A Lower Bound for Answer Set Solver Computation

Article *in* INTELIGENCIA ARTIFICIAL · January 2010

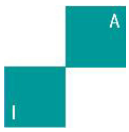
DOI: 10.4114/ia.v14i48.1630 · Source: DBLP

2 authors:

Stefania Costantini
Università degli Studi dell'Aquila
154 PUBLICATIONS **948** CITATIONS
SEE PROFILE

Alessandro Provetti
Birkbeck, University of London
88 PUBLICATIONS **1,029** CITATIONS
SEE PROFILE

Inteligencia Artificial 48(2010), 41-52
doi: 10.4114/ia.v14i48.1630



INTELIGENCIA ARTIFICIAL

http://erevista.aepia.org/

A Lower Bound for Answer Set Solver Computation

Stefania Costantini and Alessandro Provetti

Dipartimento d'Informatica, Università degli Studi di L'Aquila
Via Vetoio, I-67010 L'Aquila, Italy
stefcost@di.univaq.it

Oxford-Man Institute, University of Oxford
Eagle House, Walton Well Rd., Oxford OX2 6ED, UK.

Dip. di Fisica, sezione d'Informatica, Università degli Studi di Messina
Viale G. Stagno d'Alcontres, 31. I-98166 Messina, Italy.
ale@unime.it

Abstract We build upon recent work by Lierler that defines an abstract framework for describing the algorithm underlying many of the existing answer set solvers (for answer set programs, based upon the Answer Set Semantics), considering in particular Smodels and SUP. We define a particular class of programs, called AOH, and prove that the computation that the abstract solver performs actually represents a lower bound for deciding inconsistency of logic programs under the Answer Set Semantics. The main result is that for a given AOH program with n atoms, an algorithm that conforms to Lierler's abstract model needs $\Omega(n)$ steps before exiting with failure (no answer set exists). We argue that our result holds for every logic program that, like AOH programs, contains cyclic definitions and rules that can be seen as connecting the cycles.

Keywords: Answer Set Programming, Solvers, Complexity, Lower Bounds.

1 Introduction

Answer Set Programming (ASP) is a paradigm of logic programming which has been gaining credit from both the theoretical and practical point of view. ASP is based on the answer set semantics of [17], where solutions to a given problem are represented in terms of selected models (answer sets) of the corresponding logic program [23, 26]. A rich literature exists on applications of ASP in many areas, including problem solving, configuration, information integration, security analysis, agent systems, semantic web, and planning (see among many [4, 1, 20, 29, 16] and the references therein). Efficient inference engines, or *ASP Solvers*, are available [3] and can be freely downloaded by potential users.

Recently, Yuliya Lierler has proposed [21] an abstract framework for describing the algorithm underlying many of the existing answer set solvers, considering in particular *Smodels* [28] and her own SUP, thus introducing the notion of an "abstract solver." The abstract solver encompasses the main optimization strategies adopted by actual solvers, and primarily by Smodels, which is often taken as reference for comparison among solvers.

The expressive power of ASP, as well as its computational complexity, have been deeply investigated. The interested reader can refer, for instance, to [13]. In particular, deciding the existence of an answer set has been proved NP-complete in [24]; The same holds for deciding whether an atom is member of some answer set (proved in [25]).

ISSN: 1988-3064(on-line)

A topic that has received less attention in the literature concerns the least number of steps that a solver relying upon this type of algorithm (essentially, a branch-and-bound recursion) takes in order to establish whether a given program is inconsistent, i.e., a lower bound for ASP solvers computation. This is of interest in order to understand whether the existing strategies work well, or what could be done better[1]. Moreover, in a scenario were repeated calls to the inferential engine are needed (e.g., [re]planning as in [27]) one should be careful in evaluating the time investment related to each call, even for instances computable in polynomial time.

In this article, we introduce a particular class of (inconsistent) programs, *AOH-programs*, and prove that for any AOH program with n atoms, the abstract solver must perform no less than n steps, i.e., its lower-bound complexity is $\Omega(n)$. We prove that this class of programs is significant, as every non-trivial inconsistent program has an AOH-program at its "core," hence our result extends to every logic program of given size.

The structure of the paper is as follows: in Section 2 we provide the necessary background about lower bounds, ASP, some particular class of ASP programs and finally about the abstract ASP solver. In Section 3 we examine the behavior of the abstract solver on a particular class of programs that we suitably define and in Section 4 we argue in favor of the significance of this class and formulate a general lower-bound result. Finally, in Section 5 we conclude.

2 Background

Once algorithms for solving a specific problem have been found one may wonder whether it is possible to design a faster algorithm or not, and may wish to compare the different algorithms not only in terms of the number of steps in the worst- or average-case, but also concerning the *least* number of steps that they perform on a significant class of inputs. Often, a *lower bound* for the problem can be given, which in this context is practically intended as the number of steps that an algorithm has to execute at least in order to solve the problem on an input belonging to a given (interesting) class[2].

As usual, thanks to the Θ-notation constant factors are ignored and instances of size smaller than some n_0 are disregarded. Only the order of the lower bound is considered, as customary in terms of the function class expressing it.

Let $f : \mathbb{N} \to \mathbb{R}$; the set $\Omega(f)$ is defined as follows:

$$\Omega(f) = \{g : \mathbb{N} \to \mathbb{R} | \\ \exists c > 0 \; and \; n_0 \in \mathbb{N}, such \; that \\ \forall \; n \geq n_0 \; : \; g(n) \geq c \, f(n)\}$$

I.e., $\Omega(f)$ is the set of all functions that asymptotically grow at least as fast as f, modulo constant factors.

2.1 ASP in a nutshell

Below, we briefly recall the basics about Answer Set Programming [4, 23, 26]. In this logical framework, a problem can be encoded —by using a function-free logic language— as a set of facts, rules and constraints that describes the (candidate) solutions. More specifically, an *ASP-program*, or in the following simply a logic program Π, is a collection of *rules* of the form

$$H \leftarrow \; L_1, \ldots, L_m, \; not \; \; L_{m+1}, \ldots, not \; \; L_{m+n}. \tag{1}$$

where H is an atom, $m \geqslant 0$, $n \geqslant 0$, and each L_i is an atom. The symbol *not* stands for default negation (often also called "negation-as-failure" or simply "negation"). Various extensions to the basic paradigm exist, but we do not consider here all of them as they are not essential in the present context. The left-hand side and the right-hand side of the clause are called *head* and *body*, respectively. As customary, a *literal* can be either an atom a (positive literal) or its negation, in this context denoted by *not* a (negative literal). Then, the head of a rule is a positive literal and its body is composed of literals. A rule with empty head is a *constraint* (the literals in the body of a constraint cannot be all true, otherwise they

[1] Victor Marek and Mirek Truszczynski, personal communications.

[2] The general definition is that of a certain number of steps that *every* algorithm has to execute at least in order to solve a problem. As with the upper bounds, the notion of a step refers to an underlying machine model.

would imply falsity). To the aim of better understanding the discussion below, assume a constraint to be rewritten as a plain rule as follows, where f is a fresh atom not occurring elsewhere in the program

$$f \leftarrow \ not\, f,\, L_1, \ldots, L_m,\, not\ \ L_{m+1}, \ldots, not\ \ L_{m+n}.$$

By $Bodies(\Pi, H)$ or simply $Bodies(H)$ if Π is fixed from the context we mean the (multi-)set of the bodies of all rules with head H.

The semantics of ASP is expressed in terms of *answer sets* (also called *stable models* [17]). Consider first the case of a ground[3] ASP-program Π which does not involve negation.

If Π is a ground, *positive* (i.e., no negative conditions in the body of rules) program, the unique answer set $Cn(\Pi)$ is defined as the smallest set of literals constructed from the atoms occurring in Π which is closed under Π and it is either consistent or equal to the set of all literals. Such a definition is extended to any ground program P containing negation by considering the *reduct* Π^X of Π w.r.t. a set of atoms X obtained by means of the Gelfond-Lifschitz Γ operator introduced in [17].

First, Π^X is defined as the set of rules of type (1) of Π such that X does not contain any of the atoms $L_{m+1}, \ldots, L_{m+n}$. Clearly, Π^X does not involve negation. Let $\Gamma(P, X) = J$ where J is the unique answer set of Π^X. The set X is an answer set for Π if it is a fixed point of Γ, i.e., when $X = J$. Equivalently, X is an answer set for Π if it is the (unique) answer set of P^X. In order to obtain an answer set in the form of the set of literals which are true w.r.t. that answer set, the definition can be rephrased into $\Gamma(\Pi, X) = Cn(\Pi^X)$.

Once a problem is described as an ASP-program Π, its solutions (if any) are represented by the answer sets of Π. Unlike other semantics, a logic program may have several answer sets, or may have no answer set, because conclusions are included in an answer set only if they can be justified. The following program has no answer set (and it is said to be *inconsistent* w.r.t. *consistent* programs, which admit at least one, possibly empty, answer set): $\{a \leftarrow not\ b.\ b \leftarrow not\ c.\ c \leftarrow not\ a.\}$. The reason is that in every minimal model of this program there is a true atom that depends (in the program) on the negation of another true atom. Checking for consistency means checking for the existence of answer sets. For a survey of this and other semantics of logic programs with negation, the reader may refer to [2].

Let us now consider program π_1 consisting of three rules:

$$r \leftarrow \ p. \qquad p \leftarrow not\ q. \qquad q \leftarrow not\ p.$$

Such program has two answer sets: $\{p, r\}$ and $\{q\}$. If we add the rule (actually, a constraint) $\leftarrow q.$ to π_1, then we would rule out the second answer sets, since it violates the new constraint.

This simple example reveals the core of the usual approach followed in formalizing/solving a problem with ASP. Intuitively speaking, the programmer adopts a "generate-and-test" strategy: first (s)he provides a set of rules describing the collection of (all) potential solutions. Then, the addition of constraints rules-out all those answer sets that are not desired real solutions.

Given a rule γ in a language $\mathcal{L}$, the *grounding* of γ w.r.t. $\mathcal{L}$ is the set of all ground rules obtainable from γ through (ground) instantiation using the constant symbols of $\mathcal{L}$. Usually, given a program P and a rule $\gamma \in P$, we will consider the grounding of γ w.r.t. the language underlying P. The grounding of a set of rules is defined similarly. Given a (not necessarily ground) program P, a set of atoms is an answer set for P if it is an answer set for the grounding of P. In the following, we will always implicitly consider ground programs, i.e., equivalently, propositional logic programs.

An ASP solver is an inferential engine that computes the answer sets of an arbitrary ASP program given as input. Several ASP solvers are now available [3], each of them being characterized by valuable features and language extensions that simplify the programming activity. As it is well-known, ASP solvers produce the grounding of the input program as a first step; the actual inferential activity is hence carried out on ground programs only[4].

The expressive power of ASP, as well as, its computational complexity have been deeply investigated. The interested reader can refer, for instance, to [13]. In particular, deciding the existence of an answer

[3] As customary, a term (atom, literal, rule, ...) is ground if no variable occurs in it. A ground program is a program that contains no variables.

[4] Presently, ASP solvers perform a complete grounding before starting inference; Asperix [19] is an exception as it performs *on-the-fly* grounding. Several research efforts are now addressing this limitation.

set has been proved NP-complete in [24] and the same for deciding whether an atom is a member of some answer set (proved in [25]). The reader can also see [4, 14], among others, for a presentation of ASP as a tool for declarative problem-solving.

2.2 Kernel Programs

Below we summarize the features of a special class of logic programs, kernel programs, introduced in [7] and discussed at length in [10] and [12].

The kernel form is a *normal* form, in the sense that (as proved in [10]) any logic program under the answer set semantics admits an equivalent kernel program, i.e., one which has the same answer sets, modulo some projection. Transforming a program into the corresponding kernel normal form eliminates i) all literals that are true/false *in all answer sets* and ii) literals that would be irrelevant w.r.t. consistency and number of answer sets of the program. As it is well-known, case i) contains at least the atoms which are true (resp. false) in the *Well-founded semantics* of the program. That semantics is three-valued and provides the set of atoms which are deemed true and false, while the other atoms are assumed to have truth value *undefined*.

If Π is a logic program, we denote by $WFM(\Pi) = \langle T, F\rangle$ the well-founded model of Π. It can be obtained in the form of the set of literals which are true/false [22] by iterating the double application of the Γ operator over a set of atoms *I:* $\Gamma^2(\Pi, I) = \Gamma(\Pi, \Gamma(\Pi, I))$. The existence of a fixpoint for Γ^2 iterated from $\emptyset$ is guaranteed and the process terminates.

A program Π is said WFM-irreducible whenever $WFM(\Pi) = \langle \emptyset, \emptyset\rangle$. That is, in WFM-irreducible programs all the atoms are undefined under the well-founded semantics. As discussed in [9, 10], these are exactly the atoms that are relevant for deciding whether answer sets exist, and for finding them. Below is the definition of programs in kernel form.

Definition 2.1 *A logic program* Π *is in kernel normal form (or, equivalently,* Π *is a kernel program) if and only if the following conditions hold.*

1. Π *is WFM-irreducible;*
2. *every rule has its body composed of negative literals only;*
3. *every atom in* Π *occurs in the body of some rule;*

Clearly, kernel programs contain no facts (else the WFM would not be empty). Also, it is easy to see that, in kernel programs, each atom occurs as the head of some rule and, since they are undefined w.r.t. well-founded semantics, it is either part of a cyclic definition or defined using atoms that are part of a cycle (the notion of cycle is formally defined and developed in [12]).

For programs in kernel normal form, every *supported* model is stable [8, 9], where a supported model M is such that for every atom $a \in M$ some $B \in Bodies(a)$ is true w.r.t. M. We may also notice that kernel programs are *tight*, i.e., do not contain positive loops (which is obvious, as no atom occurs positively). For tight programs, the same result has been proved in [15].

The kernel normal form can be obtained by means of a normalization algorithm (see [12] which carefully exploits the *BDFZ* program rewriting w.r.t. the WFM described in [6]. After a preliminary grounding phase and the finding of positive cycles (a concept that is mapped of graph-theoretic properties of the dependency graph representing the program), BDFZ transforms the program into a (unique) program remainder $\hat{\Pi}$ obtained by iterating a straightforward extension of the above-mentioned Gelfond-Lifschitz operator Γ, i.e., by i) deleting every rule instance with a body literal which is false w.r.t. $WFM(\Pi)$, and ii) removing from the remaining rule instances the body literals which are true w.r.t. $WFM(\Pi)$. Atoms involved in positive cycles will be false in $WFM(\Pi)$ *unless they are also defined in (and supported by) other rules external to the cycle itself.* We exploit this concept to rid programs from positive cycles.

The kernelization algorithm (formally described in [11]) performs, as a first step, a simplification of the given program Π w.r.t. the well-founded semantics $WFS(\Pi)$ by means of BDFZ. As a second step, the kernelization algorithm performs *top elimination,* i.e., it eliminates all ground rules whose head is an atom which never appears in the body of a rule. Finally, a *Positive Condition Elimination* procedure produces the result $ker(\Pi)$. The aim of this step is to eliminate all the remaining positive body literals,

since they amount to nothing but "intermediate steps" between relevant atoms, and are immaterial to the existence and number of answer sets.

As proved in [9], the answer sets of $ker(\Pi)$ and the answer sets of Π are in correspondence, in the sense that they are in the same number, and the latter can be obtained from the former. In particular, given a answer set S of $ker(\Pi)$, a stable model of the original program Π can be obtained as follows:

(i) apply the Gelfond-Lifschitz transformation to Π w.r.t. S, and

(ii) compute the Least Model of the resulting (positive) program.

2.3 An Abstract Answer Set Solver

In the following, as it is customary in ASP solvers we will indicate $\leftarrow$ with $:-$, and we will interpret it as an implication, where if the body of the rule is true (w.r.t. a given answer set) then the head must be true as well. If instead the body is false, then the head is false as well unless it is made true via some other rule.

The complement $\bar{l}$ of literal l is such that, for atom a, we have $\overline{a} = not\ a$ and $\overline{not\ a} = a$. If B is a set of literals, by $\overline{B}$ we mean a set of literals composed of the complements of all the literals in B; by B^+ we take the set of positive literals occurring in B. For a given program Π, let $\mathbb{H}_{\not>}$ be the set of all literals that can be defined using the atoms appearing in Π.

The abstract solver is described by means of *steps*, or *transition rules*, that can be applied to *states*. A state here is either $\emptyset$, or *FailState*, or a set $M \subseteq \mathbb{H}_{\not>}$. Each literal l in M can be further *annotated* using a *'d'* superscript, as in l^d. Literals in the current state M are those that have been deemed true up to that point. Each annotated literal has been *assumed* to be true, where the others have been assigned true by some of the transition rules. A literal l is *assigned* (w.r.t. *unassigned*) in state M (or for short by M) if either l or l^d or $\bar{l}$ occur in M. Sometimes, states will be treated as sets, regardless the order of literals and the annotations.

The abstract solver starts from state $\emptyset$ and from a given formula (in this case a logic program Π) and applies transition rules until it reaches either a *FailState* state, or a *final* state M where each atom in σ occurs in some literal in M, and M is consistent, i.e., it is not the case that both a literal and its negation (whatever their annotations) occur in M. The set of possible transitions from the empty state to final states can be represented as a graph DP_Π where *terminal nodes*, i.e., nodes with no out-going arcs, are either *FailState* or states where no transition is applicable.

We summarize below from [21] the transition rules that define the basic version $ATLEAST_\Pi$ of the abstract solver. Capital letters $M, M', C, P, Q, \ldots$ denote states. A transition rule has the form $M \Rightarrow M'$ and is applicable (determining a *step* to be performed or, equivalently, a new arc of DP_Π to be created) if its condition is satisfied by M. By mentioning a rule, we implicitly assume that it is a rule occurring in Π.

Decide (D):
$M \Rightarrow M\ l^d$ if l is unassigned by M.

Fail (F):
$M \Rightarrow FailState$ if M is inconsistent and M contains no decision literal.

Backtrack (B):
$P\ l^d\ Q \Rightarrow P\ \bar{l}$ if $P\ l^d\ Q$ is inconsistent and Q contains no decision literal.

Notice that the definition of $\Rightarrow B$ includes strategic aspects that are found in most solvers. (i) Backtracking is performed to the last decision that has been taken, i.e., literal l^d: this comes from the assumption that part Q of initial state M contains no decision literal. The negation $\bar{l}$ is added to the new state $M' = P\ \bar{l}$ as a plain true literal, not as a decision. Notice that adding $\bar{l}$ to the new state prevents l to be decided again later, as it is already assigned. However, if later on a backtracking should be performed to a literal which occurs earlier than l^d in M, the assignment would then be retracted, and then l could be decided again, though in a new context.

Unit Propagate (UP):

$M \Rightarrow M\ a$ if we have a rule $a :- B$ and $B \subseteq M$.

I.e., the head of a rule with body true (w.r.t. M) is added to the new state.

All Rules Canceled (*ARC*):
$M \Rightarrow M\ not\ a$ if for all $B \in Bodies(a)$, $\overline{B} \cap M \neq \emptyset$

I.e., the negation *not a* of the head a is added to the new state if all the bodies B of rules with head a are false w.r.t. M which in fact includes the negation of some literal in B.

Backchain True (*BT*):
$M \Rightarrow M\ B$ if we have a rule $a :- B$, $a \in M$ (whatever its annotation in M),
and for all $B' \neq B$, $B' \in Bodies(a)$ we have $\overline{B'} \cap M \neq \emptyset$

I.e., if an atom a belongs to M and all but one body of a rule with head a are false (w.r.t. M), then the literals occurring in the only remaining body are added to the new state (which means that they are deemed true) as in supported models atoms may occur only if derivable via a rule.

Backchain False (*BF*):
$M \Rightarrow M\ \bar{l}$ if we have a rule $a :- l, B$ such that $not\ a \in M$, and $B \subseteq M$.

I.e., if atom a is false w.r.t. M (as M contains its negation *not a*) and if we have a rule where all literals in the body but one are true (w.r.t. M) then this last literal is deemed false, thus justifying the falsity of a.

As proved in [21], the terminal nodes of the graph DP_Π other than *FailState* generated by $ATLEAST_\Pi$ are consistent states and represent in particular all the supported models of Π (which in the case of kernel and, more generally, of tight programs correspond to all the answer sets). Moreover, *FailState* is reachable only if no supported model exists.

The Smodels solver and all the other solvers that accept programs that are not *tight* will have to apply another transition rule called *Unfounded;* it is needed in order to deem false all atoms that are involved in positive circularities in Π and cannot be deemed true by any other rule. With this additional transition rule the above results extends, i.e., the terminal nodes of the graph DP_Π other than *FailState* correspond to all the answer sets of given program Π and *FailState* is reachable only if Π is inconsistent.

3 Understanding the lower bounds for SMODELS-like Algorithms

In the discussion that follows we resort to the previous description $ATLEAST_\Pi$ of the abstract solver, as we will consider a class of programs composed of kernel programs only. This is however without loss of generality, as we may notice that any solver might in principle detect the fact that a given program is negative (if no positive literal occurs in bodies) and omit the application of *Unfounded*.

Consider the following inconsistent kernel program π_6, containing 6 distinct atoms (thus, $n = 6$).

$$p :- not\ p, not\ a_1, not\ a_2. \quad (1)$$

$$q :- not\ q.$$
$$q :- not\ a_1, not\ a_2. \quad (2)$$

$$a_1 :- not\ b_1.$$
$$b_1 :- not\ a_1.$$

$$a_2 :- not\ b_2.$$
$$b_2 :- not\ a_2.$$

The reason why this program is inconsistent relies in its structure: there are the *odd loops* $p :- not\ p$ and $q :- not\ q$ which, by themselves, would make the program inconsistent. The former cycle might in principle be *repaired* by the conjunction $not\ a_1, not\ a_2$ that is called AND-handle [12]. In fact, if at least

one literal is deemed false (i.e., if either a_1 or a_2 are true) then p becomes false as well, as there are no other alternative rules with head p. The latter cycle, could also, in principle, be *repaired* by the same conjunction $not\ a_1, not\ a_2$, called OR-handle [12]: if both literals are deemed true (i.e., if both a_1 or a_2 are false) then q becomes true, thus *overriding* the contradictory rule. However, the two conditions are clearly mutually incompatible, and thus π_6 is inconsistent. This might be easily checked either on the EDG (Extended Dependency Graph, [7]) or even better on the Cycle Graph [12] of the program itself.

Let us see how the abstract answer set solver $ATLEAST_{\Pi}$ –which is of course oblivious to such "structural" information– would behave on π_6. We assume that the algorithm does not perform a *Decide* step if some other step is possible. We also assume (as most solvers do) to decide positive literals only. As said before, backtracking is up to the last decision. Also, the execution of the algorithm stops in a final state whenever all atoms have been assigned, no decision literal occurs in the state, and no further step is possible. The final state can be *FailState* in case the last but one state is inconsistent. We finally assume, quite arbitrarily but harmlessly, that *UP* is applied according to the order of the rules in the program.

Notice that the abstract solver behavior is simplified by the fact that, except for q, each atom is the head of just one rule. Thus, after deciding the head it is immediately possible to apply $\Rightarrow BT$or $\Rightarrow ARC$. An execution of the abstract solver always starts with the empty state, an proceeds via steps corresponding to the application of a transition rule. Following [21] we indicate on the right of the current state (other than *FailState*) the transition which is applied (for conciseness, by using its label). Let us first assume that the solver tries to decide atom p first. This results in the following sequence of states:

$\emptyset \Rightarrow D$
$p^d \Rightarrow BT$
$p^d,\ not\ p,\ not\ a_1,\ not\ a_2 \Rightarrow B$
$not\ p\ \ldots$

I.e., as p is the head of just one rule, the solver applies *Backchain True* in order to try to justify its truth, but it immediately finds a contradiction which implies backtracking, i.e., retracting the decision to assume p true and asserting $not\ p$. The execution will then continue with some other decision. Let us instead assume that the solver tries to decide atom q first. This results in the following sequence of states:

(2) Decide atom q first.
$\emptyset \Rightarrow D$
$q^d,\ not\ a_1,\ not\ a_2 \Rightarrow BT$
$q^d,\ not\ a_1,\ not\ a_2\ q^d\ \ldots$

That is, assuming the truth of d leads, analogously to the case of deciding p, to a later retraction.

Therefore, as we are looking for a lower bound, we will optimistically assume that the solver will start its execution by deciding some atom other than p or q, say a_1. The execution will proceed, for instance, as follows:

$\emptyset \Rightarrow D$
${a_1}^d \Rightarrow ARC$
${a_1}^d,\ not\ p \Rightarrow BT$ (or $\Rightarrow ARC$)
${a_1}^d,\ not\ p,\ not\ b_1 \Rightarrow D$
${a_1}^d,\ not\ p,\ not\ b_1,\ q^d \Rightarrow ARC$
${a_1}^d,\ not\ p,\ not\ b_1,\ q,\ not\ q \Rightarrow B$

Notice that in the above trace it is relevant whether one decides either a_2 or q first. In particular, deciding q first leads more quickly to discovering the inconsistency, and it is what we have done as we are looking for a lower bound. This determines to backtrack the decision q^d, which implies asserting $not\ q$ that again implies q. What remains is a further backtracking, which means undoing ${a_1}^d$ and restarting from $not\ a_1$, which implies b_1.

${a_1}^d,\ not\ p,\ not\ b_1,\ q,\ not\ q \Rightarrow B$ (6 steps)
$not\ a_1 \Rightarrow UP$
$not\ a_1, b_1$ (two steps from backtracking)

If we now decide b_2 we then get q. We would quickly run into an inconsistency on p, as all conditions of its only rule become true but one ($not\ p$). The latter can be derived by means of *Backchain True* thus determining inconsistency and backtracking on the decision of p. Such backtracking, however, leads again to inconsistency, which forces us to retract the decision on f and assert $not\ b_2$, which finally yields a_2.

$not\ a_1, b_1 \Rightarrow D$ (*)
$not\ a_1, b_1,\ {b_2}^d \Rightarrow BT(\text{or} \Rightarrow ARC)$
$not\ a_1, b_1,\ {b_2}^d,\ not\ a_2 \Rightarrow UP$
$not\ a_1, b_1,\ {b_2}^d,\ not\ a_2,\ q \Rightarrow D$ (four steps, for each of the b_2's if there were many)
$not\ a_1, b_1,\ {b_2}^d,\ not\ a_2,\ q,\ p^d \Rightarrow BT$
$not\ a_1, b_1,\ {b_2}^d,\ not\ a_2,\ q,\ p^d,\ not\ p \Rightarrow B$
$not\ a_1, b_1,\ {b_2}^d,\ not\ a_2,\ q,\ not\ p \Rightarrow UP$
$not\ a_1, b_1,\ {b_2}^d,\ not\ a_2,\ q,\ not\ p,\ p \Rightarrow B$ (four steps more for each of the b_2's if there were many)
$not\ a_1, b_1,\ not\ b_2 \Rightarrow UP$
$not\ a_1, b_1,\ not\ b_2,\ a_2$ (two steps more for each of the b_2's if there were many)

At this point, p becomes false as the body of its only clause is false, where q has to be decided and as the body of its second clause is false this leads to assuming that the first rule should work, and then to inconsistency and failure.

$not\ a_1, b_1,\ not\ b_2,\ a_2 \Rightarrow ARC$
$not\ a_1, b_1,\ not\ b_2,\ a_2,\ not\ p \Rightarrow D$
$not\ a_1, b_1,\ not\ b_2,\ a_2,\ not\ p,\ q^d \Rightarrow BT$
$not\ a_1, b_1,\ not\ b_2,\ a_2,\ not\ p,\ q^d,\ not\ q \Rightarrow B$
$not\ a_1, b_1,\ not\ b_2,\ a_2,\ not\ p,\ not\ q \Rightarrow UP$
$not\ a_1, b_1,\ not\ b_2,\ a_2,\ not\ p,\ not\ q,\ q \Rightarrow F$
FailState (six final steps)

The total number of steps is 26, i.e., slightly less than $4n$. It remains to see what would happen if at point (*) one would decide a_2.

$not\ a_1, b_1 \Rightarrow D$
$not\ a_1, b_1,\ {a_2}^d \Rightarrow BT(\text{or} \Rightarrow ARC)$
$not\ a_1, b_1,\ {a_2}^d,\ not\ b_2 \Rightarrow ARC$
$not\ a_1, b_1,\ {a_2}^d,\ not\ b_2,\ not\ p \Rightarrow D$ (three steps to get rid of p)
$not\ a_1, b_1,\ {a_2}^d,\ not\ b_2,\ not\ p,\ q^d \Rightarrow ARC$
$not\ a_1, b_1,\ {a_2}^d,\ not\ b_2,\ not\ p,\ q^d,\ not\ q \Rightarrow B$
$not\ a_1, b_1,\ {a_2}^d,\ not\ b_2,\ not\ p,\ not\ q \Rightarrow UP$
$not\ a_1, b_1,\ {a_2}^d,\ not\ b_2,\ not\ p,\ not\ q,\ q \Rightarrow B$

It turns out that we must backtrack this decision; next, the execution would proceed as before (with some modifications). Had computation started by deciding a_2 instead of a_1, by reverting the indexes we would have obtained the same trace. Instead, things might be different if we start by deciding b_1:

$\emptyset \Rightarrow D$
${b_1}^d,\ not\ a_1$ (two steps) (**)

now, deciding b_2 will quickly lead to inconsistency on *p:*

${b_1}^d,\ not\ a_1 \Rightarrow D$
${b_1}^d,\ not\ a_1,\ {b_2}^d \Rightarrow BT(\text{or} \Rightarrow ARC)$
${b_1}^d,\ not\ a_1,\ {b_2}^d, not\ a_2 \Rightarrow UP$

${b_1}^d,\ not\ a_1,\ {b_2}^d, not\ a_2,\ q \Rightarrow D$
${b_1}^d,\ not\ a_1,\ {b_2}^d, not\ a_2,\ q,\ p^d \Rightarrow BT$
${b_1}^d,\ not\ a_1,\ {b_2}^d, not\ a_2,\ q,\ p^d,\ not\ p \Rightarrow B$
${b_1}^d,\ not\ a_1,\ {b_2}^d, not\ a_2,\ q,\ not\ p \Rightarrow UP$
${b_1}^d,\ not\ a_1,\ {b_2}^d, not\ a_2,\ q,\ not\ p, p \Rightarrow B$ (seven steps for each of the other b_2's, if any)
${b_1}^d,\ not\ a_1,\ not\ b_2 \Rightarrow UP$
${b_1}^d,\ not\ a_1,\ not\ b_2, a_2 \Rightarrow ARC$
${b_1}^d,\ not\ a_1,\ not\ b_2, a_2, not\ p \Rightarrow D$
${b_1}^d,\ not\ a_1,\ not\ b_2, a_2, not\ p,\ q^d \Rightarrow BT$
${b_1}^d,\ not\ a_1,\ not\ b_2, a_2, not\ p,\ not\ q \Rightarrow UP$
${b_1}^d,\ not\ a_1,\ not\ b_2, a_2, not\ p,\ not\ q,\ q \Rightarrow B$
$not\ b_1 \Rightarrow UP$
$not\ b_1,\ a_1\ \ldots$ (15 steps)

that is symmetrical to (*). Another variation variation is to decide a_2 at (**).

${b_1}^d,\ not\ a_1 \Rightarrow D$
${b_1}^d,\ not\ a_1,\ {a_2}^d \Rightarrow BT(\text{or} \Rightarrow ARC)$
${b_1}^d,\ not\ a_1,\ {a_2}^d,\ not\ b_2 \Rightarrow ARC$
${b_1}^d,\ not\ a_1,\ {a_2}^d,\ not\ b_2,\ not\ p \Rightarrow D$
${b_1}^d,\ not\ a_1,\ {a_2}^d,\ not\ b_2,\ not\ p,\ q^d \Rightarrow ARC$
${b_1}^d,\ not\ a_1,\ {a_2}^d,\ not\ b_2,\ not\ p,\ q^d,\ not\ q \Rightarrow B$
${b_1}^d,\ not\ a_1,\ {a_2}^d,\ not\ b_2,\ not\ p,\ q^d,\ not\ q \Rightarrow UP$
$not\ b_1 \Rightarrow UP$
$not\ b_1,\ a_1$ (9 steps)

that is symmetrical to (*) and takes a few less steps. If we decide a_2 (or symmetrically b_2) from the beginning we get:

$\emptyset \Rightarrow D$
${a_2}^d,\ not\ b_2 \Rightarrow ARC$
${a_2}^d,\ not\ b_2,\ not\ p \Rightarrow D$
${a_2}^d,\ not\ b_2,\ not\ p,\ q^d \Rightarrow ARC$
${a_2}^d,\ not\ b_2,\ not\ p,\ q^d,\ not\ q \Rightarrow B$
${a_2}^d,\ not\ b_2,\ not\ p,\ not\ q \Rightarrow UP$
${a_2}^d,\ not\ b_2,\ not\ p,\ not\ q,\ q \Rightarrow B$
$not\ a_2 \Rightarrow UP$
$not\ a_2,\ b_2$

which requires a decision recollecting one of the traces before. Therefore, the minimum number of steps that $ATLEAST_\Pi$ needs to decide that Π_6 is inconsistent is in $\Omega(n)$.

4 Generalization

The program above is a sample of the following class of programs, that we call *AOH-programs* where AOH stands for AND-OR-handles.

Definition 4.1 *An AOH-program Π_n has the following structure:*

$$p \text{ :}-not\ p, not\ a_1, \ldots, not\ a_k. \quad (1)$$

$$q \text{ :}-not\ q.$$
$$q \text{ :}-not\ a_1, \ldots, not\ a_k. \quad (2)$$

$$\%for\ every\ a_i, i \leq k:$$
$$a_i \text{ :}-not\ b_i.$$
$$b_i \text{ :}-not\ a_i.$$

There are $2k+2$ atoms and $n+1$ rules, as each atom occurs in the head of just one rule, except for q.

As in previous section, the body of rule (1) is an *AND handle;* for the program to be consistent, at least one of its conditions must be false (thus making the AND handle *active*), so that the head becomes false as well. The body of rule (2) is an *OR handle;* for the program to be consistent all the composing literals must be true, thus making the head true (*active* OR handle). In fact, no answer set would otherwise exists as the contradiction over p and/or q cannot be overridden.

It is easy to see that in the above program the two handles are *incompatible* in the sense that they cannot be both active, as this implies a conflict over at least a literal, that should be simultaneously true and false. Therefore we have the following proposition.

Proposition 4.1 *Every AOH-program Π_n is inconsistent, whatever the number n of composing atoms.*

Notice that the above-defined AOH programs include in rules (1) and (2) two negative odd cycles (involving atoms p and q respectively) of length 1, i.e., composed of just one rule. In this sense, we might call these program AOH1-programs, and introduce the classes of AOHn-programs with n odd, involving two odd cycles each one of length at most n, the former one exhibiting AND handles (no matter in which rules) and the latter one exhibiting OR handles (no matter for which rules). The above proposition can thus be immediately extended to AOHn-programs.

As it is well-known, inconsistency of logic programs stems from negative odd cycles (every program involving no negative odd cycle is consistent). As discussed in depth in [12], a kernel logic program is inconsistent either because there is an *unconstrained* odd cycle (i.e., an odd cycle without handles) or because, as it happens with AOH programs, there exist two odd cycles whose handles are incompatible. Kernelization does not affect the above result: in fact, on the one hand it eliminates atoms not involved in negative cycles and on the other hand removes literal true/false in every answer set and skips intermediate steps. The only effect can be that some odd cycles, which in the original program seemed to have handles, become unconstrained in its kernel version as the literals occurring therein are true w.r.t. the well-founded model. Therefore, the significance of AOHn-programs consists in the fact that they characterize inconsistency for ASP programs.
Observation: every inconsistent logic program includes in its kernel counterpart either an unconstrained odd cycle or an AOHn-program (for some n).
Thanks to the above observation, the following lower-bound result, which can be easily extended to arbitrary AOHn-program, shall hold for every ASP program.

Theorem 4.1 *The abstract solver $ATLEAST_\Pi$ on an AOH-program Π_n performs $\Omega(n)$ steps.*

Proof The proof is essentially the lifting of the detailed proof seen in the previous Section for the AOH program of size *n=6.* It suffices to notice that the given value for n is never accessed nor used in the derivation; i.e., we use n as a parameter for building the instance and for iterating over rules (1) and (2), but we never exploit any property of the values assigned to n.

Hence, we can say that in general the *winning* strategy for performing the least possible number of steps is that of making both handles true. This quickly falsifies the contradictory atom that has been decided first, and determines later the contradiction on the second one thus leading to a failure state. This effect is obtained efficiently only if one chooses to decide in the first place all of the a_i's. This choice results in getting all of them false (and thus no more decidable) via backtracking, obtaining therefore as soon as possible a failure state.

5 Concluding Remarks

As we have seen above, determining the strategy that chooses the atoms to assume true so as to result in the least possible number of steps requires information about the structure of the program. From the Cycle Graph (CG) [12] of an AOH-program [12] one would see immediately that the program is inconsistent. Granted that obtaining the CG is computationally expensive, the solver designers should evaluate whether some kind of structural analysis might actually be useful in order to reduce the number of steps, which is especially valuable on large problem instances.

We remind the reader about the existence of alternative algorithms for computing the answer sets, e.g., based on the EDG of a (kernelized) program [5] whose underlying principles have been applied in order to improve existing solvers [18].

To conclude, we have established that the lower bound of the ASP solvers that adopt the abstract solver algorithm, like SMODELS, is by no means bad. However, the integration with program analysis and transformation techniques might bring relevant advantages.

Acknowledgements

We appreciated Victor Marek and Mirek Truszczynski for raising the problem of lower bounds for Answer Set computation in informal discussions at the ICLP 2008 Conference in Udine. The authors also wish to thank Yuliya Lierler for many useful and motivating discussions on this subject. The anonymous reviewers for the LANMR'09 Conference careful reviewed our work and made important suggestions on how to improve it.

References

[1] Christian Anger, Torsten Schaub, and Miroslaw Truszczyński. ASPARAGUS – the Dagstuhl Initiative. *ALP Newsletter*, 17(3), 2004. See `http://asparagus.cs.uni-potsdam.de`.

[2] Krzysztof R. Apt and Roland N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19/20:9–72, 1994.

[3] Web references for some ASP solvers
. ASSAT: `http://assat.cs.ust.hk`;
Clasp: `http://www.cs.uni-potsdam.de/clasp`;
Cmodels: `http://www.cs.utexas.edu/users/tag/cmodels`;
DeReS and aspps: `http://www.cs.uky.edu/ai/`;
DLV: `http://www.dbai.tuwien.ac.at/proj/dlv`;
Smodels: `http://www.tcs.hut.fi/Software/smodels`.

[4] Chitta Baral. *Knowledge representation, reasoning and declarative problem solving.* Cambridge University Press, 2003.

[5] A. Bertoni, G. Grossi, A. Provetti, V. Kreinovich, and L. Tari. The prospect for answer set computation by a genetic model. In *Proc. of the AAAI Spring Symposium ASP 2001*, pages 1–5. AAAI press, 2001.

[6] S. Brass, J. Dix, B. Freitag, and U. Zukowski. Transformation-based bottom-up computation of the well-founded model. *Theory and Practice of Logic Programming*, 1(5):497–538, 2001.

[7] G. Brignoli, S. Costantini, O. D'Antona, and A. Provetti. Characterizing and computing stable models of logic programs: the non–stratified case. In *Proc. of the 1999 Conference on Information Technology, Bhubaneswar, India*, 1999.

[8] S. Costantini. Contributions to the stable model semantics of logic programs with negation. In A. Nerode and V.S. Subrahmanian, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the 2nd International Workshop LPNMR93*, 1993.

[9] S. Costantini. Contributions to the stable model semantics of logic programs with negation. *Theoretical Computer Science*, 149, 1995. (prelim. version in Proc. of LPNMR93).

[10] S. Costantini and A. Provetti. Normal forms for answer sets programming. *J. on Theory and Practice of Logic Programming*, 5(6), 2005.

[11] S. Costantini and A. Provetti. Computing the kernel normal form of answer set programs. submitted, can be obtained form the authors, 2009.

[12] Stefania Costantini. On the existence of stable models of non-stratified logic programs. *Theory and Practice of Logic Programming*, 6(1-2):169–212, 2006.

[13] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[14] Agostino Dovier, Andrea Formisano, and Enrico Pontelli. A comparison of CLP(FD) and ASP solutions to NP-complete problems. In Maurizio Gabbrielli and Gopal Gupta, editors, *Logic Programming, 21st International Conference, ICLP 2005, Proceedings*, volume 3668 of *LNCS*, pages 67–82. Springer, 2005.

[15] Francois Fages. Consistency of Clark's completion and existence of stable models. *Methods of Logic in Computer Science*, 1:51–60, 1994.

[16] Michael Gelfond. Answer sets. In *Handbook of Knowledge Representation, Chapter 7*. Elsevier, 2007.

[17] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Proc. of the 5th Intl. Conference and Symposium on Logic Programming*, pages 1070–1080. The MIT Press, 1988.

[18] G. Grossi, M. Marchi, E. Pontelli, and A. Provetti. Improving the adjsolver algorithm for asp kernel programs. In S. Costantini and R. Watson, editors, *Proc. of ASP2007, 4th International Workshop on Answer Set Programming at ICLP07*, 2007.

[19] Claire Lefèvre and Pascal Nicolas. The first version of a new asp solver : Asperix. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *LPNMR*, volume 5753 of *Lecture Notes in Computer Science*, pages 522–527. Springer, 2009.

[20] Nicola Leone. Logic programming and nonmonotonic reasoning: From theory to systems and applications. In Chitta Baral, Gerhard Brewka, and John Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007*, 2007.

[21] Y. Lierler. Abstract answer set solvers. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming, Proc. of the 24th Intl. Conf., ICLP 2008*, volume 5366 of *Lecture Notes in Computer Science*, pages 377–391. Springer-Verlag, Berlin, 2008.

[22] Vladimir Lifschitz. Foundations of logic programming. In *Principles of Knowledge Representation*. CSLI Publications, 1996.

[23] Vladimir Lifschitz. Answer set planning. In *Proc. of the 16th Intl. Conference on Logic Programming*, pages 23–37, 1999.

[24] Victor W. Marek and Miroslaw Truszczyński. Autoepistemic logic. *Journal of the ACM*, 38(3):587–618, 1991.

[25] Victor W. Marek and Miroslaw Truszczyński. Computing intersection of autoepistemic expansions. In *Proceedings of the First International Workshop on Logic Programming and Non Monotonic Reasoning*, pages 35–70. The MIT Press, 1991.

[26] Victor W. Marek and Miroslaw Truszczyński. *Stable logic programming - an alternative logic programming paradigm*, pages 375–398. Springer, 1999.

[27] Luca Padovani and Alessandro Provetti. Qsmodels: Asp planning in interactive gaming environment. In José Júlio Alferes and João Alexandre Leite, editors, *JELIA*, volume 3229 of *Lecture Notes in Computer Science*, pages 689–692. Springer, 2004.

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

[29] Miroslaw Truszczyński. Logic programming for knowledge representation. In Verónica Dahl and Ilkka Niemelä, editors, *Logic Programming, 23rd International Conference, ICLP 2007*, pages 76–88, 2007.