Learning and Using Domain-Specific Heuristics in ASP Solvers

Marcello Balduccini
Kodak Research Laboratories
Eastman Kodak Company
Rochester, NY 14650-2102 USA
marcello.balduccini@gmail.com

In spite of the improvements in the performance of many solvers for model-based languages, it is still possible for the search algorithm to focus on the wrong areas of the search space, preventing the solver from returning a solution in an acceptable amount of time. This prospect is a real concern e.g. in an industrial setting, where users typically expect consistent performance. To overcome this problem, we propose a framework that allows learning and using domain-specific heuristics in the solvers. The learning is done offline, on representative instances from the target domain, and the learned heuristics are then used for choice-point selection. In this paper we focus on Answer Set Programming (ASP) solvers. In our experiments, the introduction of domain-specific heuristics improved performance quite substantially on hard instances, and in particular made overall performance more consistent by reducing the number of cases in which the solver timed out

Keywords: answer set programming, solvers, domain-specific heuristics

1. Introduction

One of the key elements for the improvement of the performance of solvers for Answer Set Programming (ASP) [14,22] consists in the development of good heuristics for guiding the exploration of the search space. Naturally, there are cases when even the best heuristics still perform badly. This is typically due to the fact that such heuristics are general-purpose, and thus may not perform well on domains that substantially deviate from the norm. When this happens, the performance degradation is often dramatic, even to the point that the solver may need to be terminated before returning an answer. This prospect is a real concern when one is considering using such a solver in an industrial application, in which the solver will act as part of a black-box from which users typically expect consistent performance.

Various methods have been proposed in the literature to improve solver stability. A rather successful technique involves using a collection, or *portfolio*, of solvers, rather than just one. The portfolio is initially analyzed against benchmark problems in order to identify which solvers are best suited to which domains. At run-time, the syntactic features of the program in input are used, together with the information collected on the portfolio, to select the most promising solver. Portfolio-based approaches have been successfully applied in various areas, such as SAT solvers (e.g. SATzilla [20]), quantified Boolean formula (QBF) solvers (e.g. AQME [27]) and ASP (CLASPFOLIO [12]).

A related approach [30] from the area of QBFs consists in making multiple heuristics available to a single solver. At each decision point, the solver selects the most promising heuristic. An offline learning method is used to train the algorithm for the selection of the most promising heuristic.

A different approach consists in having the solver adapt to the problem in input at run-time by using *online* learning techniques. This is the case of the clause learning and conflict learning techniques that have been quite successful in SAT and ASP solvers (see e.g. [16,11], but also [23]), and have brought about substantial performance improvements. One drawback of this approach is that learning is limited to the current program, and the information that has been learned in one run cannot be used in later runs of the solver.

In this paper we propose a framework, called DORS, in which the solver learns domain-specific heuristics while solving instances from that domain. The learned heuristics can then be used in solving further instances from the domain. The learning occurs offline, in the sense that what is learned while computing the models of a program does not affect that execution of the solver. However, the updated heuristics can be immediately used in the following runs.

At this point of development, the DORS framework is aimed at DPLL-style solvers, i.e. solvers whose algorithms follow the lines of the DPLL procedure [8,7]. Moreover, here we focus on ASP solvers, although the DORS framework could be easily applied to DPLL-style solvers from other areas, such as SAT and constraint programming solvers. Although one could argue that in the past few years the performance of some non-DPLL-style solvers has been better than that of DPLL-style solvers, for various reasons DPLL-style solvers are still used for several applications (e.g. DLV, [6,17]), and developing techniques that make them more stable for practical use is important.

The specific learning technique used here is intendedly simple, but as we will show later it appears to be quite effective. In our experimental evaluation, the use of domain-specific heuristics yielded remarkable performance improvements on many hard instances, and in particular made overall performance more consistent by reducing the number of cases in which the solver timed out.

This paper is organized as follows. In the next section we give some background on ASP. Next, we discuss the basic search algorithm used in DPLL-style ASP solvers. Then, in Section 4, we present the DORS framework. Experimental results are discussed in Section 5. Section 6 discusses related work. Finally, in Section 7, we draw conclusions.

2. Syntax and Semantics of ASP

The syntax and semantics of ASP are defined as follows. More details can be found in [14]. Let Σ be a propositional signature containing constant, function and predicate symbols. Terms and atoms are formed as usual in first-order logic. A literal is either an atom a or its strong (also called classical or epistemic) negation $\neg a$. The set of literals formed from Σ is denoted by $lit(\Sigma)$. A *rule* is a statement of the form:

$$h_1 \ \vee \ \ldots \ \vee \ h_k \leftarrow l_1, \ldots, l_m, \\ \text{not } l_{m+1}, \ldots, \text{not } l_n$$

$k+m+n > 0$. h_i's and l_i's are ground literals and *not* is the so-called *default negation.* The intuitive meaning of the rule is that an agent who believes $\{l_1, \ldots, l_m\}$ and has no reason to believe $\{l_{m+1}, \ldots, l_n\}$, has to believe one of h_i's. The part of the statement to the left of $\leftarrow$ is called *head*; the part to its right is called *body.* Furthermore, $pos(r)$ denotes $\{l_1, \ldots, l_m\}$ and $neg(r) = \{l_{m+1}, \ldots, l_n\}$. Symbol $\leftarrow$ can be omitted if no l_i's are specified. Often, rules of the form $h \leftarrow \text{not } h, l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n$ are abbreviated into $\leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n$, and called *constraints.* The intuitive meaning of a constraint is that its body must not be satisfied. A *program* is a pair $\langle \Sigma, \Pi \rangle$, where Σ is a signature and Π is a set of rules over Σ. We often denote programs using only the second element of the pair, and let the signature be defined implicitly. In this case, the signature of Π is denoted by $\Sigma(\Pi)$. A non-ground rule, i.e. a rule containing variables, is interpreted as the shorthand for the set of rules obtained by replacing the variables with all the possible ground terms, called *grounding* of the rule. A non-ground program is viewed as the collection of the groundings of its rules.

The semantics of ASP is defined in two steps. The first step consists in giving the semantics of *naf-free programs*, i.e. programs that do not contain default negation.

A literal l is *satisfied* by a consistent set of literals S (denoted by $S \models l$) if $l \in S$. If l is not satisfied by S, we write $S \not\models l$. An expression *not* l, where l is a literal, is satisfied by S iff $S \not\models l$. By *extended literal* we mean a literal l or the expression *not* l. A set of extended literals is satisfied by S if each element of the set is satisfied by S. A set S of literals is *consistent* if there is no atom a such that $a \in S$ and $\neg a \in S$. A consistent set S of literals is *closed under a naf-free program* Π if, for every rule of Π such that the body of the rule is satisfied by S, $\{h_1, h_2, \ldots, h_k\} \cap S \neq \emptyset$.

A consistent set S of literals is an *answer set of a naf-free program* Π if S is closed under Π and S is set-theoretically minimal among the sets satisfying this property. Programs without default negation and whose rules have at most one literal in the head are called *definite.* It can be shown that definite programs have at most one answer set. The answer set of a definite program Π is denoted by $ans(\Pi)$.

The second step of the definition of the semantics consists in reducing the computation of answer sets of ASP programs to the computation of the answer sets of naf-free programs.

Let Π be an arbitrary ASP program. For any set S of literals, the *reduct of* Π *w.r.t.* S, denoted by Π^S, is the program obtained from Π by deleting:

- each rule r such that $neg(r) \cap S \neq \emptyset$;
- all formulas of the form not l in the bodies of the remaining rules.

Notice that Π^S is a naf-free program. A set S of literals is *an answer set of an ASP program* Π if it is an answer set of Π^S. A program that has no answer sets is called *inconsistent.*

Because a convenient representation of alternatives is often important in the formalization of knowledge, the language of ASP has been extended in various ways to allow compact encodings (e.g. [25,6,2]). One frequently used construct is that of *constraint literals* [25], available in SMODELS [25], CLASP [11], and associated grounders. Constraint literals are expressions of the form $m\{l_1, l_2, \ldots, l_k\}n$, where m and n are non-negative integers and l_i's are literals as defined above. A constraint literal is satisfied whenever the number of literals that hold from $\{l_1, \ldots, l_k\}$ is between m and n, inclusive. Using constraint literals, the choice between p and q, under some set of conditions Γ, can be compactly encoded by the rule $1\{p, q\}1 \leftarrow \Gamma$. A rule of this form is called *choice rule.*

When solving sets of problems from a given domain of interest, ASP programs are often divided into a *domain description* and a *problem instance.* Intuitively, the domain description encodes a description of the problem domain and of the solutions, while the problem instance encodes a specific problem from the domain.

3. Search in ASP Solvers

The search algorithm implemented by many ASP solvers (e.g. SMODELS [31], DLV [19]) follows the lines of the DPLL procedure [8,7,13,15,21]. The algorithm for the computation of a single answer set, which we will later refer to as *standard algorithm,* is shown in Figure 1. The algorithm is based on the idea of growing a particular set of (ground) literals, often called a partial answer set, until it is either shown to be an answer set of the program, or it becomes inconsistent. To achieve this, guesses have to be made as to which literals may be in the answer set. Let us now describe the algorithm more precisely. Given an extended literal e, $not(e)$ denotes the expression *not* l if $e = l$ and it denotes l if $e =$ *not* l. Algorithm *solve* takes as input a program Π, and a partial answer set A, which is a set of extended literals. A is initially empty. Next, function `expand` [31] uses simple properties of the answer set semantics to (1) derive a collection of extended literals that follow from Π and A, and add them to the partial answer set, or (2) determine that the program is inconsistent. If the result of `expand` is an answer set of Π, the algorithm returns it (and terminates). If instead the program is discovered to be inconsistent, the algorithm backtracks. In all other cases, the partial answer set is still incomplete but consistent. Then, function *choose_literal* selects an extended literal e such that neither e nor $not(e)$ occur in B. This is called the *choice literal* or *choice point.* The algorithm then calls itself recursively in order to find an answer set of Π from the partial answer set $B \cup \{e\}$. If one such answer set is found, then the algorithm returns it. If instead no answer set is found, then the algorithm attempts to find an answer set of Π that contains $B \cup \{not(e)\}$. If the attempt succeeds, the answer set is returned. Otherwise, the algorithm returns no model ($\bot$).

To see how the choices made by *choose_literal* influence the number of choice points picked by the algorithm, and ultimately its performance, consider the program:

$$P_1 = \begin{cases} p \leftarrow \text{not } q. \quad\quad q \leftarrow \text{not } p. \\ \\ r. \\ \leftarrow p, r. \\ \leftarrow q, \text{not } s. \\ \\ u(X) \leftarrow t(X), \text{not } v(X). \\ v(X) \leftarrow t(X), \text{not } u(X). \\ \\ t(0).\ t(1).\ \ldots\ t(1000). \end{cases}$$

The program is clearly inconsistent. In fact, the first two rules force either p or q to hold, but the next three rules forbid p and q from holding simultaneously. In state-of-the-art solvers, `expand` is actually sufficient to determine the inconsistency of the program. On the other hand, for illustrative purposes, let us hypothesize that inconsistency is not detected, and that *choose_literal* is called. Thus, if the first call to *choose_literal* were to select e.g. *not* p, then the following call to `expand` would conclude that q must hold, and that incon-

```
function solve ( $\Pi$ : $Program$, $A$ : $Set\ of\ Extended\ Literals$ )
    $A$ := expand($\Pi$, $A$);
    if ($A$ is answer set of $\Pi$) then return $A$;
    if ($A$ is not consistent or $A$ is complete) then
        return $\bot$;
    $e$ := choose_literal($\Pi$, $A$);
    $A'$ := solve($\Pi$, $A \cup \{e\}$);
    if ($A' = \bot$) then $A'$ := solve($\Pi$, $A \cup \{not(e)\}$);
    return $A'$;
```

Fig. 1. Standard Search Algorithm for ASP

sistency follows (since s is not defined by any rule and thus the body of the corresponding constraint is satisfied).

The algorithm would then backtrack and select p. This time, `expand` would derive inconsistency from the fact that the body of the first constraint is satisfied. Hence, the algorithm would return $\bot$ (no model). However, consider what would happen if *choose_literal* were to select $u(0)$ instead of *not* p. Function `expand` would derive the consequence *not* $v(0)$. The choice would not cause inconsistency.

Then, the algorithm would recurse, and possibly select say $u(1)$. As before, `expand` would not detect any inconsistency, and allow the algorithm to recurse again. Suppose now *choose_literal* were to pick *not* p. Following the same steps outlined earlier, the algorithm would derive inconsistency. Upon backtracking, the algorithm would also derive inconsistency from choosing p. However, the finding would only affect the current branch of the search stemming from the selection of $u(1)$, and the algorithm would then backtrack, select *not* $u(1)$, and recurse.

At this point, the algorithm would be again free to select any of the remaining $u(X)$ literals, which from an intuitive point of view means going in the wrong direction. Even if the algorithm were to select *not* p right away, it would still have to backtrack over the choice of $u(0)$ and explore the corresponding branch of the search tree that starts from *not* $u(0)$ before finally concluding that the program is inconsistent. If instead *choose_literal* were to choose *not* p at an even later point in the search process, the amount of backtracking that would be needed to determine the inconsistency of the program would be even larger, which would substantially affect performance of the solver.

In order to reduce the chances of *choose_literal* making "wrong" selections, modern solvers base literal selection on carefully designed heuristics. For example, in SMODELS the selection is roughly based on maximizing the number of consequences that can be derived after selecting the given extended literal [31]. These techniques work well in a number of cases, but not always. In fact, particular features of the program can confuse the heuristics. When this happens at an early stage of the search process, the effect is often disastrous, causing the solver to fail to return an answer in an acceptable amount of time. Particularly problematic from a practical perspective is the fact that small elaborations of the program in input may result in very different performances of the heuristics.

One method to improve solver performance is that of restarting the search if selections made by *choose_literal* do not appear to lead towards a solution (with additional bookkeeping required to ensure the completeness of the solver). Such determination is done using heuristics – often by considering the number of conflicts recently detected (see e.g. [24,10,18]). The benefit of using restarts varies from domain to domain, and clearly depends on the heuristics used to trigger the restarts. In Section 5 we discuss the role of restarts in our experimental evaluation.

As we mentioned in the introduction, another way of limiting the effect of wrong selections by *choose_literal* is that of allowing the solver to learn about relevant conflicts at run-time. Once learned, the information about conflicts can be used for the early pruning of other branches of the search space (e.g. [16,11]). Although this technique has proven to be extremely effective, it does not address directly the issue of *choose_literal* making wrong choices, but rather curbs the problem by making some of those choices impossible after learning has taken place, or by allowing to quickly backtrack after a wrong choice has been made. Furthermore, because the learning occurs at run-time, during the initial phase of the computation in which learning has not yet occurred, *choose_literal* may once again affect efficiency negatively by taking the search process in the wrong direction. Finally, whatever has been learned in one execution of the algorithm is discarded upon termination, and cannot be used in later runs.

In the next section, we describe a different approach, aimed at improving directly the selections made by *choose_literal* and at retaining what the algorithm has learned.

4. The DORS Framework

Our technique for learning domain-specific heuristics and using them for literal selection applies to the situation in which one is interested in solving a number of problem instances from a given domain. Such situations are rather common – in the context of ASP, a good example is provided by the Second Answer Set Programming Competition [9]. Moreover, this is particularly the case in industrial applications, where the application contains the domain description, and the user describes the instance using some interface (refer e.g. to [3]), which then automatically encodes the problem instance. For example, this happens frequently in applications such as automated planning, diagnosis, and system configuration [2,1,4,3,29].

The intuition behind the DORS framework is rather simple. Let us assume that we are given a domain description consisting of a set M of rules, and a problem instance I, so that $P_I = M \cup I$ is consistent. If the solver's heuristics match well the features of P_I, then the solver will find an answer set of P_I with little or no backtracking. Otherwise, the solver will explore a branch of the search space, detect inconsistency, backtrack over the latest choice(s), explore another branch, and so on. The larger the number of times the solver needs to backtrack, the worse the solver's performance will be.

Once the computation has been completed, let b_I be the *solving branch* for P_I, i.e. the branch that was explored last by the solver and led to an answer set. An obvious way to improve performance of the solver on instance I would be to modify the heuristics so that, next time the solver is presented with I, branch b_I will be explored first. This can be accomplished by identifying the choice points that characterize b_I, and by forcing the solver's heuristic to make those decisions first, and behave as usual if the branch does not lead to an answer set. Given a set of instances $\{I_1, \ldots, I_k\}$, and corresponding solution branches $B = \{b_{I_1}, \ldots, b_{I_k}\}$, one could tune the solver for those instances by forcing the heuristics to explore the branches in B first.

Doing so, however, is unlikely to improve performance on other instances, unless the solution branches from B happen to be solution branches for the other instances as well. On the other hand, let us consider the choice points that describe the solution branches: it seems reasonable that, if a particular choice occurs frequently in B, then that choice is likely to be a good choice for other instances as well. In the general case, one can apply machine learning techniques to extract choices from B, rank them, e.g. based on their frequency, and then have the solver's heuristics use the choices extracted, rather than directly using the branches from B. The larger the original set of instances and the more common the choices in their solution branches, the more likely it is that the solver will quickly find a solution branch for new instances.

Clearly, the learned heuristics are unlikely to yield good performance if the application domain changes, and thus they are *domain-specific*. One should also note that the learning of domain-specific heuristics relies on the availability of training instances that are satisfiable. In fact, unsatisfiable instances by definition do not have solution branches, and thus provide no information for the tuning of the heuristics. On the other hand, once the learning has taken place, the solver can be applied to satisfiable as well as to unsatisfiable instances.

One possible issue with this form of learning is that, if the set of solving branches used for learning is large and contains few common choices, then the learned heuristics may cause the solver to explore a substantial number of branches before finding an answer set, reducing performance. This is especially the case when instances from the domain are different from each other in nature (e.g. when the instances include both planning tasks and diagnostic tasks, as in [3]). In order to avoid this, the DORS framework is designed to allow identifying subsets of domain instances, and learning separate heuristics for each subset.

Next, we provide a more precise description of our approach, discussing how choices made in previous runs of the algorithm can be extracted and combined for future use.

In technical terms, the final result of the learning process can be viewed as the creation of a *policy* (see e.g. [5] for a comprehensive introduction on the topic), that is, of a mapping from states to probabilities of taking each available decision. To achieve this, the algorithm from Figure 1 is first of all modified to maintain a record of the choice points selected, and to return the list of such choice points (which characterize the solution branch) together with the answer set, whenever one is found. The modified algorithm is shown in Figure 2. In the algorithm, the list of choice points is stored in variable S. Symbol $\circ$ represents concatenation. When *solvecp* is initially invoked, S is the empty list.

Now we turn our attention to combining the information collected by *solvecp* into domain-specific heuristics. Given the domain description M and a problem instance I that is to be used to learn the domain-specific heuristics, the *decision-sequence* of I (denoted by $d(I)$) is $\bot$ if $solvecp(I \cup M, \emptyset, \emptyset) = \bot$ and S if $solvecp(I \cup M, \emptyset, \emptyset) = \langle A, S \rangle$ for some A. From now on, given a decision-sequence d, we denote its nth element by d_n. Moreover, given an extended literal e from d, $level(e, d)$ denotes the value i such that $d_i = e$ (e is guaranteed not to occur at more than one position by construction of the decision-sequence in *solvecp*). Intuitively, $level(e, d)$ represents the level in the decision tree at which e was selected. This notion is similar to that of *recursion level* from [28]. Notice that, by construction of the sequence of choice points in *solvecp*, if $d(I) \neq \bot$, then $d(I)$ only enumerates the choice points that led directly to the answer set. All the choice points that did not lead directly to it, in the sense that they were later backtracked upon, are in fact discarded every time the algorithm backtracks.

As mentioned earlier, in order to improve the accuracy of the learned heuristics, we allow dividing the class of problem instances in subclasses, and associate with each problem instance I an expression σ denoting the subclass it belongs to. The intuition is that using subclasses allows to further tailor the literal selection heuristics to the particular features of the problem instances. For example, in a planning domain, σ might be the maximum length of the plan. The subclass of a problem instance I is denoted by $\sigma(I)$.

Let $\mathcal{I}$ denote the set of all problem instances that will be used for the learning of the domain-specific heuristics. Next, we specify a way of determining how many times an extended literal e was selected at a certain level of the decision-sequences for the problem instances in $\mathcal{I}$. Given a non-negative integer δ, called the *scaling factor*, and subclass σ, the *occurrence count* of an extended literal e w.r.t. a level l is

$$\begin{aligned} o_{\delta,\sigma}(e, l, \mathcal{I}) = || \{ \ I \mid I \in \mathcal{I} \ \wedge \ \sigma(I) = \sigma \ \wedge \\ e \in d(I) \ \wedge \\ abs(l - level(e, d(I))) \leq \delta \ \} || \ . \end{aligned}$$

The scaling factor δ allows taking into account all the occurrences of e at a level in the interval $[l - \delta, l + \delta]$. If $\delta = 0$, then only the occurrences of e at level l are considered.

Let now $E = \{e_1, e_2, \ldots, e_k\}$ be a set of extended literals, representing possible choice points at some level l of the decision tree. The *set of best choice points* among E (w.r.t. l, I, σ) is:

$$\begin{aligned} best_\delta(l, E, \sigma, \mathcal{I}) = \{ e \mid e \in E \ \wedge \\ \forall e' \in E : \quad o_{\delta,\sigma}(e, l, \mathcal{I}) \geq o_{\delta,\sigma}(e', l, \mathcal{I}) \}. \end{aligned}$$

Intuitively, $best_\delta(l, E, \sigma, \mathcal{I})$ returns the choice points that, when taken at level l in the instances of subclass σ of $\mathcal{I}$ considered, most frequently led to an answer set without backtracking.

Function $best_\delta(l, E, \sigma, \mathcal{I})$ encodes the essence of the domain-specific heuristics, or, more precisely, the policy[1] for the selection of choice points. Algorithm *choose_literal* can now be extended to perform literal selection guided by the domain-specific heuristics. The modified algorithm, *choose_literal_dspec*, is shown in Figure 3. In *choose_literal_dspec*, argument T is the set of extended literals that have been returned by previous calls to the function. Elements of T are not considered in the computation of the set of best choice points; this is intended to force the function to select different extended literals upon backtracking and thus increase the breadth of the search. If $best_\delta(level, E', \sigma(I), \mathcal{I})$ is the empty

[1]We assume uniform probability of selection among the elements of the set returned by $best_\delta(l, E, \sigma, \mathcal{I})$.

```
function solvecp ( $\Pi$ : $Program$,
                   $A$ : $Set\ of\ Extended\ Literals$,
                   $S$ : $Ordered\ List\ of\ Extended\ Literals$ )
    $B$ := expand($\Pi$, $A$);
    if ($B$ is answer set of $\Pi$) then return $\langle B, S\rangle$;
    if ($B$ is not consistent or $B$ is complete) then
        return $\bot$;
    $e$ := choose_literal($\Pi$, $B$);
    $\langle B', S'\rangle$ := solve($\Pi$, $B \cup \{e\}$, $S \circ e$);
    if ($B' \neq \bot$) then return $\langle B', S'\rangle$;
    $\langle B', S'\rangle$ := solve($\Pi$, $B \cup \{not(e)\}$, $S \circ not(e)$);
    return $\langle B', S'\rangle$;
```

Fig. 2. Search Algorithm for ASP with Explicit Tracking of Choice Points

```
function choose_literal_dspec ( $\Pi$ : $Program$,
                                $\sigma$ : $Problem\ Subclass$,
                                $A$ : $Set\ of\ Extended\ Literals$,
                                $level$ : $Integer$,
                                $T$ : $Set\ of\ Extended\ Literals$,
                                $\mathcal{I}$ : $Set\ of\ Instances$,
                                $\delta$ : $Integer$)
    $L := lit(\Sigma(\Pi))$; $E := L \cup \{$not $l \mid l \in L\}$;
    $E' = \emptyset$;
    for each $e \in E$
        if ($e \notin A \wedge not(e) \notin A \wedge e \notin T$) then
            $E' := E' \cup \{e\}$;
    end for
    $B := best_\delta(level, E', \sigma, \mathcal{I})$;
    if ($B \neq \emptyset$) then $chosen := one\_element\_of(B)$;
                else $chosen := choose\_literal(\Pi, A)$;
    return $chosen$;
```

Fig. 3. Function for Literal Selection with Domain-Specific Heuristics

set, then *choose_literal_dspec* falls back to performing standard extended literal selection via *choose_literal*. This is for instances in which the learned heuristics do not prescribe any extended literal for the current decision level, or in which all the extended literals that the learned heuristics prescribed have already been tried. Modifying the standard solver's algorithm in order to use the domain-specific heuristics for choice-point selection is rather straightforward. A simple version, which for the most part follows the well-known iterative version of the SMODELS algorithm, is shown in Figure 4.

From a practical perspective, it should be noted that constraint literals are sometimes subject to transformations in the pre-processing components of ASP solvers. These transformations may involve the introduction of special atoms, which require a specific treatment within the DORS framework. More details on this topic can be found in the Appendix.

5. Experimental Evaluation

In this section we report on the experimental evaluation of the DORS framework. Our implementation was tested on several abstract problems from the Second ASP Programming Competition

```
function solve_dspec ( $\Pi$ : Program,
                       $\sigma$ : Problem Subclass,
                       $\mathcal{I}$ : Set of Instances,
                       $\delta$ : Scaling Factor )
 var $S$ : Stack of Sets of Extended Literals;
 var $B, T$ : Set of Extended Literals;
 var $terminate$ : Boolean;

    $S := \emptyset$;  $B := \emptyset$;  $T := \emptyset$;  $level := 1$;
    $terminate := false$;
    while ($terminate = false$)
        $B := expand(\Pi, B)$;
        if ($B$ is answer set of $\Pi$) then
            $terminate := true$;
        else
            if ($B$ inconsistent or $B$ complete) then
                if ($S = \emptyset$) then
                    $B := \bot$;
                    $terminate := true$;
                else
                    /* Backtrack */
                    $B := top(S)$;   $S := pop(S)$;
                    $level := level - 1$;
                end if
            else
                /* Select a choice point */
                $e := choose\_literal\_dspec(\Pi, \sigma, B, level,$
                                          $T, \mathcal{I}, \delta)$;
                $T := T \cup \{e\}$;
                $S := push(B \cup \{not(e)\}, S)$;
                $B := B \cup \{e\}$;   $level := level + 1$;
            end if
        end if
    end while
    return B;
```

Fig. 4. Search Algorithm for ASP with Domain-Specific Heuristics for Choice-Point Selection

[9], and on the task of planning for the Reaction Control System of the Space Shuttle [3].

The tests consisted in computing one answer set for every problem instance considered. The solver used in the experiments was SMODELS, which we modified to obtain implementations of algorithms *solvecp* and *solve_dspec*. It should be noted that we did not use CLASP for our experiments, since CLASP is not a DPLL-style solver, being rather based on conflict-driven clause learning (CDCL) (e.g. [16]). Although we believe that certain similarities between DPLL and CDCL make it technically possible to extend the DORS framework to CDCL-based systems, work on implementing the DORS framework within CLASP is still in the early stages. In the rest of the discussion, we refer to the implementation of *solve_dspec* within SMODELS as DSPEC.

The evaluation on the domains from the Second ASP Competition was performed as follows. For each domain, we randomly generated problem instances using the generating programs provided by the authors of the benchmarks. For the reasoning modules, we used the ones made available by the Potassco group, and available from `http://dtai.cs.kuleuven.be/events/ASP-competition/encodings.shtml`. The reasoning modules were modified to associate a name with certain rules, as explained in the Appendix. The change does not alter the semantics of the program, nor the performance of the solver.

For each domain, we used two sets of generating parameters and created 100 instances for each set of parameters. The parameters were selected randomly from the ranges recommended by each benchmark's authors. To make the instances relevant in the context of improving solver's robustness on hard instances, we constrained the parameter selection so that SMODELS would take 100 seconds or more to solve 20% to 40% of the instances. The constraint was enforced by performing random sampling over the instances defined by each set of parameters considered. The collection of instances can be found at `http://marcy.cjb.net/DORS/AIComm-2011-instances.tgz`.

As explained in Section 4, DORS allows identifying subclasses of a domain, and learning different domain-specific heuristics for each subclass. For our benchmarks, we followed the simple approach of grouping the instances in subclasses according to the parameters passed to the generating program when creating the instances. For example, all the instance of the Hierarchical Clustering domain that were generated to have 50 vertices, 10 levels, and at most 4 vertices in each cluster, were assigned to the subclass labeled $\langle 50, 10, 4\rangle$ (the reader may refer to `http://dtai.cs.kuleuven.be/events/ASP-competition/Benchmarks/HClustering.shtml` for more details on the problem description). Hence, each subclass contained 100 instances.

The first step of the evaluation consisted in running SMODELS on all the instances to form a baseline. a rather common method for testing the performance of machine learning algorithms, in which one (1) divides the available instances in a training set and a test set, (2) uses the training set for the learning part of the algorithm, and (3) uses the test set to evaluate the quality of what has been learned. Because in the practical use of ASP solvers it is unlikely for the exact same instance to be presented in input more than once, we chose to keep the training and test sets disjoint.

In the first set of benchmarks, the training and test sets were formed by random selection, so that each instance had a 10% probability of being selected for the test set. In order to compensate for any bias in the selection of the test set, we repeated the selection process twice, always starting from the original set of instances, and using a different random seed. DORS was evaluated separately on each partitioning of the set of instances (i.e. the domain-specific heuristics formed from one training set were only used for the corresponding test set), and the results from the two test sets were combined by averaging the time taken by DORS and counting the total number of timeouts triggered. For the experiments, a timeout of 6000 seconds was used.

For a given partitioning of the set of instances, DSPEC was first run on the training set in order to generate the domain-specific heuristics. Next, DSPEC was run on the test set, and using the domain-specific heuristics obtained from the training set. Finally, the performance of DSPEC on the test set was compared to the performance of SMODELS on the same set. The results are shown in Figure 5. In the figure, the instances are grouped according to the time it took SMODELS to solve them. The particular time ranges used in the figure were selected for the purpose of data visualization, but do not influence the results of the comparison. Each column corresponds to the instances that SMODELS solved within the given amount of time. So, for example, column $10s - 50s$ is for the instances that took at least 10 seconds, but less than 50. The "T/O" column is for the instances on which SMODELS timed out. For each column, row "num" gives the number of instances that belong to that category, e.g. in the Solitaire domain there were 2 instances that took SMODELS between 10 and 50 seconds. Row "T/O" contains the number of instances for which DSPEC timed out. Finally, row "avg" shows the average time taken by DSPEC to solve the instances in that category, disregarding the instances on which DSPEC timed out (if any).

The benchmarks in this figure and in the other figures in this section were run on a computer with an Intel i7 CPU at 2.93GHz running Linux. All the solvers were constrained to use at most 512MB of physical memory, and to run on a single processor.

In Figure 5, particularly interesting is the comparison between the two right-most columns, corresponding to instances that took 200 seconds or more, and the other columns, corresponding to instances that were solved relatively quickly. In the following discussion, we will refer to the instances reported in the two right-most columns as the "hard instances" and to the other instances as the "easy instances." In the figure, one can observe a remarkable difference in the performance of DSPEC between the easy instances and the hard instances.

In the easy instances, DSPEC performed either similarly to SMODELS (e.g. with a best performance of an average of 5 seconds for the instances that took SMODELS less than 10 seconds), or worse, with a worst case of 5 timeouts in the 19 instances that took SMODELS less than 10 seconds each. That is not entirely surprising, if one considers that the instances in these classes where solved by SMODELS with little or no backtracking. Hence, whenever the domain-specific heuristics do not succeed in taking the search directly to a solution, the extra backtracking that occurs as a result causes a performance degradation that is likely large compared to the SMODELS time. Investigation of the instances in which DSPEC timed out on the easy instances showed that in all cases the solver did not find a solution using the domain-specific heuristics, and reverted to the use of the standard SMODELS heuristics. The standard heuristic, because of its own brittleness, was not effective from the particular configuration of the search algorithm reached using the domain-specific heuristics.

On the other hand, in the hard instances, DSPEC performed remarkably well. Whereas SMODELS timed out 19 times out of a total of 27 hard instances, DSPEC only timed out 5 times. Moreover, the average times were low, with a best average of 49 seconds on the instances of the Solitaire domain for which SMODELS timed out.

As we explained earlier, the goal of this paper is to improve the performance of DPLL-style solvers. Nonetheless, we believe that it is important to allow the reader to understand how the performance of SMODELS and DSPEC compares to that of CLASP, which is currently one of the best performing ASP solvers. When run with CLASP, none of the instances from Figure 5 timed out. Even more remarkably, on average an answer set was found in less than 0.01 seconds.

For the second set of benchmarks, we adopted a different method for forming the training and test sets, aimed at assessing the robustness of the domain-specific heuristics learned. We began by analyzing the performance of SMODELS in the previous benchmark, and, for each domain, identified the set of generating parameters that gave the hardest set of instances; in line with the distinction made earlier between easy and hard instances, the metric for measuring the hardness of a set of generating parameters was the number of instances, among the ones generated with those parameters, for which SMODELS took 200 seconds or more. The 100 instances, obtained earlier, for the sets of generating parameters identified in this way were selected for use in the new benchmark.

Next, rather than selecting the test instances randomly, we used in the test set the instances that were solved by SMODELS in 50 seconds or more. With this selection method, we aimed at assessing how well domain-specific heuristics learned from easier instances perform on harder instances. The results of the evaluation are shown in Figure 6.

By inspecting the figure, one can observe that, once again, the domain-specific heuristics performed remarkably well on the hard instances of the test set, in spite the fact that the training process was performed using only easy instances. Overall, DSPEC timed out 18 times while SMODELS timed out 76 times; DSPEC had a best average time of 51 seconds on the instances of the Hierarchical domain for which SMODELS timed out, and improvements of one order of magnitude in the same category for the other domains. As before, the performance on the easy instances was not as good as the performance on the hard instances, although it was more consistent than in the previous experiment, as demonstrated in particular by the lack of timeouts. It should also be noted that these results were obtained using domain-specific heuristics often trained on a relatively small number of instances. For example, in the Solitaire domain the training set consisted only of 32% of the instances (across all the domains, the training set consisted on average of 61.5% of the instances).

As a reference, the performance of CLASP on the same test instances is shown in Figure 7. It is interesting to note that even CLASP timed out in 2 occasions (one instance of Solitaire and one of GraphPartitioning). Moreover, GraphPartitioning and 15Puzzle appear to be by far the hardest domains, with average times that are respectively 3 and 2 orders of magnitude larger than the times for the other domains. This is not surprising considering that, as reported in the figure, the average

Domain: Solitaire						
	$< 10s$	10s-50s	50s-100s	100s-200s	200s-6000s	T/O
num	19	2	0	1	0	15
T/O	5	0	0	0	0	3
avg	335.032	19.374	N/A	8.781	N/A	49.409

Domain: HierarchicalClustering						
	$< 10s$	10s-50s	50s-100s	100s-200s	200s-6000s	T/O
num	32	0	0	1	3	2
T/O	2	0	0	0	0	2
avg	15.451	N/A	N/A	21.686	1797.443	N/A

Domain: GraphPartitioning						
	$< 10s$	10s-50s	50s-100s	100s-200s	200s-6000s	T/O
num	20	6	7	3	3	0
T/O	0	0	0	0	0	0
avg	4.971	12.191	101.857	166.456	756.591	N/A

Domain: 15Puzzle						
	$< 10s$	10s-50s	50s-100s	100s-200s	200s-6000s	T/O
num	19	12	3	1	2	2
T/O	0	0	0	0	0	0
avg	10.716	63.331	15.219	28.097	195.021	263.482

Fig. 5. Performance comparison for 2nd ASP Competition Domains; 90% of instances used for training.

numbers of conflicts and restarts in those domains are rather large compared to the numbers for the other two domains; furthermore, all the instances in GraphPartitioning are non-tight.

Our experimental evaluation also included problem instances from the task of planning for the Reaction Control System (RCS) of the Space Shuttle [26,3].

A collection of problem instances from the domain of the RCS is publicly available, at `http://www.krlab.cs.ttu.edu/Software/Download/`, together with the ASP encodings of the model of the RCS and of various reasoning modules. The interested reader may refer to [26] for a description of the instances. For our evaluation, we used the public instances with no electrical faults, 8 and 10 mechanical faults respectively, and for which a plan of length 6 or less was found in the experiments discussed in [26,3]. The aim of the last constraint was to ensure the availability of a sufficient number of training instances to form the domain-specific heuristics. Using these instances, we compared the performance of SMODELS and DSPEC using maximum plan lengths ranging between 6 and 14 steps.

As before, first we ran all the instances with SMODELS and a timeout of 6000 seconds in order to form the baseline. In order to focus on the robustness of the heuristics, the partitioning in training and test sets was again based on the execution time of SMODELS in the baseline. In order to have training sets containing about approximately 70% of the instances, we set the selection threshold to 50 seconds.

The problem subclasses were defined by the pair $\langle maxtime, goal \rangle$, where *maxtime* specifies the maximum plan length and *goal* is the goal of the planning task (12 such goals are defined for the domain). Each subclass contained on average 12 instances. Although, as can be seen later, these sub-

Domain: Solitaire				
	50s-100s	100s-200s	200s-6000s	T/O
num	2	2	13	51
T/O	0	0	1	6
avg	52.447	443.031	678.629	665.019

Domain: HierarchicalClustering				
	50s-100s	100s-200s	200s-6000s	T/O
num	1	2	10	13
T/O	0	0	3	7
avg	12.285	117.225	1096.573	51.797

Domain: GraphPartitioning				
	50s-100s	100s-200s	200s-6000s	T/O
num	7	4	7	1
T/O	0	0	0	1
avg	220.249	135.215	222.965	N/A

Domain: 15Puzzle				
	50s-100s	100s-200s	200s-6000s	T/O
num	13	2	15	11
T/O	0	0	0	0
avg	326.851	1957.899	1268.053	782.938

Fig. 6. Performance comparison for 2nd ASP Competition Domains; easy instances used for training.

classes yielded overall good performance results, other choices are possible; however, an analysis of the strategies for subclass selection is beyond the scope of the present paper. For the first comparison, we focused on planning with maximum plan lengths of 9 and 10. Figure 8 summarizes the results of the comparison. The training set contained 151 instances, while the test set contained 81 instances. Overall, DSPEC timed out 10 times, while SMODELS timed out 49 times. The average times of DSPEC are also remarkable, being mostly in the tens of seconds. The average speedup was 259.2, with a peak of 1253.1 for an instance for which SMODELS timed out, and a peak of 544.5 for an instance for which SMODELS did not time out. It should be noted that, whenever a timeout is involved, the speedup given is only an approximation, and the actual value could be substantially higher. As a test, we have let SMODELS run on some of these instances for over $60,000$ seconds (16 hours) without getting a solution. Inspection of the instances in which DSPEC performed worse than SMODELS revealed that in most cases the size of the training set (for the particular subclass) was rather small, which affected the quality of the domain-specific heuristic formed from it.

In the next experiment, we assessed the performance improvement that can be achieved by combining domain-specific heuristics with the use of restarts in the solver (refer to Section 1 and to e.g. [24,10,18]). In the following discussion, the term SMODELS-RESTART refers to SMODELS executed with restarts enabled. The comparison was performed on the instances from the previous experiment. The domain-specific heuristics used for this experiment were the ones computed in the previous one – which *were obtained without the use of restarts*. Figure 9 summarizes the results.

As one might expect, the advantage of using the domain-specific heuristics is less evident. In

Domain: Solitaire				
	50s-100s	100s-200s	200s-6000s	T/O
num	2	2	13	51
T/O	0	0	0	1
avg	0.150	0.070	0.080	0.212
Avg conflicts: 1372				
Avg restarts: 2				
Tight instances: all				

Domain: GraphPartitioning				
	50s-100s	100s-200s	200s-6000s	T/O
num	7	4	7	1
T/O	0	0	0	1
avg	31.632	15.972	12.350	N/A
Avg conflicts: 354513				
Avg restarts: 15				
Tight instances: none				

Domain: HierarchicalClustering				
	50s-100s	100s-200s	200s-6000s	T/O
num	1	2	10	13
T/O	0	0	0	0
avg	0.050	0.055	0.055	0.062
Avg conflicts: 662				
Avg restarts: 3				
Tight instances: all				

Domain: 15Puzzle				
	50s-100s	100s-200s	200s-6000s	T/O
num	13	2	15	11
T/O	0	0	0	0
avg	1.356	4.250	2.825	2.234
Avg conflicts: 5135				
Avg restarts: 7				
Tight instances: all				

Fig. 7. CLASP on the 2nd ASP Competition Domains; easy instances used for training.

8 Mechanical Faults				
	50s-100s	100s-200s	200s-6000s	T/O
num	5	3	12	28
T/O	1	0	2	4
avg	149.633	8.756	10.553	200.491

10 Mechanical Faults				
	50s-100s	100s-200s	200s-6000s	T/O
num	2	1	9	21
T/O	1	0	1	1
avg	9.756	19.576	11.393	129.082

Fig. 8. Performance comparison for the RCS Domain; maximum plan lengths 9 and 10.

fact, restarts are known to cause performance improvements in various domains, and the RCS domain appears to be one of them. DSPEC timed out in 4 cases, versus just 2 instances for SMODELS-RESTART. Nonetheless, DSPEC showed an average speedup of 7.4 over SMODELS-RESTART – almost one order of magnitude – and a peak speedup of 32.3. The performance of DSPEC was better than that of SMODELS-RESTART in 14 cases out of 21. We believe that the fact that DSPEC performs overall well when compared to SMODELS-RESTART is quite remarkable, *because the heuristics used by* DSPEC *in this experiment were learned from the decisions made by* DSPEC *without restarts in the previous experiment.* Even more remarkable is the fact that the decisions used to build the heuristics are those corresponding to the "easy" instances, while the comparison with SMODELS-RESTART was performed on the hard instances.

The next experiment investigated the performance improvements obtained by forming the domain-specific heuristics from the decisions made by DSPEC *using* restarts. In the following discussion, DSPEC-RESTART denotes the execution

8 Mechanical Faults				
	50s-100s	100s-200s	200s-6000s	T/O
num	10	3	7	1
T/O	0	0	0	4
avg	20.548	23.075	55.085	29.789

10 Mechanical Faults				
	50s-100s	100s-200s	200s-6000s	T/O
num	8	3	5	1
T/O	0	0	0	0
avg	25.859	15.112	17.957	2372.903

Fig. 9. Performance comparison for the RCS Domain using restarts; maximum plan lengths 9 and 10.

of DSPEC with restarts enabled. Note that, because DSPEC is built on top of SMODELS, DSPEC-RESTART and SMODELS-RESTART share the same restarting technique.

In order to compensate for the performance increase that is often typical of the use of restarts, we increased the size of the search space. Thus, in this experiment we used maximum plan lengths of 12 and 14. The time threshold for the partitioning of the set of instances was kept at 50 seconds. This produced a rather small training set of 62 instances, and a test of 170 instances. Performance, however, was not particularly affected by the comparatively small size of the training set. The results are shown in Figure 10.

The number of timeouts was quite low for both solvers – 9 for SMODELS-RESTART and 4 for DSPEC-RESTART. In 127 instances, DSPEC-RESTART was faster than SMODELS-RESTART (in 9 more cases the performance was the same). The average speedup was 9.2, with a peak of 153.0 for an instance for which SMODELS-RESTART timed out, and of 76.1 for an instance for which SMODELS-RESTART did not time out. Overall, the use of domain-specific heuristics still allowed for a more consistent performance, in spite of the performance improvement due to the use of restarts.

6. Related Work

Because of the central role of solver performance and stability, a myriad different techniques for their improvement have been studied. Here we focus on those that are closest to our approach.

A rather successful approach relies on the use of multiple solvers. The intuition is that solvers often exhibit excellent performance on certain classes of problems, while they perform poorly on others, with different solvers often performing well on different classes of problems. Instead of using the same solver for all the problems, one can then analyze each problem in input and use the best solver for the task. From a technical perspective, doing this involves the following steps: (1) selecting a set of relevant solvers, which is called *solver portfolio*; (2) analyzing, offline, the performance of the solvers in the *solver portfolio* on representative instances; (3) when a program is provided in input, its syntactic features are analyzed, and (4) used for the selection of the solver that is most likely to perform well; finally, (5) the selected solver is executed, and computes the models of the program.

A popular instance of the solver portfolio approach is the SAT solver SATzilla [20]. In SATzilla, the analysis of the solvers is performed automatically, using machine learning techniques. The analysis algorithm takes in input an objective function, and optimizes the portfolio and associated selection function in such a way that maximizes the objective function. In recent works, SATzilla has also been extended with the ability to include local search solvers in the portfolio, and by a rather sophisticated method for the analysis of the program in input.

In the ASP community, a similar technique is implemented in CLASPFOLIO [12]. CLASPFOLIO differs from SATzilla in that the portfolio consists of different configurations of CLASP, rather than possibly completely different solvers. The configura-

8 Mechanical Faults				
	50s-100s	100s-200s	200s-6000s	T/O
num	31	17	42	4
T/O	0	0	0	1
avg	66.706	95.436	287.269	77.617

10 Mechanical Faults				
	50s-100s	100s-200s	200s-6000s	T/O
num	21	16	34	5
T/O	0	0	1	2
avg	59.421	68.687	550.360	576.790

Fig. 10. Performance comparison for the RCS Domain using restarts; maximum plan lengths 12 and 14.

tions are specified manually, and CLASPFOLIO automatically analyzes their performance on representative problem instances. When the answer sets of a program must be computed, the program's features are analyzed and used to select the most promising configuration of CLASP.

In the portfolio-based approaches the decision as to which solver to use is made before model computation starts. As noted in [30], doing this has the potential disadvantage that, after the initial analysis and solver selection occur, the technique provides no way of analyzing the program later on during the solving process, and possibly switching to a different solver. So, if the initial analysis is incorrect, a less performing solver will be used for the whole computation, causing the system to perform poorly. In [30] an alternative approach is proposed, in which multiple heuristics, rather than multiple solvers, are available. All the heuristics are implemented within the same QBF solver, and the program can be analyzed and the most promising heuristic selected *at each choice point*. Similarly to the previous approaches, an offline learning method is used to train the algorithm for the selection of the heuristics.

The DORS framework can be viewed as being located in between the portfolio-based approaches and the one from [30]. Like the latter, DORS selects among multiple heuristics. The main selection occurs at the beginning of the computation, when the domain is identified and the relevant domain-specific heuristics selected. Similarly to [30], at each choice point DORS considers the possible decisions under the current domain-specific heuristics and selects the most promising one. On the other hand, unlike [30] and similarly to the portfolio-based approaches, DORS never reconsiders the initial classification of the program. An additional feature of DORS, which differentiates it from both the portfolio-based approaches and [30], is that the collection of domain-specific heuristics can be quickly, incrementally updated at the end of each execution of the solver.

All of the above approaches (including DORS) can be viewed as offline techniques, in the sense that most of the learning occurs separately from the computation of the models of the program (or theory), typically before or after it. A very different approach consists in adapting the solver's heuristics to the problem in input at run-time by using online learning techniques. This is the case of the clause learning and conflict learning techniques that are used in SAT and ASP solvers (see e.g. [16,11]), and whose introduction has brought about substantial performance improvements. The idea behind these learning techniques is to record information about the conflicts that are detected during the exploration of the search space, and to use the information to avoid descending similar branches later. Hence, the basic heuristic is still general-purpose, but it is tuned, during execution, depending on the features of the problem in input. On the other hand, the tuned heuristic only applies to the current execution of the solver; in fact, the information that has been acquired by the learning algorithms is discarded when the solver terminates.

7. Conclusions

In this paper we have described a framework that allows learning and using domain-specific heuristics for choice-point selection, and we have demonstrated its application to DPLL-style ASP solvers. Our experimental comparison with SMODELS has shown that domain-specific heuristics can give remarkable speedups, allow to find answers that cannot otherwise be computed in a reasonable amount of time, and make performance overall more consistent. In the case of the RCS domain, a large number of the instances for which the standard solver timed out, could be solved in a matter of seconds using the domain-specific heuristics, with an average speedup of more than 2 orders of magnitude and peaks of more than 3. When combined with restarts, and applied to substantially harder instances, domain-specific heuristics gave a speedup of 1 order of magnitude on average, with peaks of 2. We believe that an appealing feature of the DORS framework is that in principle it can be applied to any DPLL-style solver. Hence, it is possible to extend the approach shown here to other ASP solvers, or even to e.g. constraint solvers. Work is also ongoing on extending the DORS framework to solvers based on conflict-driven clause learning, such as CLASP. As a final note, we would like to point out that the method used here to learn the domain-specific heuristics is a very simple instance of policy learning. It may be interesting to investigate how more sophisticated techniques from reinforcement learning, but also from machine learning and data mining, can be applied within the DORS framework.

References

[1] M. Balduccini. USA-Smart: Improving the Quality of Plans in Answer Set Planning. In B. Jayaraman, editor, *6th International Symposium on the Practical Aspects of Declarative Languages (PADL'04)*, volume 3057 of *Lecture Notes in Artificial Intelligence (LNCS)*. Springer Verlag, Berlin, Jun 2004.

[2] M. Balduccini and M. Gelfond. Logic Programs with Consistency-Restoring Rules. In P. Doherty, J. McCarthy, and M.-A. Williams, editors, *International Symposium on Logical Formalization of Commonsense Reasoning*, AAAI 2003 Spring Symposium Series, pages 9–18, Mar 2003.

[3] M. Balduccini, M. Gelfond, and M. Nogueira. Answer Set Based Design of Knowledge Systems. *Annals of Mathematics and Artificial Intelligence*, 47(1–2):183–219, 2006.

[4] C. Baral, M. Gelfond, and N. Rushton. Probabilistic reasoning with answer sets. *Journal of Theory and Practice of Logic Programming (TPLP)*, 9(1):57–144, 2009.

[5] A. G. Barto and R. S. Sutton. *Reinforcement learning: an introduction.* MIT Press, 1998.

[6] F. Calimeri, T. Dell'Armi, T. Eiter, W. Faber, G. Gottlob, G. Ianni, G. Ielpa, C. Koch, N. Leone, S. Perri, G. Pfeifer, and A. Polleres. The DLV System. In S. Flesca and G. Ianni, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA 2002)*, Sep 2002.

[7] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[8] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Communications of the ACM*, 7:201–215, 1960.

[9] M. Denecker, J. Vennekens, S. Bond, M. Gebser, and M. Truszczynski. The Second Answer Set Programming Competition. In E. Erdem, F. Lin, and T. Schaub, editors, *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR09)*, volume 5753 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 637–654. Springer Verlag, Berlin, Sep 2009.

[10] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Clasp: A Conflict-Driven Answer Set Solver. In C. Baral, G. Brewka, and J. Schlipf, editors, *9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR07)*, volume 4483 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 260–265. Springer Verlag, Berlin, May 2007.

[11] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-Driven Answer Set Solving. In M. M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392, 2007.

[12] M. Gebser, B. Kaufmann, and T. Schaub. The Conflict-Driven Answer Set Solver clasp: Progress Report. In E. Erdem, F. Lin, and T. Schaub, editors, *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR09)*, volume 5753 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 509–514. Springer Verlag, Berlin, Sep 2009.

[13] M. Gebser and T. Schaub. Tableau Calculi for Answer Set Programming. In S. Etalle and M. Truszczynski, editors, *22nd International Conference on Logic Programming (ICLP06)*, volume 4079 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 11–25. Springer Verlag, Berlin, Aug 2006.

[14] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[15] E. Giunchiglia, N. Leone, and M. Maratea. On the Relation Among Answer Set Solvers. *Annals of Mathematics and Artificial Intelligence*, 53(1–4):169–204, 2008.

[16] E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust Sat-Solver. In *Proceedings of Design, Automation and Test in Europe Conference (DATE-2002)*, pages 142–149, Mar 2002.

[17] G. Grasso, N. Leone, M. Manna, and F. Ricca. ASP at Work: Spin-off and Applications of the DLV System. In M. Balduccini and T. C. Son, editors, *Symposium on Constructive Mathematics in Computer Science*, Oct 2010.

[18] J. Huang. The Effect of Restarts on the Efficiency of Clause Learning. In M. M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 2318–2323, 2007.

[19] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[20] K. Leyton-Brown, H. H. Hoos, F. Hutter, and L. Xu. SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

[21] Y. Lierler. Abstract Answer Set Solvers. In M. G. de la Banda and E. Pontelli, editors, *Proceedings of the 24th International Conference on Logic Programming (ICLP08)*, volume 5366 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 377–391. Springer Verlag, Berlin, Dec 2008.

[22] V. W. Marek and M. Truszczynski. *The Logic Programming Paradigm: a 25-Year Perspective*, chapter Stable models and an alternative logic programming paradigm, pages 375–398. Springer Verlag, Berlin, 1999.

[23] J. P. Marques-Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[24] D. Mitchell. A SAT Solver Primer. *Bulletin of the EATCS*, 85:112–133, 2005.

[25] I. Niemelä and P. Simons. *Logic-Based Artificial Intelligence*, chapter Extending the Smodels System with Cardinality and Weight Constraints, pages 491–521. Kluwer Academic Publishers, 2000.

[26] M. Nogueira. *Building Knowledge Systems in A-Prolog*. PhD thesis, University of Texas at El Paso, May 2003.

[27] L. Pulina and A. Tacchella. A Self-Adaptive Multi-Engine Solver for Quantified Boolean Formulas. *Constraints*, 14(1):80–116, 2009.

[28] F. Ricca, W. Faber, and N. Leone. A Backjumping Technique for Disjunctive Logic Programming. *AI Communications*, 19(2):155–172, 2006.

[29] C. Sakama and T. C. Son. Negotiation Using Logic Programming with Consistency Restoring Rules. In C. Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 930–935, 2009.

[30] H. Samulowitz and R. Memisevic. Learning to Solve QBF. In *Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI'07)*, pages 255–260. AAAI Press/The MIT Press, 2007.

[31] T. Soininen, I. Niemelä, and P. Simons. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.

Appendix: Grounding in DORS

ASP solvers typically expect in input ground (i.e. variable-free) programs. However, because using variables in ASP programs is convenient, programs are first pre-processed by a *grounder* (LPARSE and GRINGO in the systems considered here), which replaces each non-ground rule by the set of its ground instances. The grounder also translates certain rules containing constraint literals into rules with a simpler structure. The main difficulty in implementing our technique in state-of-the-art ASP systems (the following discussion is based on the architecture of the LPARSE+SMODELS system but can be extended to other ASP systems as well) is that the grounders often introduce "unnamed atoms" during the grounding process.

An unnamed atom is an atom that does not occur in the original program, and is used internally by the ASP system. Because of their local use, unnamed atoms are assigned identifiers that are only valid for the current run of the system. There is no guarantee that unnamed atoms will be assigned the same identifiers when the system is run on a different problem instance. Because such unnamed atoms may be used as choice points by the solver, one needs to ensure that unnamed atoms are given a unique, known identifier, so that choice-point information regarding them can be used in the formation of the domain-specific heuristics.

One possible solution is to modify the ASP grounders so that unnamed atoms are given identifiers that remain valid across multiple executions of the solver. Although conceptually simple, this solution requires modifying each grounder that one is interested in using. In this section we present instead a relatively simple, indirect method that consists of a pre-processing phase and a post-processing phase, and does not involve modifications to the grounders.

As we mentioned earlier, in LPARSE and GRINGO, unnamed atoms are introduced during the grounding of rules containing certain constraint literals, in order to simplify their structure. (Details on the translation can be found in [25].) For example, the choice rule in the program:

$$\begin{cases} p(1).\ p(2).\ p(3). \\ 1\{a(X) : p(X)\}2. \end{cases}$$

is translated by the grounder as:

$$\begin{cases} \{a(1), a(2), a(3)\}. \\ \leftarrow \mu_1. \\ \mu_1 \leftarrow 3\{\text{not } a(1), \text{not } a(2), \text{not } a(3)\}. \\ \leftarrow \mu_0. \\ \mu_0 \leftarrow 3\{a(3), a(2), a(1)\}. \end{cases}$$

where μ_0 and μ_1 are unnamed atoms. As we mentioned earlier, no assumptions can be made about which identifiers are used for the unnamed atoms. If we were for example to add to the program a second choice rule, or if the number of ground instances of the choice rule of our example were to change because of changes in the problem instance, the grounding of the new program could use some new identifiers μ_2, μ_3 for the above translation.

On the other hand, because of the structure of the grounding algorithm, *the relative order of the rules belonging to the grounding of the choice rule is independent from the changes made to the rest of the program.* Moreover, whenever multiple unnamed atoms occur in the body of a rule, *their relative order is independent of changes made to the rest of the program.* We will make use of these two properties later.

In the pre-processing phase, the user specifies a name for each rule whose grounding may cause the introduction of unnamed atoms. Because we want to avoid modifications to the grounder, we do not extend the syntax of the language to allow specifying rule names explicitly. Instead, the name of a rule is specified in the body of the rule itself, using a special relation ν.[2] So, choice rule $1\{a(X) : p(X)\}2$ can be re-written as:

$$1\{a(X) : p(X)\}2 \leftarrow \nu(r_1). \qquad (1)$$

Generally speaking, given a list, $\vec{X}$, of all the free variables in the rule, and some fresh constant ρ, the name is specified by the atom $\nu(\rho, \vec{X})$. A rule whose name is specified as above is called an *augmented* rule.

To ensure that the meaning of a rule is not altered by the augmentation, a definition of atom $\nu(\cdot)$ must also be provided (otherwise the body of the augmented rule is never satisfied). Because state-of-the-art grounders usually drop trivially-true atoms from the body of the rules, we define the new atom by a choice rule with no bounds and suitable domain predicates for the arguments of re-

[2]Notice that the specification of the name of the rule in the body is purely a technical device, and should not be intended as conveying any semantic information.

lation ν, such as $\{ \ \nu(r_1) \ \}$. This choice rule will be removed later, to avoid affecting the performance of the solver. When processing (1), the grounder produces:

$$\begin{cases} \{a(1), a(2), a(3)\} \leftarrow \nu(r_1). \\ \leftarrow \mu_1, \nu(r_1). \\ \mu_1 \leftarrow 3\{\text{not } \ a(1), \text{not } \ a(2), \text{not } \ a(3)\}. \\ \leftarrow \mu_0, \nu(r_1). \\ \mu_0 \leftarrow 3\{a(3), a(2), a(1)\}. \end{cases}$$

Notice how the unnamed atoms co-occur with the $\nu(\cdot)$ atom in the body of some of the rules. Because of the structure of the grounding algorithm, *this is the case for the grounding of any rule that introduces unnamed atoms.* The reader should also notice that the addition of $\nu(\cdot)$ atoms to the program can be easily automated. A user could then specify a name for the rule using a more convenient syntax, and have a simple pre-processor introduce the $\nu(\cdot)$ atoms in the program as shown above.

The post-processing phase is based on the algorithm shown in Figure 11. The algorithm works as follows. First, the ground rules are scanned for co-occurrences of unnamed atoms and ν atoms. The goal is to use the information provided by the ν atoms to give a name to the unnamed atoms they co-occur with. The association of names to unnamed atoms is stored in variable *Assoc*. Because multiple unnamed atoms may be introduced by the grounding of a single rule, an extra integer argument is added to relation ν when naming unnamed atoms. Values for that argument are assigned on a first-come, first-serve basis. Because, as we noted above, the relative order of unnamed atoms in the ground rules does not change, we are guaranteed that the naming of unnamed atoms will be consistent throughout multiple runs of the grounder with different input programs (as long as the domain description remains the same). The second *for* loop is for the support of rules of the form

$$h \leftarrow \text{not } \ lower[atom\text{-}list]upper, \Gamma.$$

The grounding of these rules involves the generation of a rule of the form $h \leftarrow \Gamma$ where h can be guaranteed to have been given a name during the execution of previous loop, and Γ may still contain unnamed atoms. The names of these atoms are obtained from the one assigned to h, using the technique described for the previous loop. In the third *for* loop, all ν atoms and their definitions are removed from the program. Finally, the unnamed atoms are renamed according to the associations encoded by variable *Assoc*.

```
function postp ( $G : GroundProgram$ )
    $Assoc := \emptyset$;  $G' := G$;
    for each rule $\rho \in G$ and unnamed atom $\mu$ in $\rho$
        if $\rho$ contains an atom $\nu(\vec{X})$ for some $\vec{X}$ then
            $i :=$ smallest positive integer such that
                        $\forall \mu' \ \langle \mu', \nu(i, \vec{X}) \rangle \notin Assoc$;
            $Assoc := Assoc \cup \{\langle \mu, \nu(i, \vec{X}) \rangle\}$;
        end if
    end for
    for each rule $\rho \in G$ of the form $h \leftarrow \Gamma$
           and unnamed atom $\mu$ in $\Gamma$
        if $\langle h, \nu(i, \vec{X}) \rangle \in Assoc$ for some $i, \vec{X}$ then
            $i' :=$ smallest positive integer such that
                        $\forall \mu' \ \langle \mu', \nu(i', \vec{X}) \rangle \notin Assoc$;
            $Assoc := Assoc \cup \{\langle \mu, \nu(i', \vec{X}) \rangle\}$;
        end if
    end for
    for each atom of the form $\nu(\vec{X})$
        Remove from $G'$ rule $\{\nu(\vec{X})\} \leftarrow \Gamma$ (for some $\Gamma$);
        Remove every occurrence of $\nu(\vec{X})$ from $G'$;
    end for
    for each $\langle \mu, \nu(i, \vec{X}) \rangle \in Assoc$
        Replace all occurrences of $\mu$ in $G'$ by $\nu(i, \vec{X})$;
    end for
    return $G'$;
```

Fig. 11. Post-processing algorithm