Submitted to the Technical Communications of the International Conference on Logic Programming (ICLP'10)
http://www.floc-conference.org/ICLP-home.html

LEARNING DOMAIN-SPECIFIC HEURISTICS FOR ANSWER SET SOLVERS

MARCELLO BALDUCCINI [1]

[1] Intelligent Systems, KRL
Eastman Kodak Company
Rochester, NY 14650-2102 USA
E-mail address: `marcello.balduccini@gmail.com`

ABSTRACT. In spite of the recent improvements in the performance of Answer Set Programming (ASP) solvers, when the search space is sufficiently large, it is still possible for the search algorithm to mistakenly focus on areas of the search space that contain no solutions or very few. When that happens, performance degrades substantially, even to the point that the solver may need to be terminated before returning an answer. This prospect is a concern when one is considering using such a solver in an industrial setting, where users typically expect consistent performance. To overcome this problem, in this paper we propose a technique that allows learning domain-specific heuristics for ASP solvers. The learning is done off-line, on representative instances from the target domain, and the learned heuristics are then used for choice-point selection. In our experiments, the introduction of domain-specific heuristics improved performance on hard instances by up to 3 orders of magnitude (and 2 on average), nearly completely eliminating the cases in which the solver had to be terminated because the wait for an answer had become unacceptable.

1. Introduction

In recent years, solvers for Answer Set Programming (ASP) [Gel91, Mar99] have become amazingly fast. Mostly, that is due to good heuristics that direct the search toward the most promising areas of the search space, and to learning algorithms that discover features of the search space on-the-fly (see e.g. [Geb07]). Unfortunately, when the search space is sufficiently large, it is still possible for the search algorithm to mistakenly focus on areas of the search space that contain no solutions or very few. When that happens, performance degrades substantially, even to the point that the solver may need to be terminated before returning an answer. This prospect is a 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. It should be noted that the phenomenon of performance degradation is often due to the fact that the heuristics used in choice-point selection are general-purpose, and thus can be side-tracked by peculiar features of a given domain. To overcome this problem, in this paper we propose a technique that allows learning domain-specific heuristics for ASP solvers. The technique is mainly aimed at improving the efficiency of the computation of one answer set (as opposed to multiple answer sets of a program) of consistent programs, but could be extended further. The learning is done off-line, on representative instances from the target domain. In our experiments, the introduction of domain-specific heuristics improved performance on hard instances by up to 3 orders of magnitude (and 2 on average), nearly

1998 ACM Subject Classification: I.2.3, I.2.4, I.2.5.
Key words and phrases: answer set programming, solvers, domain-specific heuristics.

completely eliminating the situations in which the solver had to be terminated because the wait for an answer had become unacceptable.

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 most ASP solvers. Then, in Section 3, we present our technique for learning domain-specific heuristics. Experimental results are discussed in Section 4. Finally, in Section 5, we draw conclusions.

2. Answer Set Programming

Let us start by giving some background on ASP. We define the syntax of the language precisely, but only give the informal semantics of the language in order to save space. We refer the reader to [Gel91, Nie00] for a specification of the formal semantics. Let Σ be a signature containing constant, function and predicate symbols. Terms and atoms are formed as usual in first-order logic. A (basic) 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:

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

where 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 a reasoner who believes $\{l_1, \dots, l_m\}$ and has no reason to believe $\{l_{m+1}, \dots, l_n\}$, 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*. Symbol $\leftarrow$ can be omitted if no l_i's are specified. Often, rules of the form $h \leftarrow \text{not } \ h, l_1, \dots, \text{not } \ l_n$ are abbreviated into $\leftarrow l_1, \dots, \text{not } \ l_n$, and called *constraints*. The intuitive meaning of a constraint is that its body must not be satisfied. 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 *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 just by the second element of the pair, and let the signature be defined implicitly. In that case, the signature of Π is denoted by $\Sigma(\Pi)$. Finally, an *answer set* (or *model*) of a program Π is one of the possible collections of its consequences under the answer set semantics. Notice that the semantics of ASP is defined in such a way that programs may have multiple answer sets, intuitively corresponding to alternative views of the specification given by the program. In that respect, the semantics of default negation allows for a simple way to encode choices. For example, the set of rules $\{p \leftarrow \text{not } \ q. \ \ q \leftarrow \text{not } \ p.\}$ intuitively states that either p or q hold, and the corresponding programs has two answer sets, $\{p\}$, $\{q\}$. Because a convenient representation of alternatives is often important in the formalization of knowledge, the language of ASP has been extended with *constraint literals* [Nie00], which are expressions of the form $m\{l_1, l_2, \dots, l_k\}n$, where m, n are arithmetic expressions and l_i's are basic literals as defined above. A constraint literal is satisfied whenever the number of literals that hold from $\{l_1, \dots, l_k\}$ 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 each problem instance encodes a different problem from the domain. In this paper we will make the simplifying assumption (usually satisfied even in practical applications) that the signature of every problem instance of interest is contained in the signature of the domain description. Another notion that is important for practical purposes is that of *domain predicate*. Domain predicates are relations whose definition is given with rules following syntactic restrictions, in such a way that the definition of the relation can be derived from the rules without performing a complete answer set computation for the containing program. Domain predicates are used by the grounding procedures in order to determine the ranges of the variables that occur in the program. The precise definition of the syntactic restrictions varies depending on the grounding procedure used. A commonly used definition is the one given in [Syr98].

3. Learning Domain-Specific Heuristics

The search algorithm used by most ASP solvers (e.g. SMODELS [Nie02], DLV [Cal02], CLASP [Geb07]) builds upon the DPLL procedure [Dav60, Dav62]. The basic algorithm for the computation of a single answer set, which we will later refer to as *standard algorithm*, is show in Figure 1. The term *extended literal*, used in the algorithm, identifies a literal l or the expression *not* l (intuitively meaning that l is known not to hold in the answer set, but its complement, $\bar{l}$, may or may not hold). Given an extended literal e, $not(e)$ denotes the expression *not* l if $e = l$ and it denotes l if $e =$ *not* l. The algorithm is based on the idea of growing a particular set of (ground) literals, often

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

Figure 1: Basic Search Algorithm for ASP

called 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.

It is not difficult to see how the choices made by `choose_literal` greatly influence the number of choice points picked by the algorithm, and ultimately its performance. 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 [Nie02]. These techniques work well in a number of cases, but not always. In fact, particular features of the program can confuse the heuristics. When that happens in the 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 frustrating is the fact that the efficiency of the heuristics may change largely in correspondence of small elaborations of the program in input. For example, the $choose\_literal$ heuristics may make good selections for one problem instance, while they may cause the search to take an unacceptable amount of time for a not-too-different problem instance.

One way to limit 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. [Geb07]). 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.

A different, more straightforward, way of limiting the wrong selections made by $choose\_literal$ is to directly improve the choice-making algorithm. In this paper, we adopt the approach of learning domain-specific heuristics from a number of sample problems, and of using them for literal selection in a modified version of $choose\_literal$. This technique is suitable for situations in which one is interested in solving a number of problem instances from a given problem domain. Such situations are very common in the ASP community – see e.g. the Second Answer Set Programming Competition [Den09]. 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 [Bal06]), which then automatically encodes the problem instance.

Consider program P_1:

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

The program can be viewed as consisting of a domain description and a problem instance: the first 7 rules constitute the former, while the definition of predicate t is the problem instance. A different problem instance might then define t as $\{t(5),\ t(6),\ t(7)\}$. In this case, it is obvious that a good strategy for the selection of the literals consists in first choosing among $\{p, \text{not } p, q, \text{not } q\}$ and only later (if necessary) considering the extended literals formed by u and v.

In general, the domain-specific heuristics for $choose\_literal$ will be learned – rather than manually specified – by analyzing the choices made by the standard solver *solve* when solving representative problem instances from the domain. This approach is particularly useful in applications in which a number of problem instances from the same class of problems will have to be solved over time – for example, in the setting of an industrial application, or in a programming/solver competition in which benchmarking is involved – and computational power is available off-line to allow learning the domain-specific heuristics (e.g. before deploying the application, or before submitting the solver or solutions to a competition).

Let us now describe in more detail our technique for learning and using domain-specific heuristics. We start with the learning phase. First of all, the algorithm from Figure 1 is modified to maintain a record of the choice points, and to return the list of choice points together with the answer set, when 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 how the information collected by *solvecp* is used to guide the 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 $\perp$ if *solvecp*$(I \cup M, \emptyset, \emptyset) = \perp$ 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

```
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 $\langle B', S'\rangle$;
```

Figure 2: Search Algorithm for ASP with Explicit Tracking of Choice Points

e, $level(e, d)$ denotes the index 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. 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 sets. 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.

In order to improve the efficiency of the learned heuristics, we divide 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 peculiar features of the problem instances. For example, in a planning domain, σ might be the maximum length of the plan (often called $lasttime$ or $maxtime$ in ASP-based planning). 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 to determine how many times an extended literal e was selected at a certain level of the decision-sequences for the problem instances in $\mathcal{I}$. More precisely, given a positive integer δ, called the *scaling factor*, and subclass σ, the *occurrence count* of an extended literal e w.r.t. a level l and set of instances $\mathcal{I}$ is

$$\begin{aligned} o_{\delta,\sigma}(e, l, \mathcal{I}) = |\, \{\, I \mid I \in \mathcal{I} \;\wedge\; \sigma(I) = \sigma \;\wedge\; d(I) \neq \bot \;\wedge \\ l - \delta/2 \leq index(e, d(I)) < l + \delta/2 \,\}\, |. \end{aligned}$$

The scaling factor δ allows taking into account all the occurrences of e at a level in the interval $[l - \delta/2, l + \delta/2)$. If $\delta = 1$, then only the occurrences of e with level equal to l are considered. Values of δ greater than 1 can be useful in those cases in which all or most permutations of a sub-sequence of choice points lead to an answer set.

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 is:

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

Intuitively, $best_\delta(l, E, \sigma, \mathcal{I})$ returns the choice points that, if taken at level l, are most likely to lead to an answer set without backtracking, based on the information collected about the instances of subclass σ in $\mathcal{I}$. Algorithms for the computation of $best_\delta(l, E, \sigma, \mathcal{I})$ and $o_{\delta,\sigma}(e, l, \mathcal{I})$ are simple and are omitted to save space.

Function $best_\delta(l, E, \sigma, \mathcal{I})$ encodes the essence of the domain-specific heuristics. 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

```
function choose_literal_dspec ( $\Pi$ : $Program$,
                                $\sigma$ : Problem Subclass,
                                $A$ : Set of Extended Literals,
                                $level$ : Integer /* Current Level in the Decision Tree */,
                                $T$ : Set of Extended Literals,
                                $\mathcal{I}$ : Set of Instances,
                                $\delta$ : Integer /* Scaling Factor*/ )

    $L := lit(\Sigma(\Pi))$;
    $E := L \cup \{\texttt{not}\ l \,|\, l \in L\}$;
    $E' = \emptyset$;
    for each $e \in E$
        if ($e \notin A \wedge not(e) \notin A \wedge e \notin T$) then
            $E' := E' \cup \{e\}$;
        end if
    end for
    $B := best_\delta(level, E', \sigma, \mathcal{I})$;
    if ($B \neq \emptyset$) then
        $chosen := one\_element\_of(B)$;
    else
        $chosen := choose\_literal(\Pi, A)$;
    end if

    return $chosen$;
```

Figure 3: Function for Literal Selection with Domain-Specific Heuristics

$choose\_literal\_dspec$, argument T is the set of extended literals that have previously been selected by $choose\_literal\_dspec$. If $best_\delta(level, E', \sigma(I), \mathcal{I})$ is the empty 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.

4. Experimental Evaluation

In this section we discuss the experiments we ran in order to evaluate our technique for learning domain-specific heuristics and using them in computing answer sets. To ensure coverage of a wide variety of cases, we have tested our implementation on both abstract problems and on problems from industrial applications of ASP. Here we show the results of testing on the task of planning for the Reaction Control System of the Space Shuttle.

The system used in the experiments is LPARSE+SMODELS, which we modified to obtain implementations of algorithms $solvecp$ and $solve\_dspec$. One complication of the implementation process is due to the fact that LPARSE often introduces unnamed atoms during the grounding of rules containing constraint literals, where by unnamed atoms we mean atoms that do not occur in the original program, and that are assigned an identifier that is only meaningful in the context of the current computation. Dealing with unnamed atoms is problematic because, in order to be used in the learning of the domain heuristics, all atoms must be assigned identifiers that are meaningful throughout

```
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;

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

Figure 4: Search Algorithm for ASP with Domain-Specific Heuristics for Choice-Point Selection

multiple computations (normally, the atoms' own string representation satisfies this requirement). We have thus developed a technique that uses pre-processing and post-processing for the execution of LPARSE to assign unnamed atoms identifiers satisfying this requirement. Space limitations prevent us from giving more details on this technique.

It should also be noted that we did not use CLASP for our experiments: although CLASP is based, like SMODELS, on the DPLL procedure, and thus technically viable for the implementation of our algorithms, such implementation is complicated by the fact that, in CLASP, literal selection is allowed to select special literals denoting the whole body of a rule. A further complication of the implementation is due to the use of clause learning in CLASP. Work is ongoing on implementing $solvecp$ and $solve\_dspec$ within this solver, and results will be discussed in a longer paper. In the rest of the discussion, we refer to the implementation of $solve\_dspec$ within SMODELS as DSPEC.

As described in e.g. [Nog03, Bal06], the RCS is the Shuttle's system that has primary responsibility for maneuvering the Shuttle while it is in space. It consists of fuel and oxidizer tanks, valves, and other plumbing needed to provide propellant to the maneuvering jets of the Shuttle. The RCS also includes electronic circuitry, both to control the valves in the fuel lines and to prepare the jets to receive firing commands.

In order to configure the Shuttle for an orbital maneuver, the RCS must be configured by opening and closing appropriate valves. This is accomplished by either changing the position of the associated switches, or by issuing computer commands. In normal conditions, the procedures for the configuration of the RCS for a given maneuver are known in advance by the astronauts. However, if components of the RCS are faulty, then the standard procedures may not be applicable. Moreover, because of the amount of possible combinations of faults, it is impossible to prepare in advance a set of configuration procedures for faulty situations. In those cases, ground control needs to carefully examine the problem and manually come up with a configuration procedure. The system described in [Nog03, Bal06] uses a model of the RCS, as well as ASP-based reasoning algorithms, to provide ground control with a decision-support system that automatically generates configuration procedures for the RCS and that can be used when faulty components are present (incidentally, the system can also perform diagnostic reasoning [Bal06]).

A collection of problem instances from the domain of the RCS is publicly available, together with the ASP encoding of the model of the RCS.[1] The interested reader may refer to [Nog03] for a description of the instances. For our testing, we have selected a set of 425 instances from the collection, corresponding to the public instances with no electrical faults and 3, 8, and 10 mechanical faults respectively, for which a plan of length 6 or less (determined by parameter $lasttime$) was found in the experiments discussed in [Nog03, Bal06], and we have analyzed the performance of the solver on planning with maximum lengths ranging between 6 and 10.

The comparison between SMODELS and DSPEC was conducted as follows. First of all, for each instance we found one plan using SMODELS. Each computation was set up in such a way as to timeout after 6000 seconds, if no answer set had yet been found. Next, we generated the domain-specific heuristics. The set of instances used for learning consisted of all the instances for which our implementation of $solvecp$ found a solution in 50 seconds of less, while the remaining "hard instances" were used for the evaluation phase. The problem subclasses were defined by the pair $\langle lasttime, maneuver \rangle$, where $lasttime$ specifies the maximum plan length and $maneuver$ is the maneuver that the RCS must be configured for (in our experiments, using the maneuver in the subclass definition substantially improved the performance of the learned heuristics). Figure 5 shows the results of the comparison for the 58 hard instances with 8 mechanical faults and values of lasttime of 9 and 10. The results were obtained with $\delta = 1$. We believe the speedup obtained with the domain-specific heuristics is remarkable. First of all, out of 32 instances for which the standard solver timed out before finding a solution, in 28 cases the domain-specific heuristics allowed to find a solution within the time limit, and in some cases in under 10 seconds. The average speedup is 232.3, with a peak of 1253.1 for an instance for which SMODELS timed out[2], and a peak of 544.5 for an instance for which SMODELS did not time out. In 4 cases (out of 32) DSPEC performed worse than the standard solver. We believe that these outliers can be eliminated if more samples are made available for learning.

[1]The files are available from `http://www.krlab.cs.ttu.edu/Software/Download/`.

[2]The actual speedup could in fact be higher, since SMODELS timed out. As a test, we have let SMODELS run on some of these instances for over $60,000$ seconds (16 hours) without getting a solution.

8 Mechanical Faults

Lasttime/ Instance	SMODELS (sec)	DSPEC (sec)	Speedup (times)
9 / 025	6000	17.643	340.1
9 / 027	6000	9.597	625.2
9 / 038	125.244	8.616	14.5
9 / 044	1439.027	6.846	210.2
9 / 053	6000	13.599	441.2
9 / 059	85.151	551.806	0.2
9 / 074	6000	8.961	669.6
9 / 075	736.134	3.837	191.9
9 / 087	6000	6000	1.0
9 / 090	6000	14.111	425.2
9 / 093	2451.649	6.477	378.5
9 / 098	114.643	10.529	10.9
9 / 103	52.219	12.544	4.2
9 / 122	6000	4.788	1253.1
9 / 140	6000	11.493	522.1
9 / 165	6000	13.027	460.6
9 / 170	6000	6000	1.0
9 / 179	6000	14.304	419.5
9 / 184	6000	20.254	296.2
9 / 188	6000	6000	1.0
9 / 191	4829.019	8.869	544.5
9 / 199	437.379	7.144	61.2
10 / 013	94.623	21.663	4.4
10 / 022	6000	423.565	14.2
10 / 025	6000	2035.089	2.9
10 / 027	6000	10.248	585.5
10 / 032	2949.169	13.82	213.4
10 / 037	6000	12.218	491.1
10 / 044	6000	18.162	330.4
10 / 050	72.596	12.521	5.8
10 / 053	1907.445	23.37	81.6
10 / 059	6000	15.163	395.7
10 / 061	266.024	7.756	34.3
10 / 070	519.583	16.343	31.8
10 / 074	6000	13.903	431.6
10 / 077	251.754	7.518	33.5
10 / 087	6000	24.962	240.4
10 / 088	3830.141	18.512	206.9
10 / 092	318.83	11.712	27.2
10 / 093	6000	494.85	12.1
10 / 096	789.351	13.787	57.3
10 / 103	6000	16.781	357.5
10 / 110	6000	255.421	23.5
10 / 113	264.419	6000	0.044
10 / 120	1983.466	20.254	97.9
10 / 140	64.451	6000	0.011
10 / 147	187.8	7.125	26.4
10 / 154	942.008	6000	0.157
10 / 165	6000	30.008	199.9
10 / 166	6000	820.789	7.3
10 / 177	6000	12.605	476.0
10 / 178	6000	6000	1.0
10 / 179	6000	16.74	358.4
10 / 188	5235.985	12.74	411.0
10 / 189	3773.981	11.765	320.8
10 / 190	6000	1010.51	5.9
10 / 194	6000	12.407	483.6
10 / 199	6000	9.452	634.8

Figure 5: Performance Comparison on the RCS Domain. Machine specs: Intel i7 CPU, 2.93GHz, 8GB RAM.

5. Conclusions

In this paper we have demonstrated how domain-specific heuristics for choice-point selection can be learned and used in ASP solvers. Our experimental evaluation has shown that domain-specific heuristics can give remarkable speedups, and allow to find answer sets that otherwise cannot be computed in a reasonable time. In the case of the RCS domain, a large number of the instances for which the standard solver timed out, could be solver 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. This is the type of consistent performance that makes a solver viable for industrial applications.

We believe that an appealing feature of our approach is that in principle it can be applied to any solver built around the DPLL procedure. Hence, it is technically possible to apply the same approach shown here to other ASP solvers, or even to, say, SAT solvers and constraint solvers. Work is ongoing on implementing our technique within 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 will be interesting to investigate how more sophisticated techniques from reinforcement learning, but also from machine learning and data mining, can be applied to the learning of the domain-specific heuristics. We expect that doing so will allow to improve performance of the solvers even further.

References

[Bal06] Marcello Balduccini, Michael Gelfond, and Monica Nogueira. Answer Set Based Design of Knowledge Systems. *Annals of Mathematics and Artificial Intelligence*, 2006.

[Cal02] Francesco Calimeri, Tina Dell'Armi, Thomas Eiter, Wolfgang Faber, Georg Gottlob, Giovanbattista Ianni, Giuseppe Ielpa, Christoph Koch, Nicola Leone, Simona Perri, Gerard Pfeifer, and Axel Polleres. The DLV System. In Sergio Flesca and Giovanbattista Ianni (eds.), *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA 2002)*. 2002.

[Dav60] Martin Davis and Hillary Putnam. A Computing Procedure for Quantification Theory. *Communications of the ACM*, 7:201–215, 1960.

[Dav62] Martin Davis, Geroge Logemann, and Donald Loveland. A Machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[Den09] Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Miroslaw Truszczynski. The Second Answer Set Programming Competition. In *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR09)*, pp. 637–654. 2009.

[Geb07] Martin Gebser, B. Kaufmann, A. Neumann, and Torsten Schaub. Conflict-driven answer set solving. In Manuela M. Veloso (ed.), *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pp. 386–392. MIT Press, 2007.

[Gel91] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[Mar99] Victor W. Marek and Miroslaw Truszczynski. *The Logic Programming Paradigm: a 25-Year Perspective*, chap. Stable models and an alternative logic programming paradigm, pp. 375–398. Springer Verlag, Berlin, 1999.

[Nie00] Ilkka Niemela and Patrik Simons. *Logic-Based Artificial Intelligence*, chap. Extending the Smodels System with Cardinality and Weight Constraints, pp. 491–521. Kluwer Academic Publishers, 2000.

[Nie02] Ilkka Niemela, Patrik Simons, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.

[Nog03] Monica Nogueira. *Building Knowledge Systems in A-Prolog*. Ph.D. thesis, University of Texas at El Paso, 2003.

[Syr98] Tommi Syrjanen. Implementation of logical grounding for logic programs with stable model semantics. Tech. Rep. 18, Digital Systems Laboratory, Helsinki University of Technology, 1998.