**0.14**			0.47	3.73	2.38	1221.53

Table 3. Performances on non random, non large programs. Benchmarks (40)-(60) are tight, while the others are non tight.

defined as N = $\{1, 2, \ldots n\}$ has to be partitioned into K bins such that each bin is sum-free, i.e., for each Z∈N and Y∈N (*i*) Z and Z+Z are in different bins, and (*ii*) if Z and Y are in the same bin, then Z+Y is in a different bin. We denote with X=1 the basic encoding and with X=2 the non basic encoding.

3. Benchmarks (55)-(57) are programs encoding the 15 puzzle problem. In a label "15puz.M", M denoted the number of moves in which the final configuration has to be reached. The initial configuration is not fixed and varies from program to program.
4. Benchmarks (58)-(60) are tight programs encoding pigeons problems. In a label "pige.*h*.*p*", *h* denotes the number of holes and *p* the number of pigeons.
5. Benchmarks (61)-(72) are blocks world planning problems encoded as basic programs in lines (61)-(66), and as non basic programs in lines (67)-(72)), the formulations due to Erdem ([7]). In the tables, in the column PB the "8" or "11" represents the number of blocks; while an "*i*" (standing for "number of steps") means that the instance corresponds to the problem of finding a plan in "*i*" steps, where "*i*" is the minimum integer for which a plan exists. Thus, the instances with "*i*" and "$i + 1$" in the label admit at least one answer set, while those with "$i - 1$" do not have answer sets. Technically speaking, these programs are non tight. However, these problems are "tight on their completion models" [1]: If Π is one such program, each model of the completion of Π is guaranteed to be also an answer set of Π.

For these benchmarks results are mixed: On BMC problems, SMODELS$_{cc}$ has the best performances overall, while on the other benchmarks it is ulv which has the best performances overall. What is most interesting is that there is no version of CMODELS2 dominating the others on the BMC problems. Given this fact and SMODELS$_{cc}$ good performances on BMC instances, we believe that on non random, non large problems the "overall best" solver is somewhere in between ulv and fbu, i.e., that it can can be obtained by adding a little bit of failed-literal detection to ulv. This can be done is several ways, e.g., by checking if a literal is failed only if it belongs to a pool of "most promising" literals (as, e.g., it is done by SATZ), or by checking all the literals but not at each branching node. All of this is subject of future research.

It is also worth noting that, overall, flu is better than flv: This can be explained by the bad interaction between failed-literal and VSIDS. For non random, large formulas, this phenomena was already showed to hold in SAT [16].

5 Conclusions and future work

The experimental evaluation shows that:

1. CMODELS2 is competitive with other state-of-the-art systems;
2. depending on the type of program different search strategies are best.

This suggests that future development of answer set solvers should be done by focusing on certain classes of problems. In our analysis we identified two classes

of programs that need completely different strategies, i.e., random and large programs. This also implies that benchmarking should be done by considering the application domain which they have been developed for. This reflects what is nowadays a standard in the SAT competition, where there is a track for solvers designed for random problems, and a separate track for solvers designed for large industrial benchmarks. Solvers get designed and specialized for one track, and indeed the top performers in one track behave very badly in the other.

Considering the future, there are several directions in which this work can be improved. First CMODELS2 can be improved as a solver for non disjunctive programs. One way to do this is by improving the SAT solving part.Performances can be improved by implementing better failed literal detection strategies and/or heuristics. Another possibility is to incorporate another SAT solver with the latest advancements, e.g., MINISAT [6] the winner of the last SAT competition.

Another direction of work is to extend CMODELS2 in order to deal with disjunctive logic programming. A preliminary implementation and analysis are encouraging [23], but more work has to be done in order to improve the overall efficiency of the solver.

Acknowledgements

We are grateful to Paolo Ferraris, Nicola Leone, Vladimir Lifschitz and the anonymous reviewers of this paper for their helpful comments and/or discussions on the subject of the paper; to Esra Erdem and Keijo Heljanko for providing us with the benchmarks; and to Francesco Calimeri for his support on DLV. This work is partially supported by MIUR (Italian Ministry of Education, University and Research), and Texas Higher Education Coordinating Board under Grant 003658-0322-2001.

References

1. Babovich, Y., E. Erdem, and V. Lifschitz: 2000, 'Fages' Theorem and Answer Set Programming'. In: *Proc. NMR*.
2. Baral, C. Gelfond, M. and R. Scherl: 2004, 'Using answer set programming to answer complex queries'. In: *Workshop on Pragmatics of Question Answering at HLT-NAAC2004*.
3. Clark, K.: 1978, 'Negation as failure'. In: H. Gallaire and J. Minker (eds.): *Logic and Data Bases*. New York: Plenum Press, pp. 293–322.
4. Ştefănescu, A., J. Esparza, and A. Muscholl: 2003, 'Synthesis of Distributed Algorithms Using Asynchronous Automata'. In: *Proceedings of CONCUR'03*, Vol. 2761. pp. 27–41, Springer.
5. Dowling, W. and J. Gallier: 1984, 'Linear-time algorithms for testing the satisfiability of propositional Horn formulae'. *Journal of Logic Programming* **3**, 267–284.
6. Eén, N. and N. Sörensson: 2003, 'An Extensible SAT-solver'. In: *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*. pp. 502–518.
7. Erdem, E.: 2002, 'Theory and applications of answer set programming'. Ph.D. thesis, University of Texas at Austin.

8. Erdem, E. and V. Lifschitz: 2001, 'Fages' theorem for Programs with Nested Expressions'. In: *Proc. International Conference on Logic Programming*. pp. 242–254.
9. Faber, W., N. Leone, and G. Pfeifer: 2001, 'Experimenting with Heuristics for Answer Set Programming.'. In: *IJCAI*. pp. 635–640.
10. Fages, F.: 1994, 'Consistency of Clark's completion and existence of stable models'. *Journal of Methods of Logic in Computer Science* **1**, 51–60.
11. Gelfond, M. and V. Lifschitz: 1988, 'The stable model semantics for logic programming'. In: R. Kowalski and K. Bowen (eds.): *Logic Programming: Proc. Fifth Int'l Conf. and Symp.* pp. 1070–1080.
12. Gelfond, M. and V. Lifschitz: 1991, 'Classical negation in logic programs and disjunctive databases'. *New Generation Computing* **9**, 365–385.
13. Giunchiglia, E. and M. Maratea: 2005a, 'On the relation between SAT and ASP procedures (or, between smodels and cmodels)'. In: *Proc. of the 21th International Conference on Logic Programming (ICLP)*. pp. 37–51, Springer.
14. Giunchiglia, E. and M. Maratea: 2005b, 'Evaluating Search Strategies and Heuristics for Efficient Answer Set Programming'. In: *Advanced in Artificial Intelligence: Conference of the Italian Association for Artificial Intelligence, AI*IA '05, Milan, Italy, September 20–23, 2005: proceedings*. pp. 37–51, Springer.
15. Giunchiglia, E., M. Maratea, and Y. Lierler: 2004, 'SAT-Based Answer Set Programming'. In: *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*. AAAI Press / The MIT Press.
16. Giunchiglia, E., M. Maratea, and A. Tacchella: 2003, '(In)Effectiveness of Look-Ahead Techniques in a Modern SAT Solver'. In: *9th International Conference on Principles and Practice of Constraint Programming (CP-03)*. pp. 842–846.
17. Giunchiglia, E., M. Maratea, A. Tacchella, and D. Zambonin: 2001, 'Evaluating Search Heuristics and Optimization Techniques in Propositional Satisfiability.'. In: *Automated Reasoning, First International Joint Conference (IJCAR)*, Vol. 2083 of *Lecture Notes in Computer Science*. pp. 347–363, Springer Verlag.
18. Heljanko, K. and I. Niemelä: 2003, 'Bounded LTL Model Checking with Stable Models'. *Theory and Practice of Logic Programming* **3**(4&5), 519–550. Also available as (CoRR: arXiv:cs.LO/0305040).
19. Le Berre, D. and L. Simon: 2003, 'The essentials of the SAT'03 Competition'. In: *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, Vol. 2919 of *LNCS*.
20. Lee, J. and V. Lifschitz: 2003, 'Loop formulas for disjunctive logic programs'. In: *Proc. ICLP-03*.
21. Leone, N., G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello: 2005, 'The DLV System for Knowledge Representation and Reasoning'. *Accepted to ACM Transaction on Computational Logic (ToCL)*.
22. Li, C. M. and Anbulagan: 1997, 'Heuristics Based on Unit Propagation for Satisfiability Problems'. In: *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*. San Francisco, pp. 366–371, Morgan Kaufmann Publishers.
23. Lierler, Y.: 2005, 'Disjunctive Answer Set Programming via Satisfiability.'. In: *Answer Set Programming*, Vol. 142 of *CEUR Workshop Proceedings*.
24. Lierler, Y. and V. Lifschitz: 2003, 'Computing Answer Sets Using Program Completion'. Available at `http://www.cs.utexas.edu/users/tag/cmodels.html`.
25. Lifschitz, V.: 1996, 'Foundations of logic programming'. In: G. Brewka (ed.): *Principles of Knowledge Representation*. CSLI Publications, pp. 69–128.

26. Lifschitz, V., L. R. Tang, and H. Turner: 1999, 'Nested expressions in logic programs'. *Annals of Mathematics and Artificial Intelligence* **25**, 369–389.
27. Lin, F. and Y. Zhao: 2002, 'ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers'. In: *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI-02)*. Menlo Parc, CA, USA, pp. 112–118, AAAI Press.
28. Lin, F. and Y. Zhao: 2003b, 'Answer Set Programming Phase Transition: A study on Randomly Generated Programs'. In: *Proc. ICLP*.
29. Lin, F. and Y. Zhao: 2004, 'ASSAT: computing answer sets of a logic program by SAT solvers.'. *Artificial Intelligence* **157**(1-2), 115–137.
30. Lloyd, J. and R. Topor: 1984, 'Making Prolog more expressive'. *Journal of Logic Programming* **3**, 225–240.
31. Marek, V. and V. Subrahmanian: 1989, 'The Relationship Between Logic Program Semantics and Non-Monotonic Reasoning'. In: G. Levi and M. Martelli (eds.): *Logic Programming: Proc. Sixth Int'l Conf.* pp. 600–617.
32. Marek, V. and M. Truszczynski: 1999, 'Stable models as an alternative programming paradigm'. In: *The Logic Programming Paradigm: a 25.Years perspective*, Lecture Notes in Computer Science. Springer Verlag.
33. Moskewicz, M. W., C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik: 2001, 'Chaff: Engineering an Efficient SAT Solver'. In: *Proceedings of the 38th Design Automation Conference (DAC'01)*. pp. 530–535.
34. Niemelä, I.: 1999, 'Logic programs with stable model semantics as a constraint programming paradigm'. *Annals of Mathematics and Artificial Intelligence* **25**, 241–273.
35. Nogueira, M., M. Balduccini, M. Gelfond, R. Watson, and M. Barry: 2001, 'An A-Prolog decision support system for the space shuttle'. In: *Working Notes of the AAAI Spring Symposium on Answer Set Programming*.
36. Simons, P., I. Niemelä, and S. Timo: 2002, 'Extending and Implementing the Stable Model Semantics'. *Artificial Intelligence* **138**(1–2), 181–234.
37. Ward, J. and J. S. Schlipf: 2004, 'Answer Set Programming with Clause Learning.'. In: *Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, January 6-8, 2004, Proceedings*. pp. 302–313.