An Experimental Study of Search Strategies and Heuristics in Answer Set Programming

Enrico Giunchiglia and Marco Maratea

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

Abstract. Answer Set Programming (ASP) and propositional satisfiability (SAT) are closely related. In some recent work we have shown that, on a wide set of logic programs called "tight", the main search procedures used by ASP and SAT systems are equivalent, i.e., that they explore search trees with the same branching nodes. In this paper, we focus on the experimental evaluation of different search strategies, heuristics and their combinations that have been shown to be effective in the SAT community, in ASP systems. Our results show that, despite the strong link between ASP and SAT, it is not always the case that search strategies, heuristics and/or their combinations that currently dominate in SAT are also bound to dominate in ASP. We provide a detailed experimental evaluation for this phenomenon and we shed light on future development of efficient Answer Set solvers.

1 Introduction

Answer Set Programming [15, 17] (ASP) and propositional satisfiability (SAT) are closely related. If a logic program Π is "tight"[6] there exists a 1 to 1 correspondence between the solutions (Answer Sets) of the logic program (under the answer set semantics [8]) and the propositional formula given by its Clarke's completion [2] $Comp(\Pi)$. In some recent work [9], we have shown that, on the wide set of tight logic programs, the relation goes up to the point that the main search ASP and SAT procedures are equivalent, i.e., that they explore search trees with the same branching nodes, when running on Π and $Comp(\Pi)$ respectively. Given the above result, state-of-the-art ASP systems like SMODELS,[1] CMODELS2[2] and ASSAT[3] are equivalent because they are based on the main search procedures for ASP and SAT: SMODELS is a native procedure working directly on a logic program, while CMODELS2 and ASSAT are based on the Davis-Logemann-Loveland (DLL) procedure.

In this paper we focus on the experimental evaluation of different search strategies, heuristics and their combinations that have been shown to be effective in the SAT community, in ASP systems. The analysis is performed using CMODELS2 as a common platform: CMODELS2 is an AS solver based on SAT, strengthening in this way the

[1] http://www.tcs.hut.fi/Software/smodels
[2] http://www.cs.utexas.edu/users/tag/cmodels.html
[3] http://assat.cs.ust.hk

relationship, and already incorporates most state-of-the-art SAT techniques and heuristics. Given the equivalence on search procedures, the results obtained for CMODELS2 extend to ASSAT and SMODELS if enhanced with corresponding techniques. For the search strategies, we evaluate both look-ahead strategies (used while descending the search tree) and look-back strategies (used for recovering from a failure in the search tree). In particular we analyze

- Look-ahead: basic unit-propagation, based on lazy data structures;
- Look-ahead: unit-propagation+failed-literal detection.
- Look-back: basic backtracking;
- Look-back: backtracking+backjumping+learning.

In SAT, failed-literal detection [7] has been shown to be effective on randomly generated benchmarks, while optimized look-back techniques like backjumping [18] and learning [19, 1] have been shown to be effective on propositional formulas arising from real-world applications (such as planning and model checking). Among the SAT heuristics, we analyze

- Static: based on the order induced by the appearance in the SAT formula.
- VSIDS (Variable State Independent Decaying Sum): based on the information extracted from the optimized look-back phase of the search.
- Unit: based on the information extracted from the failed-literal detection technique.
- Unit with pool: Unit heuristic restricted to a subset of the open (not yet assigned) atoms.

The static heuristic is used for evaluating the contribution of individual look-ahead and look-back strategies *independently* from the heuristic. VSIDS [16] heuristic has been shown to be very effective on real-world benchmarks, while unit and unit with pool heuristics [13] have been shown effective on randomly generated benchmarks.

Finally, we also evaluate several combinations of look-head, look-back strategies and heuristics. There are 16 (2x2x4 for look-ahead, look-back and heuristics respectively) possible combinations of techniques we have presented, but only 10 among them make sense. This is because

- VSIDS heuristic makes sense only if learning is enabled
- Unit-based heuristics make sense only if failed-literal is enabled

The analysis has been performed by means of the following methodology: First, we fixed the heuristic (static) and analyzed the 4 remaining possible combinations (all the combinations between look-ahead and look-back strategies). The goal here is to understand the impact of each single strategy independently from (the interaction with) the heuristics. Second, we added the remaining heuristics where possible. The goal here is to evaluate how "real" ASP solvers (the results of the combinations of look-ahead, look-back strategies and heuristics) perform on different benchmarks. We have used both randomly generated logic programs and logic programs arising from real-world applications. Besides tight logic programs, we have taken into account also non-tight logic programs. They are interesting because most of the state-of-the-art ASP systems,

such as CMODELS2, ASSAT, SMODELS and DLV[4], can also solve non-tight logic programs.

The results of our experimental analysis point out that

1. on "small but relatively hard", randomly generated logic programs, failed-literal detection is very effective, especially in conjunction with unit-based heuristics. This result reflects what happens in SAT.
2. on "big but relatively easy", real-world logic programs of "medium" size (in the number of atoms in the logic programs), learning is very effective. A combination of learning, failed-literal and unit (with pool) heuristic is the best combination on these benchmarks. This is very different to what happens in SAT.
3. on real-world logic programs of large size, e.g. with more than about 15000 atoms, learning is again very effective, but now it leads to the best results in combination with simple unit-propagation and VSIDS heuristic, reflecting the results in the SAT community.

The division in two categories, random and real-world, follow from the literature, in particular from SAT [12]. Here we have introduced a further division in the real-world category, related to the size, i.e. the number of atoms, of the logic programs. This further (sub)division is useful for isolating and underlying different behaviors in the real-world benchmarks.

This is the first paper that we know of, in which a variety of look-ahead, look-ahead strategies and heuristics are evaluated and combined in the ASP community. Previous works (such as [5]) mostly considered and evaluated only one technique (the heuristic in the paper cited). The evaluation of a single technique is often not sufficient, because it is well-known that for the performances of systems what is crucial is the *combination* of techniques: For example, VSIDS heuristic is effective on real-world problems in conjunction with unit-propagation and learning, but becomes ineffective when failed-literal is added and does not make even sense with basic backtracking. Moreover, it is important to remark that a significant analysis ought to be performed on a *unique* platform, otherwise the results can be biased by implementation issues. The same results extend to ASSAT and SMODELS if enhanced with corresponding techniques (at least on tight programs), given the strong link between ASP and SAT procedures outlined in this paper.

2 Answer Set Programming

A *rule* is an expression of the form

$$A_0 \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n \tag{1}$$

where A_0 is an atom or the symbol $\bot$ (standing for false), and $A_1, \ldots, A_n$ are atoms ($0 \leq m \leq n$). A_0 is the *head* of the rule, $A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n$ is the *body*. A *(non disjunctive) logic program* is a finite set of rules.

[4] `http://www.dbai.tuwien.ac.at/proj/dlv`

CMODELS2(Γ, S)
if $\Gamma = \emptyset$ **then return** *test*(S, Π);
if $\emptyset \in \Gamma$ **then return** *False*;
if $\{l\} \in \Gamma$ **then return** CMODELS2$(assign(l, \Gamma), S \cup \{l\})$;
$A :=$ an atom occurring in Γ;
CMODELS2$(assign(A, \Gamma), S \cup \{A\})$;
CMODELS2$(assign(\neg A, \Gamma), S \cup \{\neg A\})$.

Fig. 1. The CMODELS2 procedure

In order to give the definition of an answer set, we consider first the special case in which the program Π does not contain negation as failure (not) (i.e., such that for each rule (1) in Π, $n = m$). Let Π be such a program, and let X be a consistent set of atoms. We say that X is *closed* under Π if, for every rule (1) in Π, $A_0 \in X$ whenever $\{A_1, \ldots, A_m\} \subseteq X$. We say that X is an *answer set* for Π if X is the smallest set closed under Π.

To extend this definition to programs with negation as failure, take any program Π, and let X be a consistent set of atoms. The *reduct* Π^X of Π relative to X is the set of rules

$$A_0 \leftarrow A_1, \ldots, A_m$$

for all rules (1) in Π such that X does not contain any of $A_{m+1}, \ldots, A_n$. Thus Π^X is a program without negation as failure. We say that X is an *answer set* for Π if X is an answer set for Π^X.

Now we want to introduce the relation between the answer sets of a program Π and the models of the completion of Π. In the following, we represent an interpretation (in the sense of propositional logic) as the set of atoms true in it. With this convention, a set of atoms X can denote both an answer set and an interpretation. Consider a program Π.

If A_0 is an atom or the symbol $\bot$, the *completion of* Π *relative to* A_0 is the formula

$$A_0 \equiv \bigvee (A_1 \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n)$$

where the disjunction extends over all rules (1) in Π with head A_0. The *completion* *Comp*(Π) of Π consists of one formula *Comp*(Π, A_0) for each atom A_0 and the symbol $\bot$.

For the wide set of tight logic programs, if X is an answer set of Π, then X satisfies *Comp*(Π), and the converse is also true. In the following, we say that a program Π is *tight* if there exists a function λ from atoms to ordinals such that, for every rule (1) in Π whose head is not $\bot$, $\lambda(A_0) > \lambda(A_i)$ for each $i = 1, \ldots, m$.

2.1 CMODELS2 and ASSAT: SAT-based Answer Set Programming

In this section we review the SAT-based approach to ASP. We present CMODELS2's algorithm, and then we say how it extends to the algorithm of ASSAT. There are various versions of CMODELS, all of them with the same behavior on tight programs. Here we

SMODELS(Π,S)
$\langle \Pi, S\rangle$:= *simplify*(Π, S);
if $(\{l, not\ l\} \subseteq S)$ **return** *False*;
if $(\{A : A \in A_\Pi, \{A, not\ A\} \cap S \neq \emptyset\} = A_\Pi)$ exit with *True*;
A := an atom occurring in Γ;
SMODELS(*A-assign*$(A, \Pi)), S \cup \{A\}$);
SMODELS(*A-assign*$(not\ A, \Pi)), S \cup \{not\ A\}$);

Fig. 2. A recursive version of the algorithm of SMODELS.

consider the one proposed in [10], represented in Figure 1, in which l denotes a literal (a literal is an atom or its negation); Γ a set of clauses (each clause defined as a set of literals); S an *assignment*, i.e., a consistent set of literals. Given an atom A, $assign(A, \Gamma)$ is the set of clauses obtained from Γ by removing the clauses to which A belongs, and by removing $\neg A$ from the other clauses in Γ. $assign(\neg A, \Gamma)$ is defined similarly. In the initial call, $\Gamma =$ *Comp*(Π) and S is the empty set. Here we consider *Comp*(Π) after clausification. We assume that (i) *Comp*(Π) signature extends the signature A_Π of Π, and (ii) for each set X of atoms in *Comp*(Π) signature, X satisfies *Comp*(Π) iff $X \cap A_\Pi$ satisfies *Comp*(Π) before clausification. Standard clausification methods satisfy such conditions.

The algorithm of CMODELS2 is very similar to the well-known DLL decision procedure for SAT [3]. The only difference is in the basic case when $\Gamma = \emptyset$, where "exit with *True*" is substituted with "return *test*(S, Π)", a new function which has to return

– *True*, and exit the procedure, if the set of atoms in S is an answer set of Π, and
– *False*, otherwise.

CMODELS2(*Comp*$(\Pi), \emptyset$) returns *True* if and only if Π has an answer set.

ASSAT has the same behavior on tight logic programs, while on non-tight logic programs the approaches are different. Moreover, CMODELS2 has a number of advantages in comparison with ASSAT. For more details, see [10].

2.2 Relation with SMODELS

Consider now Fig. 2. This is a recursive version of the algorithm of SMODELS. In the Figure, Π is a program, initially set to the program of which we want to determine the existence of answer sets; S is an assignment, initially set to $\{\top\}$; A denotes an atom, r a rule, and l a literal. *A-assign*(l, Π) returns the program obtained from Π by (i) deleting the rules r such that $not\ l \in$ *body*(r); and (ii) deleting l from the body of the other rules in Π.

In the paper [9], has been formally proved that, for tight logic programs, the algorithm of CMODELS2 and SMODELS are equivalent: They explore search trees with the same branching nodes (considering the heuristics return the same atom). What it is interesting to say, is that in the mentioned paper the rules considered in Fig. 2 to extend the assignment S in function $simplify$ are exactly the same used in SMODELS (procedure $expand$, see [20] pagg. 17, 32-37).

3 Experimental analysis

Due to the strong link between the solving procedures of state-of-the-art ASP solvers, the experimental results are independent from the chosen solver (at least on tight programs). We have used our solver, CMODELS2, also because

- its front-end is LPARSE [20], a widely used grounder for logic programs;
- its back-end solver already incorporates lazy data structures for fast unit propagation as well as some state-of-the-art strategies and heuristics evaluated in the paper; and
- can be also run on non-tight programs.

Moreover, it is based on SAT, strengthening in this way the relation between ASP and SAT.

There is no other publicly available AS system having the above features, and that we know of. SMODELS does not contain lazy data structures, and adding them to SMODELS would basically boil down to re-implement the entire solver.

The experimental results we present (at least the ones on tight programs) extend to ASSAT and SMODELS if enhanced with reasoning strategies corresponding to the ones that we considered.

The analysis is focused on tight logic programs, but we also run non-tight logic programs in order to understand (at least on the experimental side) if the results on the tight domain can be extended to the non-tight domain. We considered several domains of publicly available, currently challenges for ASP solvers, benchmarks for our investigation; in particular

- Randomly generated logic programs: The tight programs are (modular) translation from classical random 3SAT instances; the non-tight are randomly generated according to the methodology proposed in [14].
- tight blocks-worlds, queens and 4-coloring problems;[5] tight bounded model checking (BMC) problems.[6]
- non-tight blocks-world problems and non-tight Hamiltonian Circuit on complete graphs.[7]

In the introduction, we already introduced the various strategies and heuristics used in the experimental evaluation. In more details

- "U" (unit-propagation), assigns repeatedly open literals in unit clauses until either (i) there are no more unit clauses, or (ii) a contradiction is found. It is based on two-literal watching, an efficient lazy data structure for propagate unit clauses [16];
- "F" (unit-propagation+failed-literal detection), failed-literal detection is applied if unit-propagation has not reached a contradiction. For each unassigned atom A, A is assigned to *True* and then unit-propagation is called again: If a contradiction is found (and A is said to be a *failed literal*), $\neg A$ can be safely assigned. Otherwise, $\neg A$ is checked. If both branches fail, backtracking occurs;

[5] Publicly available at `http://www.tcs.hut.fi/Software/smodels/tests/.`
[6] Available at `http://www.tcs.hut.fi/~kepa/tools/boundsmodels/.`
[7] Encoding due to Esra Erdem [4] and Ilkka Niemela [17] respectively.

– "B" (basic backtracking), performs chronological backtracking;
– "L" backtracking+backjumping+learning, when a contradiction is found, a clause (called *reason*) is created. The reason is a clause, unsatisfied under the current assignment, that contains only literals "responsible" for the conflict. Instead of just backtrack chronologically, the atoms not in the reason are skipped until we encounter an atom in the reason that was chosen by the heuristic. Reasons are updated during backtracking via resolution with the clauses that caused the atoms to be assigned. The idea here is to avoid the visit of useless parts of the search tree. Learning adds,[8] under given conditions, some of the reasons in order to avoid the repetition of the same mistakes in other parts of the search tree. CMODELS2 implements 1-UIP learning [21].

For the heuristics

– "S" (static), is based on the order induced by the appearance in the SAT formula: The first an atom is in the formula, the sooner is selected;
– "V" (VSIDS), is the acronym for Variable State Independent Decaying Sum. It is based on the information extracted from learning. Each literals has a weight associated with it. The weight is initialized with the occurrences of the literal in the formula and incremented if the literal appears in a learned clause. Periodically the score is divided by a constant (2 in our case). The atom associated to the literal with maximum weight is chosen. The rational here is to put focus on atoms involved in recent conflicts;
– "U" (Unit), is based on the failed-literal detection technique. Given an unassigned atom A, while doing failed-literal on A we count the number $u(A)$ of unit-propagation caused, and then we select the atom with maximum $1024 \times u(A) \times u(\neg A) + u(A) + u(\neg A)$;
– "P" (Unit with pool), unit heuristic restricted to a subset of the unassigned atoms. It is similar to "U" except that (i) we first select a pool of 10 "most watched" atoms, and (ii) we perform failed-literal and score accordingly only with the atoms in the pool. Our simple pooling criteria is motivated by the fact that we are using a solver with lazy data structures. State of the art SAT solvers (e.g. SATZ) that implements failed-literal detection use much more sophisticated criteria. Because of this, results using the pool should be considered significant only if they are positive: Negative results could be biased by the simplicity of our criteria.

The chosen atom is assigned to *True* by default in the "S", "U" and "P" heuristics, while for "V" the value depends on which literal built on the chosen atom has higher weight with ties broken with *False*.
We will refer to the actual combination of search strategies and heuristics using an acronym where the first, second and third letter denote the look-ahead, look-back and heuristic respectively, used in the combination. For example, ulv is a standard look-back, "CHAFF"-like, solver similar to CMODELS2. fbu is a standard look-ahead solver. flv and flu have both a powerful look-ahead and look-back but different heuristic. All

[8] A policy to delete reasons when they became useless is also needed in order to maintain in polynomial space the procedure.

	PB	# VAR	uls	ubs	fls	fbs
1	4	300	TIME	TIME	230.86	338.05
2	5.5	300	TIME	TIME	478.46	TIME
3	6	300	371.28	TIME	120.02	84.16
4	bw-large.d9	9956	0.9	2497.02	2.68	2.62
5	bw-large.e9	12260	1.11	1928.43	1.95	1.9
6	bw-large.e10	13482	1.61	TIME	5.28	19.52
7	queens21	925	0.20	0.23	0.36	0.38
8	queens24	1201	0.46	1.14	0.67	0.74
9	queens50	5101	3.67	TIME	12.41	TIME
10	dp-12.fsa-i-b9	1186	12.51	2651.28	20.30	TIME
11	key-2-i-b29	3199	157.29	TIME	111.61	293.37
12	mmgt-3.fsa-i-b10	1933	TIME	TIME	1570.27	3241.45
13	mmgt-4.fsa-s-b8	1586	1004.36	TIME	1054.06	TIME
14	q-1.fsa-i-b17	2201	165.07	TIME	301.16	TIME
15	p1000	14955	7.69	TIME	377.02	TIME
16	p3000	44961	178.26	TIME	TIME	TIME
17	p6000	89951	1275.62	TIME	TIME	TIME

Table 1. Performances for uls, ubs, fls and fbs on tight programs. Problems (1-3), are randomly generated; (4-6) are blocks-world; (7-9) are queens; (10-14) are bounded model checking; (15-17) are 4-colorability.

the tests were run on a Pentium IV PC, with 2.8GHz processor, 1024MB RAM, running Linux. The timeout has been set to 600 seconds of CPU time for random problems, and 3600 for real-world problems.

We present results using CPU time (remember that we performed the experiments on a unique platform: Our results are not biased by implementation issues). For randomly generated problems the result in the Tables is the median time over 10 runs.

We have considered far more benchmarks for each category than the ones we show. In the Tables are only shown the bigger benchmarks we run for each category when significant (i.e., when at least one of the combinations in each table does not reach the time limit, denoted with TIME). In the Tables, the second column is the ratio between number of rules and number of atoms for random problems, and the name of the benchmarks for real-world problems. The third column contains the number of atoms after grounding.

3.1 Tight logic programs

In Table 1 the results about the CMODELS2's versions with plain heuristic "S" on tight programs are shown. Randomly generated tight programs are modular translation of classical random 3SAT benchmarks with 300 atoms, 10 instances per point. Here we have not taken into account ratios 4.5 and 5 because all the medians are in timeout. For these problems (1-3), we immediately see that failed-literal is very effective, being

	PB	# VAR	ulv	flv	flu	fbu	ulp	ubp
18	4	300	0.41	0.52	0.85	0.66	21.79	3.01
19	4.5	300	TIME	TIME	81.92	22.53	TIME	54.7
20	5	300	448.21	485.36	8.27	4.72	452.75	14.35
21	bw-large.d9	9956	1.02	5.84	2.69	2.75	1.01	TIME
22	bw-large.e9	12260	0.98	1.91	1.92	1.93	1.03	1.54
23	bw-large.e10	13482	1.29	7.51	5.03	4.95	1.55	TIME
24	queens21	925	786.14	1864.49	384.87	47.33	0.24	0.24
25	queens24	1201	TIME	TIME	TIME	368.76	0.28	0.29
26	queens50	5101	TIME	TIME	TIME	TIME	347.98	43.16
27	dp-12.fsa-i-b9	1186	223.93	383.66	353.53	TIME	2910.96	1051.17
28	key-2-i-b29	3199	415.54	204.87	44.14	589.45	1329.53	TIME
29	mmgt-3.fsa-i-b10	1933	16.23	32.23	26.71	16.55	6.19	372.54
30	mmgt-4.fsa-s-b8	1586	17.02	27.59	421.30	327.55	13.79	2492.62
31	q-1.fsa-i-b17	2201	1539.96	505.15	259.05	816.26	TIME	TIME
32	p1000	14955	0.48	37.86	15.41	15.23	3.69	TIME
33	p3000	44961	8.86	369.27	144.12	142.83	223.62	TIME
34	p6000	89951	99.50	TIME	583.55	578.98	2549.50	TIME

Table 2. Performances for ulv, flv, flu, fbu, ulp and ubp on tight programs. The problems presented are the same as in Table 1.

much faster than the versions using only unit-propagation. fls is slightly better than fbs because, with a static heuristic, "L" can help to escape from unsatisfied portion of the search tree where the uninformed static heuristic could be trapped. These results are in accordance to those from the SAT community. Lines (4-17) show the results for real-world problems. From the comparison between the 4th and 5th columns, and the 6th and 7th columns, we can conclude that "L" is of fundamental importance on real-world problems, being often faster by orders of magnitude w.r.t. the same combination but using "B". Also this result reflects what happens in the SAT community. The effects of adding failed-literal follow from the comparison between 4th and 6th, and 5th and 7th columns. When "L" is enabled, adding failed-literal does not help (except for two BMC problems) in improving the performances. This phenomenon was already partly encountered in the SAT community in [11]. Otherwise, when using simple backtracking failed-literal helps in general in improving performances, avoiding (with a forward reasoning) the visit of useless parts of the search tree that otherwise (due to the absence of "L") the solver would explore (the results are confirmed by some smaller experiments on 4-coloring problems not shown here).

In Table 2, there are the results on tight programs when using CMODELS2 with a non-static heuristic. For randomly generated logic programs (18-20), using a non-static heuristic in general helps for increasing the performances. It is also clear that using failed-literal in combination with a look-head based heuristic is the best choice. In particular fbu is the best, but flu and ubp are not far. Here, using "L" is not effective in conjunction with failed-literal: The positive effects it had with the static heuristic

are shadowed by the unit-based heuristics. Even more, now it leads to negative results, with a huge difference when using heuristic "P". Rows (21-34) in Table 2 are results for tight real-world logic programs. The situation here is far less similar w.r.t. the SAT case. Indeed, combinations based on look-back (in particular ulv) perform quite well on a wide variety of benchmarks, but not as well as one would expect. In particular, the performances on the BMC instances (problems (27-31)) of ulv (resp. the version using failed-literal) are worse (resp. better) than expected: In SAT, BMC instances are the benchmarks where look-back solvers (resp. solvers with powerful look-ahead) give their best (resp. their worst). ulp is often very competitive with ulv. This is indeed explained if we look at the number of variables "# VAR" of these instances, which is in the order of a few thousands. Indeed, for such "# VAR" it still makes sense to perform an aggressive look-ahead at each branching node. On the other hand, as "# VAR" increases this is no longer the case, as the results on the 4 colorability instances (lines 32-34) show.

Summing up about the experimental analysis for tight programs

- on randomly generated logic programs, failed-literal is very effective, especially in conjunction with unit-based heuristics;
- on real-world logic programs, learning is usually very effective;
- on real-world logic programs of "medium" size, a combination of powerful look-ahead and powerful look-back like ulp is very competitive even if not the overall most effective alternative;
- on real-world logic programs of "large" size, e.g. with more than 15000 atoms, a look-back based solver like ulv becomes the most effective combination.

3.2 Non-tight logic programs

Besides the analysis on tight logic programs, we also analyzed non-tight logic programs. The analysis is motivated by trying to understand if the results obtained on the tight domain can be extended to the non-tight domain (at least from the experimental point of view).

In Table 3 and 4 the results for non-tight logic programs using plain heuristic "s" and non-static heuristics respectively. Problems (35-37) and (44-46) are randomly generated logic programs with 300 atoms, 10 instances per point. The instances were generated using the method proposed in [14]. The ratios from 3.5 to 7 have not been shown in Table 3 because all the medians are in timeout. Lines (38-43) and (47-52) contain the results for blocks-world and complete graphs problems.

Summing up, on the non-tight domain

- results obtained in the tight domain extend to the non-tight for CMODELS2; but
- it can be the case that results on non-tight benchmarks do not extend to other solvers given that the strong theoretical link has been established on tight programs.

4 Conclusions

In this paper, motivated by the strong existing link between ASP and SAT, we have investigated several search strategies, heuristics and their combinations that have been

	PB	# VAR	uls	ubs	fls	fbs
35	3	300	9.75	31.63	4.69	4.4
36	7.5	300	TIME	TIME	TIME	567.78
37	8	300	544.83	TIME	199.05	178.98
38	bw-basic-P4-i	5301	2.08	43.19	4.07	6.91
39	bw-basic-P4-i-1	4760	1.73	15.55	2.54	2.57
40	bw-basic-P4-i+1	5842	2.29	47.09	5.04	8.17
41	np60c	10742	6.8	TIME	125.83	TIME
42	np70c	14632	12.34	TIME	326.34	TIME
43	np80c	19122	19.89	TIME	745.26	TIME

Table 3. Performances for uls, ubs, fls and fbs on non-tight programs. Problems (35-37), are randomly generated; (38-40) are blocks-world; (41-43) are Hamiltonian Circuit on complete graphs.

	PB	# VAR	ulv	flv	flu	fbu	ulp	ubp
44	4	300	265.43	218.48	41.97	31.05	77.41	123.31
45	5	300	TIME	TIME	136.67	99.75	439.71	323.15
46	6	300	TIME	TIME	107.34	65.83	591.3	337.45
47	bw-basic-P4-i	5301	2.16	15.54	6.07	5.79	2.54	79.64
48	bw-basic-P4-i-1	4760	1.64	4.92	2.47	2.44	1.86	13.44
49	bw-basic-P4-i+1	5842	2.49	24.27	22.01	19.71	2.41	11.60
50	np60c	10742	2.83	1611.32	44.12	44.12	4.77	597.82
51	np70c	14632	4.69	TIME	97.44	97.89	5.91	TIME
52	np80c	19122	6.91	TIME	192.29	196.32	12.88	TIME

Table 4. Performances for ulv, flv, flu, fbu, ulp and ubp on non-tight programs. The problems presented are the same as in Table 3.

shown to be effective for SAT, in ASP systems. Our results have shown that on randomly generated problems look-ahead solvers dominate, like in SAT, while on logic programs arising from real-world applications, despite the strong link, a combination of powerful look-ahead and look-back is currently what dominate in ASP systems.

Given the relatively low number of variables in most currently challenging instances in ASP, we believe that if the goal is to develop a general purpose ASP solver, an ulp-based solver is, at the moment, the way to go. We have also shed light on future development: As soon as the number of variables in the challenges benchmarks will increase, for real-world problems we expect that ulv-based solvers, leaders in the SAT community, become leaders also in ASP.

As a future work, we are planning to investigate the interplay between grounding techniques and the SAT strategies and heuristics presented in this paper.

Finally, we believe that this paper is a major step in the direction of closing the gap between ASP and SAT.

References

1. Roberto J. Bayardo, Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of AAAI-97*, pages 203–208, Menlo Park, July 27–31 1997. AAAI Press.
2. Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.
3. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.
4. Erdem Esra. *Theory and applications of answer set programming*. PhD thesis, University of Texas at Austin, 2002. PhD thesis.
5. W. Faber, N. Leone, and G. Pfeifer. Experimenting with heuristics for asp. In *Proc. IJCAI*, 2001.
6. François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.
7. Jon W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, 1995.
8. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Logic Programming: Proc. Fifth Int'l Conf. and Symp.*, pages 1070–1080, 1988.
9. E. Giunchiglia and M. Maratea. On the relation between sat and asp procedures. Submitted to ICLP 2005, 2005.
10. E. Giunchiglia, M. Maratea, and Y. Lierler. SAT-based answer set programming. In *American Association for Artificial Intelligence*, 2004.
11. E. Giunchiglia, M. Maratea, and A. Tacchella. (In)Effectiveness of look-ahead techniques in a modern SAT solver. In *9th International Conference on Principles and Practice of Constraint Programming (CP-03)*, pages 842–846, 2003.
12. D. LeBerre and L. Simon. Fifty-five solvers in vancouver: The sat 2004 competition. In *8th International Conference on Theory an Applications of Satisfiability Testing. Selected Revised Papers.*, Lecture Notes in Computer Science. Springer Verlag, 2005. To appear.
13. Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 366–371, San Francisco, August 23–29 1997. Morgan Kaufmann Publishers.
14. F. Lin and Y. Zhao. Asp phase transition: A study on randomly generated programs. In *Proc. ICLP-03*, 2003.
15. Victor Marek and Miroslaw Truszczynski. Stable models as an alternative programming paradigm. In *The Logic Programming Paradigm: a 25.Years perspective*, Lecture Notes in Computer Science. Springer Verlag, 1999.
16. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.
17. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.
18. Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.
19. João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. Technical report, University of Michigan, 1996.
20. Patrick Simons. *Extending and implementing the stable model semantics*. PhD thesis, Helsinky University, 2000. PhD thesis.
21. L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, 2001.