A Lower-Bound for Answer Set Solver Computation

Stefania Costantini[1] and Alessandro Provetti[2]

[1] Dip. d'Informatica, Università di L'Aquila
stefania.costantini@univaq.it
[2] Dip. di Fisica, Università di Messina
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 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. For that class of programs, given a program composed of n atoms a solver adopting this algorithm must perform no less than n steps. We then argue that this result can be extrapolated to every logic program.

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 [1], where solutions to a given problem are represented in terms of selected models (answer sets) of the corresponding logic program [2, 3]. 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 [4–8] and the references therein). Efficient inference engines, or *ASP Solvers*, are available [9] and can be freely downloaded by potential users.

Recently, Yuliya Lierer has proposed [10] an abstract framework for describing the algorithm underlying many of the existing answer set solvers, considering in particular SMODELS and SUP (we can say that [10] describes an "abstract solver"). The abstract solver encompasses the main optimization strategies adopted by actual solvers, and primarily by SMODELS (described in [11]). In fact, SMODELS 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 [12]. In particular, deciding the existence of an answer set has been proved NP-complete in [13] and the same for deciding whether an atom is a member of some answer set (proved in [14]).

A topic that has received less attention in the literature concerns the least number of steps that a solver relying upon this algorithm actually performs in order to establish whether a given program is inconsistent, i.e., a lower bound for ASP solvers computation[3]. This is of interest in order to understand whether the existing strategies work well, or what could be done better. In this paper, we define a particular class of (inconsistent) programs, *OAH-programs*, and prove that for programs in this class the abstract solver must perform no less than n steps, i.e., its lower-bound complexity is $\Omega(n)$. We demonstrate that this class of program is significant as every non-trivial inconsistent program has an OAH-program as its "core", and therefore the result can be extrapolated to every logic program.

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

2.1 Lower Bounds

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[4].

As usual, constant factors c is disregarded and problems of size smaller than some n_0 are disregarded as well. Only the order of the lower bound is considered, as customary in terms of the function class expressing it. Let $f : \textit{natural numbers} \rightarrow \textit{real numbers}$. The set $\Omega(f)$ is defined as follows:

$$\Omega(f) = \{g : \textit{natural numbers} \rightarrow \textit{real numbers} \mid \textit{there exists } c > 0 \textit{ and } n_0 \textit{ natural numbers such that for all } n \geq n_0 : g(n) \geq c\, f(n)\}$$

I.e., $\Omega(f)$ comprises all functions g such that $g(n)$ is greater or equal to $c\, f(n)$ for some constant c and for instance size at least n_0. Equivalently, $\Omega(f)$ is the set of all functions that asymptotically grow at least as fast as f, disregarding constant factors.

[3] Victor Marek and Mirek Truszczynski, private communication.

[4] 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.

2.2 ASP in a nutshell

Below, we briefly recall the basics about Answer Set Programming [4, 2, 3]. In this logical framework, a problem can be encoded —by using a function-free logic language— as a set of properties and constraints which describe 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

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

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 $\mathit{not}\, a$ (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 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

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

By $\mathit{Bodies}(\Pi, H)$ or simply $\mathit{Bodies}(H)$ 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* [1]). Consider first the case of a ground[5] ASP-program P which does not involve negation. In this case, a set of atoms X is said to be an answer set for P if it is the (unique) least model of P. Such a definition is extended to any ground program P containing negation by considering the *reduct* P^X of P w.r.t. a set of atoms X obtained by means of the Gelfond-Lifschitz γ operator introduced in [1]. First, P^X is defined as the set of rules of the form $H \leftarrow L_1, \ldots, L_m$ for all rules of P such that X does not contain any of the atoms $L_{m+1}, \ldots, L_{m+n}$. Clearly, P^X does not involve negation. Let $\gamma(P, X) = J$ where J is the unique answer set of P^X. The set X is an answer set for P if it is a fixed point of γ, i.e., if $X = J$. Equivalently, X is an answer set for$P$ 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 $\Gamma(P, X) = Cn(P^X)$ where given set R of rules without negation, $Cn(R)$ is the smallest set of literals constructed from the atoms occurring in R which is closed under R and it is either consistent or equal to the set of all such literals.

Once a problem is described as an ASP-program P, its solutions (if any) are represented by the answer sets of P. Unlike other semantics, a logic program

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

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 [15].

Let us consider the program P consisting of the 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 P, then we rule-out the second of these answer sets, because 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.

To find the solutions of an ASP-program, an ASP-solver is used. Several solvers have became available [9], each of them being characterized by its own prominent valuable features. As it is well-known, ASP solvers produce the grounding of the given program as a first step, as they are able to find the answer sets of ground programs only[6].

The expressive power of ASP, as well as, its computational complexity have been deeply investigated. The interested reader can refer, for instance, to [12]. In particular, deciding the existence of an answer set has been proved NP-complete in [13] and the same for deciding whether an atom is a member of some answer set (proved in [14]). The reader can also see [4, 16], among others, for a presentation of ASP as a tool for declarative problem-solving.

2.3 Kernel Programs

Below we summarize the features of a special class of logic programs, kernel programs, introduced in [17] and discussed at length in [18] and [19].

[6] Work is under way both theoretically and practically to overcome at least partially this limitation. However, at present almost all ASP solvers perform the grounding.

The kernel form is a *normal* form, in the sense that (as proved in [18]) 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 on the one hand the literals that are certainly true/false in all the answer set and on the other hand the literals which are irrelevant for checking the consistency of the given program. As it is well-known, the former literals can be constructed from the atoms which are true and false w.r.t. the *well-founded semantics* of logic programs [20]. This semantics is three-valued and provides the set of atoms which are deemed true and false, where the other atoms are assumed to have truth value *undefined*. If Π is a logic program, we denote by $WFS(\Pi) = \langle T, F\rangle$ the well-founded model of Π. The well-founded model of program Π can be obtained in the form of the set of literal which are true/false [21] by computing $\Gamma^2(P, \emptyset)$ $= \Gamma(P, \Gamma(P, \emptyset))$. A program Π is said WFS-irreducible whenever $WFS(\Pi) = \langle \emptyset, \emptyset\rangle$. That is, in WFS-irreducible programs all the atoms are undefined under the well-founded semantics. As discussed in [22], 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 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 WFS-irreducible;*
2. *every rule has its body composed of negative literals only;*
3. *every atom in Π occurs in the body of some rule;*

It is easy to see that, in kernel programs, each atoms occurs as the head of some rule and, being undefined under the 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 [19]). Clearly, there are no facts.

For programs in kernel normal form, every *supported* model is stable [23, 22], where a supported model M is such that for every atom $a \in M$ some $B \in \mathit{Bodies}(a)$ 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 [24].

The kernel normal form can be obtained by means of a normalization algorithm which is computable in polynomial time modulo however a preliminary call to the program rewriting w.r.t. the WFS semantics described in [25]. This algorithm, that we call BDFZ, transforms a given program Π into a (unique) program remainder $\hat{\Pi}$ obtained by means of a straightforward extension of the above-mentioned Gelfond-Lifschitz operator, i.e., by deleting every rule instance with a body literal which is false w.r.t. $WFS(\Pi)$, and removing from the remaining rule instances the body literals which are true w.r.t. $WFS(\Pi)$. As atoms involved in positive cycles, if not supported differently, are false w.r.t. $WFS(\Pi)$, the algorithm is able to get rid of positive cycles. In particular, the BDFZ algorithm has the following features:

– it performs the preliminary grounding of Π;
– its main part consists of a confluent rewriting system, and
– it performs loop detection, thus deleting from the program all positive loops.

The kernelization algorithm (formally described in [26]) 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 rule instances 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 that of eliminating all the remaining positive atoms, that constitute "intermediate steps" between relevant atoms, and are immaterial to the existence and number of answer sets.

As proved in [22], 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.

2.4 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. For any set M of literals, by M^+ we take the set of positive literals occurring in M.

The abstract solver is described by means of *steps*, or *transition rules*, that can be applied to *states*. Given set σ of atoms, a state is either $\emptyset$, or *FailState*, or a list M of literals (without repetitions). Each literal l in M can be *annotated* as 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 the empty state 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 annotation) occur in M. In the rest of the paper, we

will assume σ to be the atoms occurring in the given program Π. 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 [10] 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):
$M \Rightarrow \mathit{FailState}$ 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 some strategic aspects proper of 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) as this has proved advantageous in practice. 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 prior than l^d in M the assignment would be canceled, and then l would be decided again, though in a new context.

Unit Propagate (UP):
$M \Rightarrow M\ a$ if $a :- B$ 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):
$M \Rightarrow M\ \mathit{not}\, a$ if for all $B \in \mathit{Bodies}(a)$, $\overline{B} \cap M \neq \emptyset$

I.e., the negation $\mathit{not}\, a$ 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 $a :- B$, $a \in M$ (whatever its annotation in M), and for all $B' \neq B$, $B' \in \mathit{Bodies}(a)$ 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 $a :- l, B, not\, a \in M$, 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 [10], 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 apply another transition rule called *Unfounded* that is needed in order to deem false all the atoms that in Π are involved in positive circularities and cannot be deemed true by any 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 Lower Bound for SMODELS-like Algorithm

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 two *odd loops*, namely $p :- not\, p$ and $q :- not\, q$, and as it is well-known the existence of odd loops may cause inconsistency. The former one might in principle be "rescued" by the conjunction $not\, a_1, not\, a_2$ that in terms of [19] can be called AND-handle: 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 instead might in principle be "rescued" by the same conjunction $not\, a_1, not\, a_2$ that in terms of [19] can be called OR-handle: if both literals are deemed true (i.e., if both a_1 or a_2 are false) then q becomes true, thus overcoming the contradictory rule. However, the two conditions are clearly in contrast with each other, and thus the whole program is inconsistent. This might be easily seen either on the EDG (Extended Dependency Graph, [17]) or even better on the Cycle Graph [19] corresponding to the program itself.

Without this "structural" information, let us try to assess how the abstract answer set solver $ATLEAST_{\Pi}$ will behave. 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 more step is possible. The final state can be *FailState* in case the last but final is an inconsistent state. 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 *BT* or *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 [10] we indicate on the right of the current state (other than *FailState*) the transition which is applied (for coinciseness, 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.
$\emptyset \Rightarrow D$
$q^d \Rightarrow BT$

$q^d, not\, a_1, not\, a_2 \Rightarrow ARC$
$q^d, not\, a_1, not\, a_2 \Rightarrow UP$
$q^d, not\, a_1, not\, a_2, b_1 \Rightarrow UP$
$q^d, not\, a_1, not\, a_2, b_1, b_2 \Rightarrow D$
$q^d, not\, a_1, not\, a_2, b_1, b_2, p^d \Rightarrow BT$
$q^d, not\, a_1, not\, a_2, b_1, b_2, p^d, p \Rightarrow B$
$q^d, not\, a_1, not\, a_2, b_1, b_2, not\, p \Rightarrow UP$
$q^d, not\, a_1, not\, a_2, b_1, b_2, not\, p, p \Rightarrow B$
$not\, q\ \ldots$

I.e., as q is the head of two rules where the body of the first one ($not\, q$) is in constrast with the decision, the solver can apply *Backchain True* on the second rule in order to try to justify its truth. After two steps of unit propagation, the solver is forced to decide p, which leads to an inconsistency (via *Backchain True* on the unique rule defining p) and to a backtracking, which via the assertion of $not\, p$ leads to another inconsistency and thus to backtrack the decision on q.

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:

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

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 get q but we then run quickly into a contradiction on p, as all conditions of its only rule become true but one ($not\, p$) that can be derived by means of *Backchain True* thus determining inconsistency and backtracking on the decision of p which however leads to inconsistency again, which forces to retract the decision on f and assert $not\, b_2$ which gives a_2.

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

$not\, a_1, b_1,\ {b_2}^d,\ not\, a_2,\ q \Rightarrow D$ (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.

$not\, a_1, b_1 \Rightarrow D$
$not\, a_1, b_1,\ {a_2}^d \Rightarrow BF (\text{or} \Rightarrow ARC)$
$not\, a_1, b_1,\ {a_2}^d,\ not\, b_2 \Rightarrow ARC$
$not\, a_1, b_1,\ {a_2}^d,\ not\, b_2,\ not\, p \Rightarrow D$ (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 should backtrack this decision and then the execution would proceed as before (with some modifications) after having performed more steps. If the whole computation would have 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 starting by deciding b_1

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

Now, deciding b_2 will quickly lead to an inconsistency on p.

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

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

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

${b_1}^d,\ not\, a_1 \Rightarrow D$
${b_1}^d,\ not\, a_1,\ {a_2}^d \Rightarrow BF$(or $\Rightarrow ARC$)
${b_1}^d,\ not\, a_1,\ {a_2}^d,\ not\, b_2 \Rightarrow ARC$
${b_1}^d,\ not\, a_1,\ {a_2}^d,\ not\, b_2,\ not\, p \Rightarrow D$
${b_1}^d,\ not\, a_1,\ {a_2}^d,\ not\, b_2,\ not\, p,\ q^d \Rightarrow ARC$
${b_1}^d,\ not\, a_1,\ {a_2}^d,\ not\, b_2,\ not\, p,\ q^d,\ not\, q \Rightarrow B$
${b_1}^d,\ not\, a_1,\ {a_2}^d,\ not\, b_2,\ not\, p,\ not\, q \Rightarrow UP$
$not\, b_1 \Rightarrow UP$
$not\, b_1,\ a_1$ (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 $ATLEAST_{\Pi}$ can perform before deciding that Π_6 is inconsistent belongs to $\Sigma(n)$.

4 Generalization

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

Definition 2. *An OAH-program Π_n has the following structure:*

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

$$q : - \, not\, q. \quad (2')$$
$$q : - \, not\, a_1, \ldots, not\, a_k. \quad (2'')$$

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

The number of composing atoms is $n = 2k + 2$ and there are $n + 1$ rules. as each atom occurs in the head of just one rule, except q which occurs as the head of two rules.

As said in previous section: the body of rule (1) is called, in the terminology of [19], an AND handle, and for the program to be consistent at least one of the composing literals must be false (thus making the AND handle *active*), so that the head becomes false as well; the body of rule (2") is called instead an OR handle, and for the program to be consistent all the composing literals must be true, thus making the head true as well (*active* OR handle); otherwise in fact, no answer set 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 one literal, that should be simultaneously true and false. Therefore we have the following.

Proposition 1. *Every OAH-program Π_n is inconsistent, whatever the number n of composing atoms.*

Notice that the above-defined OAH programs include, in rules (1) and (2') respectively, 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 OAH1-programs, and introduce the classes of OAHn-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 OAHn-programs.

As it is well-known, inconsistency of logic programs stems from negative odd cycles: in fact, every program involving no negative odd cycle is consistent (the reader may refer, e.g., to [27] for a discussion). As discussed in depth in [19], 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 in OAH programs, there exists two odd cycles whose handles are incompatible. The same can be said for any logic program, as in fact Kernelization does not affect its underlying structure: 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 stable

models and skips intermediate steps. The only effect of kernelization 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 OAHn-programs consists in the following:
Observation
Every inconsistent logic program includes in its kernel counterpart either an unconstrained odd cycle or an OAHn-program (for some n).

Therefore, the following lower-bound result, that can be easily extended to every OAHn-program, can be extrapolated to hold for every logic program.

Theorem 1. *The abstract solver $ATLEAST_{\Pi}$ on an OAH-program Π_n performs $\Omega(n)$ steps.*

Proof (sketch) A simple possible strategy for performing the least possible number of steps is that of making the AND handle not active. This quickly falsifies the contradictory atom in rule (1), 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) [19] of an OAH-program [19] 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 [28] whose underlying principles have been applied in order to improve existing solvers [29].

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

The authors wish to thank Victor Marek and Mirek Truszczynski that raised this problem in the course of a private discussion at ICLP08 in Udine. The authors also wish to thank Yuliya Lierler for many useful and exciting discussions on this subject.

References

1. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In Kowalski, R., Bowen, K., eds.: Proc. of the 5th Intl. Conference and Symposium on Logic Programming, The MIT Press (1988) 1070–1080
2. Lifschitz, V.: Answer set planning. In: Proc. of the 16th Intl. Conference on Logic Programming. (1999) 23–37
3. Marek, V.W., Truszczyński, M. In: Stable logic programming - an alternative logic programming paradigm. Springer (1999) 375–398
4. Baral, C.: Knowledge representation, reasoning and declarative problem solving. Cambridge University Press (2003)
5. Anger, C., Schaub, T., Truszczyński, M.: ASPARAGUS – the Dagstuhl Initiative. ALP Newsletter **17**(3) (2004) See `http://asparagus.cs.uni-potsdam.de`.
6. Leone, N.: Logic programming and nonmonotonic reasoning: From theory to systems and applications. In Baral, C., Brewka, G., Schlipf, J., eds.: Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007. (2007)
7. Truszczyński, M.: Logic programming for knowledge representation. In Dahl, V., Niemelä, I., eds.: Logic Programming, 23rd International Conference, ICLP 2007. (2007) 76–88
8. Gelfond, M.: Answer sets. In: Handbook of Knowledge Representation, Chapter 7. Elsevier (2007)
9. : Web references for some ASP solvers
   ASSAT: `http://assat.cs.ust.hk`;
   Ccalc: `http://www.cs.utexas.edu/users/tag/ccalc`;
   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`.
10. Lierler, Y.: Abstract answer set solvers. In de la Banda, M.G., Pontelli, E., eds.: Logic Programming, Proc. of the 24th Intl. Conf., ICLP 2008. Volume 5366 of Lecture Notes in Computer Science., Springer-Verlag, Berlin (2008) 377–391
11. Simons, P.: Extending and Implementing the Stable Model Semantics. PhD thesis, Helsinki University of Technology (2000)
12. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Computing Surveys **33**(3) (2001) 374–425
13. Marek, V.W., Truszczyński, M.: Autoepistemic logic. Journal of the ACM **38**(3) (1991) 587–618
14. Marek, V.W., Truszczyński, M.: Computing intersection of autoepistemic expansions. In: Proceedings of the First International Workshop on Logic Programming and Non Monotonic Reasoning, The MIT Press (1991) 35–70
15. Apt, K.R., Bol, R.N.: Logic programming and negation: A survey. Journal of Logic Programming **19/20** (1994) 9–72
16. Dovier, A., Formisano, A., Pontelli, E.: A comparison of CLP(FD) and ASP solutions to NP-complete problems. In Gabbrielli, M., Gupta, G., eds.: Logic Programming, 21st International Conference, ICLP 2005, Proceedings. Volume 3668 of LNCS., Springer (2005) 67–82
17. Brignoli, G., Costantini, S., D'Antona, O., Provetti, A.: Characterizing and computing stable models of logic programs: the non–stratified case. In: Proc. of the 1999 Conference on Information Technology, Bhubaneswar, India. (1999)

18. Costantini, S., Provetti, A.: Normal forms for answer sets programming. J. on Theory and Practice of Logic Programming **5**(6) (2005)
19. Costantini, S.: On the existence of stable models of non-stratified logic programs. J. on Theory and Practice of Logic Programming **6**(1-2) (2006)
20. Gelder, A.V., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. Journal of the ACM **38**(3) (1991) 620–650
21. Lifschitz, V.: Foundations of logic programming (1996)
22. Costantini, S.: Contributions to the stable model semantics of logic programs with negation. Theoretical Computer Science **149** (1995) (prelim. version in Proc. of LPNMR93).
23. Costantini, S.: Contributions to the stable model semantics of logic programs with negation. In Nerode, A., Subrahmanian, V., eds.: Logic Programming and Non-Monotonic Reasoning, Proceedings of the 2nd International Workshop LPNMR93. (1993)
24. Fages, F.: Consistency of Clark's completion and existence of stable models. Methods of Logic in Computer Science **1** (1994) 51–60
25.
26. Costantini, S., Provetti, A.: Computing the kernel normal form of answer set programs. submitted, can be obtained form the authors (2009)
27. F. Fages, title = Consistency of Clark's completion and existence of stable models, j..M.v...p...y...
28. Bertoni, A., Grossi, G., Provetti, A., Kreinovich, V., Tari, L.: The prospect for answer set computation by a genetic model. In: Proc. of the AAAI Spring Symposium ASP 2001, AAAI press (2001) 1–5
29. Grossi, G., Marchi, M., Pontelli, E., Provetti, A.: Improving the adjsolver algorithm for asp kernel programs. In Costantini, S., Watson, R., eds.: Proc. of ASP2007, 4th International Workshop on Answer Set Programming at ICLP07. (2007)