On the relation between Answer Set and SAT procedures (or, between CMODELS and SMODELS)

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) is a declarative paradigm for solving search problems. State-of-the-art systems for ASP include SMODELS, DLV, CMODELS, and ASSAT.
In this paper, our goal is to study the computational properties of such systems both from a theoretical and an experimental point of view. From the theoretical point of view, we start our analysis with CMODELS and SMODELS. We show that though these two systems are apparently different, they are equivalent on a significant class of programs, called tight. By equivalent, we mean that they explore search trees with the same branching nodes, (assuming, of course, a same branching heuristic). Given our result and that the CMODELS search engine is based on the Davis Logemann Loveland procedure (DLL) for propositional satisfiability (SAT), we are able to establish that many of the properties holding for DLL also hold for CMODELS and thus for SMODELS. On the other hand, we also show that there exist classes of non-tight programs which are exponentially hard for CMODELS, but "easy" for SMODELS. We also discuss how our results extend to other systems.
From the experimental point of view, we analyze which combinations of reasoning strategies work best on which problems. In particular, we extended CMODELS in order to obtain a unique platform with a variety of reasoning strategies, and conducted an extensive experimental analysis on "small" randomly generated and on "large" non randomly generated programs. Considering these programs, our results show that the reasoning strategies that work best on the small problems are completely different from the ones that are best on the large ones. These results point out, e.g., that we can hardly expect to develop one solver with the best performances on all the categories of problems. As a consequence, (i) developers should focus on specific classes of benchmarks, and (ii) benchmarking should take into account whether solvers have been designed for specific classes of programs.

1 Introduction

Answer Set Programming (ASP) is a declarative paradigm for solving search problems. State-of-the-art systems for ASP include SMODELS, DLV, CMODELS, and ASSAT.[1] Our

[1] See `http://www.tcs.hut.fi/Software/smodels`, `http://www.dbai.tuwien.ac.at/proj/dlv`, `http://assat.cs.ust.hk`, `http://www.cs.utexas.edu/users/tag/cmodels.html`, respectively.

goal is to study the computational properties of such systems both from a theoretical and an experimental point of view.

From the theoretical point of view, we start our analysis with CMODELS and SMODELS. Given a program Π, while SMODELS (and also DLV) is a native procedure which directly operate on Π, CMODELS (and ASSAT) computes a set of clauses corresponding to the Clark's completion of Π, and then invoke a propositional satisfiability (SAT) solver based on Davis Logemann Loveland procedure (DLL). We show that though CMODELS and SMODELS are apparently different, they are equivalent on a significant class of programs, called tight. By equivalent, we mean that they explore search trees with the same branching nodes, (assuming, of course, a same branching heuristic). Given our equivalence result and that CMODELS search engine is based on DLL, we are able to establish that many of the properties holding for DLL also hold for CMODELS and thus, when considering tight programs, also for SMODELS. For instance we show that:

1. There exist classes of tight formulas which are exponentially hard both for CMODELS and SMODELS.
2. There exist classes of non tight programs which are exponentially hard for CMODELS but very easy (i.e., solved without search) by SMODELS.
3. In SMODELS, deciding the "best" literal to branch on, is both NP-hard and co-NP hard and in PSPACE for tight programs.

These are just a few of the many results (i) that are already known for DLL, (ii) that can be easily shown to hold for CMODELS, and (iii) that –thanks to our equivalence result– can be easily shown to hold also for SMODELS.

From the experimental point of view, we analyze which combinations of reasoning strategies work best on which problems. In particular,

– we extended CMODELS in order to obtain a unique platform with various "look-ahead" strategies (used while descending the search tree); "look-back" strategies (used for recovering from a failure in the search tree); and "heuristic" (used for selecting the next literal to branch on), and
– we considered various combinations of strategies, and conducted an extensive experimental analysis, on a wide variety of tight and non tight programs.

Our experimental results show that:

1. On "small" (i.e., with a few hundreds variables), randomly generated problems, 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) are best.
2. On "large" (i.e., with tens of thousands variables) problems,"look-back" solvers (featuring a simple but efficient look-ahead –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 [1], our comparison is “fair” because all the reasoning strategies are realized on a common platform (thus, our experimental evaluation is not biased by the differences due to the quality of the implementation) and is “significant” because our solver implements current state-of-the-art look-ahead/look-back strategies and heuristics.

As discussed in more details in the conclusions, our experimental results have some 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 one solver with the best performances on all the categories of problems. As a consequence, (i) developers should focus on specific classes of benchmarks (e.g., on randomly generated programs), and (ii) 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, real-world) programs on large, real-world (resp. random) programs.

The paper is structured as follows. In Section 2 we give the basic definitions. Sections 3 and 4 are devoted to the definition of the algorithms of CMODELS and SMODELS respectively, and that are used in our formal analysis of their computational properties (done in Section 5). Section 6 is dedicated to the experimental analysis of different look-ahead/look-back strategies and heuristics. We end the paper in Section 7 with the conclusions.

2 Basic definitions

Let P be a set of atoms. If p is a an atom, $\overline{p}$ is the *negation* of p, and $\overline{\overline{p}}$ is p. We will also use the logical symbols $\bot$ and $\top$ (standing for FALSE and TRUE respectively), and assume that $\overline{\bot} = \top$ and $\overline{\top} = \bot$. Atoms, their negations, and the symbols $\bot$, $\top$ form the set of *literals*. If S is a set of literals, we define $\overline{S} = \{\bar{l} : l \in S\}$.

A *rule* is an expression of the form

$$p_0 \leftarrow p_1, \dots, p_m, \overline{p}_{m+1}, \dots, \overline{p}_n \tag{1}$$

where $p_0 \in P \cup \{\bot\}$, and $\{p_1, \dots, p_n\} \subseteq P$ ($0 \leq m \leq n$). If r is a rule (1), *head*$(r) = p_0$ is the *head* of r, and *body*$(r) = \{p_1, \dots, p_m, \overline{p}_{m+1}, \dots, \overline{p}_n\}$ is the *body* of r. A *(logic) program* is a finite set of rules.

Consider a program Π, and let X be a set of atoms. In order to give the definition of an answer set we consider first the special case in which the body of each rule in Π contains only atoms (i.e., for each rule (1) in Π, $m = n$). Under these assumptions, we say that

- X is *closed* under Π if for every rule (1) in Π, $p_0 \in X$ whenever $\{p_1, \dots, p_m\} \subseteq X$, and that
- X is an *answer set* for Π if X is the smallest set closed under Π.

Now we consider the case in which Π is an arbitrary program. The *reduct* Π^X of Π relative to X is the set of rules

$$p_0 \leftarrow p_1, \dots, p_m$$

for all rules (1) in Π such that $X \cap \{p_{m+1}, \ldots, p_n\} = \emptyset$. X is an *answer set* for Π if X is an answer set for Π^X.

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 $\bot$, $\lambda(p_0) > \lambda(p_i)$ for each $i = 1, \ldots, m$.

3 CMODELS

```
function CMODELS($\Pi$) return DLL-REC(lp2sat($\Pi$),$\emptyset$,$\Pi$);

function DLL-REC($\Gamma$,$S$,$\Pi$)
 1 $\langle \Gamma, S \rangle$ := unit-propagate($\Gamma, S$);
 2 if ($\emptyset \in \Gamma$) return FALSE;
 3 if ($\Gamma = \emptyset$) return test($S$,$\Pi$);
 4 $l$ := ChooseLiteral($S$);
 5 return DLL-REC(s-assign($l, \Gamma$)), $S \cup \{l\}, \Pi$) or
 6          DLL-REC(s-assign($\overline{l}, \Gamma$)), $S \cup \{\overline{l}\}, \Pi$);

function unit-propagate($\Gamma$,$S$)
 7 if ($\{l\} \in \Gamma$) return unit-propagate(s-assign($l, \Gamma$), $S \cup \{l\}$);
 8 return $\langle \Gamma, S \rangle$;
```

Fig. 1. The algorithm of CMODELS.

CMODELS reduces the problem of answer set computation to the satisfiability problem of propositional formulas via Clark's completion, and uses a SAT solver as search engine. Formally, a *clause* is a finite set of literals different from $\bot, \top$, and a *(propositional) formula* is a finite set of clauses. An *assignment* is a set of literals. An assignment S *satisfies* a formula Γ if S is consistent and for each clause C in Γ, $C \cap S \neq \emptyset$. If S satisfies Γ then we also say that S is a *model* of Γ and that Γ is *satisfiable*.

There are various versions of CMODELS (see the web page of CMODELS). Here we consider the one proposed in [2] (called ASP-SAT in that paper), and it is represented in Figure 1, in which

- Π is the input program; Γ is a set of clauses; S is an assignment; p and l are an atom and a literal respectively.
- *lp2sat*(Π) is the set of clauses –corresponding to the Clark's completion of Π– formally defined below.
- *s-assign*(l, Γ) returns the formula obtained from Γ by (i) deleting the clauses $C \in \Gamma$ with $l \in C$, and (ii) deleting $\overline{l}$ from the other clauses in Γ.
- *test*(S, Π) returns TRUE if $S \cap P$ is an answer set of Π, and FALSE otherwise.
- *ChooseLiteral*(S) returns a literal not assigned by S. We say that a *literal l is assigned by an assignment S* if $\{l, \overline{l}\} \cap S \neq \emptyset$. For simplicity, we assume that *ChooseLiteral*(S) returns the first –according to a fixed total order ρ on $P \cup \overline{P}$– literal in $P \cup \overline{P}$ which is unassigned by S.

We assume that parameters are passed to a procedure by value, as in [3].

CMODELS(Π) simply invokes DLL-REC(*lp2sat*(Π),$\emptyset$,Π). It is easy to see that DLL-REC(Γ,S,Π) is a variation of the standard DLL procedure. In particular, at line 3, instead of just returning TRUE as in the standard DLL (meaning that the input set of clauses is satisfiable), it invokes *test*(S, Π) (see [2] for more details): such a modification is needed only if the input program Π is non tight. Indeed, if Π is tight we are guaranteed that any model of *lp2sat*(Π) corresponds to an answer set of Π [4], and thus SAT solvers can be used as black-box (as it is the case for some versions of CMODELS).

In order to precisely define *lp2sat*(Π) we need the following definitions. If p_0 is an atom, the *translation of Π relative to* p_0, denoted with *lp2sat*(Π, p_0), consists of

1. for each rule $r \in \Pi$ of the form (1) and whose head is p_0, the clauses:
$$\{p_0, \overline{n}_r\},$$
$$\{n_r, \overline{p}_1, \dots, \overline{p}_m, p_{m+1}, \dots, p_n\},$$
$$\{\overline{n}_r, p_1\}, \dots, \{\overline{n}_r, p_m\}, \{\overline{n}_r, \overline{p}_{m+1}\}, \dots, \{\overline{n}_r, \overline{p}_n\},$$
where n_r is a newly introduced atom, and
2. the clause $\{\overline{p}_0, n_{r_1}, \dots, n_{r_q}\}$ where $n_{r_1}, \dots, n_{r_q}$ $(q \geq 0)$ are the new symbols introduced in the previous step.

The *translation of Π relative to* $\bot$, denoted with *lp2sat*($\Pi, \bot$), consists of a clause
$$\{\overline{p}_1, \dots, \overline{p}_m, p_{m+1}, \dots, p_n\},$$
one for each rule in Π of the form (1) with head $\bot$. Finally, the *translation of Π*, denoted with *lp2sat*(Π), is $\cup_{p \in P \cup \{\bot\}}$*lp2sat*($\Pi, p$).

Proposition 1. *Let* CMODELS *be the procedure in Figure 1. For each program Π,* CMODELS(Π) *returns* TRUE *if Π has an answer set, and* FALSE *otherwise.*

A few remarks are in order:

1. As we said, there are various versions of CMODELS. However, if the input program Π is tight, all the versions are equivalent at the algorithmic level. In other words, the presentation of CMODELS in Figure 1 can be considered as representative for all the various versions of CMODELS, in the case of tight programs.
2. Figure 1 is indeed a simple presentation of CMODELS. CMODELS incorporates, e.g., a pre-processing for the simplification of the input program. Analogously, DLL-REC is based on the standard simple recursive presentation of DLL: actual SAT solvers (including the ones used by CMODELS) feature far more sophisticated look-ahead/look-back strategies and heuristics.
3. Given a program Π, its translation *lp2sat*(Π) to SAT is exactly the one used by CMODELS (see [5]).

Considering other ASP systems, ASSAT also computes a set Γ of clauses corresponding to the Clark's completion of the input program Π, and then invokes a SAT solver on Γ. Assuming that Γ is computed as *lp2sat*(Π), ASSAT and CMODELS have different behavior only if Π is non tight.[2].

[2] Unfortunately, for ASSAT the way a program Π is converted into a set of clauses is not specified (see [6])

```
function SMODELS($\Pi$) return SMODELS-REC($\Pi$, $\{\top\}$);

function SMODELS-REC($\Pi$,$S$)
 1 $\langle \Pi, S \rangle$ := $expand(\Pi, S)$;
 2 if ($\{l, \bar{l}\} \subseteq S$) return FALSE;
 3 if ($\{p : p \in P, \{p, \overline{p}\} \cap S \neq \emptyset\} = P$) return TRUE;
 4 $p$ := $ChooseLiteral(S)$;
 5 return SMODELS-REC($p\text{-}elim(p, \Pi)), S \cup \{p\}$) or
 6          SMODELS-REC($p\text{-}elim(\overline{p}, \Pi)), S \cup \{\overline{p}\}$);

function $expand$($\Pi$,$S$)
 7 $S'$ := $S$;
 8 $S$ := $AtLeast(\Pi, S)$;
 9 $\Pi$ := $p\text{-}elim(S, \Pi)$;
10 $S$ := $S \cup \{\overline{p} : p \in P, p \notin AtMost(\Pi^{\emptyset}, S)\}$;
11 $\Pi$ := $p\text{-}elim(S, \Pi)$;
12 if ($S \neq S'$) return $expand$($\Pi$,$S$);
13 return $\langle \Pi, S \rangle$;

function $AtLeast$($\Pi$,$S$)
14 if ($r \in \Pi$ and $body(r) \subseteq S$ and $head(r) \notin S$)
       return $AtLeast(p\text{-}elim(head(r), \Pi), S \cup \{head(r)\})$;
15 if ($\{p, \overline{p}\} \cap S = \emptyset$ and $\nexists r \in \Pi : head(r) = p$)
       return $AtLeast(p\text{-}elim(\overline{p}, \Pi), S \cup \{\overline{p}\})$;
16 if ($r \in \Pi$ and $head(r) \in S$ and $body(r) \not\subseteq S$ and $\nexists r' \in \Pi, r' \neq r : head(r') = head(r)$)
       return $AtLeast(p\text{-}elim(body(r), \Pi), S \cup body(r))$;
17 if ($r \in \Pi$ and $\overline{head(r)} \in S$ and $body(r) \setminus S = \{l\}$)
       return $AtLeast(p\text{-}elim(\bar{l}, \Pi)), S \cup \{\bar{l}\})$;
18 return $S$;

function $AtMost$($\Pi$,$S$)
19 if ($r \in \Pi$ and $body(r) \subseteq S$ and $head(r) \notin S$)
       return $AtMost(\Pi, S \cup \{head(r)\})$;
20 return $S$;
```

Fig. 2. The algorithm of SMODELS.

Given a program Π, SMODELS searches for answer sets by extending an assignment S till either S becomes inconsistent (in which case backtracking occurs) or each atom is assigned by S (in which case $S \cap P$ is an answer set). A simple, recursive presentation of SMODELS is given in Figure 2, where

– Π is a program; S is an assignment; p is an atom; r is a rule; and l is a literal.

– *p-elim*(S, Π) returns the program obtained from Π by eliminating the rules $r \in \Pi$ such that for some literal $l \in S$, $\bar{l} \in$ *body*(r). For simplicity, when S is a singleton $\{l\}$, we write *p-elim*(l, Π) for *p-elim*$(\{l\}, \Pi)$.
– *ChooseLiteral*(S) is the same function used by CMODELS at line 4 in Figure 1. Thus, our presentation of CMODELS and SMODELS incorporates the assumption that the two systems use the same heuristic.

The computation of SMODELS-REC(Π, S) proceeds as follows (in the following, we say that a set of atoms X *extends an assignment* S if $S \cap P \subseteq X$ and $\overline{S} \cap X = \emptyset$):

– *Line 1:* The program Π is simplified and the assignment S is extended by the routine *expand*(Π, S), explained below.
– *Line 2:* if S is inconsistent, no answer set extending S exists, and FALSE is returned,
– *Line 3:* if each atom $p \in P$ is assigned, then (i) $S \cap P$ is an answer set of the initial program, and (ii) TRUE is returned.
– *Lines 4-6:* if none of the above applies, an atom p is selected (line 4), an answer set extending $S \cup \{p\}$ (line 5) or $S \cup \{\overline{p}\}$ (line 6) is searched.

expand(Π, S) extends the assignment S generated so far by recursively invoking *AtLeast* (line 8) and then *AtMost* (line 10) till it is no longer possible to extend S (lines 12- 13). *AtLeast* encodes the following facts:

– *Line 14:* if there exists a rule r whose body is a subset of S, then every answer set extending S includes the head of r.
– *Line 15:* if an unassigned atom p is not the head of any rule, then every answer set extending S does not include p.
– *Line 16:* if there is only one rule with head p, and $p \in S$, then each answer set extending S, also extends $S \cup$ *body*(r).
– *Line 17:* if there is a rule with head p and whose body contains only one literal l which is not in S, then if $\overline{p}$ is in S, then every answer set extending S also extends $S \cup \{\bar{l}\}$.

When no further simplification is possible, (i) the set S is returned by *AtLeast*(Π, S) (line 18); (ii) the program Π is simplified accordingly (line 9); and (iii) *AtMost* is invoked with $\Pi^{\emptyset}$ –the reduct of Π relative to the empty set– and S as arguments (line 10). *AtMost* incrementally adds to (the local copy of) S the heads of the rules in $\Pi^{\emptyset}$ whose body is a subset of S (line 19). If S' is the set returned by *AtMost*$(\Pi^{\emptyset}, S)$ (i.e., if S' is the set returned at line 20), if an atom p does not belong to S' then $\overline{p}$ can be safely added to the current assignment S (line 10) (see [7] for more details). To get an intuition of why this is the case, assume for simplicity that the head of each rule in Π is not $\perp$:

1. $\Pi^{\emptyset}$ has a unique answer set, and
2. any answer set of Π which extends S has to be a subset of S union the answer set of $\Pi^{\emptyset}$.

Proposition 2. *Let* SMODELS *be the procedure in Figure 2. For each program Π,* SMODELS(Π) *returns* TRUE *if Π has an answer set, and* FALSE *otherwise.*

The above presentation of SMODELS is a recursive reformulation of the description of SMODELS provided in [7], pag. 17. As for CMODELS, the actual implementation of SMODELS features more complex look-ahead/look-back strategies and heuristic. SMODELS has been extended with clause learning in [8], and SMODELS-CC is the name given to the resulting system.

5 Relating CMODELS and SMODELS

Consider a program Π. Our goal is to prove that the computations of CMODELS and SMODELS are tightly related if Π is tight, and that this is not necessarily the case otherwise. To this end, we will compare the search trees of CMODELS and SMODELS on Π, i.e., the search trees of SMODELS-REC$(\Pi, \{\top\})$ and DLL-REC$(\mathit{lp2sat}(\Pi), \emptyset, \Pi)$ respectively. In doing this, the first problem is that the translation *lp2sat* introduces additional atoms not in P. In the following we assume that both SMODELS-REC and DLL-REC operate in the signature of the input program and formula respectively. However, we still assume that *ChooseLiteral*(S) returns the first literal in $P \cup \overline{P}$ which is unassigned by S: notice that once all the atoms in P are assigned, also the atoms introduced by *lp2sat* will be assigned by *unit-propagate* in DLL-REC.

Given this, one possibility for achieving our goal would be to consider the search trees corresponding to the assignments generated by the two procedures, and try to prove that they are the same. However, this is not the case:

– *lp2sat* introduces additional atoms not in P and also these atoms get assigned, and
– The order followed by *expand* and *unit-propagate* to assign literals may differ.

However, if we do not take into account the above differences, we have that the two procedures generate the “same” search tree. In order to formally state this result we introduce the following definitions.

We say that a set of literals S is a *branching node of* SMODELS(Π) (resp. *of* CMODELS(Π)) if there is a call to SMODELS-REC(Π', S) (resp. DLL-REC(Γ', S, Π)), following the invocation of SMODELS(Π) (resp. CMODELS(Π)). If $proc$ is SMODELS(Π) or CMODELS(Π), we define

$$\mathit{Branches}(proc) = \{S \cap (P \cup \overline{P}) : S \text{ is a branching node of } proc\}.$$

Finally, we say that SMODELS(Π) and CMODELS(Π) are *equivalent* if

$$\mathit{Branches}(\text{SMODELS}(\Pi)) = \mathit{Branches}(\text{CMODELS}(\Pi)).$$

Theorem 3. *Let* CMODELS *and* SMODELS *be the procedures in Figures 1 and 2 respectively. For each tight program Π,* CMODELS(Π) *and* SMODELS(Π) *are equivalent.*

The idea underlying the proof is that the atoms in P assigned by *expand* in SMODELS-REC correspond exactly to those assigned by *unit-propagate* in DLL-REC, and vice-versa. Indeed, for such result to hold, it is essential that *lp2sat*() is defined as in Section 3.

Theorem 3 states a strong relation between SMODELS and CMODELS, and, ultimately, between SMODELS and DLL: to a certain extent, SMODELS() and DLL(*lp2sat*()) are the same procedure on tight programs. Further, the results hold independently from the specific heuristic used by SMODELS-REC and DLL-REC, as long as they are guaranteed to return the same literal at every point of the two search trees. Because of this, similar results would hold if we enhance SMODELS-REC and DLL-REC with more powerful look-ahead techniques based on *expand* and *unit-propagate* respectively. For instance, SMODELS has been enhanced with the following check, performed before each branch: for every unassigned literal l in the program, check whether assigning l would "fail", i.e., if $\textit{expand}(\textit{p-elim}(l, \Pi), S \cup \{l\})$ returns (as second argument) an inconsistent set of literals. If this is the case, we can safely assign $\bar{l}$ before branching. However, if l fails, then also branching on l would fail, and the tree generated by SMODELS-REC extended with such "failed literal" strategy corresponds to the tree generated by SMODELS-REC with a specific heuristic. Using the same heuristic in DLL-REC (i.e., using a similar "failed literal" strategy based on *unit-propagate*) would lead to an equivalent search tree.

The established correspondence between CMODELS and SMODELS gives us the possibility to derive lower/upper bounds and average case results for CMODELS and SMODELS. Here there are a few.

First, observe that the search tree explored by CMODELS and SMODELS when run on a program Π, critically depends on the specific heuristic used, i.e., in our terminology and with reference to Figures 1 and 2, by the fixed total ordering ρ on the set $P \cup \overline{P}$ used by $\textit{ChooseLiteral}(S)$. In order to highlight the dependency from ρ, we now write $\textit{Branches}^{\rho}(\text{SMODELS}(\Pi))$ (resp. $\textit{Branches}^{\rho}(\text{CMODELS}(\Pi))$) to indicate the set of branching nodes of SMODELS (resp. CMODELS) when run on a program Π, assuming that ρ is the total order on the set $P \cup \overline{P}$ used by $\textit{ChooseLiteral}(S)$. We are now ready to define the *complexity of* SMODELS *on a program* Π as the smallest number in

$$\{|\textit{Branches}^{\rho}(\text{SMODELS}(\Pi))| : \rho \text{ is a total order on } P \cup \overline{P}\}.$$

Analogously, the *complexity of* CMODELS *on a program* Π is the smallest number in

$$\{|\textit{Branches}^{\rho}(\text{CMODELS}(\Pi))| : \rho \text{ is a total order on } P \cup \overline{P}\}.$$

Intuitively, the complexity of SMODELS (resp. CMODELS) on Π is the minimum number of branching nodes that SMODELS (resp. CMODELS) has to explore for solving Π.

Consider the formula PHP_n^m $(n \geq 0, m \geq 0)$ consisting of the clauses

$$\begin{array}{ll} \{p_{i,1}, p_{i,2}, \ldots, p_{i,n}\} & (i \leq m), \\ \{\overline{p}_{i,k}, \overline{p}_{j,k}\} & (i, j \leq m, k \leq n, i \neq j). \end{array}$$

The formulas PHP_n^m are from [9] and encode the pigeonhole principle. If $n < m$, PHP_n^m are unsatisfiable and it is well known that any procedure based on resolution (like DLL) has an exponential behavior. Here we state a similar result for CMODELS and SMODELS. First, if C is a clause $\{l_1, \ldots, l_l\}$ $(l \geq 0)$ we define $\textit{sat2tlp}(C)$ to be the rule $\bot \leftarrow \bar{l}_1, \ldots, \bar{l}_l$. Then, if Γ is a formula, the *translation of* Γ, denoted with $\textit{sat2tlp}(\Gamma)$, is $\cup_{C \in \Gamma} \textit{sat2tlp}(C) \cup \cup_{p \in P}\{p \leftarrow \overline{p'}, p' \leftarrow \overline{p}\}$, where, for each atom $p \in P$, p' is a new atom associated to p. For each n, $\textit{sat2tlp}(PHP_{n-1}^n)$ is tight and has no answer sets.

Corollary 4. *The complexity of* SMODELS *and* CMODELS *on sat2tlp*(PHP^n_{n-1}) *is exponential in* n.

The above result can be easily proved for CMODELS starting from [9]. For SMODELS, it relies on the fact that *sat2tlp*(PHP^n_{n-1}) is tight, and thus on such programs SMODELS and CMODELS are equivalent. The pigeonhole formulas give us the opportunity to define a class of formulas which are exponentially hard for CMODELS but easy for SMODELS. For each formula Γ, define *sat2nlp*(Γ) to be the program $\cup_{C\in\Gamma}$*sat2tlp*$(C) \cup \cup_{p\in P}\{p \leftarrow p\}$.

Corollary 5. *The complexity of* SMODELS *and* CMODELS *on sat2nlp*(PHP^n_{n-1}) *is* 0 *and exponential in* n *respectively.*

In this case, *sat2nlp*(PHP^n_{n-1}) is non tight, and SMODELS can determine the non existence of answer sets without branching mainly thanks to the procedure *AtMost*.[3] To see why this is the case, notice that, if $\Pi =$ *sat2nlp*(PHP^n_{n-1}), then

– *AtLeast*$(\Pi, \{\top\})$ returns $\{\top\}$,
– $\Pi^{\emptyset}$ consists of the rules

$$\bot \leftarrow; \quad \bot \leftarrow p_{i,k}, p_{j,k} \ (i, j \leq n, k \leq n{-}1, i \neq j); \quad p_{i,k} \leftarrow p_{i,k} \ (i \leq n, k \leq n{-}1)$$

  and thus *AtMost*$(\Pi^{\emptyset}, \{\top\})$ returns $\{\bot, \top\}$.
– At line 10 in Figure 2, the set S is set to $S' = \overline{P} \cup \{\top\}$, thus causing one more recursive call to *expand*.
– If $\Pi' =$ *p-assign*(S', Π), *AtLeast*(Π', S') returns the set $S' \cup \{\bot\} = \overline{P} \cup \{\bot, \top\}$, and this is also the set returned by *expand*.
– At line 2 in Figure 2, SMODELS returns FALSE, without performing any branch.

Indeed, the above results can be easily generalized to any formula Γ which is known to be exponentially hard for DLL: *sat2tlp*(Γ) will be exponentially hard for both SMODELS and CMODELS, while *sat2nlp*(Γ) will be exponentially hard for CMODELS but easy for SMODELS. We mention one more of such results, because it involves a class of programs that have been frequently used in the literature as a benchmark for ASP systems.

Define a formula Γ to be a k*-cnf* if each clause in Γ consists of k literals. The *random family of* k*-cnf formulas* is a k-cnf whose clauses have been randomly selected with uniform distribution among all the clauses C of k literals and such that, for each two distinct literals l and l' in C, $\bar{l} \neq l'$.

Corollary 6. *Consider a random* k*-cnf formula* Γ *with* n *atoms and* m *clauses. With probability tending to one as* n *tends to infinity, the complexity of* SMODELS *and* CMODELS *on sat2tlp*(Γ) *is exponential in* n *if the density* $d = m/n$ *is* $d \geq 0.7 \times 2^k$.

[3] In the real implementation of CMODELS, rules like $p \leftarrow p$ will be removed during the preprocessing, and thus the implementation of CMODELS concludes that *sat2nlp*(PHP^n_{n-1}) does not have answer sets without a single branch. However, instead of $p \leftarrow p$, we could have considered, e.g., the two rules $p \leftarrow p'$, $p' \leftarrow p$, (where p' is a newly introduced atom associated to p) and the result in the corollary would still hold.

As in the case of Corollary 4, this result is easy to show for CMODELS starting from [10], and then it follows for SMODELS from Theorem 3. Programs corresponding to random k-cnf formulas have been used, e.g., in [11, 12, 8]. Also notice that since the results in [9] and [10] hold for any proof system based on resolution, enhancing SMODELS and CMODELS with "learning" look-back strategies does not lower the exponential complexity of the procedures. Thus, the above corollaries also hold for SMODELS-CC, and all the different versions of CMODELS. (assuming that CMODELS use a procedure based on DLL as search engine, as it is indeed the case in practice).

Other results that have been proven for DLL can now easily be shown to hold also for SMODELS. Define a literal l as *optimal for a program Π* if there exists a minimal search tree of SMODELS(Π) whose root is labeled with l. The following result echoes the one in [13] for DLL.

Corollary 7. *In* SMODELS, *deciding the optimal literal to branch on, is both NP-hard and co-NP hard, and in PSPACE for tight programs.*

There are many other results in the SAT literature studying the proof-complexity of DLL and/or resolution that are applicable also to SMODELS and CMODELS. See, e.g., [14] for a study on the average complexity of coloring randomly generated graphs with DLL, and [15], which derives exponential lower bounds on the running time of DLL on random 3-SAT formulas also for densities significantly below the satisfiability threshold $d \approx 4.23$. The first result applies also to SMODELS and CMODELS when run on a program Π being the standard tight formulation of a graph coloring problem:[4] *lp2sat*(Π) corresponds to the SAT formulation considered in [14]. Analogously for the second result.

6 On the relation between AS and SAT solvers

Given the results established in the previous Section, we can expect that the combinations of reasoning strategies that work best in SAT, should also work best in ASP, at least when considering tight programs. We show that this is indeed the case, also on non tight programs. We now report about an extensive experimental comparison that we have conducted on a wide variety of programs, and using the combinations of reasoning strategies that, along the years, proved to be more effective in SAT. 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" instances, typically random k-cnf formulas. A solver in this category is SATZ [16].
– "look-back" solvers, featuring a simple but efficient 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

[4] See, e.g., the formulation in `http://www.tcs.hut.fi/~ini/papers/niemela-iclp04-tutorial.ps.gz`.

the look-back phase. These solvers are best for dealing with "large but relatively easy" instances, typically encoding real-world problems. A solver in this category is ZCHAFF [17].

The terminology "small but relatively difficult" and "large but relatively easy" refer to the number of variables and are used to convey the basic intuitions about the instances. To get a more precise idea, consider that in the SAT2003 competition, instances in the random and industrial categories had, on average, 442 and 42703 atoms respectively [18]. Given this, the reasoning strategies that we considered are:

– *Look-ahead:* fast unit-propagation based on 2 literal watching (denoted with "u"); and unit-propagation+failed literal (denoted with "f").
– *Look-back:* basic backtracking (denoted with "b"); and backtracking+1-UIP learning from [17] (denoted with "l").
– *Heuristic:* VSIDS from [17] (denoted with "v"); unit (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 $1024u(p) \times u(\overline{p}) + u(p) + u(\overline{p})$. This heuristic is denoted with "u").

The above search strategies and heuristics are not novel: they have been already presented and implemented in the literature. For example, failed literal is already incorporated in SMODELS, and the heuristic of SMODELS-CC is similar to VSIDS. However, here, for the first time, all these techniques are implemented, combined and analyzed in a common platform.

We considered 4 combinations of reasoning strategies: ulv, flv, flu and fbu, where the first, second and third letter denote the look-ahead, look-back and heuristic respectively, used in the combination. ulv is a standard look-back, "ZCHAFF"-like, solver, similar to SMODELS-CC and CMODELS2. fbu is a look-ahead, "SATZ"-like, solver. flv and flu have both a powerful look-ahead and look-back but different heuristic. As we already anticipated, we can expect that ulv (resp. fbu) has good performances on "large but relatively easy" (resp. "small but relatively difficult") programs. By comparing flv with ulv (resp. flu with fbu) we will see under which conditions a more powerful look-ahead (resp. look-back) leads to better performances. Also notice that the 4 combinations of reasoning strategies that we consider, are the only meaningful. Indeed, the "v" heuristic requires learning, while the "u" heuristic requires that failed literal is enabled.

All 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 to 3600 seconds for real-world problems. In order to have our results not biased by the differences due to the quality of the implementation, we implemented all the reasoning strategies in CMODELS ver. 2 [2]. CMODELS ver. 2, besides being the solver that we knew best, had the following features:

– Its front-end is LPARSE [7], 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.

	PB	#VAR	ulv	flv	flu	fbu
1	4	300	**0.41**	0.52	0.85	0.66
2	4.5	300	TIME	TIME	81.92	**22.53**
3	5	300	448.21	485.36	8.27	**4.72**
4	bw*d9	9956	**1.02**	5.84	2.69	2.75
5	bw*e9	12260	**0.98**	1.91	1.92	1.93
6	bw*e10	13482	**1.29**	7.51	5.03	4.95
7	p1000	14955	**0.48**	37.86	15.41	15.23
8	p3000	44961	**8.86**	369.27	144.12	142.83
9	p6000	89951	**99.50**	TIME	583.55	578.98
10	4	300	265.43	218.48	41.97	**31.05**
11	5	300	TIME	TIME	136.67	**99.75**
12	6	300	TIME	TIME	107.34	**65.83**
13	np60c	10742	**2.83**	1611.32	44.12	44.12
14	np70c	14632	**4.69**	TIME	97.44	97.89
15	np80c	19122	**6.91**	TIME	192.29	196.32

Table 1. Performances on tight (1-9) and non-tight (10-15) problems. For each row, the best result is in bold, and the results within a factor of 2 from the best, are underlined.

There is no other publicly available AS system having the above features, and that we know of. In particular, SMODELS does not contain lazy data structures, and adding them to SMODELS would basically boil down to re-implement the entire solver. Though our analysis has been conducted using CMODELS ver. 2, thanks to the equivalence result established in Theorem 3, analogous results are to be expected for any system based on SMODELS and implementing the techniques that we consider.

Table 1 shows the results on "small" randomly generated programs (lines 1-3, 10-12), and "large" non random programs (lines 4-6, 7-9, 13-15). More in details,

1. Benchmarks (1-3) are tight programs being the translation of randomly generated 3-SAT instances with a ratio of clauses to atoms as in the column "PB". They have been used in [11, 12, 8].
2. Benchmarks (4-6) and (7-9) are tight programs encoding blocks world planning problems and 4-colorability graph problems, respectively. These benchmarks are publicly available at `http://www.tcs.hut.fi/Software/smodels/tests/`.
3. Benchmarks (10-12) are non tight programs, randomly generated according to the methodology proposed in [19]. As before, the number in the column "PB" is the ratio of clauses to atoms.
4. Benchmarks (13-15) are non tight programs encoding Hamiltonian Circuit problems on complete graphs. The encoding is from [20].

For the randomly generated programs, for each ratio, we generated 10 instances and show the median results. In each row, #VAR represents the number of atoms in the instance.

The first observation is that we get the results that we expected, (except for the results on the first row, where the positive results of ulv are due to the relative simplicity of the problems): on “small but relatively difficult” programs fbu is best, while on “large but relatively easy” programs ulv is best. The second observation is that adding failed literal (resp. learning) to ulv (resp. fbu) does not improve performances when considering the “large” (resp. “small”) programs.

We also considered other classes of programs, both non large and non randomly generated. For these programs, the situation of which reasoning strategy is best is less clear, and (as it can be expected) it varies from class to class.

7 Conclusions

We studied the relation existing between SMODELS and CMODELS, and, ultimately, between AS and SAT solvers. From a theoretical point of view, we proved that the two systems have the same behavior on tight programs. Given that CMODELS is based on DLL, our equivalence results allow to easily derive many other interesting properties about the two procedures, and in particular about SMODELS. We also conducted an extensive experimental analysis showing that the combination of reasoning strategies that are best in SAT, are also best in ASP on randomly generated or on large real world problems.

We believe that our paper is particularly important for ASP researchers who are interested in formally establishing the computational behavior of systems, but also for developers and, more in general, for people involved in benchmarking ASP systems. In particular, for developers, our theoretical results should foster the design of systems incorporating reasoning strategies that provably allow to easily solve problems otherwise exponential: in SAT, this led to the development, e.g., of ZAP [21]. Further our experimental results suggest that developers (in order to advance the state-of-the-art) should focus either on randomly generated problems (and thus develop a look-ahead solver) or on real-world problems (and thus develop a look-back solver): this already happened in SAT. Finally, the results in this paper are particularly important also to people interested in benchmarking systems (see the recent ASPARAGUS initiative [22]). Our theoretical results tell us, e.g., that there exist classes of programs on which SMODELS and/or CMODELS (but also ASSAT) are *bound to* be exponential. Our conclusive experimental analysis points out that it hardly makes sense to run a solver like SMODELS-CC [8] on randomly generated programs, and, vice-versa, that it hardly makes sense to use CMODELS with SATZ [16] as SAT solver on large problems coming from real-world applications.

Finally, we believe that this paper is a major step in the direction of closing the gap between SAT and ASP, as advocated by Miroslaw Truszczyński in his invited talk at the last NMR workshop in Whistler, Canada.[5]

Acknowledgments

We would like to thank Nicola Leone, Vladimir Lifschitz and Mirek Truszczynski for discussions related to the subject of this paper. This work has been partially supported by MIUR.

[5] Slides available at `http://cs.engr.uky.edu/~mirek/stuff/nmr-inv.pdf`.

References

1. E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. In *Proc. IJCAR*, 2001.
2. E. Giunchiglia, M. Maratea, and Y. Lierler. SAT-based answer set programming. In *Proc. AAAI*, 2004.
3. T. H. Cormen, C. E. Leiserson,R. L. Rivest and C. Stein Introduction to Algorithms. MIT Press, 2001.
4. François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.
5. Yu. Babovich and V. Lifschitz. Computing Answer Sets Using Program Completion. Available at `http://www.cs.utexas.edu/users/tag/cmodels/cmodels-1.ps`, 2003.
6. F. Lin and Y. Zhao ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proc. AAAI*, 2002.
7. P. Simons. Extending and implementing the stable model semantics. *PhD Thesis*, 2000.
8. Jeffrey Ward and John S. Schlipf. Answer set programming with clause learning. In *Proc. LPNMR*, 2004.
9. Haken. The intractability of resolution. *TCS*, 39:297-308, 1985.
10. V. Chvátal and E. Szemerédi. Many hard examples for resolution. *J. ACM*, 35(4):759–768, 1988.
11. W. Faber, N. Leone, and G. Pfeifer. Experimenting with heuristics for ASP. In *Proc. IJCAI*, 2001.
12. P. Simons, I. Niemelä, and S. Timo. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.
13. Paolo Liberatore. On the complexity of choosing the branching literal in DPLL. *Artificial Intelligence*, 116(1-2):315–326, 2000.
14. Rémi Monasson. On the analysis of backtrack procedures for the coloring of random graphs. In *Complex Networks*, Lecture Notes in Physics, pages 232–251. Springer, 2004.
15. D. Achlioptas, P. Beame, and M. Molloy. A sharp threshold in proof complexity. In *Proc. STOC*, pages 337–346, 2001.
16. Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proc. IJCAI*, 1997.
17. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. DAC*, 2001.
18. D. Le Berre and L. Simon. The essentials of the SAT'03 competition. In *Proc. SAT*, 2003.
19. Fangzhen Lin and Yuting Zhao. ASP phase transition: A study on randomly generated programs. In *Proc. ICLP*, 2003.
20. I. Niemelä Logic programs with stable model semantics as a constraint programming paradigm. In *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.
21. H. Dixon, M. Ginsberg, E. Luks, and A. Parkes. Generalizing Boolean satisfiability II: Theory. *JAIR*, 22:481–534, 2004.
22. P. Borchert, C. Anger, T. Schaub, and M. Truszczynski. Towards systematic benchmarking in answer set programming: The dagstuhl initiative. In *Proc. LPNMR*, 2004.