Domain-specific Heuristics in Answer Set Programming

M. Gebser[*] and **B. Kaufmann**[*] and **R. Otero**[⋆] and **J. Romero**[*,⋆] and **T. Schaub**[*] and **P. Wanko**[*]

Institute for Informatics[*]
University of Potsdam
14482 Potsdam, Germany

Department of Computer Science[⋆]
University of Corunna
15071 Corunna, Spain

Abstract

We introduce a general declarative framework for incorporating domain-specific heuristics into ASP solving. We accomplish this by extending the first-order modeling language of ASP by a distinguished heuristic predicate. The resulting heuristic information is processed as an equitable part of the logic program and subsequently exploited by the solver when it comes to non-deterministically assigning a truth value to an atom. We implemented our approach as a dedicated heuristic in the ASP solver *clasp* and show its great prospect by an empirical evaluation.

Introduction

The success of modern Boolean constraint technology was greatly boosted by Satisfiability Testing (SAT; (Biere et al. 2009)). Meanwhile, this technology has been taken up in many related areas, like Answer Set Programming (ASP; (Baral 2003)). This is because it provides highly performant yet general-purpose solving techniques for addressing demanding combinatorial search problems. Sometimes, it is however advantageous to take a more application-oriented approach by including domain-specific information. On the one hand, domain-specific knowledge can be added for improving deterministic assignments through propagation. And on the other hand, domain-specific heuristics can be used for making better non-deterministic assignments.

In what follows, we introduce a general declarative framework for incorporating domain-specific heuristics into ASP solving. The choice of ASP is motivated by its first-order modeling language offering an easy way to express and process heuristic information. To this end, we use a dedicated predicate `_h` whose arguments allow us to express various modifications to the solver's heuristic treatment of atoms. The respective heuristic rules are seamlessly processed as an equitable part of the logic program and subsequently exploited by the solver when it comes to choosing an atom for a non-deterministic truth assignment. For instance, the rule

```
_h(occ(A,T),factor,T) :- action(A), time(T).
```

favors later action occurrences over earlier ones (via multiplication by `T`). That is, when making a choice between two unassigned atoms `occ(a,2)` and `occ(b,3)`, the solver's heuristic value of `occ(a,2)` is doubled while that of `occ(b,3)` is tripled. This results in a bias on the system heuristic that may or may not take effect. Besides `factor`, our heuristic language extension offers the primitive heuristic modifiers `init`, `level`, and `sign`, from which even further modifiers can be defined. Our approach provides an easy and flexible access to the solver's heuristic, aiming at its modification rather than its replacement. Note that the effect of the modifications is generally dynamic, unless the truth of a heuristic atom is determined during grounding (as with the rule above). As a result, our approach offers a declarative framework for expressing domain-specific heuristics. As such, it appears to be the first of its kind.

Background

We assume some basic familiarity with ASP, its semantics as well as its basic language constructs, like normal rules, cardinality constraints, and optimization statements. Although our examples are self-explanatory, we refer the reader for details to (Gebser et al. 2012). For illustrating our approach, we consider selected rules of a simple planning encoding, following (Lifschitz 2002). We use predicates `action` and `fluent` to distinguish the corresponding entities. The length of the plan is given by the constant `l`, which is used to fix all time points via the statement `time(1..l).` Moreover, suppose our ASP encoding contains the rule

```
1 { occ(A,T) : action(A) } 1 :- time(T).
```

stating that exactly one action occurs at each time step. Also, it includes a frame axiom of the following form.[1]

```
holds(F,T) :- holds(F,T-1), not -holds(F,T).
```

In such a setting, actions and fluents are prime subjects to planning-specific heuristics. As we show below, these can be elegantly expressed by heuristic statements about atoms formed from predicates `occ` and `holds`, respectively.

For computing the stable models of a logic program, we use a Boolean assignment, that is, a (partial) function mapping propositional variables in $\mathcal{A}$ to truth values $\boldsymbol{T}$ and $\boldsymbol{F}$. We represent such an assignment A as a set of signed literals of form $\boldsymbol{T}a$ or $\boldsymbol{F}a$, standing for $a \mapsto \boldsymbol{T}$ and $a \mapsto \boldsymbol{F}$, respectively. We access the true and false variables in A via $A^{\boldsymbol{T}} = \{a \in \mathcal{A} \mid \boldsymbol{T}a \in A\}$ and $A^{\boldsymbol{F}} = \{a \in \mathcal{A} \mid \boldsymbol{F}a \in A\}$,

[1] We use '−/1' to stand for classical negation.

loop
propagate // compute deterministic consequences
if no conflict **then**
if all variables assigned **then return** variable assignment
else *decide* // non-deterministically assign some literal
else
if top-level conflict **then return** unsatisfiable
else
analyze // analyze conflict and add a conflict constraint
backjump // undo assignments until conflict constraint is unit

Figure 1: Basic decision algorithm: CDCL

respectively. A is conflicting, if $A^{\boldsymbol{T}} \cap A^{\boldsymbol{F}} \neq \emptyset$; A is total, if it is non-conflicting and $A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}} = \mathcal{A}$. For generality, we represent Boolean constraints by *nogoods* (Dechter 2003). A nogood is a set $\{\sigma_1, \ldots, \sigma_m\}$ of signed literals, expressing that any assignment containing $\sigma_1, \ldots, \sigma_m$ is inadmissible. Accordingly, a total assignment A is a *solution* for a set Δ of nogoods if $\delta \not\subseteq A$ for all $\delta \in \Delta$. While clauses can be directly mapped into nogoods, logic programs are subject to a more involved translation. For instance, an atom a defined by two rules $a \texttt{ :- } b, \texttt{not } c$ and $a \texttt{ :- } d$ gives rise to three nogoods: $\{\boldsymbol{T}a, \boldsymbol{F}x_{\{b,\texttt{not } c\}}, \boldsymbol{F}x_{\{d\}}\}$, $\{\boldsymbol{F}a, \boldsymbol{T}x_{\{b,\texttt{not } c\}}\}$, and $\{\boldsymbol{F}a, \boldsymbol{T}x_{\{d\}}\}$, where $x_{\{b,\texttt{not } c\}}$ and $x_{\{d\}}$ are auxiliary variables for the bodies of the two previous rules. Similarly, the body $\{b, \texttt{not } c\}$ leads to nogoods $\{\boldsymbol{F}x_{\{b,\texttt{not } c\}}, \boldsymbol{T}b, \boldsymbol{F}c\}$, $\{\boldsymbol{T}x_{\{b,\texttt{not } c\}}, \boldsymbol{F}b\}$, and $\{\boldsymbol{T}x_{\{b,\texttt{not } c\}}, \boldsymbol{T}c\}$. See (Gebser et al. 2012) for full details. Note that translating logic programs into nogoods adds auxiliary variables. For simplicity, we restrict our formal elaboration to atoms in $\mathcal{A}$ (also because our approach leaves such internal variables unaffected anyway).

Conflict-driven constraint learning

Given that we are primarily interested in the heuristic machinery of a solver, we only provide a high-level description of the basic decision algorithm for conflict-driven constraint learning (CDCL; (Marques-Silva and Sakallah 1999; Zhang et al. 2001)) in Figure 1. CDCL starts by extending a (partial) assignment by deterministic (unit) propagation. Although propagation aims at forgoing nogood violations, assigning a literal implied by one nogood may lead to the violation of another nogood; this situation is called *conflict*. If the conflict can be resolved, it is analyzed to identify a conflict constraint. The latter represents a "hidden" conflict reason that is recorded and guides backjumping to an earlier stage such that the complement of some formerly assigned literal is implied by the conflict constraint, thus triggering propagation. Only when propagation finishes without conflict, a (heuristically chosen) literal can be assigned provided that the assignment at hand is partial, while a solution has been found otherwise. See (Biere et al. 2009) for details.

A characteristic feature of CDCL is its look-back based approach. Central to this are conflict-driven mechanisms scoring variables according to their prior conflict involvement. These scores guide heuristic choices regarding literal selection as well as constraint learning and deletion.

A decision heuristic is used to implement the non-deterministic assignment done via *decide* in the CDCL algorithm in Figure 1. In fact, the selection of an atom along with its sign relies on two such functions:

$$h : \mathcal{A} \to [0, +\infty) \quad \text{and} \quad s : \mathcal{A} \to \{\boldsymbol{T}, \boldsymbol{F}\} .$$

Both functions vary over time. To capture this, we use h_i and s_i to denote the specific mappings in the ith iteration of CDCL's main loop. Analogously, we use A_i to represent the ith assignment (after *propagation*). We use $i = 0$ to refer to the initialization of both functions via h_0 and s_0; similarly, A_0 gives the initial assignment (after *propagation*).

The following lines give a more detailed yet still high-level account of the non-deterministic assignment done by *decide* in the CDCL algorithm for $i \geq 1$ (and a given h_0):[2]

1. $h_i(a) := \alpha_i \times h_{i-1}(a) + \beta_i(a)$ for each $a \in \mathcal{A}$
2. $U := \mathcal{A} \setminus ({A_{i-1}}^{\boldsymbol{T}} \cup {A_{i-1}}^{\boldsymbol{F}})$
3. $C := \mathit{argmax}_{a \in U} h_i(a)$
4. $a := \tau(C)$
5. $A_i := A_{i-1} \cup \{s_i(a)a\}$

The first line describes the development of the heuristic depending on a global decay parameter α_i and a variable-specific scoring function β_i. The set U contains all atoms unassigned at step i. Among them, the ones with the highest heuristic value are collected in C. Whenever C contains several (equally scored) variables, the solver must break the tie by selecting one atom $\tau(C)$ from C.

Look-back based heuristics rely on information gathered during conflict analysis in CDCL. Starting from some initial heuristic values in h_0, the heuristic function is continued as in Item 1 above, where $\alpha_i \in [0, 1]$ is a global parameter decaying the influence of past values and $\beta_i(a)$ gives the conflict score attributed to variable a within conflict analysis. The value of $\beta_i(a)$ can be thought of being 0 unless a was scored by *analyze* in CDCL. Similarly, α_i usually equals 1 unless it was lowered at some system-specific point, such as after a *restart*. Occurrence-based heuristics like *moms* (Pretolani 1996) furnish initial heuristics. Prominent look-back heuristics are *berkmin* (Goldberg and Novikov 2002) and *vsids* (Moskewicz et al. 2001).

For illustration, let us look at a rough trace of atoms a, b, and c in a fictive run of the CDCL algorithm.

i	*operation*	A	h	s	α	β
		$a \;\; b \;\; c \ldots$	$a \; b \; c \ldots$			$a \; b \; c \ldots$
0			0 1 1	$\boldsymbol{T}$	1	0 0 0
1	*propagate*	$\boldsymbol{F}$	0 1 1	$\boldsymbol{T}$	1	0 0 0
	decide	$\boldsymbol{F} \;\; \boldsymbol{T}$	0 1 1	$\boldsymbol{T}$	1	0 0 0

The initial heuristic h_0 prefers b, c over a; the sign heuristic s constantly assigns $\boldsymbol{T}$. Initial propagation assigns $\boldsymbol{F}$ to a. This leaves all heuristics unaffected. When invoking *decide*, we find b and c among the unassigned variables in U (in Item 2 above). Assuming the maximum value of h_1 to be 1,

[2] For clarity, we keep using indexes in this algorithmic setting although this is unnecessary in view of assignment operator ':='.

both are added to C. This tie is broken by selecting $\tau(C) = b$ in C. Given that the (constant) sign heuristic yields $\boldsymbol{T}$, Item 5 adds signed literal $\boldsymbol{T}b$ to the current assignment.

Next suppose we encounter a conflict involving c at step 8. This leads to an incrementation of $\beta_8(c)$.

		a b c ...	a b c ...			a b c ...
8	*propagate*	$\boldsymbol{F}$ $\boldsymbol{T}$ $\boldsymbol{F}$	0 2 2	$\boldsymbol{T}$	1	0 0 0
	analyze	$\boldsymbol{F}$ $\boldsymbol{T}$ $\boldsymbol{F}$	0 2 2	$\boldsymbol{T}$	1	0 0 1
	backjump	$\boldsymbol{F}$	0 2 3	$\boldsymbol{T}$	1	0 0 0
9	*propagate*	$\boldsymbol{F}$	0 2 3	$\boldsymbol{T}$	1	0 0 0
	decide	$\boldsymbol{F}$ $\boldsymbol{T}$	0 2 3	$\boldsymbol{T}$	1	0 0 0

As at step 1, b and c are unassigned after backjumping. Unlike above, c is now heuristically preferred to b since it occurred more frequently within conflicts.

Without going into detail, we mention that at certain steps i, parameter α_i is decreased for decaying the values of h_i and the conflict scores in β_i are re-set (eg. after *analyze*).

Also, look-back based sign heuristics take advantage of previous information. The common approach is to choose the polarity of a literal according to the higher number of occurrences in recorded nogoods (Moskewicz et al. 2001). Another effective approach is *progress saving* (Pipatsrisawat and Darwiche 2007), caching truth values of (certain) retracted variables and reusing them for sign selection.

Although we focus on look-back heuristics, we mention that look-ahead heuristics aim at shrinking the search space by selecting the (signed) variable offering most implications. This approach relies on failed-literal detection (Freeman 1995) for counting the number of propagations obtained by (temporarily) adding in turn the variable and its negation to the current assignment. This count can be used in Item 1 above for computing the values $\beta_i(a)$, while all α_i are set to 0 (because no past information is taken into account).

Heuristic language elements

We express heuristic modifications via a set $\mathcal{H}$ of *heuristic atoms* disjoint from $\mathcal{A}$. Such a heuristic atom is formed from a dedicated predicate `_h` along with four arguments: a (reified) atom $a \in \mathcal{A}$, a heuristic modifier m, and two integers $v, p \in \mathbb{Z}$. A heuristic modifier is used to manipulate the heuristic treatment of an atom a via the modifier's value given by v. The role of this value varies for each modifier. We distinguish four primitive heuristic modifiers:

`init` for initializing the heuristic value of a with v,

`factor` for amplifying the heuristic value of a by factor v,

`level` for ranking all atoms; the rank of a is v,

`sign` for attributing the sign of v as truth value to a.

While v allows for changing an atom's heuristic behavior relative to *other* atoms, the second integer p allows us to express a priority for disambiguating similar heuristic modifications to the *same* atom. This is particularly important in our dynamic setting, where varying heuristic atoms may be obtained in view of the current assignment. For instance, the heuristic atoms $\_\mathtt{h}(b, \mathtt{sign}, 1, 3)$ and $\_\mathtt{h}(b, \mathtt{sign}, -1, 5)$ aim at assigning opposite truth values to atom b. This conflict can be resolved by preferring the heuristic modification with the higher priority, viz. 5 in $\_\mathtt{h}(b, \mathtt{sign}, -1, 5)$. Obviously such priorities can only support disambiguation but not resolve conflicting values sharing the same priority.

For accommodating priorities, we define for an assignment A the *preferred values* for modifier m on atom a as

$$V_{a,m}(A) = \mathit{argmax}_{v \in \mathbb{Z}}\{p \mid \boldsymbol{T}\_\mathtt{h}(a, m, v, p) \in A\}.$$

Heuristic values are dynamic; they are extracted from the current assignment and may thus vary during solving. Note that $V_{a,m}(A)$ returns the singleton set $\{v\}$, if the current assignment A contains a single true heuristic atom $\_\mathtt{h}(a, m, v, p)$ involving a and m. $V_{a,m}(A)$ is empty whenever there are no such heuristic atoms. And whenever all heuristic atoms regarding a and m have the same priority p, $V_{a,m}(A)$ is equivalent to $\{v \mid \boldsymbol{T}\_\mathtt{h}(a, m, v, p) \in A\}$.

Here are a few examples. We obtain $V_{b,\mathtt{sign}}(A_1) = \{-1\}$ and $V_{c,\mathtt{init}}(A_1) = \emptyset$ from assignment $A_1 = \{\boldsymbol{F}a, \boldsymbol{T}\_\mathtt{h}(b, \mathtt{sign}, 1, 3), \boldsymbol{T}\_\mathtt{h}(b, \mathtt{sign}, -1, 5)\}$, while assignment $A_2 = \{\boldsymbol{T}\_\mathtt{h}(b, \mathtt{sign}, 1, 3), \boldsymbol{T}\_\mathtt{h}(b, \mathtt{sign}, -1, 3)\}$ results in $V_{b,\mathtt{sign}}(A_2) = \{1, -1\}$.

For ultimately resolving ambiguities among alternative values for heuristic modifiers, we propose for a set $V \subseteq \mathbb{Z}$ of integers the function $\nu(V)$ as

$$\mathit{max}\big(\{v \in V \mid v \geq 0\} \cup \{0\}\big) + \mathit{min}\big(\{v \in V \mid v \leq 0\} \cup \{0\}\big).$$

Note that $\nu(\emptyset) = 0$, attributing 0 the status of a neutral value. Alternative options exist, like taking means or median of V or even time specific criteria relating to the emergence of values in the assignment. In the above examples, we get $\nu(V_{b,\mathtt{sign}}(A_1)) = -1$ and $\nu(V_{b,\mathtt{sign}}(A_2)) = 0$.

Given this, we proceed by defining the *domain-specific extension* d to the heuristic function h for $a \in \mathcal{A}$ as

$$d_0(a) = \nu(V_{a,\mathtt{init}}(A_0)) + h_0(a)$$

and for $i \geq 1$

$$d_i(a) = \begin{cases} \nu(V_{a,\mathtt{factor}}(A_i)) \times h_i(a) & \text{if } V_{a,\mathtt{factor}}(A_i) \neq \emptyset \\ h_i(a) & \text{otherwise} \end{cases}$$

First of all, it is important to note that d is merely a modification and not a replacement of the system heuristic h. In fact, d extends the range of h to $(-\infty, +\infty)$. Negative values serve as penalties. The values of the `init` modifiers are added to h_0 in d_0. The use of addition rather than multiplication allows us to override an initial value of 0. Also, the higher the absolute value of the `init` modifier, the longer lasts its effect (given the decay of heuristic values). Unlike this, `factor` modifiers rely on multiplication because they aim at de- or increasing conflict scores gathered during conflict analysis. In view of h's range, a factor greater than 1 amplifies the score, a negative one penalizes the atom, and 0 resets the atom's score. Enforcing a factor of 1 transfers control back to the system heuristic h.

Heuristically modified logic programs are simply programs over $\mathcal{A} \cup \mathcal{H}$, the original vocabulary extended by heuristic atoms (without restrictions). As a first example, let us extend our planning encoding by a rule favoring atoms expressing action occurrences close to the goal situation.

```
_h(occ(A,T),factor,T,0) :- action(A),time(T).
```

With `factor`, we impose a bias on the underlying heuristic function h. Rather than comparing, for instance, the plain values $h(\texttt{occ(a,2)})$ and $h(\texttt{occ(a,3)})$, a decision is made by looking at $2 \times h(\texttt{occ(a,2)})$ and $3 \times h(\texttt{occ(a,3)})$, even though it still depends on h. A further refined strategy may suggest considering climbing actions as early as possible.

```
_h(occ(climb,T),factor,l-T,1) :- time(T).
```

Clearly, this rule conflicts with the more general rule above. However, this conflict is resolved in favor of the more specific rule by attributing it a higher priority (viz. 1 versus 0).

For capturing a *domain-specific extension* t to the sign heuristic s, we define for $a \in \mathcal{A}$ and $i \geq 0$:

$$t_i(a) = \begin{cases} \boldsymbol{T} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) > 0 \\ \boldsymbol{F} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) < 0 \\ s_i(a) & \text{otherwise} \end{cases}$$

As with d above, the extension t to the sign heuristic is dynamic. The sign of the modifier's preferred value determines the truth value to assign to an atom at hand. No `sign` modifier (or enforcing a value of 0) leaves sign selection with the system's sign heuristic s. For example, the heuristic rule

```
_h(holds(F,T),sign,-1,0) :- fluent(F),time(T).
```

tells the solver to assign false to non-deterministically chosen fluents. The next pair of rules is a further refinement of our strategy on climbing actions, favoring their effective occurrence in the first half of the plan.

```
_h(occ(climb,T),sign, 1,0) :- T<l/2,time(T).
_h(occ(climb,T),sign,-1,0) :- T>l/2,time(T).
```

Thus, while the atom $\texttt{occ(climb,1)}$ is preferably made true, false should rather be assigned to $\texttt{occ(climb,1)}$.

Finally, for accommodating rankings induced by `level` modifiers, we define for an assignment A and $\mathcal{A}' \subseteq \mathcal{A}$:

$$\ell_A(\mathcal{A}') = \mathit{argmax}_{a \in \mathcal{A}'} \nu(V_{a,\texttt{level}}(A))$$

The set $\ell_A(\mathcal{A}')$ gives all atoms in $\mathcal{A}'$ with the highest `level` values relative to the current assignment A. Similar to d and t above, this construction is also dynamic and the rank of atoms may vary during solving. The function ℓ_A is then used to modify the selection of unassigned atoms in the above elaboration of *decide*. For this purpose, we replace Item 2 by $U := \ell_A(\mathcal{A} \setminus (A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}}))$ in order to restrict U to unassigned atoms of (current) highest rank. Unassigned atoms at lower levels are only considered once all atoms at higher levels have been assigned. Atoms without an associated level default to level 0 because $\nu(\emptyset) = 0$. Hence, negative levels act as a penalty since the respective atoms are only taken into account once all atoms with non-negative or no associated level have been assigned.

For a complementary example, consider a `level`-based formulation of the previous (`factor`-based) heuristic rule.

```
_h(occ(A,T),level,T,0) :- action(A),time(T).
```

Unlike the above, $\texttt{occ(a,2)}$ and $\texttt{occ(a,3)}$ are now associated with different ranks, which leads to strictly preferring $\texttt{occ(a,3)}$ over $\texttt{occ(a,2)}$ whenever both atoms are unassigned. Hence, `level` modifiers partition the set of atoms and restrict h to unassigned atoms at the highest level.

The previous replacement along with the above amendments of h and s through the domain-specific extensions d and t yields the following elaboration of CDCL's heuristic choice operation *decide* for $i \geq 1$ (and given d_0).[2]

0. $h_{i-1}(a) := d_{i-1}(a)$ for each $a \in \mathcal{A}$
1. $h_i(a) := \alpha_i \times h_{i-1}(a) + \beta_i(a)$ for each $a \in \mathcal{A}$
2. $U := \ell_{A_{i-1}}(\mathcal{A} \setminus ({A_{i-1}}^{\boldsymbol{T}} \cup {A_{i-1}}^{\boldsymbol{F}}))$
3. $C := \mathit{argmax}_{a \in U} d_i(a)$
4. $a := \tau(C)$
5. $A_i := A_{i-1} \cup \{t_i(a)a\}$

Although we formally model both h and d (as well as s and t) as functions, there is a substantial conceptual difference in practice in that h is a system-specific data structure while d is an associated method. This is also reflected above, where h is subject to assignments. Item 0 makes sure that our heuristic modifications take part in the look-back based evolution in Item 1, and are thus also subject to decay. We added this as a separate line rather than integrating it into Item 1 in order to stress that our modifications are modular in leaving the underlying heuristic machinery unaffected. Item 2 gathers in U all unassigned atoms of highest rank. Among them, Item 3 collects in C all atoms a with a maximum heuristic value $d_i(a)$. Since this is not guaranteed to yield a unique element, the system-specific tie-breaking function τ is evoked to return a unique atom. Finally, the modified sign heuristic t_i determines a truth value for a, and the resulting signed literal $t_i(a)a$ is added to the current assignment.

Note that so far all sample heuristic rules were *static* in the sense that they are turned into facts by the grounder and thus remain unchanged during solving. Examples of dynamic heuristic rules are given at the end of next section.

Our simple heuristic language is easily extended by further heuristic atoms. For instance, $\texttt{\_h}(a, \texttt{true}, v, p)$ and $\texttt{\_h}(a, \texttt{false}, v, p)$ have turned out to be useful in practice.

```
_h(A,level,V,P) :- _h(A,true, V,P).
_h(A,sign, 1,P) :- _h(A,true, V,P).
_h(A,level,V,P) :- _h(A,false,V,P).
_h(A,sign,-1,P) :- _h(A,false,V,P).
```

For instance, the heuristic atom $\texttt{\_h}(a, \texttt{true}, 3, 3)$ expands to $\texttt{\_h}(a, \texttt{level}, 3, 3)$ and $\texttt{\_h}(a, \texttt{sign}, 1, 3)$, expressing a preference for both making a decision on a and assigning it to true. On the other hand, $\texttt{\_h}(a, \texttt{false}, -3, 3)$ expands to $\texttt{\_h}(a, \texttt{level}, -3, 3)$ and $\texttt{\_h}(a, \texttt{sign}, -1, 3)$, thus suggesting not to make a decision on a but to assign it to false if there is no "better" decision variable.

Another shortcut of pragmatic value is the abstraction from specific priorities. For this, we use the following rule.

```
_h(A,M,V,#abs(V)) :- _h(A,M,V).
```

With it, we can directly describe the heuristic restriction used in (Rintanen 2011) to simulate planning by iterated deepening A^* (Korf 1985) in SAT solving through limiting choices to action variables, assigning those for time `T` before those for time `T+1`, and always assigning truth value `true` (where `l` is a constant indicating the planning horizon):

```
_h(occ(A,T),true,l-T) :- action(A), time(T).
```

Although we impose no restriction on the occurrence of heuristic atoms within logic programs, it seems reasonable

Setting	*Labyrinth*	*Sokoban*	*Hanoi Tower*
base configuration	9,108s (14)	2,844s (3)	9,137s (11)
	24,545,667	19,371,267	41,016,235
_h(a,init,2)	95%(12) 94%	91%(**1**) 84%	85% (9) 89%
_h(a,factor,4)	**78**% (**8**) 30%	120%(**1**)107%	109%(11)110%
_h(a,factor,16)	**78**%(10) 23%	120%(**1**)107%	109%(11)110%
_h(a,level,1)	90%(12) **5**%	119%(2) 91%	126%(15)120%
_h(f,init,2)	103%(14)123%	**74**%(2) **71**%	97%(10)109%
_h(f,factor,2)	98%(12) 49%	116%(3)134%	**55**% (**6**) **70**%
_h(f,sign,-1)	94%(13) 89%	105%(**1**)100%	92%(12) 92%

Table 1: Selection from evaluation of heuristic modifiers

to require that the addition of rules containing heuristic atoms does not alter the stable models of the original program. That is, given a logic program P over $\mathcal{A}$ and a set of rules H over $\mathcal{A} \cup \mathcal{H}$, we aim at a one-to-one correspondence between the stable models of P and $P \cup H$ and their identity upon projection on $\mathcal{A}$. This property is guaranteed whenever heuristic atoms occur only in the head of rules and thus only depend upon regular atoms. In fact, so far, this class of rules turned out to be expressive enough to model all heuristics of interest, including the ones presented in this paper. It remains future work to see whether more sophisticated schemes, eg., involving recursion, are useful.

Experiments

We implemented our approach as a dedicated heuristic module within the ASP solver *clasp* (2.1; available at (hclasp)). We consider *moms* (Pretolani 1996) as initial heuristic h_0 and *vsids* (Moskewicz et al. 2001) as heuristic function h_i. Accordingly, the sign heuristic *s* is set to the one associated with *vsids*. As base configuration, we use *clasp* with options `--heu=vsids` and `--init-moms`. To take effect, the heuristic atoms as well as their contained atoms must be made visible to the solver via `#show` directives. Once the option `--heu=domain` is passed to `clasp`, it extracts the necessary information from the symbol table and applies the heuristic modifications when it comes to non-deterministic assignments. Our experiments ran under Linux on dual Xeon E5520 quad-core processors with 2.26GHz and 48GB RAM. Each run was restricted to 600s CPU time. Timeouts account for 600s and performed choices.

To begin with, we report on a systematic study comparing single heuristic modifications. A selection of best results is given in Table 1; full results are available at (hclasp). We focus on well-known ASP planning benchmarks in order to contrast heuristic modifications on comparable problems: *Labyrinth*, *Sokoban*, and *Hanoi Tower*, each comprising 32 instances from the third ASP competition (Calimeri et al. 2011).[3] We contrast the aforementioned base configuration with 38 heuristic modifications, (separately) promoting the choice of actions (*a*) and fluents (*f*) via the heuristic modifiers `factor` (1,2,4,8,16), `init` (2,4,8,16), `level` (1,-1), `sign` (1,-1), as well as attributing values to `factor`, `init`, and `level` by ascending and descending time points. The first line of Table 1 gives the sum of times,

[3]All instances are satisfiable except for one third in *Sokoban*.

Setting	*Diagnosis*	*Expansion*	*Repair (H)*	*Repair (S)*
base config.	111.1s(115)	161.5s(100)	101.3s(113)	33.3s(27)
s,-1	324.5s(407)	7.6s (3)	8.4s (5)	3.1s (**0**)
s,-1 f,2	310.1s(387)	7.4s (**2**)	3.5s (**0**)	3.2s (1)
s,-1 f,8	305.9s(376)	7.7s (**2**)	3.1s (**0**)	2.9s (**0**)
s,-1 l,1	**76.1**s (**83**)	**6.6**s (**2**)	**0.8**s (**0**)	2.2s (1)
l,1	77.3s (86)	12.9s (5)	3.4s (**0**)	**2.1**s (**0**)

Table 2: Abductive problems with optimization

timeouts, and choices obtained by the base configuration on all 32 instances of each problem class. The results of the two configurations using `factor,1` differ from these figures in the low per mille range, demonstrating that the infrastructure supporting heuristic modifications does not lead to a loss in performance. The seven configurations in Table 1 yield best values in at least one category (indicated in boldface). We express the accumulated times and choices as percentage wrt the base configuration; timeouts are total. We see that the base configuration can always be dominated by a heuristic modification. However, the whole spectrum of modifiers is needed to accomplish this. In other words, there is no dominating heuristic modifier and each problem class needs a customized heuristic. Looking at *Labyrinth*, we observe that a preferred choice of action occurrences (a) pays off. The stronger this is enforced, the fewer choices are made. However, the extremely low number of choices with `level` does not result in less time or timeouts (compared to a "lighter" `factor`-based enforcement). While with `level` *all* choices are made on heuristically modified atoms, both `factor`-based modifications result in only 43% such choices and thus leave much more room to the solver's heuristic. For a complement, *a*,`init`,2 as well as the base configuration (with *a*,`factor`,1) make 14% of their choices on heuristically modified atoms (though the former produces in total 6% less choices than the latter). Similar yet less extreme behaviors are observed on the two other classes. With *Hanoi Tower*, a slight preference of fluents yields a strictly dominating configuration, whereas no dominating improvement was observed with *Sokoban*.

Next, we apply our heuristic approach to problems using abduction in combination with a `#minimize` statement minimizing the number of abducibles. We consider Circuit *Diagnosis*, Metabolic Network *Expansion*, and Transcriptional Network *Repair* (including two distinct experiments, *H* and *S*). The first uses the ISCAS-85 benchmark circuits along with test cases generated as in (Siddiqi 2011); this results in 790 benchmark instances. The second one considers the completion of the metabolic network of *E.coli* with reactions from *MetaCyc* in view of generating target from seed metabolites (Schaub and Thiele 2009). We selected the 450 most difficult benchmarks in the suite. Finally, we consider repairing the transcriptional network of *E.coli* from *RegulonDB* in view of two distinct experiment series (Gebser et al. 2010). Selecting the most difficult triple repairs provided us with 1000 instances. Our results are summarized in Table 2. Each entry gives the average runtime and number of timeouts. Here, heuristic modifiers apply only to abducibles subject to minimization. For supporting minimization, we

Problem	*base*		*base*+_h		*base (SAT)*		*base*+_h *(SAT)*	
Blocks'00	134.4s	(180/61)	9.2s	(239/3)	163.2s	(59)	2.6s	(0)
Elevator'00	3.1s	(279/0)	0.0s	(279/0)	3.4s	(0)	0.0s	(0)
Freecell'00	288.7s	(147/115)	184.2s	(194/74)	226.4s	(47)	52.0s	(0)
Logistics'00	145.8s	(148/61)	115.3s	(168/52)	113.9s	(23)	15.5s	(3)
Depots'02	400.3s	(51/184)	297.4s	(115/135)	389.0s	(64)	61.6s	(0)
Driverlog'02	308.3s	(108/143)	189.6s	(169/92)	245.8s	(61)	6.1s	(0)
Rovers'02	245.8s	(138/112)	165.7s	(179/79)	162.9s	(41)	5.7s	(0)
Satellite'02	398.4s	(73/186)	229.9s	(155/106)	364.6s	(82)	30.8s	(0)
Zenotravel'02	350.7s	(101/169)	239.0s	(154/116)	224.5s	(53)	6.3s	(0)
Total	252.8s	(1225/1031)	158.9s	(1652/657)	187.2s	(430)	17.1s	(3)

Table 3: Planning Competition Benchmarks '00 and '02

assign false to such abducibles (`s,-1`)[4] and gradually increase the bias of their choice by imposing `factor` 2 and 8 (`f`) or enforce it via a `level` modifier (`l,1`). The second last setting[5] in Table 2 is the winner, leading to speedups of one to two orders of magnitude over the base configuration. Interestingly, merely fixing the sign heuristics to $\boldsymbol{F}$ leads at first to a deterioration of performance on *Diagnosis* problems. This is finally overcome by the constant improvement observed by gradually strengthening the bias of choosing abducibles. The stronger the preference for abducibles, the faster the solver converges to an optimum solution. This limited experiment already illustrates that sometimes the right combination of heuristic modifiers yields the best result.

Finally, let us consider true PDDL planning problems. For this, we selected 20 instances from the STRIPS domains of the 2000 and 2002 planning competition (ICAPS).[6] In turn, we translated these PDDL instances into facts via *plasp* (Gebser et al. 2011) and used a simple planning encoding with 15 different plan lengths (`l=5,10,..,75`) to generate 3000 ASP instances. Inspired yet different from (Rintanen 2012), we devised a dynamic heuristic that aims at propagating fluents' truth values backwards in time. Attributing levels via `l-T+1` aims at proceeding depth-first from the goal fluents.

```
_h(holds(F,T-1),true, l-T+1) :- holds(F,T).
_h(holds(F,T-1),false,l-T+1) :-
        fluent(F), time(T), not holds(F,T).
```

Our results are given in Table 3. Each entry gives the average runtime along with the number of (solved satisfiable instances and) timeouts (in columns two and three). Our heuristic amendment (*base*+`_h`) greatly improves over the base configuration in terms of runtime and timeouts. On the overall set of benchmarks, it provides us with 427 more plans and 374 less timeouts. As already observed by (Rintanen 2012), the heuristic effect is stronger on satisfiable instances. This is witnessed by the two last columns restricting results to 1655 satisfiable instances solved by either system setup. Our heuristic extension allows us to reduce the total number of timeouts from 430 to 3; the reduction in solving time would be even more drastic with a longer timeout.

Interestingly, the previous dynamic heuristic has no overwhelming effect on our initial ASP planning problems. An improvement was only observed on *Hanoi Tower* problems (being susceptible to choices on fluents), viz. '54%(7) **57%**' in terms of the format used in Table 1. However, restricting the heuristic to positive fluents by only using the first rule gives a substantial improvement, namely '**19%**(**2**) 66%', in terms of runtime and timeouts. A direct comparison of both heuristics shows that, although the latter performs 15% more choices, it encounters 75% fewer conflicts than the former.

[4]Assigning $\boldsymbol{T}$ instead leads to a deterioration of performance.

[5]This corresponds to using `_h`$(a,$ `false`$, 1)$ for an abducible a.

[6]We discard *Schedule'00* due to grounding issues.

Discussion

Various ways of adding domain-specific information have been explored in the literature. A prominent approach is to implement forms of preferential reasoning by directing choices through a given partial order on literals (Castell et al. 1996; Di Rosa, Giunchiglia, and Maratea 2010; Giunchiglia and Maratea 2012). To some degree, this can be simulated by heuristic modifiers like `_h`$(a,$ `false`$, 1)$ that allow for computing a (single) inclusion-minimal model. However, as detailed in (Di Rosa, Giunchiglia, and Maratea 2010), enumerating all such models needs additional constraints or downstream tester programs. Similarly, (Balduccini 2011) modifies the heuristic of the ASP solver *smodels* to accommodate learning from smaller instances. See also (Faber, Leone, and Pfeifer 2001; Faber et al. 2007). Most notably, (Rintanen 2012) achieves impressive results in planning by equipping a SAT solver with planning-specific heuristics. All aforementioned approaches need customized changes to solver implementations. Hence, it will be interesting to investigate how these approaches can be expressed and combined in our declarative framework. Declarative approaches to incorporating control knowledge can be found in heuristic planning. For instance, (Bacchus and Kabanza 2000) harness temporal logic formulas, while (Sierra-Santibáñez 2004) also uses dedicated predicates for controlling backtracking in a forward planner. However, care must be taken when it comes to modifying a solver's heuristics. Although it may lead to great improvements, it may just as well lead to a degradation of search. In fact, the restriction of choice variables may result in exponentially larger search spaces (Järvisalo, Junttila, and Niemelä 2005). This issue is reflected in our choice of heuristic modifiers, ranging from an `init`ial bias, over a continued yet scalable one by `factor`, to a strict preference with `level`.

To sum up, we introduced a declarative framework for incorporating domain-specific heuristics into ASP solving. The seamless integration into ASP's input language provides us with a general and flexible tool for expressing domain-specific heuristics. As such, we believe it to be the first of its kind. Our heuristic framework offers completely new possibilities of applying, experimenting, and studying domain-specific heuristics in a uniform setting. Our example heuristics merely provide first indications on the prospect of our approach, but much more systematic empirical studies are needed to exploit its full power.

Acknowledgments. This work was partly funded by DFG grants SCHA 550/8-3 and SCHA 550/9-1.

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116(1-2):123–191.

Balduccini, M. 2011. Learning and using domain-specific heuristics in ASP solvers. *AI Communic.* 24(2):147–164.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds. 2009. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Calimeri, F. et al. 2011. The third answer set programming competition: Preliminary report of the system competition track. In Delgrande and Faber (2011), 388–403.

Castell, T.; Cayrol, C.; Cayrol, M.; and Le Berre, D. 1996. Using the Davis and Putnam procedure for an efficient computation of preferred models. In Wahlster, W., ed., *Proceedings of the Twelfth European Conference on Artificial Intelligence (ECAI'96)*, 350–354. John Wiley & sons.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann.

Delgrande, J., and Faber, W., eds. 2011. *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*. Springer.

Di Rosa, E.; Giunchiglia, E.; and Maratea, M. 2010. Solving satisfiability problems with preferences. *Constraints* 15(4):485–515.

Faber, W.; Leone, N.; Maratea, M.; and Ricca, F. 2007. Experimenting with look-back heuristics for hard ASP programs. In Baral, C.; Brewka, G.; and Schlipf, J., eds., *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, 110–122. Springer.

Faber, W.; Leone, N.; and Pfeifer, G. 2001. Experimenting with heuristics for answer set programming. In Nebel, B., ed., *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, 635–640. Morgan Kaufmann.

Freeman, J. 1995. *Improvements to Propositional Satisfiability Search Algorithms*. Ph.D. Dissertation, University of Pennsylvania.

Gebser, M.; Guziolowski, C.; Ivanchev, M.; Schaub, T.; Siegel, A.; Thiele, S.; and Veber, P. 2010. Repair and prediction (under inconsistency) in large biological networks with answer set programming. In Lin, F., and Sattler, U., eds., *Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR'10)*, 497–507. AAAI Press.

Gebser, M.; Kaminski, R.; Knecht, M.; and Schaub, T. 2011. plasp: A prototype for PDDL-based planning in ASP. In Delgrande and Faber (2011), 358–363.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Schaub, T. 2012. *Answer Set Solving in Practice*. Morgan and Claypool.

Giunchiglia, E., and Maratea, M. 2012. Algorithms for solving satisfiability problems with qualitative preferences. In Erdem, E.; Lee, J.; Lierler, Y.; and Pearce, D., eds., *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, 327–344. Springer.

Goldberg, E., and Novikov, Y. 2002. BerkMin: A fast and robust SAT solver. In *Proceedings of the Fifth Conference on Design, Automation and Test in Europe (DATE'02)*, 142–149. IEEE Computer Society Press.

hclasp. `http://www.cs.uni-potsdam.de/hclasp`.

ICAPS. `http://ipc.icaps-conference.org`.

Järvisalo, M.; Junttila, T.; and Niemelä, I. 2005. Unrestricted vs restricted cut in a tableau method for Boolean circuits. *Annals of Mathematics and Artificial Intelligence* 44(4):373–399.

Korf, R. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Lifschitz, V. 2002. Answer set programming and plan generation. *Artificial Intelligence* 138(1-2):39–54.

Marques-Silva, J., and Sakallah, K. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers* 48(5):506–521.

Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Thirty-eighth Conference on Design Automation (DAC'01)*, 530–535. ACM Press.

Pipatsrisawat, K., and Darwiche, A. 2007. A lightweight component caching scheme for satisfiability solvers. In Marques-Silva, J., and Sakallah, K., eds., *Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, 294–299. Springer.

Pretolani, D. 1996. Efficiency and stability of hypergraph SAT algorithms. In Johnson, D., and Trick, M., eds., *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26, 479–498. American Mathematical Society.

Rintanen, J. 2011. Planning with SAT, admissible heuristics and A*. In Walsh (2011), 2015–2020.

Rintanen, J. 2012. Planning as satisfiability: heuristics. *Artificial Intelligence* 193:45–86.

Schaub, T., and Thiele, S. 2009. Metabolic network expansion with ASP. In Hill, P., and Warren, D., eds., *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, 312–326. Springer.

Siddiqi, S. 2011. Computing minimum-cardinality diagnoses by model relaxation. In Walsh (2011), 1087–1092.

Sierra-Santibáñez, J. 2004. Heuristic planning: A declarative approach based on strategies for action selection. *Artificial Intelligence* 153(1-2):307–337.

Walsh, T., ed. 2011. *Proceedings of the Twenty-second International Joint Conference on Artificial Intelligence (IJCAI'11)*. IJCAI/AAAI.

Zhang, L.; Madigan, C.; Moskewicz, M.; and Malik, S. 2001. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'01)*, 279–285. ACM Press.