Ann Math Artif Intell (2008) 53:169–204
DOI 10.1007/s10472-009-9113-1

On the relation among answer set solvers

Enrico Giunchiglia · Nicola Leone · Marco Maratea

Published online: 24 January 2009

Abstract In this paper, we study the relation among Answer Set Programming (ASP) systems from a computational point of view. We consider SMODELS, DLV, and CMODELS ASP systems based on stable model semantics, the first two being native ASP systems and the last being a SAT-based system. We first show that SMODELS, DLV, and CMODELS explore search trees with the same branching nodes (assuming, of course, a same branching heuristic) on the class of *tight* logic programs. Leveraging on the fact that SAT-based systems rely on the deeply studied Davis–Logemann–Loveland (DLL) algorithm, we derive new complexity results for the ASP procedures. We also show that on nontight programs the SAT-based systems are computationally different from native procedures, and the latter have computational advantages. Moreover, we show that native procedures can guarantee the "correctness" of a reported solution when reaching the leaves of the search trees (i.e., no stability check is needed), while this is not the case for SAT-based procedures on nontight programs. A similar advantage holds for DLV in comparison with SMODELS if the "well-founded" operator is disabled and only Fitting's operator is used for negative inferences. We finally study the "cost" of achieving such advantages and comment on to what extent the results presented extend to other systems.

A preliminary version of a part of the results presented in this work has been published in [25].

E. Giunchiglia (✉) · M. Maratea
STAR-Lab, DIST, University of Genova, viale Francesco Causa,
13-16145 Genoa, Italy
e-mail: enrico@dist.unige.it

M. Maratea
e-mail: marco@dist.unige.it

N. Leone
Department of Mathematics, University of Calabria, viale P. Bucci,
Cubo 30b-87030 Arcavacata di Rende, Cosenza, Italy
e-mail: leone@mat.unical.it

Keywords Answer set solvers · Answer set programming · Search trees

Mathematics Subject Classifications (2000) 68N17 · 68T20 · 68T27

1 Introduction

Answer set programming (ASP) [36, 41, 46] is a declarative approach to programming, which has been recently proposed in the area of nonmonotonic reasoning and logic programming. The ASP language is based on logic rules and characterized by the use of nonmonotonic negation [42]. The intended models of an ASP program (i.e., the semantics of the program) are subset-minimal models which are "grounded" in a precise sense, and are called *answer sets* or *stable models* [22, 23]. The idea of answer set programming is to represent a given computational problem by an ASP program whose answer sets correspond to solutions, and then use an answer set solver to find such a solution [36].

ASP systems based on stable model semantics include both native ASP systems (the ones which work directly on a logic program, i.e., CLASP, DLV, NoMore++, SMODELS, and SMODELS-CC[1]), SAT-based systems (the ones which rely on a translation into a propositional formula and on a SAT solver to solve such formula, i.e., ASSAT and CMODELS[2]) and the Pseudo Boolean (PB)-based system PBMODELS[3] [39], which relies on a translation into a PB formula, and is targeted for logic programs with cardinality and weight atoms.[4]

It is worthwhile noting that, in a broader view, ASP includes also other systems that deal with logic programs but are based on different semantics, like, for instance, ASPPS [11], which is based on a logic for propositional schemata, that is somehow "closer" to classical logic. Here, we do not consider these systems because our focus is on systems based on the stable model semantics.

In this paper, we study the relation among ASP systems from a computational point of view. Until recently, the procedures underlying ASP solvers had been often advocated to be "similar", but without formal results. In [25], for the first time, we provided such a needed, theoretical foundation, but only limited to CMODELS and SMODELS. From there on, other works have been performed in this direction with different approaches, i.e., the one based on proof system ASP tableaux proposed in [18] and then extended in [3, 19, 31], and another relying on transition systems, recently proposed in [35].

[1] See http://www.cs.uni-potsdam.de/clasp, http://www.dbai.tuwien.ac.at/proj/dlv, http://www.cs.uni-potsdam.de/nomore, http://www.tcs.hut.fi/Software/smodels, and http://www.nku.edu/~wardj1/Research/smodels_cc.html, respectively.

[2] See http://assat.cs.ust.hk and http://www.cs.utexas.edu/users/tag/cmodels.html, respectively.

[3] See http://www.cs.uky.edu/ai/pbmodels/.

[4] Note that in this work we have considered CLASP as a "native" solver, given it does not use off-the-shelf SAT solvers. But, in some sense, CLASP can be viewed as a unique system given that it is neither really "native" nor, as we said, "SAT-based". Unlike native systems, it does not perform its inference directly on a logic program. CLASP, like SAT-based solvers, computes the completion of a program and then performs its inference on the completion. Unlike SAT-based systems, CLASP does not use off-the-shelf SAT solvers but implements its own inference engine.

In this paper, starting from [25], we include DLV in the picture, and present other relevant results.

We first analyze the computational behaviour of CMODELS, SMODELS, and DLV ASP systems on the class of *tight* [12] logic programs. This class is characterized by the fact that the answer sets of a tight program Π are in one-to-one correspondence with the solutions of the propositional formula representing the completion $Comp(\Pi)$ of Π [7].[5] We prove that the three systems explore search trees with the same branching nodes (assuming, of course, a same branching heuristic) on tight programs. Since SAT-based systems are based on the Davis–Logemann–Loveland (DLL) algorithm, which is one of the best studied procedures in Artificial Intelligence and Computer Science, the equivalence results on tight programs allow us to derive new complexity results for CMODELS, SMODELS, and DLV procedures, related to, e.g., (a) logic program translations of SAT formulas encoding the pigeons principle and k-CNF randomly generated formulas, and (b) deciding the best literal to branch on. Note that [31] have recently shown, among other results, that it is possible to define a proof system for (nondisjunctive) logic programs which can polynomially simulate tree resolution, and vice versa (assuming translations between ASP and SAT similar to those we use in this paper). Given that CMODELS (resp. SMODELS) can be seen as an implementation of tree resolution (resp. of the proof system for nondisjunctive programs), their work can be seen as an alternative way of extending results known for DLL to ASP systems. We strengthen this result showing that the procedures produce exactly the same search trees (assuming the same branching heuristics).

We then turn our attention to nontight programs. We show that on nontight programs the SAT-based systems are computationally different from native procedures, and the latter have some computational advantage. This is accomplished by finding nontight programs which are exponentially hard for CMODELS but are very easy for SMODELS and DLV.

Moreover, we outline further advantages of native procedures over SAT-based ones. Consider the points of the algorithms where the computation ends at the leaves of a search tree without a contradiction (that we call *consistent leaf interpretations*): on these leaves each atom in the Herbrand Base has been assigned either to True or to False. We prove that such interpretations are guaranteed to be answer sets for DLV and SMODELS but not for CMODELS, which additionally needs a stability check for their verification. Intuitively, native procedures do not need a stability check because they employ the "well-founded" operator for negative inferences,[6] which detects in advance possible reasons for "unstability" (recall that, as shown in [32], a model is stable iff it is a fixpoint of the "well-founded" operator). An interesting difference among the native procedures arises when their "well-founded" operator is disabled, and negative inferences are performed by their "Fitting" operator only. In this case, only the DLV system still guarantees this property, that is, even if the "well-founded" operator is disabled, the consistent leaves of the computation of DLV are automatically guaranteed to be answer sets, without the need of any stability check.

[5] There are other reductions, i.e., the ones presented in [29, 30] which guarantee the one-to-one correspondence. Here, we consider Clark's completion because it is the one used by the SAT-based systems we analyze and because, in general, it leads to better results; see, e.g., the results of the First Answer Set Programming Competition held in 2007 and even the analysis in [29].

[6] The "well-founded" operator is called *AtMost* in the original presentation of SMODELS.

This is mainly due to a peculiarity of DLV, which is more "cautious" in the derivation of positive literals and better ensures their support also employing a four-valued interpretation for atoms. Starting from these results, we study the complexity of finding such a consistent leaf interpretation, which is shown to be linear for CMODELS, SMODELS and DLV on tight programs; and (a) becomes quadratic for native procedures on nontight programs; (b) remains linear when disabling well-founded inferences.

We finally comment on to what extent all the results presented so far extend to the remaining ASP systems based on stable model semantics.

An important consideration, which is suggested by the equivalence results of this paper, is that native and SAT-based ASP systems can take advantages from each others. ASP solvers could benefit (at least on tight programs) from the great amount of work done in SAT: search strategies, heuristics and combinations, which have been shown to be effective in SAT, can be imported (with the necessary adaptations) in ASP systems leading to similar improvements. This is indeed the case as shown in [24, 27] (where results are shown to experimentally extend to nontight programs), and this is the current research trend that the main groups working on native procedures are following, by implementing optimized look-back techniques and heuristics in order to more efficiently handling "real-world" problems (see, e.g., [20, 40, 48][7]).

On the other hand, given our results on nontight programs, SAT-based ASP solvers should evaluate the possibility to import from native solvers (with the needed adaptations) the techniques for exploiting "well-founded" inferences.

Our considerations have been recently confirmed by CLASP [20], which aims at combining SAT heuristics and optimizations with the powerful techniques specifically designed for ASP. The effectiveness of CLASP's approach is confirmed both in [20] and in the results of the First ASP Competition [21].[8]

Summing up, the main contributions of this paper are:

- We prove that CMODELS, SMODELS and DLV ASP systems, based on stable model semantics, are equivalent on tight programs.
- We show that they are not equivalent on nontight programs, and native procedure have advantages over SAT-based procedures.
- We show new complexity results for such systems.

When dealing with consistent leaf interpretations,

- We highlight computational advantages of native over SAT-based procedures.
- We highlight further computational advantages of DLV over SMODELS, when considering the systems without well-founded inferences (i.e., using only Fitting operator for negative inferences).
- We study the computational cost of achieving such advantages.

The paper is structured as follows. In Section 2 we give the basic definitions. Section 3 is devoted to the presentation of the algorithms of CMODELS, SMODELS, and DLV, respectively, which are used in Section 4 for the formal analysis of their computational properties. Section 5 compares the systems with Fitting's operator for

[7] Regarding SMODELS, these are personal communications from Ilkka Niemelä.

[8] http://asparagus.cs.uni-potsdam.de/contest/.

negative inferences (well-founded operator disabled). Section 6 discusses the impact of our results on other systems. Section 7 is dedicated to the study of the related work. We end the paper in Section 8 with the conclusions and possible topics for future research.

2 Basic definitions

In this section, we introduce the basic concepts and definitions that will be used in the rest of the paper. We first present ASP syntax and semantics, and then important properties that hold on some classes of logic programs.

2.1 Syntax

Let P be a set of atoms.[9] If p is a an atom, $\overline{p}$ is the *negation* of p, and $\overline{\overline{p}}$ is p. We will also use the logic symbol $\bot$ (standing for False).

Atoms and their negations form the set of *literals*. If S is a set of literals, we define $\overline{S} = \{\overline{l} : l \in S\}$.

A *rule* is an expression of the form

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

where $p_0 \in P$, and $\{p_1, \ldots, p_n\} \subseteq P$ $(0 \leq m \leq n)$. If r is a rule (1), $head(r) = p_0$ is the *head* of r, and $body(r) = \{p_1, \ldots, p_m, \overline{p}_{m+1}, \ldots, \overline{p}_n\}$ is the *body* of r.[10] We also define $posBody(r) = \{p_1, \ldots, p_m\}$ (resp. $negBody(r) = \{p_{m+1}, \ldots, p_n\}$) as the positive (resp. negative) part of the body. A *(logic) program* is a finite set of rules.

2.2 Semantics

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 positive 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 r (1) in Π, $p_0 \in X$ whenever $posBody(r) \subseteq X$, and that
- X is an *answer set* for Π if X is the smallest set (i.e., minimal under subset inclusion) 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

$$p_0 \leftarrow p_1, \ldots, p_m$$

[9] Since our analysis compares the ASP *solvers* whose input are propositional programs, we deal with propositional programs in this paper. Atoms, literals, rules, programs are implicitly assumed to be ground here.

[10] This formulation does not (directly) allow for constraints like $\bot \leftarrow body(r)$, but they can be expressed with a rule $p' \leftarrow body(r), \overline{p'}$, provided p' is a newly introduced atom.

for all rules r (1) in Π such that $X \cap negBody(r) = \emptyset$. X is an *answer set* for Π if X is an answer set for Π^X.

2.3 Program properties

The *dependency graph* of a program Π is a directed graph G such that

- the vertexes of G are the atoms in Π, and
- G has an edge from p_0 to $p_1, \ldots, p_m$ for each rule (1) in Π.

A *loop* of Π is a nonempty set L of atoms such that for each pair p, p' of atoms in L there is a path of length > 0 from p to p' in the dependency graph of Π, whose intermediate nodes belong to L. A program Π is *tight* if G does not contain any loop.

Example 1 Consider the following two simple programs:

$$\Pi_1 : p \leftarrow q, not\ r.$$
$$q.$$

$$\Pi_2 : p \leftarrow p.$$

Program Π_1 is tight, while Π_2 is nontight. Indeed, the dependency graph of Π_1 does not contain any loop, while the dependency graph of Π_2 contains a loop on p. Note that Π_1 has the answer set $\{p, q\}$, while the unique answer set of Π_2 is the empty set.

If p_0 is an atom, we define *completion of* Π *relative to* p_0, i.e., $Comp(\Pi, p_0)$, as the propositional formula

$$p_0 \equiv \bigvee (p_1 \wedge \cdots \wedge p_m \wedge \overline{p}_{m+1} \wedge \cdots \wedge \overline{p}_n)$$

where the disjunction extends over all rules (1) in Π with head p_0. The *completion* $Comp(\Pi)$ of Π consists of formulas $Comp(\Pi, p_0)$ for each atom p_0.

The following theorem exploits an important correspondence between the answer sets of Π and the models of $Comp(\Pi)$.

Theorem 1 [17] *For any tight program* Π*, there is a one-to-one correspondence between the answer sets of* Π *and the models of* $Comp(\Pi)$.

As a side effect of the theorem, enumerators of SAT models can be used, on tight programs, as ASP solvers if run on the completion of Π. Note that for nontight program Π_2, $Comp(\Pi_2)$ is the propositional formula $p \equiv p$, which has two models, but the one where p is assigned to True does not correspond to an answer set because of the loop on p in the dependency graph.

3 ASP procedures

In this Section we present the solving procedures for the ASP systems CMODELS, SMODELS and DLV. Besides being well-known, we focus on these systems because their

solving procedures are both representative (at least considering basic algorithms without learning) of other ASP procedures (e.g., CMODELS for ASSAT and SMODELS for SMODELS-CC), and have some peculiarities (i.e., SAT-based vs. native procedures, and based on four-valued interpretations vs. three-valued interpretations).

Remark 1 In the description of the algorithms reported in this paper, we assume that parameters are passed to a procedure by value, as in [8].

In the following, given a program Π, we denote with P the set of atoms appearing in Π.

3.1 CMODELS

CMODELS [27] is a SAT-based ASP system which 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. In order for the formula to be fed to a SAT solver, it needs to be converted into Conjunctive Normal Form (CNF), usually by adding variables (denoted with the set N). Formally, a *clause* is a finite set of literals and a *(propositional) formula* is a finite set of clauses. A *three-valued interpretation* is a pair $I := \langle \mathbf{T}, \mathbf{F} \rangle$, where $\mathbf{T}$ and $\mathbf{F}$ are sets of atoms included in the signature of the SAT formula. An atom p is True/False/Undef w.r.t. I if $p \in \mathbf{T}/\mathbf{F}/((P \cup N) \setminus (\mathbf{T} \cup \mathbf{F}))$. We denote by $\mathbf{T}(I)/\mathbf{F}(I)/\mathbf{U}(I)$ the set of True/False/Undef atoms w.r.t. I.

In the following, $I + true(p)$ denotes the extension of the interpretation I assigning p to True, and similarly for $I + false(p)$ and $I + undef(p)$.

An interpretation I *satisfies* a formula Γ if for each clause C in Γ, $C \cap (\mathbf{T}(I) \cup \overline{\mathbf{F}(I)}) \neq \emptyset$. If I satisfies Γ then we also say that I is a *model* of Γ and that Γ is *satisfiable*. Finally, I is inconsistent iff $\mathbf{T}(I) \cap \mathbf{F}(I) \neq \emptyset$.

There are various versions of CMODELS. Here we consider the one proposed in [27] (called ASP-SAT in that paper) without learning, which also corresponds to the description on the CMODELS's home page, and it is represented in Fig. 1, where

- Π is the input program; Γ is a set of clauses; I is an interpretation; p and l are an atom and a literal, respectively.
- *lp2sat*(Π) is the set of clauses—corresponding to the CNF translation (using [52] clause form transformation) of *Comp*(Π)—formally defined below.

Fig. 1 The algorithm of CMODELS

```
function CMODELS(Π) return DLL-REC(lp2sat(Π),I∅,Π);

function DLL-REC(Γ,I,Π)
 1  I := unit-propagate(Γ, I);
 2  if (∅ ∈ Γ_I) return False;
 3  if (P \ (T(I) ∪ F(I)) = ∅) return test(I,Π);
 4  l := ChooseLiteral(Π, I);
 5  return DLL-REC(Γ, I + true(l), Π); or
 6         DLL-REC(Γ, I + false(l), Π);

function unit-propagate(Γ,I)
 7  if ({l} ∈ Γ_I) return unit-propagate(Γ, I + true(l));
 8  return I;
```

- Γ_I is the formula obtained from Γ by (a) deleting the clauses $C \in \Gamma$ such that for some literal l, $l \in (\mathbf{T}(I) \cup \overline{\mathbf{F}(I)}) \cap C$, and (b) deleting $\bar{l}$ from the other clauses in Γ such that $\bar{l} \in (\overline{\mathbf{T}(I)} \cup \mathbf{F}(I)) \cap C$.
- $I_\emptyset$ is the function which extends the interpretation I with $I + undef(p)$ for each $p \in P \cup N$.
- *test*(I, Π) returns True if $\mathbf{T}(I) \cap P$ is an answer set of Π, and False otherwise. More in details, it works by (a) computing the reduct $\Pi^{\mathbf{T}(I)\cap P}$ of Π with respect to the set of atoms $\mathbf{T}(I) \cap P$; (b) computing the answer set X of $\Pi^{\mathbf{T}(I)\cap P}$ via the Dowling–Gallier procedure [10]; and, finally, (c) if $(\mathbf{T}(I) \cap P) \setminus X$ is empty returns True, otherwise False.
- *ChooseLiteral*(Π, I) returns a literal in $P \cup \overline{P}$, according to some heuristic, such that $\{l, \bar{l}\} \cap \mathbf{U}(I) \neq \emptyset$.
  Moreover, we assume that the left part of the **or** at line 5 is evaluated before the right part at line 6, as customary.

CMODELS(Π) simply invokes DLL-REC(*lp2sat*(Π),$I_\emptyset$,Π). It is easy to see that DLL-REC(Γ,I,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*(I, Π): such a modification is needed only if the input program Π is nontight. Indeed, if Π is tight we are guaranteed that any model of *lp2sat*(Π) corresponds to an answer set of Π [17], and *test*(I, Π) always succeeds.

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) whose head is p_0, the clauses:

$$\{p_0, \overline{n}_r\}, \tag{2}$$

$$\{n_r, \overline{p}_1, \ldots, \overline{p}_m, p_{m+1}, \ldots, p_n\}, \tag{3}$$

$$\{\overline{n}_r, p_1\}, \ldots, \{\overline{n}_r, p_m\}, \{\overline{n}_r, \overline{p}_{m+1}\}, \ldots, \{\overline{n}_r, \overline{p}_n\}, \tag{4}$$

   where n_r is a newly introduced atom, and
2. the clause

$$\{\overline{p}_0, n_{r_1}, \ldots, n_{r_q}\} \tag{5}$$

   where $n_{r_1}, \ldots, n_{r_q}$ $(q \geq 0)$ are the new symbols introduced in the previous step.

Finally, the *translation of* Π, denoted with *lp2sat*(Π), is $\cup_{p\in P}$*lp2sat*(Π, p).

The following proposition states the correctness of CMODELS procedure.

Proposition 1 [27] *Let* CMODELS *be the procedure in Fig. 1. Then, 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 Fig. 1 can be considered as representative for all the various versions of CMODELS, in 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: modern SAT solvers (including the ones used by CMODELS) feature more sophisticated look-ahead, look-back strategies, such as backjumping and learning, and heuristics.
3. Given a program Π, its translation *lp2sat*(Π) to SAT is exactly the one used by CMODELS (see [4]).

3.2 SMODELS

SMODELS [50] is a native ASP procedure which, given a program Π, searches for answer sets by "extending" an interpretation I till either I becomes inconsistent (in which case backtracking occurs) or each atom is not assigned to $\mathbf{U}(I)$ (in which case $\mathbf{T}(I)$ is an answer set for Π). $\mathbf{T}$ and $\mathbf{F}$ are now sets of atoms in P, and the set of undefined atoms $\mathbf{U}$ is defined as $P \setminus (\mathbf{T} \cup \mathbf{F})$. Given an interpretation I, the body of a rule is (a) True if the set of its literals is empty, or (b) equal to the minimal truth value of its literals considering the order True > Undef > False. We denote the truth value of a literal or set of literals S w.r.t. I by $val_I(S)$, and we will omit the index I (writing $val(S)$) when I is clear from the context. A simple, recursive presentation of SMODELS is given in Fig. 2, where

- Π is a program; I is an interpretation; p is an atom; r is a rule; and l is a literal.
- Π_I is the program obtained from Π by eliminating the rules $r \in \Pi$ such that $val_I(body(r)) = \mathsf{False}$.
- *ChooseLiteral*(Π, I) is the same function used by CMODELS at line 4 in Fig. 1.
- Given a rule r, *assignToTrue*(I, *posBody*(r)) is a procedure that assigns to True all atoms in *posBody*(r) that are assigned to Undef. Similarly for *assignToFalse*(I, *negBody*(r)), where undefined atoms in *negBody*(r) are assigned to False.

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

- *Line 1:* the interpretation I is extended by the routine *expand*(Π, I), explained below.
- *Line 2:* if I is inconsistent, no answer set extending I exists, and False is returned.
- *Line 3:* if each atom $p \in P$ is assigned either to True or to False, then (a) $\mathbf{T}(I)$ is an answer set of the initial program, and (b) True is returned.
- *Lines 4–6:* if none of the above applies, a literal l is selected (line 4), an answer set extending $I + true(l)$ (line 5) is searched, and, upon failure, $I + false(l)$ (line 6) is searched.

expand(Π, I) extends the assignment I generated so far by recursively invoking *AtLeast* (line 7) and then *WFInf* (line 10) till it is no longer possible to extend I (lines 12–13). *AtLeast* encodes the following facts:

- *Line 14:* if there exists a rule r whose body is True, and the head is not True, then the head is assigned to True.
- *Line 15:* if an atom p, which is not false, is not the head of any potentially applicable rule, then p can be safely assigned to False.

```
function SMODELS(Π) return SMODELS-REC(Π, I_∅);

function SMODELS-REC(Π,I)
 1 I := expand(Π, I);
 2 if (T(I) ∩ F(I) ≠ ∅) return False;
 3 if (P \ (T(I) ∪ F(I)) = ∅) return True;
 4 l := ChooseLiteral(Π, I);
 5 return SMODELS-REC(Π, I + true(l)); or
 6        SMODELS-REC(Π, I + false(l));

function expand(Π,I)
 7 I := AtLeast(Π, I);
 8 I' := I;
 9 if "Π is nontight" {
10    X := WFInf(Π_I^∅, ∅);
11    foreach p ∈ P { if(p ∉ X) I + false(p);}}
12 if (I ≠ I') return expand(Π,I);
13 return I;

function AtLeast(Π,I)
14 if (∃r ∈ Π_I : val(body(r)) = True and (val(head(r)) ≠ True)
      return AtLeast(Π, I + true(head(r)));
15 if (∃p ∈ P : val(p) ≠ False and ∄r ∈ Π_I : head(r) = p)
      return AtLeast(Π, I + false(p));
16 if (∃r ∈ Π_I : val(head(r)) = True and val(body(r)) = Undef and ∄r' ∈ Π_I, r' ≠ r : head(r') = head(r))
      assignToTrue(I, posBody(r)); assignToFalse(I, negBody(r));
      return AtLeast(Π, I);
17 if (∃r ∈ Π_I, l ∈ body(r), val(l) = Undef : val(head(r)) = False and val(body(r) \ {l}) = True)
      return AtLeast(Π, I + false(l));
18 return I;

function WFInf(Π,X)
19 if (∃r ∈ Π : body(r) ⊆ X and head(r) ∉ X)
      return WFInf(Π, X ∪ {head(r)});
20 return X;
```

Fig. 2 The algorithm of SMODELS

- *Line 16:* if there is only one rule with head p, the head is positively assigned (i.e., it evaluates to True) and the body is not false, then each undefined atom in the positive part of the body is assigned to True, while each atom in the negative part of the body is assigned to False.
- *Line 17:* if there is a rule with head p whose body contains only one literal l which is undefined, if p evaluates to False, then l has to be assigned to False.

When no further simplification is possible,

1. I is returned by *AtLeast*(Π, I) (line 18); and, if the program is nontight,
2. *WFInf* is invoked with $\Pi_I^{\emptyset}$—the reduct of Π_I relative to the empty set—and $\emptyset$ as arguments (line 10).

WFInf incrementally adds to a set X of atoms first initialized to $\emptyset$ the heads of the rules in $\Pi_I^{\emptyset}$ whose body is a subset of X (line 19). If X is the set returned by *WFInf*($\Pi_I^{\emptyset}$, $\emptyset$) (i.e., if X is the set returned at line 20), if an atom p does not belong to X then p can be safely assigned to False (line 11) (see [49] for more details).

The following proposition states the correctness of SMODELS procedure.

Proposition 2 [49] *Let* SMODELS *be the procedure in Fig. 2. Then, 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 [49], page 17. As for CMODELS, the actual implementation of SMODELS features more complex look-ahead/look-back strategies and heuristic.

3.3 DLV

DLV [33] is, like SMODELS, a native ASP procedure which works directly on the program at hand. Actually, DLV can work with the wider set of *disjunctive* logic programs, where a rule r is generalized such that the head is a disjunction of atoms. Here, for the sake of the comparison to the other systems, we restrict the algorithm of DLV to work on logic programs without disjunction in the head.

A main peculiarity of DLV is the fact that it uses a four-valued interpretation (instead of the three-valued interpretation of CMODELS and SMODELS) for atoms. Indeed, DLV can assign the value "must-be-true" (denoted Mbt) to an atom p. Intuitively, an atom p is assigned the value must-be-true if it has to become True, but at the moment it misses a "supporting" rule, which is a rule for p whose body is true. The effect of negation on an Mbt atom is like a "*True*", while assigning to Mbt a negated literal corresponds to the negated literal itself, and a double negation of an atom means to assign it to Mbt. Formally, a four-valued interpretation is a triple $I := \langle \mathbf{T}, \mathbf{M}, \mathbf{F} \rangle$, where $\mathbf{T}$, $\mathbf{M}$, $\mathbf{F}$ are set of atoms included in P. $\mathbf{M}$ indicates the set of atoms assigned to Mbt ($\mathbf{M}(I)$ if it is referred to an interpretation I), $\mathbf{T}$ and $\mathbf{F}$ have the same meaning as in the three-valued interpretation. The set of undefined atoms is now defined as $P \setminus (\mathbf{T} \cup \mathbf{M} \cup \mathbf{F})$.

For simplicity, we will omit the specification of the number of truth values of the interpretations when it is clear from the context. CMODELS and SMODELS interpretations are implicitly three-valued, while DLV interpretations are four-valued. Interpretations valid for both DLV, SMODELS and CMODELS are obviously three-valued.

The body of a rule is now (a) True if the set of literals is empty, or (b) equal to the minimal truth value of the literals considering the order True > Mbt > Undef > False (from [13]).

A simple, recursive presentation of DLV is given in Fig. 3, where

- Π is a program; I is an interpretation; p is an atom; r is a rule; and l is a literal.
- Π_I is defined similarly as in the SMODELS algorithm.
- *ChooseLiteral*(Π, I) is a function similar to the analogous used by CMODELS and SMODELS, i.e., *ChooseLiteral*(Π, I) returns, given the same program and a "corresponding" interpretation, the same literal chosen by the analogous functions in CMODELS and SMODELS algorithms. Here we assume that *ChooseLiteral*(Π, I) can choose among all the undefined atoms (in the real implementation of DLV, it chooses among the so called "PT-literals").
- similarly to $I + true(p)$, $I + false(p)$ and $I + undef(p)$, $I + mbt(p)$ denotes the extension of the interpretation I assigning p to Mbt; and *assignToMBT*(I, *posBody*(r)) is a procedure that assigns to Mbt all atoms in *posBody*(r) which are assigned to Undef.

function DLV(Π) **return** DLV-REC(Π, $I_\emptyset$);

function DLV-REC(Π,I)
1 $I := DetCons(\Pi, I)$;
2 **if** $((\mathbf{T}(I) \cup \mathbf{M}(I)) \cap \mathbf{F}(I) \neq \emptyset)$ **return** False;
3 **if** $(P \setminus (\mathbf{T}(I) \cup \mathbf{M}(I) \cup \mathbf{F}(I)) = \emptyset)$ { **if** $(\mathbf{M}(I) = \emptyset)$ **return** True; **else return** False;}
4 $l := ChooseLiteral(\Pi, I)$;
5 **return** DLV-REC($\Pi, I + mbt(l)$); **or**
6 DLV-REC($\Pi, I + false(l)$);

function $DetCons(\Pi,I)$
7 $I := DetConsInf(\Pi, I)$;
8 $I' := I$;
9 **if** "Π is nontight" {
10 $X := WFInf(\Pi_I^\emptyset, \emptyset)$;
11 **foreach** $p \in P$ { **if**$(p \notin X)$ $I + false(p)$;}}
12 **if** $(I \neq I')$ **return** $DetCons(\Pi,I)$;
13 **return** I;

function $DetConsInf(\Pi,I)$
14 **if** $(\exists r \in \Pi_I : val(body(r)) \geq$ Mbt **and** $(val(body(r)) > val(head(r)))$
return $DetConsInf(\Pi, (val(body(r)) =$ True$?\ I + true(head(r)) : I + mbt(head(r))))$;
15 **if** $(\exists p \in P : val(p) \neq$ False **and** $\nexists r \in \Pi_I : head(r) = p)$
return $DetConsInf(\Pi, I + false(p))$;
16 **if** $(\exists r \in \Pi_I : val(head(r)) \geq$ Mbt **and** $val(body(r)) =$ Undef **and** $\nexists r' \in \Pi_I, r' \neq r : head(r') = head(r))$
$assignToMBT(I, posBody(r))$; $assignToFalse(I, negBody(r))$;
return $DetConsInf(\Pi, I)$;
17 **if** $(\exists r \in \Pi_I, l \in body(r), val(l) =$ Undef $: val(head(r)) =$ False **and** $val(body(r) \setminus \{l\}) \geq$ Mbt)
return $DetConsInf(\Pi, I + false(l))$;
18 **return** I;

Fig. 3 The algorithm of DLV

The computation of DLV-REC(Π, I) proceeds in a very similar way to the one of SMODELS. Here we would like to underline two crucial differences between the algorithm of DLV and the one of SMODELS, which are mainly due to the presence of the Mbt value in DLV, and that will play a fundamental role in some of the results we will obtain:

- the only point of the algorithm where an atom can be assigned True is line 14, if the atom is guaranteed to be supported. In every other point, assigning an atom "positively" means to set it to Mbt.
- in *Line 3*, if each atom $p \in P$ is assigned, we have also to check, differently from SMODELS, that there is no atom $p \in P$ such that $val(p) =$ Mbt, i.e., no atom is assigned to Mbt. If this is the case, $\mathbf{T}(I)$ is an answer set of Π, and True is returned, otherwise False is returned.[11]

The following proposition states the correctness of DLV procedure.

Proposition 3 [32] *Let* DLV *be the procedure in Fig. 3. Then, for each program* Π, DLV(Π) *returns* True *if* Π *has an answer set, and* False *otherwise.*

[11] Note that, when dealing with *disjunctive* programs DLV calls *StabilityCheck*(Π, I), which checks the "stability" of the (candidate) answer set (composed by the atoms assigned to True by I) and returns (a) True if it is an answer set of Π, or (b) False otherwise. For the class of nondisjunctive logic programs studied in this work, *StabilityCheck*(Π, I) is not needed at all for DLV.

The above presentation of DLV is a recursive reformulation of the description of DLV provided in [13], pages 40–43, enhanced with the Well-founded operator for implementing *WFInf*, from [5], Section 4.1 .

As for CMODELS and SMODELS, the actual implementation of DLV features more complex look-ahead/look-back strategies and heuristic.

Remark 2 The atoms assigned to True by an interpretation I in DLV computation coincide with the minimal model of $\{r \in \Pi \mid val_I(body(r)) \geq \mathsf{Mbt}\}^{\emptyset}$, that is, the minimal model of the program obtained by deleting the negative bodies from rules r whose bodies are satisfied. Thus, True atoms are guaranteed to be "founded", $\mathbf{T}(I)$ collects all consequences of the reduct during the computation; while this property does not hold for Mbt atoms, because $\mathbf{M}(I)$ collects the residual atoms that must necessarily be True in order to let rules be satisfied, but are missing a founded justification.

Remark 3 CMODELS and SMODELS support types of rules other than (1), namely choice rules and cardinality and weight constraints rules [51], while DLV supports aggregates [16]. The relation in this paper has been studied using the common type of rule supported by the three systems.

Remark 4 About the implementation, SMODELS and DLV implement in a more efficient way some of the points presented in the algorithms, e.g., (a) the check about the tightness of the program (at line 9 of both procedures) is done once, and a flag is used; and (b) *WFInf* procedure does not work on the entire set of atoms but only on atoms involved in strongly connected components.

4 Relating CMODELS, SMODELS and DLV

In this section we study the relation among the ASP procedures presented in Section 3. We first formalize the notion of "equivalent" procedures, that is, we specify under which conditions two procedures are considered to be equivalent. Then, we present equivalence results on tight programs, and we eventually illustrate the results on nontight programs.

4.1 The notion of equivalent procedures

Our goal is to prove that the computations of SMODELS, DLV and CMODELS on a program Π are strongly related if Π is tight, while there are relevant differences among the computations if Π is not tight. We will compare the search trees of the three procedures on Π, i.e., the search trees of SMODELS-REC(Π, $J_{\emptyset}$), DLV-REC(Π, $J_{\emptyset}$) and DLL-REC($lp2sat(\Pi)$, $I_{\emptyset}$, Π), given J and I be an interpretation for CMODELS/SMODELS and DLV, respectively. These procedures operate in the signature of the input program and formula, respectively. A technical problem to be dealt with, to compare the search trees generated by the three procedures, is that the translation *lp2sat* introduces additional atoms which do not belong to the set P of atoms in Π. However, we know that, once all the atoms in P are assigned, also the additional atoms introduced by *lp2sat* will be assigned by *unit-propagate* in DLL-REC as proved in

[26]. Thus, in the following, we assume that *ChooseLiteral* is guaranteed to return the same, undefined, literal in $P \cup \overline{P}$, given Π and "same" interpretations, i.e., the additional atoms introduced by *lp2sat* are disregarded.

We say that a set X of literals is a *branching node of* SMODELS(Π)/CMODELS(Π)/DLV(Π), if, in the computation of SMODELS(Π)/CMODELS(Π)/DLV(Π), there is a call to SMODELS-REC(Π, J) /DLL-REC(*lp2sat*(Π), J, Π)/DLV-REC(Π, I), and X equals $\mathbf{T}(J) \cup \overline{\mathbf{F}(J)}$/$(\mathbf{T}(J) \cup \overline{\mathbf{F}(J)}) \cap (P \cup \overline{P})$/$\mathbf{T}(I) \cup \mathbf{M}(I) \cup \overline{\mathbf{F}(I)}$, respectively.

We then define *Branches*(*proc*) to be the sequence of branching nodes generated in the computation of procedure *proc* for program Π.

Finally, we say that *proc*1 and *proc*2 are *equivalent* if

$$Branches(proc1) = Branches(proc2).$$

4.2 Some useful lemmas

In this subsection, we state the first results of this work, by proving some general lemmas which state the relations among *unit-propagate*, *AtLeast* and *DetConsInf*. Then, in the subsections below devoted to tight and nontight programs, we show the main results on the comparison among CMODELS, SMODELS, and DLV algorithms, by exploiting these lemmas.

To start, we prove that *AtLeast* and *unit-propagate* compute corresponding results.

Lemma 1 *Let* Π *be a program,* I *and* I' *be two interpretations, such that* $\mathbf{F}(I) = \mathbf{F}(I') \cap P$ *and* $\mathbf{T}(I) = \mathbf{T}(I') \cap P$. *Then,* $\mathbf{T}(AtLeast(\Pi, I)) = \mathbf{T}(unit\text{-}propagate(lp2sat(\Pi), I')) \cap P$ *and* $\mathbf{F}(AtLeast(\Pi, I)) = \mathbf{F}(unit\text{-}propagate(lp2sat(\Pi), I')) \cap P$.

Proof The proof is by induction on $|\mathbf{T}(I) \cup \mathbf{F}(I)| = |(\mathbf{T}(I') \cup \mathbf{F}(I')) \cap P| = n$.

Base case. For $n = 0$, $I = I_\emptyset$, $I' = I'_\emptyset$. In *AtLeast*, and correspondingly in *unit-propagate*, there are two possibilities to make inference at this point:

a if "facts", i.e., rules of the type (1) with $m = n = 0$ exist in Π, which corresponds to an application of rule (14) in *AtLeast* where the *body* is trivially true.
Assuming such rules r_i, $1 \leq i \leq q$ exist, each $p_0 := head(r_i)$ is then added to $\mathbf{T}(I)$.
Correspondingly, in DLL-REC, each such a rule corresponds to the clauses (2)–(3) of *lp2sat*(Π):

$$\{p_0, \overline{n}_r\}, \{n_r\}.$$

and n_r and p_0 are assigned by *unit-propagate* and added to $\mathbf{T}(I')$. The thesis thus holds.

b if, for a not yet false atom p_0 there is no rule having p_0 in the head. Assuming such atoms exist, rule (15) applies and p_0 is added to $\mathbf{F}(I)$.
Correspondingly, in DLL-REC, clause (5) corresponds to:

$$\{\overline{p}_0\}$$

and $\overline{p}_0$ is assigned by *unit-propagate*, thus p_0 is added to $\mathbf{F}(I')$. The thesis thus holds.

If items *a* and *b* do not hold, the thesis trivially holds.

Step case. Consider two interpretations I and I' such that $|\mathbf{T}(I) \cup \mathbf{F}(I)| = |(\mathbf{T}(I') \cup \mathbf{F}(I')) \cap P| = n$, and the thesis holds for n. In the following, we show that each rule application in *AtLeast* corresponds to the same assignments in *unit-propagate*, and vice versa. Ultimately, given the hypothesis, the two procedures assign the same atoms in Π.

Consider now each line of *AtLeast*. If rule

(14) applies, $p_0 = head(r)$ is added to $\mathbf{T}(I)$, because $\{p_1, \ldots, p_m\} \subseteq \mathbf{T}(I)$ and $\{p_{m+1}, \ldots, p_n\} \subseteq \mathbf{F}(I)$.
Correspondingly, in DLL-REC, (a) n_r is assigned by *unit-propagate* and added to $\mathbf{T}(I')$ from clause (3), (b) p_0 is assigned by *unit-propagate* and added to $\mathbf{T}(I')$ from clause (2), and (c) clause (5) is simplified (note that all clauses of type (4) were previously deleted). The thesis thus holds.

(15) applies, p_0 is added to $\mathbf{F}(I)$.
Correspondingly, in DLL-REC, clause (5) is $\{\overline{p}_0\}$: thus $\overline{p}_0$ is assigned by *unit-propagate* and p_0 is added to $\mathbf{F}(I')$. The thesis thus holds.

(16) applies, already $p_0 \in \mathbf{T}(I)$, $body(r)$ is not satisfied, and all the undefined literals in $body$ are added to I, i.e., the undefined atoms in the positive part are added to $\mathbf{T}(I)$ and the undefined atoms in the negative part are added to $\mathbf{F}(I)$.
Correspondingly, in DLL-REC, the clause (5) is of the form $\{n_r\}$, while clause (3) and the set of clauses of type (4) have been reduced in accordance to I, and (2) previously deleted. Thus (a) n_r is assigned by *unit-propagate* and added to $\mathbf{T}(I')$, (b) clause (3) is deleted, and (c) all the still remaining clauses of type (4) (i.e., the ones for which $p_i \notin (\mathbf{T}(I) \cup \mathbf{F}(I))$, $1 \leq i \leq n$) are assigned by *unit-propagate*, the various p_i added to the related set, and the thesis again holds.

(17) applies, $head(r) \in \mathbf{F}(I)$, $\{p_1, \ldots, p_m, \overline{p}_{m+1}, \ldots, \overline{p}_n\} \setminus (\mathbf{T}(I) \cup \overline{\mathbf{F}(\mathbf{I})}) = \{l\}$, l is added to $\mathbf{F}(I)$ if $l = p$, and $\overline{l}$ is added to $\mathbf{T}(I)$, otherwise.
Correspondingly, in DLL-REC, clause (3) is unit because $\{n_r, p_1, \ldots, p_m, \overline{p}_{m+1}, \ldots, \overline{p}_n\} \setminus (\mathbf{T}(I) \cup \overline{\mathbf{F}(\mathbf{I})}) = \{l\}$ (clause (5) was canceled, $\overline{n}_r$ was assigned by *unit-propagate* from clause (2) thus erasing clauses (4)). Thus, the thesis holds, because again l is added to $\mathbf{F}(I')$ if $l = p$, and $\overline{l}$ is added to $\mathbf{T}(I')$, otherwise.

Conversely, consider that a unit clause appears in $P \cup \overline{P}$. This can be because:

1. clause (2) is unit: this is possible if n_r has been assigned to True by clause (3) because $\{p_1, \ldots, p_m, \overline{p}_{m+1}, \ldots, \overline{p}_n\} \subseteq (\mathbf{T}(I) \cup \overline{\mathbf{F}(\mathbf{I})})$, and p_0 is unit and is added to $\mathbf{T}(I')$.
   Correspondingly, in *AtLeast*, p_0 is assigned and added to $\mathbf{T}(I)$ because rule (14) is applied, and thus the thesis holds.
2. clause (3) is unit: this is possible if n_r has been assigned to False, and $\{p_1, \ldots, p_m, \overline{p}_{m+1}, \ldots, \overline{p}_n\} \setminus (\mathbf{T}(I') \cup \overline{\mathbf{F}(\mathbf{I}')}) = \{l\}$, thus l is unit, and it is added to $\mathbf{F}(I')$ if $l = p$, otherwise $\overline{l}$ is added to $\mathbf{T}(I')$. Note that in this case n_r could have been assigned to False only because of clause (2), because all clauses in (4) but the one containing l are deleted, thus $p_0 \in \mathbf{F}(I')$.
   Correspondingly, in *AtLeast*, rule (17) is applied, l is added to $\mathbf{F}(I)$ if $l = p$, and $\overline{l}$ is added to $\mathbf{T}(I)$ otherwise, and the thesis again holds.
3. (at least) one of the clauses in (4) is unit: this is possible if n_r has been assigned to True, thus each p_i, $1 \leq i \leq m$ (resp. p_j, $m + 1 \leq j \leq n$) which are not assigned

by I', is assigned to True (resp. False), and added to $\mathbf{T}(I')$ (resp. $\mathbf{F}(I')$). Note that, clause (2) was canceled by p_0, thus the only possibility to assign n_r was from clause (5), which did hold because only one rule with p_0 as the *head* was not canceled by $\Pi_{I'}$.
Correspondingly, in *AtLeast*, rule (16) applies, and the undefined atoms in $posBody(r)$ are added to $\mathbf{T}(I)$ and the undefined atoms in $negBody(r)$ are added to $\mathbf{F}(I)$, thus the thesis holds.

4. clause (5) is unit: this is possible if each n_{r_i}, $1 \leq i \leq q$ has been assigned to False, thus $\{\overline{p}_0\}$ is unit and p_0 is added to $\mathbf{F}(I')$.
Correspondingly, in *AtLeast*, all rules which could support p_0 to be assigned have been deleted in Π_I, because the n_{r_i} variables can be assigned to False only from clauses (4), i.e., the related rule was canceled. Thus, rule (15) applies, and p_0 is added to $\mathbf{F}(I)$. □

We continue with a lemma confirming that a program is simplified (i.e., rules are canceled) "accordingly" by SMODELS and DLV, if all Mbt literals in DLV correspond to True atoms in SMODELS.

Lemma 2 *Let* Π *be a program,* J *and* I *be an interpretation for* SMODELS *and* DLV, *respectively, such that* $\mathbf{F}(J) = \mathbf{F}(I)$ *and* $\mathbf{T}(J) = \mathbf{T}(I) \cup \mathbf{M}(I)$. *Then,* $\Pi_J = \Pi_I$.

Proof It follows from the definitions of Π_J and Π_I. □

Lemma 2 allows us, when working on the relation between SMODELS and DLV, and when its hypothesis hold (and this will be always the case), to consider that if a rule $r \in \Pi_J$, this implies $r \in \Pi_I$; and vice versa. Ultimately, we do not have to check this condition when comparing, e.g., the application of rules in *AtLeast* vs. *DetConsInf*.

We now prove a lemma which shows the "equivalence" between *AtLeast* and *DetConsInf* - its DLV analogous.

Lemma 3 *Let* Π *be a program,* J *and* I *be an interpretation for* SMODELS *and* DLV, *respectively, such that* $\mathbf{F}(J) = \mathbf{F}(I)$ *and* $\mathbf{T}(J) = \mathbf{T}(I) \cup \mathbf{M}(I)$. *Then,* $\mathbf{F}(AtLeast(\Pi, J)) = \mathbf{F}(DetConsInf(\Pi, I))$ *and* $\mathbf{T}(AtLeast(\Pi, J)) = \mathbf{T}(DetConsInf(\Pi, I)) \cup \mathbf{M}(DetConsInf(\Pi, I))$.

Proof Let n be the number of atoms assigned to True or to False in SMODELS and, respectively, assigned to True, Mbt or False in DLV, i.e., $n = |\mathbf{T}(J)| + |\mathbf{F}(J)| = |\mathbf{T}(I)| + |\mathbf{M}(I)| + |\mathbf{F}(I)|$. The proof is by induction on n.

Base Case. $n = 0$. In both *AtLeast* and *DetConsInf*, there are only two possibilities to make inferences at this point:

a if "facts", i.e., rules of type (1) with $m = n = 0$ exists in Π, which corresponds to an application of rules (14) in *AtLeast* and *DetConsInf* where the *body* is trivially true.

Assuming such rules r_i, $1 \leq i \leq q$, exist, each $p_0 := head(r_i)$ is then added to $\mathbf{T}(J)$ by *AtLeast* and to $\mathbf{T}(I)$ by *DetConsInf*, and the thesis holds.

b if, for an not yet false atom p_0 there is no rule having p_0 in the head. Assuming such atoms exist, rule (15) applies in both *AtLeast* and *DetConsInf*, where p_0 is added to both $\mathbf{F}(J)$ and $\mathbf{F}(I)$, and the thesis holds.

If items *a* and *b* do not hold, then the thesis trivially holds.

Step case. Suppose now that $n = |\mathbf{T}(J)| + |\mathbf{F}(J)| = |\mathbf{T}(I)| + |\mathbf{M}(I)| + |\mathbf{F}(I)|$ and the thesis holds for n. In the following, we show that each rule application in *AtLeast* corresponds to a rule application in *DetConsInf*, and vice versa, and they make corresponding assignments. Ultimately, given the hypothesis, the two procedures assign the same set of atoms with corresponding values. Since the hypothesis of Lemma 2 hold, in the following we assume that Π_J and Π_I coincide and the two procedures deal precisely with the same sets of rules.

Consider now each line of *AtLeast*. If rule

(14) applies, $J + true(head(r))$ is assigned because $head(r) \notin \mathbf{T}(J)$, $posBody(r) \subseteq \mathbf{T}(J)$ and $negBody(r) \subseteq \mathbf{F}(J)$.
Correspondingly, in *DetConsInf*, for each $p_i, 1 \leq i \leq m$, $val(p_i) \geq \mathsf{Mbt}$ and for each $p_j, m+1 \leq j \leq n$, $val(p_j) = \mathsf{False}$, $val(head(r)) < \mathsf{Mbt}$ thus $val(body(r)) > val(head(r))$ holds. Thus, if each p_i is such that $val(p_i) = \mathsf{True}$, $val(body(r)) = \mathsf{True}$ and $I + true(head(r))$ is assigned otherwise $val(body(r)) = \mathsf{Mbt}$ and $I + mbt(head(r))$ is assigned by rule (14), and the thesis holds in both cases.

(15) applies, $val(p)$ is not False, $\nexists r \in \Pi_J$ such that $head(r) = p$ and $J + false(p)$ is assigned.
Correspondingly, $val(p)$ is not False by induction hypothesis, $\nexists r \in \Pi_I$ such that $head(r) = p$ also in *DetConsInf* by Lemma 2, thus $I + false(p)$ is assigned by rule (15) in the DLV algorithm, keeping the same behavior of *AtLeast*. The thesis thus holds.

(16) applies, $head(r) \in \mathbf{T}(J)$, for each $p \in posBody(r)$, $p \in \mathbf{T}(J) \cup \mathbf{U}(J)$ and for each $p' \in negBody(r)$, $p \in \mathbf{F}(J) \cup \mathbf{U}(J)$. Thus, from *assignToTrue*, for each $p \in posBody(r)$ s.t. $p \in \mathbf{U}(J)$, $J + true(p)$ is assigned, and, from *assignToFalse*, for each $p' \in negBody(r)$ s.t. $p' \in \mathbf{U}(J)$, $J + false(p')$ is assigned.
Correspondingly, in *DetConsInf*, $val(head(r)) \geq \mathsf{Mbt}$ and each $p \in posBody(r)$ with $val(p) \neq \mathsf{Undef}$ is such that $val(p) \geq \mathsf{Mbt}$, and each $p' \in negBody(r)$, with $val(p') \neq \mathsf{Undef}$ is such that $val(p') = \mathsf{False}$. Consequently, by rule (16), from *assignToMBT*, for each $p \in posBody(r)$ s.t. $val(p) = \mathsf{Undef}$, $I + mbt(p)$ is assigned, and, from *assignToFalse*, for each $p' \in negBody(r)$ s.t. $val(p') = \mathsf{Undef}$, $I + false(p')$ is assigned, and the thesis again holds.

(17) applies, $head(r) \in \mathbf{F}(J)$ and $(body(r) \setminus (\mathbf{T}(J) \cup \overline{\mathbf{F}(J)})) = \{l\}$. Thus, if $l = p$, with $p \in posBody(r)$ then $J + false(p)$ is assigned, otherwise $J + true(p)$ is assigned.
Correspondingly, in *DetConsInf*, $val(head(r)) = \mathsf{False}$, $val(body(r) \setminus \{l\}) \geq \mathsf{Mbt}$, and thus if $l = p$, with $p \in posBody(r)$ then, by rule (17), $I + false(p)$ is assigned, otherwise $I + mbt(p)$ is assigned, and the thesis still holds.

Conversely, consider now each line of *DetConsInf*. If rule

(14) applies, each $p \in negBody(r)$ is such that $val(p) = \mathsf{False}$, and for each p_i, $1 \leq i \leq m$, it is true that $p_i \in \mathbf{T}(I) \cup \mathbf{M}(I)$, i.e., $val(p_i) \geq \mathsf{Mbt}$. There are three options for the head: (*a*) $val(head(r)) = \mathsf{False}$; (*b*) $val(head(r)) = \mathsf{Undef}$; or (*c*) $val(head(r)) = \mathsf{Mbt}$.
In the case (*a*) or (*b*), if each p_i, $1 \leq i \leq m$, is such that $val(p_i) = \mathsf{True}$, $I + true(head(r))$ is assigned, otherwise $I + mbt(head(r))$ is assigned.
Correspondingly, in *AtLeast*, $head(r) \notin \mathbf{T}(J)$, $\{p_1, \ldots, p_m\} \subseteq \mathbf{T}(J)$ and $\{p_{m+1}, \ldots, p_n\} \subseteq \mathbf{F}(J)$, thus $J + true(head(r))$ is assigned by rule (14), and the thesis still holds.
In the case (*c*), from the second condition $val(body(r)) > val(head(r))$ we know that $val(body(r)) = \mathsf{True}$, thus $I + true(head(r))$ is assigned, Note that this assignment does not alter the set $\mathbf{T}(I) \cup \mathbf{M}(I)$ and, obviously, $\mathbf{F}(I)$.
Correspondingly, in *AtLeast* no rule is applied, and the thesis holds.

(15) applies, $val(p)$ is not False, $\nexists r \in \Pi_I$ such that $head(r) = p$ and $I + false(p)$ is assigned.
Correspondingly, $val(p)$ is not False by induction hypothesis, $\nexists r \in \Pi_J$ such that $head(r) = p$ also in *AtLeast* by Lemma 2, thus $J + false(p)$ is assigned by rule (15) in the SMODELS algorithm, keeping the same behavior of *DetConsInf*. The thesis thus holds.

(16) applies, $val(head(r)) \geq \mathsf{Mbt}$, for each $p \in posBody(r)$ s.t. $val(p) \neq \mathsf{Undef}$, $val(p) \geq \mathsf{Mbt}$, and each $p' \in negBody(r)$, with $val(p') \neq \mathsf{Undef}$ is such that $val(p') = \mathsf{False}$. Consequently, from *assignToMBT*, for each $p \in posBody(r)$ s.t. $val(p) = \mathsf{Undef}$, $I + mbt(p)$ is assigned, and, from *assignToFalse*, for each $p' \in negBody(r)$ s.t. $val(p') = \mathsf{Undef}$, $I + false(p')$ is assigned.
Correspondingly, $head(r) \in \mathbf{T}(J)$, for each $p \in posBody(r)$, $p \in \mathbf{T}(J) \cup \mathbf{U}(J)$ and for each $p' \in negBody(r)$, $p \in \mathbf{F}(J) \cup \mathbf{U}(J)$. Thus, from *assignToTrue*, for each $p \in posBody(r)$ s.t. $p \in \mathbf{U}(J)$, $J + true(p)$ is assigned, and, from *assignToFalse*, for each $p' \in negBody(r)$ s.t. $p' \in \mathbf{U}(J)$, $J + false(p')$ is assigned, and the thesis again holds.

(17) applies, $val(head(r)) = \mathsf{False}$, $val(body(r) \setminus \{l\}) \geq \mathsf{Mbt}$ and thus, if $l = p$, with $p \in posBody(r)$, $I + false(p)$ is assigned, otherwise $I + mbt(p)$.
Correspondingly, in *AtLeast*, $head(r) \in \mathbf{F}(J)$, $(body(r) \setminus (\mathbf{T}(J) \cup \overline{\mathbf{F}(J)})) = \{l\}$ and thus, by rule (17), if $l = p$, with $p \in posBody(r)$ then p is added to $\mathbf{F}(J)$, otherwise is added to $\mathbf{T}(J)$, and the thesis holds. □

We finally show a lemma which states the "equivalence" between *lp2sat* and *DetConsInf*.

Lemma 4 *Let* Π *be a program,* J *and* I *be an interpretation for* CMODELS *and* DLV, *respectively, such that* $\mathbf{F}(I) = \mathbf{F}(J) \cap P$ *and* $\mathbf{T}(I) \cup \mathbf{M}(I) = \mathbf{T}(J) \cap P$. *Then,* $\mathbf{F}(DetConsInf(\Pi, I)) = \mathbf{F}(unit\text{-}propagate(lp2sat(\Pi), J)) \cap P$ *and* $\mathbf{T}(DetConsInf(\Pi, I)) \cup \mathbf{M}(DetConsInf(\Pi, I)) = \mathbf{T}(unit\text{-}propagate(lp2sat(\Pi), J)) \cap P$.

Proof It follows from Lemma 1 and 3. □

4.3 Tight programs

In this subsection, we show equivalence results for CMODELS, SMODELS and DLV on tight programs. We reach such results by comparing two systems at a time. We devote one subsection to each comparison, where we prove equivalence (in the sense explained in Section 4.1) of the search trees of the systems involved. We also show some complexity results that hold for CMODELS, SMODELS and DLV on some classes of tight programs.

4.3.1 Relating CMODELS *and* SMODELS

By using Lemma 1, we are now ready to state and prove our main result on the relation between CMODELS and SMODELS procedures.

Theorem 2 *Let* CMODELS *and* SMODELS *be the procedures in Figs.* 1 *and* 2*, respectively. Then, for each tight program* Π*,* CMODELS(Π) *and* SMODELS(Π) *are equivalent.*

Proof We have to show that

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

where *Branches*(SMODELS(Π)) is a sequence of branching nodes $s_1, \ldots, s_n$ and *Branches*(CMODELS(Π)) is the sequence $s'_1, \ldots, s'_n$.

The proof is by induction on n.

Base case. For $n = 0$, the thesis obviously holds, since both s_1 and s'_1 equal $\emptyset$ (they are the nodes of the initial calls SMODELS-REC(Π, $I_\emptyset$) and DLL-REC(*lp2sat*(Π), Π, $I'_\emptyset$)).

Step case. By induction hypothesis we know that $s_n = s'_n$ holds. We have to show that $s_{n+1} = s'_{n+1}$ holds as well.

Let I_1 and I'_1 to be the interpretations corresponding to the branching nodes s_n and s'_n, respectively, such that $\mathbf{F}(I_1) = \mathbf{F}(I'_1) \cap P$ and $\mathbf{T}(I_1) = \mathbf{T}(I'_1) \cap P$. We next trace, step by step, the computations of SMODELS-REC(Π, I_1) and DLL-REC(Γ, Π, I'_1), showing that they lead to equal branching nodes s_{n+1} and s'_{n+1}.

a SMODELS-REC(Π, I_1) calls *expand*(Π, I_1), which returns $I_2 := AtLeast(\Pi, I_1)$. Indeed: (a) the instructions of lines 10 and 11 in the algorithm of SMODELS (Fig. 2) are not executed since Π is tight; (b) the recursive call to *expand* (line 12) is not executed because the **if** condition does not hold; (c) the **return** statement at line 13 is executed. DLL-REC(Γ, Π, I'_1) generates $I'_2 :=$ *unit-propagate*(Γ, I'_1). Thus, from Lemma 1, we can conclude that $\mathbf{T}(I_2) = \mathbf{T}(I'_2) \cap P$ and $\mathbf{F}(I_2) = \mathbf{F}(I'_2) \cap P$ hold.

b Then, both procedures execute precisely the same instruction (namely, instruction 2). If $(\mathbf{T}(I_2) \cap \mathbf{F}(I_2)) \neq \emptyset$, from our hypothesis and the structure of *lp2sat* an empty clause appears in $\Gamma_{I'_2}$, and vice versa: thus both algorithms backtrack; otherwise step to the respective instruction 3.

c if $(P \setminus (\mathbf{T}(I_2) \cup \mathbf{F}(I_2)) = \emptyset)$ in SMODELS, obviously the same holds for CMODELS w.r.t. I'_2. Moreover, by definition, $test(\Pi, I'_2)$ returns True on tight programs. Thus both algorithms returns True.

d If the above mentioned **if** conditions of the instructions 3 are not satisfied, then both procedures choose a literal l (which is precisely the same and belongs to $P \cup \overline{P}$, as we assumed that the same heuristic is used), and call SMODELS-REC(Π, $I_2 + true(l)$) and DLL-REC(Γ, Π, $I'_2 + true(l)$), respectively. Thus, the next branching nodes of the two procedures are $s_{n+1} = \mathbf{T}(I_2 + true(l)) \cup \overline{\mathbf{F}(I_2 + true(l))}$ and $s'_{n+1} = (\mathbf{T}(I'_2 + true(l)) \cup \overline{\mathbf{F}(I'_2 + true(l))}) \cap (P \cup \overline{P})$. We can therefore conclude that $s_{n+1} = s'_{n+1}$ (i.e., the induction is proven), as $l \in P \cup \overline{P}$ and we know that $\mathbf{T}(I_2) = \mathbf{T}(I'_2) \cap P$ and $\mathbf{F}(I_2) = \mathbf{F}(I'_2) \cap P$ from *item b* above. □

In the following we comment about the impact of restricting CMODELS heuristic to a subset of the atoms that appear in $lp2sat(\Pi)$.

From one side, this restriction is not a limitation, not only because of the results in [26] we cited above, but also because we can reach similar results to Lemma 1 and Theorem 2 by considering a (simple) modification of the logic program at hand, in the spirit of the one in [3], and by avoiding to limit CMODELS' splitting heuristic.

Consider a rewriting Π' of a program Π with $|\Pi|$ rules where, for each rule $r \in \Pi$ of type (1) there exist two rules in Π':

1. $p_0 \leftarrow n_{r_i}$.
2. $n_{r_i} \leftarrow p_1, \ldots, p_m, \overline{p}_{m+1}, \ldots \overline{p}_n$.

with $1 \leq i \leq |\Pi|$, where n_{r_i} is a newly introduced atom for the rule.

Let Π be a program, Π' be the program constructed from Π as showed before, and I be an interpretation. Then, $AtLeast(\Pi', I) = unit\text{-}propagate(lp2sat(\Pi), I)$.

Such a result could be then used for an alternative, but similar, proof of Theorem 2.

But from another side, instead, the restriction is a significant limitation. CMODELS, if not restricted to choose on the atoms in Π, can be implicitly seen as a procedure which allows the decision of the heuristic not only on atoms, but also on bodies, if an atom n_r is selected. Given this, [2, 3] and, in particular, Corollary 1 in [18] show that by not restricting to the atoms in Π can significantly contribute to the efficiency of the procedure.

In the relation between CMODELS and SMODELS, we have considered CMODELS heuristic restricted to atoms in Π, because of our assumption on the branching heuristics. Obviously, the same holds in the comparison with DLV.

4.3.2 Relating SMODELS *and* DLV

By using Lemma 3 (and Lemma 2), we can state and prove our main result on the relation between SMODELS and DLV procedures on tight programs.

Theorem 3 *Let* SMODELS *and* DLV *be the procedures in Figs. 2 and 3, respectively. Then, for each tight program* Π, SMODELS(Π) *and* DLV(Π) *are equivalent.*

Proof We have to show that

$$Branches(\text{SMODELS}(\Pi)) = Branches(\text{DLV}(\Pi)),$$

where $Branches(\text{SMODELS}(\Pi))$ is a sequence of branching nodes $s_1, \ldots, s_n$ and $Branches(\text{DLV}(\Pi))$ is the sequence $s'_1, \ldots, s'_n$.

The proof is by induction on n.

Base case. For $n = 0$, the thesis obviously holds, since both s_1 and s'_1 equal $\emptyset$ (they are the nodes of the initial calls SMODELS-REC(Π, $J_\emptyset$) and DLV-REC(Π, $I_\emptyset$)).

Step case. By induction hypothesis we know that $s_n = s'_n$ holds. We have to show that $s_{n+1} = s'_{n+1}$ holds as well.

Let J_1 (resp. I_1) to be the interpretation for SMODELS (resp. DLV) corresponding to the branching node s_n (resp. s'_n). We next trace, step by step, the computations of SMODELS-REC(Π, J_1) and DLV-REC(Π, I_1), showing that they lead to equals branching node s_{n+1} and s'_{n+1}.

a SMODELS-REC(Π, J_1) calls *expand*(Π, J_1), which returns $J_2 := AtLeast(\Pi, J_1)$. Indeed: (a) the instructions of lines 10 and 11 in the algorithm of SMODELS (Fig. 2) are not executed since Π is tight; (b) the recursive call to *expand* (line 12) is not executed because the **if** condition does not hold; (c) the **return** statement at line 13 is executed. DLV-REC(Π, I_1) calls *DetCons*(Π, I_1), which returns $I_2 := DetConsInf(\Pi, I_1)$. Indeed: (a) the instructions of lines 10 and 11 in the algorithm of DLV (Fig. 3) are not executed since Π is tight; (b) the recursive call to *DetCons* (line 12) is not executed because the **if** condition does not hold; (c) the **return** statement at line 13 is executed. Thus, from Lemma 3, we can conclude that $\mathbf{F}(J_2) = \mathbf{F}(I_2)$ and $\mathbf{T}(J_2) = \mathbf{T}(I_2) \cup \mathbf{M}(I_2)$ hold.

b Then, both procedures execute precisely the same instruction (namely, instruction 2). If $(\mathbf{T}(J_2) \cap \mathbf{F}(J_2)) \neq \emptyset$ in SMODELS-REC, this means that the same atom p belongs to both $\mathbf{T}(J_2)$ and $\mathbf{F}(J_2)$. From item *a*, this means that this atom p belongs to both $\mathbf{T}(I_2) \cup \mathbf{M}(I_2)$ and $\mathbf{F}(I_2)$ in DLV-REC, and thus both algorithms backtrack; otherwise the procedures step to the respective instruction 3.

c if $(P \setminus (\mathbf{T}(J_2) \cup \mathbf{F}(J_2)) = \emptyset)$ in SMODELS-REC the procedure returns True. In this case, $\mathbf{M}(I_2) = \emptyset$ in DLV-REC, as we know that $\mathbf{F}(J_2) = \mathbf{F}(I_2)$ and $\mathbf{T}(J_2) = \mathbf{T}(I_2) \cup \mathbf{M}(I_2)$ from *item a* above. We next show that no atom in I_2 is assigned to Mbt and DLV-REC returns True.
From Remark 2, it follows that $\mathbf{T}(I_2) = \{r \in \Pi \mid val_{I_2}(body(r)) \geq \mathsf{Mbt}\}^{\mathbf{T}(I_2) \cup \mathbf{M}(I_2)}$.[12] Since $\mathbf{F}(J_2) = \mathbf{F}(I_2)$ and $\mathbf{T}(J_2) = \mathbf{T}(I_2) \cup \mathbf{M}(I_2)$, we have that $\mathbf{T}(I_2) = \{r \in \Pi | body(r) \subseteq (\mathbf{T}(J_2) \cup \overline{\mathbf{F}(J_2)})\}^{\mathbf{T}(J_2)}$, which is, in turn, equal to $\mathbf{T}(J_2)$ since we know that J_2 is an answer set from the correctness of SMODELS procedure. Consequently, $\mathbf{M}(I_2)$ is empty, and DLV-REC returns True.

d If the above mentioned **if** conditions of the instructions 3 are not satisfied, then both procedures choose a literal l (which is precisely the same from

[12] Note that, since I_2 is consistent, $\{r \in \Pi \mid val_{I_2}(body(r)) \geq \mathsf{Mbt}\}^{\emptyset}$ is equal to $\{r \in \Pi \mid val_{I_2}(body(r)) \geq \mathsf{Mbt}\}^{\mathbf{T}(I_2) \cup \mathbf{M}(I_2)}$.

our assumptions), and call SMODELS-REC($\Pi, J_2 + true(l)$) and DLV-REC($\Pi, I_2 + mbt(l)$), respectively. Thus, the next branching nodes of the two procedures are $s_{n+1} = \mathbf{T}(J_2 + true(l)) \cup \overline{\mathbf{F}(J_2 + true(l))}$ and $s'_{n+1} = \mathbf{T}(I_2 + mbt(l)) \cup \mathbf{M}(I_2 + mbt(l)) \cup \overline{\mathbf{F}(I_2 + mbt(l))}$. We can therefore conclude that $s_{n+1} = s'_{n+1}$ (i.e., the induction is proven), as we know that $\mathbf{F}(J_2) = \mathbf{F}(I_2)$ and $\mathbf{T}(J_2) = \mathbf{T}(I_2) \cup \mathbf{M}(I_2)$ from *item a* above. □

It should be also noted that, on tight programs, *WFInf* does not make inferences.

4.3.3 Relating CMODELS *and* DLV

By using Theorems 2 and 3, we can now state a result on the relation between CMODELS and DLV procedures, on tight programs.

Corollary 1 *Let* CMODELS *and* DLV *be the procedures in Figs. 1 and 3, respectively. Then, for each tight program* Π, CMODELS(Π) *and* DLV(Π) *are equivalent.*

The above corollary readily follows from Theorems 2 and 3.

Remark 5 Here we would like to underline that the use of a further value to be assigned to atoms by DLV is in some sense "too powerful" for tight programs, because it does not bring any relevant advantage. We will see that this is not the case on nontight programs.

4.3.4 Some complexity results for ASP procedures on tight programs

Theorems 2 and 3 and Corollary 1 state a strong relation between CMODELS, SMODELS and DLV algorithms. The established correspondence between CMODELS, SMODELS and DLV, and the fact the CMODELS is based on DLL, i.e., one of the most studied procedures in Artificial Intelligence, gives us the possibility to derive new lower/upper bounds for the ASP systems involved. In particular, those bounds (a) are known for DLL, (b) thus immediately generalize to CMODELS, and (c) thanks to our results, can be shown to hold also for SMODELS and DLV.

First, observe that the search tree explored by CMODELS, SMODELS and DLV, when run on a program Π, critically depends on the specific heuristic used, i.e., in our terminology and with reference to Figs. 1, 2 and 3, by the ordering on the set $P \cup \overline{P}$ induced by *ChooseLiteral*. In order to highlight the dependency from the particular heuristic used, we now write $Branches^h(proc)$ to indicate the set of branching nodes of *proc* when run on a program Π, using heuristic h. We are now ready to define the *complexity of proc on a program* Π as the smallest number in

$$\{|Branches^h(proc)| : h \text{ is the heuristic which returns a value in } P \cup \overline{P}\}.$$

Intuitively, the complexity of *proc* on Π is the minimum number of branching nodes that *proc* has to explore for solving Π.

In the following, we show some of the new complexity results for ASP systems that can be derived thanks to our study and from the corresponding SAT references cited below in the explanation of the results.

We now define the translation of a set of clauses into a logic program we will use in the rest of the paper.

Definition 1 Given a clause $C = \{l_1, \ldots, l_l\}$ $(l \geq 0)$, *sat2tlp*(C) is the rule $f \leftarrow \bar{l}_1, \ldots, \bar{l}_l, \overline{f}$. Given a CNF formula Γ, the *translation of* Γ, denoted with *sat2tlp*(Γ), is composed by the following set of rules:

a $\cup_{C \in \Gamma} sat2tlp(C)$; and
b $\cup_{p \in P}\{p \leftarrow \overline{p'}, p' \leftarrow \overline{p}\}$, where, for each atom $p \in P$, p' is a newly introduced atom associated to p.

Γ is satisfiable iff *sat2tlp*(Γ) has an answer set. Note that, in the definition, *item a* corresponds to the direct translation of every clause in Γ into a rule, while *item b* is the "guessing" part, where values for the atoms are guessed. Moreover, note that *sat2tlp*(Γ) is tight.

We now define randomly generated formulas.

Definition 2 A formula Γ is a *k-CNF* if each clause in Γ consists of k literals. The *random family of k-CNF formulas* consists of k-CNFs 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, $\bar{l} \neq l'$.

The following result states the behavior of CMODELS, SMODELS and DLV on random k-CNF benchmarks with particular ratio between clauses and variables.

Proposition 4 *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* CMODELS, SMODELS *and* DLV *on sat2tlp*(Γ) *is exponential in n if the density* $d = m/n$ *is* $d \geq 0.7 \times 2^k$.

Proof Given that *sat2tlp*(Γ) is tight, the result holds for CMODELS starting from [6], and then it follows for SMODELS and DLV from Theorem 2 and Corollary 1. □

Programs corresponding to random k-CNF formulas have been widely used in literature, e.g., in [15, 27, 50, 54].

Other results on tight programs, which have been proved for DLL, can be easily shown to hold also for CMODELS (because it is based on DLL), and for SMODELS and DLV (thanks to our "equivalence" results on tight programs).

Following [34], we next define the notion of "optimal" literal to branch on, intuitively representing the best choice for the heuristic.

Definition 3 A literal l is defined as *optimal for a program* Π and procedure *proc* if there exists a minimal search tree of *proc* whose root is labeled with l.

Proposition 5 *In* CMODELS, SMODELS *and* DLV, *deciding the optimal literal to branch on is both NP-hard and co-NP hard, and in PSPACE for tight programs.*

Proof Given that we consider tight programs, the result holds for CMODELS starting from [34] where it is shown to hold for DLL, and then it follows for SMODELS and DLV from Theorem 2 and Corollary 1. □

In the following, we show a third result which, again, is known to hold for DLL and can be extended to CMODELS, SMODELS and DLV. We consider a class of CNF formulas that has been widely studied in the SAT literature.

Definition 4 Define PHP_n^m $(n \geq 0, m \geq 0)$ to be the CNF formula consisting of the following clauses:

$$\{p_{i,1}, p_{i,2}, \ldots, p_{i,n}\} \qquad (i \leq m), \qquad \{\overline{p}_{i,k}, \overline{p}_{j,k}\} \qquad (i, j \leq m, k \leq n, i \neq j).$$

The formulas PHP_n^m are from [28] and encode the pigeonhole principle. If $n < m$, PHP_n^m is unsatisfiable and it is well known that any procedure based on resolution (like DLL) has an exponential behavior. For each n, $sat2tlp(PHP_{n-1}^n)$ is tight and has no answer sets.

Proposition 6 *The complexity of* CMODELS, SMODELS *and* DLV *on* $sat2tlp(PHP_{n-1}^n)$ *is exponential in n.*

Proof This result holds for CMODELS starting from [28], and then it follows for SMODELS and DLV from Theorem 2 and Corollary 1. □

There are many other results, known for DLL, that can be easily shown to apply to CMODELS, and thanks to our results, could be applicable to SMODELS and DLV algorithms, e.g., [44] about the average complexity of coloring randomly generated graphs, and [1] on lower bounds on random 3-CNF formulas also for densities significantly below the satisfiability threshold $d \approx 4.23$.

Finally, we remind that [31] gives an alternative way for producing the complexity results presented in this subsection.

4.4 Nontight programs

In this subsection we compare CMODELS, SMODELS, and DLV on the class of nontight programs. Similarly to Subsection 4.3, at the end of the subsection we also show a result about the complexity of these systems on nontight programs.

To start, we observe that, considering J and I be an interpretation for SMODELS and DLV, respectively, such that $\mathbf{F}(J) = \mathbf{F}(I)$ and $\mathbf{T}(J) = \mathbf{T}(I) \cup \mathbf{M}(I)$, the *WFInf* functions in the SMODELS and DLV algorithms are invoked with the same parameters, thus return the same set.

Now we are ready to state another main theoretical result of our work, on nontight programs. We use the result about SMODELS and DLV procedures in Theorem 3, and the observation we made before.

Theorem 4 *Let* SMODELS *and* DLV *be the procedures in Figs. 2 and 3, respectively. Then, for each nontight program* Π, SMODELS(Π) *and* DLV(Π) *are equivalent.*

Proof Theorem 3 has already shown that SMODELS and DLV procedures are equivalent on tight programs. On nontight programs, the two procedures keep a very similar behaviour as in the tight case, the only difference being that instructions 10–11 are executed on nontight programs, as the "**if**" condition of instruction 9 is verified. *WFInf* (called in instruction 10 of both algorithms) computes precisely the same set of atoms, and both procedures switch the same set of atoms to False in instruction 11. □

We can also show that, on nontight programs, CMODELS is not equivalent to SMODELS and DLV. The reason of such difference is that CMODELS does not employ the well-founded operator, thus its negative inferences are weaker than the ones of SMODELS and DLV. The difference shows up when loops are involved in the program, and this fact has an impact also on the computational side: there exist families of logic programs which are easy for SMODELS and DLV, but "difficult" for CMODELS. An example will be presented below in Proposition 7.

However, it is important to note that, given that the difference is solely due to well-founded inferences, if we disable the well-founded operator in both SMODELS and DLV (by disabling lines 10–11 in Figs. 2 and 3, respectively), resulting in two systems that we call DLV^F and SMODELS^F in the next sections, then the three systems are "equivalent" on nontight programs (apart for the presence of Mbt atoms in DLV, which allow for some properties of the DLV algorithm as shown in Theorem 7 below).

In particular, Lemma 4 holds as well if we substitute DLV with DLV^F, and it could be then used to state the equivalence between CMODELS and DLV^F also on nontight programs.

Regarding this last result, we have nonetheless to stress, about the usefulness of Mbt atoms, that the difference between Mbt and True atoms allows to perform several optimizations, e.g., in [14] Mbt atoms are effectively used to guide the DLV heuristic, besides the fact that Mbt atoms also allow to avoid useless "stability" checks both on disjunctive and on nondisjunctive programs when dealing with consistent leaf interpretations, as we will see in the next section.

4.4.1 Some complexity results for ASP procedures on nontight programs

In this subsection, we infer new complexity results for SMODELS and DLV (and CMODELS) on nontight programs.

First note that if we consider the logic program which corresponds to only the rules in *item a* of the Definition 1 (i.e., no guess), both CMODELS, SMODELS and DLV solve the program easily.

In the following definition, instead, we define a logic program which is exponentially more difficult for CMODELS in comparison to SMODELS and DLV.

Definition 5 Given a CNF formula Γ, define $sat2nlp(\Gamma)$ to be the program $\cup_{C\in\Gamma} sat2tlp(C) \cup \cup_{p\in P}\{p \leftarrow p\}$.

The following result states the complexity of CMODELS, SMODELS and DLV on the *sat2nlp*(Γ) program.

Proposition 7 *The complexity of* SMODELS *and* DLV *on sat2nlp*(PHP^n_{n-1}) *is* 3*, while it is exponential in n for* CMODELS.[13]

Proof For CMODELS, the result again follows from the behavior of DLL on PHP^n_{n-1} formulas [28]. For SMODELS and DLV, the result holds thanks to the *WFInf* procedure, as we show in the following.

To see why this is the case for SMODELS notice that $\Pi = sat2nlp(PHP^n_{n-1})$ is nontight and corresponds to the set of rules (where P denotes now the set of atoms of type $p_{i,j}$):

$$\begin{array}{l} f \leftarrow \overline{p}_{i,1}, \overline{p}_{i,2}, \ldots, \overline{p}_{i,n-1}, \overline{f} \qquad (1 \leq i \leq n), \\ f \leftarrow p_{i,k}, p_{j,k}, \overline{f} \qquad (1 \leq i, j \leq n, 1 \leq k \leq n-1, i \neq j) \\ p_{i,k} \leftarrow p_{i,k} \qquad (1 \leq i \leq n, 1 \leq k \leq n-1) \end{array}$$

Then, following SMODELS behavior step by step:

- SMODELS(Π) calls SMODELS-REC($\Pi, J_\emptyset$).
- $AtLeast(\Pi, J_\emptyset)$ returns $\emptyset = J$ (line 7).
- $J' = J = \emptyset$ in *AtLeast*.
- Π is nontight, thus *WFInf* is called (line 10) where $\Pi^\emptyset_J$ consists of the rules

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

  and thus $WFInf(\Pi^\emptyset_J, \emptyset)$ returns $\{f\}$ at line 20.
- At line 11 of Fig. 2, $\mathbf{F}(J) = P$ and $\mathbf{T}(J) = \emptyset$.
- $J' \neq J$ causes a recursive call to $expand(\Pi, J)$ (line 12).
- Π_J corresponds to the set of rules

$$f \leftarrow \overline{p}_{i,1}, \overline{p}_{i,2}, \ldots, \overline{p}_{i,n-1}, \overline{f} \qquad (1 \leq i \leq n)$$

  and $AtLeast(\Pi, J)$ returns the same interpretation (line 7).
- $J' = J$.
- $\Pi^\emptyset_J$ consists of the rule $f.$. but $J = J'$, thus J is returned by *expand* at line 13 and assigned at line 1 of Fig. 2.
- Now, the conditions at line 2 and 3 do not hold, the literal $l = f$ is chosen at line 4, and SMODELS-REC($\Pi, J + true(f)$), $expand(\Pi, J + true(f))$ and $AtLeast(\Pi, J + true(f))$ are called in sequence;
- in $AtLeast(\Pi, J + true(f))$, $\Pi_{J+true(f)}$ is the empty program: thus there is no rule with f in the head (and f is not false), and f is added to $\mathbf{F}(J)$;

[13] Rules like $p \leftarrow p$ can be removed during pre-processing, resulting in CMODELS having same complexity. However, we could have replaced $p \leftarrow p$ with, e.g., $p \leftarrow p'$, $p' \leftarrow p$, (where p' is a newly introduced atom associated to p) and the result in the proposition would still hold.

– after one recursive call to *AtLeast*, J is checked for consistency at line 2 of SMODELS algorithm, and False is returned because an inconsistency is detected given that f appears in both $\mathbf{F}(J)$ and $\mathbf{T}(J)$;
– upon backtracking, $AtLeast(\Pi, J + false(f))$ is invoked, and $\Pi_{J+false(f)}$ corresponds to the set of rules

$$f \leftarrow \overline{p}_{i,1}, \overline{p}_{i,2}, \ldots, \overline{p}_{i,n-1}, \overline{f} \qquad (1 \leq i \leq n),$$

which, from rule (14), assigns f to True from $J + true(f)$. Another inconsistency will be detected, and the algorithm stops returning False.

Thus, *Branches*(SMODELS) is the following sequence of branching nodes: $\emptyset$, $\overline{P} \cup f$, $\overline{P} \cup \overline{f}$; and the complexity of SMODELS on $sat2nlp(PHP^n_{n-1})$ is indeed 3. Similarly for DLV, by Theorem 4. □

Intuitively, on this program CMODELS algorithm is "confused" by the added loops, while SMODELS and DLV maintain their good behavior thanks to *WFInf*. On the theoretical side, the work of [37], which shows that, given a nontight program, exponentially many formulas might be required to discard all supported models that are not answer sets, explains the same effects of the CMODELS behavior in an alternative way.

5 On the need of loop detection

We discuss on some properties of the algorithms we have shown in the previous sections. In particular, we will point out that, when a leaf of the computation is reached (i.e., a total consistent interpretation is generated), the native ASP procedures do not need to make a stability check. The computed interpretation is guaranteed to be an answer set. This is not true for CMODELS, as for native procedures this feature is mainly due to the employment of the well-founded operator which, besides pruning the search space, implicitly performs also a check of the stability condition. We extend our analysis observing the behaviour of the native procedures when the well-founded operator is disabled and negative inferences are performed by Fitting's operator only (which is incorporated in both *DetConsInf* and *AtLeast*). It turns out that consistent leaf interpretations are still guaranteed to be answer sets for DLV; while they are not such for SMODELS. This relevant property of the DLV system is due to the exploitation of "must-be-true"—the fourth truth value of DLV.

To start, we formalize the concept of "consistent" leaf interpretation, which is of fundamental importance for the rest of this section.

Definition 6 A "consistent" leaf interpretation is a total[14] interpretation which is reached at line 3 of CMODELS, SMODELS or DLV computation.[15]

[14] An interpretation is total if all atoms are assigned either to True or to False.

[15] Note that in CMODELS, when each $p \in P$ is assigned (to True or to False), then also all the added atoms get assigned by unit propagation.

Note that, following the computation of the three algorithms, when they reach the instruction at line 3, we are guaranteed that there is no inconsistency in the interpretation, because the condition at line 2 was not satisfied. Moreover, for DLV, given a consistent leaf interpretation J^t, the condition that for each $p \in P$, $p \in \mathbf{T}(J^t) \cup \mathbf{F}(J^t)$, implies that no atom is assigned to Mbt.

Observe also that, given a consistent leaf interpretation J^t for SMODELS, CMODELS and DLV, for a program Π, a "candidate" answer set is the set $\mathbf{T}(J^t)$ for all the systems.

Our first result concerns tight programs.

Theorem 5 *Let Π be a tight program, and consider a consistent leaf interpretation J^t of Π. Then, the set of its true atoms $\mathbf{T}(J^t)$ is guaranteed to be an answer set of Π for* DLV, SMODELS *and* CMODELS.

Proof Immediate for SMODELS and DLV, from the correctness of the algorithms (lines 3 are satisfied); and for CMODELS, from the correctness of its algorithm and the fact that on tight programs *test* always succeeds. □

The second result on consistent leaf interpretations deals with nontight programs.

Theorem 6 *Let Π be a nontight program, and consider a consistent leaf interpretation J^t of Π. Then, the set of its true atoms $\mathbf{T}(J^t)$ is guaranteed to be an answer set of Π for* DLV *and* SMODELS*, but it is not guaranteed to be an answer set for* CMODELS.

Proof Immediate for SMODELS and DLV, from the correctness of the algorithms (lines 3 are satisfied). To show that the property does not hold for CMODELS we exhibit the following counter example:

$$\begin{aligned} \Pi_c : b &\leftarrow \overline{a}, \overline{b}. \\ a &\leftarrow a. \end{aligned}$$

In fact, *lp2sat*(Π_c) is

$$\{b, \overline{n}_1\}, \tag{6}$$

$$\{n_1, a, b\}, \tag{7}$$

$$\{\overline{n}_1, \overline{a}\}, \{\overline{n}_1, \overline{b}\}, \tag{8}$$

$$\{\overline{b}, n_1\}, \tag{9}$$

$$\{a, \overline{n}_2\}, \tag{10}$$

$$\{n_2, \overline{a}\}, \tag{11}$$

$$\{\overline{n}_2, a\}, \tag{12}$$

$$\{\overline{a}, n_2\} \tag{13}$$

clauses (6)–(9) being the translation of the first rule, and clauses (10)–(13) the translation of the second rule.

Consider that CMODELS chooses "$\overline{b}$" at line 4, clause (9) and the second clause of (8) are deleted, while clauses (6) and (7) are simplified. *unit-propagate* then assigns

"$\overline{n}_1$" from clause (6), that causes (6) and the first clause in (8) to be deleted, and clause (7) to be simplified. Again, *unit-propagate* assigns "a" from clause (7) (which is thus deleted): this causes clauses (10) and (12) to be deleted and clauses (11) and (13) to be simplified. Now "n_2" is assigned by *unit-propagate*, all clauses have been deleted, $J^t = \langle\{a, n_2\}, \{b, n_1\}\rangle$, $test(J^t, \Pi_c)$ is invoked and returns False because of the loop on "a" in the second rule. □

Note that the result of CMODELS on Π_c would have been the same even if applying simplifications adopted (*i*) by all SAT solvers; and/or (*ii*) for the "optimization" of the completion:

(i) SAT solvers delete repeated clauses (like (12) and (13)). In this case J_t would be the same as without simplifications.
(ii) for rule like $a \leftarrow a$. the introduction of a new atom can be avoided, with the rule translated as $\{a, \overline{a}\}$. There are now two possibilities: the clause is deleted in pre-processing, and the computation is similar to the first part of the behavior above, leading to $J^t = \langle\{a\}, \{b, n_1\}\rangle$ which does not correspond to an answer set of Π_c, or it is not deleted, but again $J^t = \langle\{a\}, \{b, n_1\}\rangle$.

In the real implementation of CMODELS, rules like $a \leftarrow a$ (and more in general rules r like $a \leftarrow body(r)$, with $a \in body(r)$) can be deleted from the logic program because this transformation does not alter its answer sets. For Π_c, $lp2sat(\Pi_c)$ would have now clauses (6)–(9) plus the unit clause $\{\overline{a}\}$, and thus the program would be correctly determined not to have answer sets. Nonetheless, as pointed out before, it is enough to slightly modify Π_c by replacing the rule $a \leftarrow a$. with the rules $a \leftarrow a'$. $a' \leftarrow a$., where a' is a newly introduced atom associated to a, to let CMODELS to not satisfy the property.

Let us now define DLV^F (resp. SMODELS^F) to be the algorithm in Fig. 3 (resp. Fig. 2) where lines 10–11 are inactive, i.e., only Fitting's operator for negative inferences is used and the system is deprived of *WFInf* (note that Fitting's operator is implemented by line 15 of *DetConsInf* and *AtLeast*).

The last result on consistent leaf interpretations deals with DLV^F and SMODELS^F.

Theorem 7 *Let* Π *be a nontight program, and consider a consistent leaf interpretation* J^t *of* Π. *Then, the set of its true atoms* $\mathbf{T}(J^t)$ *is guaranteed to be an answer set of* Π *for* DLV^F, *but it is not guaranteed to be an answer set for* SMODELS^F.

Proof For SMODELS^F, it is sufficient to show that the property does not hold for some logic program. This is the case for program Π_c defined in the proof of Theorem 6. In fact, suppose that SMODELS^F initially chooses "$\overline{b}$" at line 4, then *AtLeast* will assign "a" at line 17 from the first rule, obtaining $J^t = \langle\{a\}, \{b\}\rangle$. Now, J^t is a consistent leaf interpretation, but it does not correspond to an answer set, since a has only a "self support".

Consider now a consistent leaf interpretation J^t for DLV^F. Since J^t is a total consistent interpretation which has been returned by *DetCons*, J^t satisfies all rules of Π, otherwise *DetCons* would have returned some inconsistency in J^t. Thus, we know that J^t is a model of Π, and only the stability of J^t remains to be proven. (Actually, J^t

is a model of Π also for SMODELS, but it misses the stability property.) From the results in [32], we can equivalently prove that $\mathbf{T}(J^t)$ does not contain any atom belonging to an unfounded set for Π w.r.t. $\mathbf{T}(J^t)$. To this end, we proceed by induction and show that, at each step of DLV^F computation, each atom in the positive part $\mathbf{T}(J)$ of the interpretation J computed by DLV^F is not unfounded in any total interpretation J^t extending J. The base case is trivial, since $J = \emptyset$. For the step case, assuming that the property holds for J, we have to show that it still holds after the application of rule (14), since the application of (14) is the only case where an atom is made True in DLV^F computation. Let p be the atom which is set to True in the application of (14) on a rule r. Then, the body of r is True w.r.t. J. Consequently, the body cannot be False in any (consistent) extension J^t of J (thus the first unfoundedness condition is inapplicable to this rule in J^t). Moreover, each positive body literal of r is in $\mathbf{T}(J)$ and, by inductive hypothesis, is not unfounded in any total interpretation J^t extending J. Thus, p is not unfounded in any total interpretation J^t extending J, and we are done, as the property holds by inductive hypothesis for the other atoms in $\mathbf{T}(J + true(p))$. □

Note that part of this property has been recently established also in [35], where it is discussed that consistent leaf interpretations are supported models for SMODELS, thus they are not guaranteed to be answer sets.

At this point, it is important to underline two factors:

1. on tight programs, the systems are in some sense "too powerful". This is clear from the fact that for SMODELS^F and DLV^F (and also CMODELS^F, i.e., CMODELS modified in line 3 to return True instead of "*test*(Π, *J*)") the related Theorems would still hold.
2. well-founded operator *WFInf* is a checking operator and makes also inferences.

We want also to stress that there are classes of programs in which SAT-based procedures have advantages vs. native procedure. In particular, on the class of tight programs SAT-based systems are very efficient, often more efficient than native solvers, see, e.g., [27, 38] and the results of the First Answer Set Programming competition.

We have studied so far some computational advantages of DLV and SMODELS with respect to CMODELS. In the rest of the section we focus on some final observations on the computational cost of loop detection.

We have seen that a main difference between DLV/SMODELS and CMODELS in the "computation" of loops is that while the first two systems deal with them as soon as they appear (thus eagerly), CMODELS has a "lazy" evaluation (delegated to the *test* procedure).

The question is:

> "What is the cost of performing an eager evaluation of loops w.r.t. a lazy evaluation?".

Let Π be a program. The cost[16] of computing a consistent leaf interpretation in SMODELS, CMODELS and DLV depends on the "tightness" of Π.

If Π is tight, the cost is

– $O(|P|)$ for DLV, SMODELS and CMODELS.

In the more general case when Π is nontight

1. $O(|P|^2)$ for DLV and SMODELS; and
2. $O(|P|)$ for DLV^F, SMODELS^F and CMODELS.

Thus, in the case Π is tight the cost corresponds to the assignments to the atoms in Π which lead from the root of the search to a leaf where the computation ends (i.e., to a consistent leaf interpretation). Given this, exactly $|P|$ atoms are assigned.

In the case that Π is nontight, the same complexity as in the case of tight programs holds for DLV^F, SMODELS^F and CMODELS, because (a) DLV^F and SMODELS^F on nontight programs do the same amount of work as DLV and SMODELS on tight programs, and (b) CMODELS does not "add" any inference rule in DLL-REC in the case of nontight programs. For DLV and SMODELS, function *WFInf* assigns up to P atoms each time, thus reaching the above mentioned cost.

6 Results application to other ASP systems

In the previous sections, we have taken into account CMODELS, SMODELS and DLV in our comparison. In this section, we discuss to what extent the results presented for such systems are applicable to other ASP solvers.

The other SAT-based system, ASSAT [38] 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. Γ can be safely considered to be computed as *lp2sat*(Π).[17] ASSAT and CMODELS have different behavior only if Π is nontight: thus, if we assume that the same heuristic is used, Propositions 4, 5 and 6 hold also for ASSAT, as well as Theorem 5 and the complexity results on tight programs in Section 5.

About splitting heuristic, we have to note that, in their real implementations, CMODELS, ASSAT, CLASP and NoMore++ can also choose atoms which represent bodies: following [2, 3, 18], this fact can lead to significant improvements w.r.t. the setting we have analyzed in the paper.

About learning, first notice that it is common practice in this context not to take into account the specific learning strategies of the solvers, by focusing on solving algorithms with plain backtracking. The question on how the results presented extend to systems featuring learning, i.e., ASSAT, CLASP, SMODELS-CC and CMODELS with learning, depends on the particular learning strategies implemented by the solvers, both on failed leaves of the search tree, and on failed *test* procedures (for

[16]Note that: (a) we consider the cost of computing a successful path of the computation tree, from the root to a consistent leaf interpretation (no backtracking); (b) DLV has also a constant factor for the transition from Mbt to True; (b) for CMODELS P should be replaced by $P \cup N$.

[17]Fangzhen Lin, personal communications at ASP'05.

SAT-based systems): extending (or not) the results to the procedures with learning would need further investigations. In general, assuming that the learning mechanism implemented by native ASP systems resembles the learning mechanism implemented in SAT-based ASP systems (based on DLL), the results that hold for any systems based on resolution, e.g., [28] and the one on random k-CNFs, hold also for native procedures with learning. For SAT-based systems, we can consider that CMODELS and ASSAT rely on SAT solvers that implement similar learning techniques, borrowed from [45].

Finally, we have seen that the results we have shown hold for the ASP systems presented, as long as they rely on the "same" *ChooseLiteral* function, which is guaranteed to return the same literal at every point of the search trees for all systems. Because of this, similar results would hold if we enhance the procedures with more powerful look-ahead techniques based, e.g., on *expand*, *DetCons* and *unit-propagate*, respectively. For instance, SMODELS (and DLV) has been enhanced with the "failed literal" strategy. Using the same strategy in DLL-REC (i.e., using a similar "failed literal" strategy based on *unit-propagate*) would lead to an equivalent search tree.

7 Related work

The work in [24], starting from the equivalence results on tight programs, extends CMODELS with SAT heuristics and optimizations techniques, and evaluates its effectiveness.

In [18] the authors have presented tableaux calculi for ASP. The algorithms and techniques used by many of the solvers presented also in our paper are expressed in the framework, leading to similar results we reported in Section 4, using similar concepts for expressing "complexity" of algorithms (first used in this context in [25]). Nonetheless, there are many differences with our work: (a) we use an algorithmic view vs. a proof-theoretic account used in [18]. This, e.g., thanks to the DLL algorithm, allowed us to derive new complexity results for ASP procedures; (b) we carry out an in-depth analysis on the relation among CMODELS, SMODELS and DLV; and among the systems without *test*/*WFInf*; and (c) we use a more detailed characterization of the DLV behavior, by exploiting its peculiarities, while their work have fewer simplifying assumptions, i.e., they allowed choosing atoms that represented bodies, or bodies themselves. In [19], the same authors enhanced the tableaux calculi by taking into account cardinality constraints (as used in SMODELS) and disjunction in the rule heads. However, here the calculi is not used to express "real" ASP systems and thus no comparison among systems is presented. Extended ASP tableaux [31] is another extension of the tableaux calculi in [18] inspired by the Tseitin's Extended Resolution proof system [52]. In the same paper, the authors also showed how it is possible to define a proof system for (nondisjunctive) logic programs which can polynomially simulate tree resolution, and vice versa (assuming translations between ASP and SAT similar to those we use in our paper). Given that CMODELS (resp. SMODELS) can be seen as an implementation of tree resolution (resp. of the proof system for nondisjunctive programs), their work can be seen as an alternative way of extending results known for DLL to ASP systems. However, we strengthen this result showing that the procedures produce exactly the same search trees (assuming the same branching heuristics).

In [55] the authors present a detailed comparison of (the pruning power of) look-ahead in SMODELS with local consistencies in CSP under two different encodings from CSP to ASP. In comparison with our work, this is both (a) different, because the relation is only on pruning techniques and not on solving procedures (thus our work can be seen as a generalization); and (b) complementary, studying relations between ASP and CSP instead of ASP vs. SAT (DLL), and taking into account also look-ahead.

In [53], the author compares the relative power of unit-propagation and arc-consistency in CSP on both SAT-encodings of CSP problems, and CSP-encodings of SAT problems. These results are then used to understand the relative power between (general) CSP algorithms which maintain arc-consistency at each node and the DLL algorithm, by using a similar concepts (with respect to our paper) of both "equivalent" branching heuristics, and complexity of a procedure. More recently, in [43] more CSP algorithms have been taken into account, by studying relative power of such algorithms, by using again a similar concept about complexity of a procedure w.r.t. our paper, and by identifying, similarly to our work, families of instances which are easier for an algorithm with respect to another (with a limit on super-polynomial differences).

In [35], the author presents an abstract framework for designing DLL-like ASP procedures in a precise mathematical way: the work adapts to logic programs under the stable model semantics the work presented in [47] for SAT and Satisfiability Modulo Theories (SMT).[18] One of the results in the paper about the relation between CMODELS and SMODELS algorithms leads to similar conclusions as our Theorem 2; another result gives a different characterization of part of Theorem 7. DLV and other systems are not considered in the paper.

8 Conclusions

We have studied the existing relation among CMODELS, SMODELS and DLV ASP solvers from a computational point of view. We have shown that the ASP systems are equivalent on tight programs, but not equivalent on nontight programs where native procedures have advantages over SAT-based systems. We have also shown further advantages of both native procedures, and DLV in particular, when dealing with consistent leaf interpretations, and have studied the related cost.

We believe that our paper is relevant for ASP researchers who are interested in formally establishing the computational behavior of systems, but also for developers, because 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 [9].

In the future, we plan first to extend the formal analysis in Section 4 to other systems, e.g., CLASP, and then to study similar relations on disjunctive logic programs, with a comparison between disjunctive ASP solvers and solvers for quantified Boolean formulas (QBF) when solving problems on the second level of the polynomial hierarchy.

[18] http://combination.cs.uiowa.edu/smtlib/.

Acknowledgements We would like to thank Wolfgang Faber, Vladimir Lifschitz, Yuliya Lierler, Fangzhen Lin, Ilkka Niemelä and Mirek Truszczynski for discussions related to the subject of this paper, and the anonymous referees for their comments and suggestions. This work was supported by M.I.U.R. within projects "Potenziamento e Applicazioni della Programmazione Logica Disgiuntiva" and "Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione."

References

1. Achlioptas, D., Beame, P., Molloy, M.: A sharp threshold in proof complexity. In: Proc. of the 36th Annual ACM Symposium on Theory of Computing (STOC), pp. 337–346. ACM, New York (2001)
2. Anger, C., Gebser, M., Linke, T., Neumann, A., Schaub, T.: The nomore++ approach to answer set solving. In: Sutcliffe, G., Voronkov, A. (eds.) Proc. of the 12th International Conference on Logic Programming, Artificial Intelligence, and Reasoning (LPAR), Lecture Notes in Computer Science, vol. 3835, pp. 95–109. Springer, New York (2005)
3. Anger, C., Gebser, M., Janhunen, T., Schaub, T.: What's a head without a body? In: Brewka, G., Coradeschi, S., Perini, A., Traverso, P. (eds.) Proc. of the 17th European Conference on Artificial Intelligence (ECAI), pp. 769–770. IOS, Amsterdam (2006)
4. Babovich, Y., Lifschitz, V.: Computing answer sets using program completion. Available at http://www.cs.utexas.edu/users/tag/cmodels/cmodels-1.ps (2003)
5. Calimeri, F., Faber, W., Leone, N., Pfeifer, G.: Pruning operators for disjunctive logic programming systems. Fundam. Inform. **71**(2-3), 183–214 (2006)
6. Chvátal, V., Szemerédi, E.: Many hard examples for resolution. J. ACM **35**(4), 759–768 (1988)
7. Clark, K.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Plenum, New York (1978)
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT, Cambridge (2001)
9. Dixon, H.E., Ginsberg, M.L., Hofer, D.K., Luks, E.M., Parkes, A.J.: Generalizing Boolean satisfiability III: implementation. J. Artif. Intell. Res. (JAIR) **23**, 441–531 (2005)
10. Dowling, W.F., Gallier, J.H.: Linear-time algorithms for testing the satisfiability of propositional Horn formulae. J. Log. Program. **1**(3), 267–284 (1984)
11. East, D., Truszczynski, M.: The aspps system. In: Flesca, S., Greco, S., Leone, N., Ianni, G. (eds.) Proc. of 8th European Conference on Logics in Artificial Intelligence (JELIA), Lecture Notes in Computer Science, vol. 2424, pp. 533–536. Springer, New York (2002)
12. Erdem, E., Lifschitz, V.: Tight logic programs. Theory Pract. Log. Program. **3**(4-5), 499–518 (2003)
13. Faber, W.: Enhancing efficiency and expressiveness in ASP systems. Ph.D. thesis, University of Wien (2002)
14. Faber, W., Leone, N., Pfeifer, G.: Pushing goal derivation in DLP computation. In: Gelfond, M., Leone, N., Pfeifer, G. (eds.) Proc. of the 5th International Conference on Logic Programming and Nonmonostonic Reasoning (LPNMR), Lecture Notes in Computer Science, vol. 1730, pp. 107–116. Springer, New York (1999)
15. Faber, W., Leone, N., Pfeifer, G.: Experimenting with heuristics for ASP. In: Nebel, B. (ed.) Proc. of the 17th International Joint Conference on Artificial Intelligence (IJCAI), pp. 635–640. Morgan Kaufmann, San Francisco (2001)
16. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: semantics and complexity. In: Alferes, J.J., Leite, J. (eds.) Proc. of the 9th European Conference on Logics in Artificial Intelligence (JELIA), Lecture Notes in Computer Science, vol. 3229, pp. 200–212. Springer, New York (2004)
17. Fages, F.: Consistency of Clark's completion and existence of stable models. Methods Logic Comput. Sci. **1**, 51–60 (1994)
18. Gebser, M., Schaub, T.: Tableaux calculi for answer set programming. In: Etalle, S., Truszczynski, M. (eds.) Proc. of the 22nd International Conference on Logic Programming (ICLP), Lecture Notes in Computer Science, vol. 4079, pp. 11–25. Springer, New York (2006)

19. Gebser, M., Schaub, T.: Generic tableaux for answer set programming. In: Dahl V, Niemelä I (eds) Proc. of the 23rd International Conference on Logic Programming (ICLP), Lecture Notes in Computer Science, vol. 4670, pp. 119–133. Springer, New York (2007)
20. Gebser, M., Kaufmann, B., Neumann, B., Schaub, T.: Conflict-driven answer set solving. In: Veloso, M.M. (ed.) Proc. of the 20th International Joint Conference on Artificial Intelligence (IJCAI), pp. 386–392 (2007a)
21. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczynski, M.: The first answer set programming system competition. In: Baral, C., Brewka, G., Schlipf, J.S. (eds.) Proc. of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), Lecture Notes in Computer Science, vol. 4483, pp. 3–17. Springer, New York (2007b)
22. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski R, Bowen K (eds.) Proc. of 5th International Conference and Symposium on Logic Programming (ICLP/SLP), pp. 1070–1080. MIT, Cambridge (1988)
23. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Gener. Comput. **9**(3/4), 365–385 (1991)
24. Giunchiglia, E., Maratea, M.: Evaluating search strategies and heuristics for efficient answer set programming. In: Bandini, S., Manzoni, S. (eds.) Advanced in Artificial Intelligence: 9th Congress of the Italian Association for Artificial Intelligence (AI*IA), Lecture Notes in Computer Science, vol. 4733, pp. 122–134. Springer, New York (2005a)
25. Giunchiglia, E., Maratea, M.: On the relation between ASP and SAT procedures (or, between smodels and cmodels). In: Gabbrielli, M., Gupta, G. (eds.) Proc. of the 21st International Conference on Logic Programming (ICLP), Lecture Notes in Computer Science, vol. 3668, pp. 37–51. Springer, New York (2005b)
26. Giunchiglia, E., Sebastiani, R.: Applying the Davis-Putnam procedure to non-clausal formulas. In: Lamma, E., Mello, P. (eds.) Advances in Artificial Intelligence: 6th Congress of the Italian Association for Artificial Intelligence (AI*IA), Lecture Notes in Computer Science, vol. 1792, pp. 84–94. Springer, New York (1999)
27. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer set programming based on propositional satisfiability. J. Autom. Reason. **36**(4), 345–377 (2006)
28. Haken, A.: The intractability of resolution. Theor. Comp. Sci. **39**, 297–308 (1985)
29. Janhunen, T.: Representing normal programs with clauses. In: de Mántaras, R.L., Saitta, L. (eds.) Proc. of the 16th Eureopean Conference on Artificial Intelligenc (ECAI), pp. 358–362. IOS, Amsterdam (2004)
30. Janhunen, T.: Some (in)translatability results for normal logic programs and propositional theories. J. Appl. Non-Class. Log. **16**(1-2), 35–86 (2006)
31. Järvisalo, M., Oikarinen, E.: Extended ASP tableaux and rule redundancy in normal logic programs. In: Dahl, V., Niemelä, I. (eds.) Proc. of the 23rd International Conference on Logic Programming (ICLP), Lecture Notes in Computer Science, vol. 4670, pp. 134–148. Springer, New York (2007)
32. Leone, N., Rullo, P., Scarcello, F.: Disjunctive stable models: unfounded sets, fixpoint semantics and computation. Inf. Comput. **135**(2), 69–112 (1997)
33. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Log. (TOCL) **7**(3), 499–562 (2006)
34. Liberatore, P.: On the complexity of choosing the branching literal in DPLL. Artif. Intell. **116**(1–2), 315–326 (2000)
35. Lierler, Y.: Abstract answer set solver. Accepted at the 24th International Conference on Logic Programming (ICLP). Available at http://www.cs.utexas.edu/users/yuliya/papers/aasp.pdf (2008)
36. Lifschitz, V.: Answer set planning. In: Schreye, D.D. (ed.) Proc. of the 16th International Conference on Logic Programming (ICLP), pp. 23–37. MIT, Cambridge (1999)
37. Lifschitz, V., Razborov, A.: Why are there so many loop formulas? ACM Trans. Comput. Log. **7**(2), 183–214 (2006)
38. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. Artif. Intell. **147**((1–2)), 115–137 (2004)
39. Liu, L., Truszczynski, M.: Pbmodels - software to compute stable models by pseudoboolean solvers. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) Proc. of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), Lecture Notes in Computer Science, vol. 3662, pp. 410–415. Springer, New York (2005)
40. Maratea, M., Ricca, F., Faber, W., Leone, N.: Look-back techniques and heuristics in DLV: Implementation, evaluation and comparison to QBF solvers. J. Algorithms **63**(1-3), 70–89 (2008)

41. Marek, V., Truszczynski, M.: Stable models as an alternative programming paradigm. In: The Logic Programming Paradigm: a 25.Years perspective, Lecture Notes in Computer Science, Springer, New York (1999)
42. Marek, V.W., Truszczyński, M.: Nonmonotonic Logics—Context-Dependent Reasoning. Springer, New York (1993)
43. Mitchell, D.G.: Resolution and constraint satisfaction. In: Rossi, F. (ed.) Proc. of the 9th International Conference on Principles and Practice of Constraint Programming (CP), Lecture Notes in Computer Science, vol. 2833, pp. 555–569. Springer, New York (2003)
44. Monasson, R.: On the analysis of backtrack procedures for the coloring of random graphs, Chap Complex Networks. Lecture Notes in Physics. Springer, New York (2004)
45. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proc. of the 38th Design Automation Conference (DAC'01), pp. 530–535. ACM (2001)
46. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Ann. Math. Artif. Intell. **25**(3-4), 241–273 (1999)
47. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: from an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM **53**(6), 937–977 (2006)
48. Ricca, F., Faber, W., Leone, N.: A backjumping technique for disjunctive logic programming. AI Commun. **19**(2), 155–172 (2006)
49. Simons, P.: Extending and implementing the stable model semantics. Ph.D. thesis, Helsinky University (2000)
50. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artif. Intell. **138**(1-2), 181–234 (2002)
51. Syrjanen, T.: Lparse manual. http://www.tcs.hut.fi/software/smodels/lparse.ps.gz (2003)
52. Tseitin, G.: On the complexity of proofs in propositional logics. Semin. Math. **8** (1970)
53. Walsh, T.: SAT v CSP. In: Dechter, R. (ed.) Proc. of 6th International Conference on Principles and Practice of Constraint Programming (CP), Lecture Notes in Computer Science, vol. 1894, pp. 441–456. Springer, New York (2000)
54. Ward, J., Schlipf, J.: Answer set programming with clause learning. In: Lifschitz, V., Niemelä, I. (eds.) Proc. of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), Lecture Notes in Computer Science, vol. 2923, pp. 302–313. Springer, New York (2004)
55. You, J.H., Liu, G., Yuan, L.Y., Onuczko, C.: Lookahead in smodels compared to local consistencies in CSP. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) Proc. of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning, (LPNMR), Lecture Notes in Computer Science, pp. 266–278. Springer, New York (2005)