ASP: Heuristic-driven Solving **and** Optimization and Preference Handling

Yucong He
20250717

Heuristic-driven Solving

Command Line Options of heuristic

```
clingo psign.lp \
--heuristic=Domain
```

--heuristic={Berkmin,Vmtf,Vsids,Unit,None,Domain}

Use *BerkMin*-like decision heuristic [55] (with argument `Berkmin`), *Siege*-like decision heuristic [76] (with argument `Vmtf`), *Chaff*-like decision heuristic [69] (with argument `Vsids`), *Smodels*-like decision heuristic [78] (with argument `Unit`), or (arbitrary) static variable ordering (with argument `None`). Finally, argument `Domain` enables a *domain-specific* decision heuristic as described in Section 10.

clasp and clingo provide means for incorporating domain-specific heuristics into ASP solving. This allows for modifying the heuristic of the solver from within a logic program or from the command line.

Heuristic information is represented within a logic program my means of heuristic directives of the form

$$\texttt{\#heuristic}\ A : B.\quad [w\texttt{@}p, m]$$

where A is an atom, B a rule body, and w, p and m are terms. The priority '@p' is optional. Different types of heuristic information can be controlled using the modifiers `sign`, `level`, `true`, `false`, `init` and `factor` for m. We introduce them below step by step.

Heuristic modifier: sign

```
#heuristic a. [1,sign]
{a}.
```

The modifier `sign` allows for controlling the truth value assigned to variables subject to a choice within the solver.

```
#heuristic a. [-1,sign]
{a}.
```

At the start of the search, the solver updates its heuristic knowledge about atom `a` assigning to it the `sign` value `1`. Then, it has to decide on `a`, making it either true or false. Following the current heuristic knowledge, the solver makes `a` true and returns the answer set {`a`}. ■

- The result would be the same if in the heuristic directive we used any positive integer instead of 1.

Heuristic modifier: **level**

```
#heuristic a. [ 1, sign]
#heuristic b. [ 1, sign]
#heuristic a. [10,level]
{a;b}.
:- a, b.
```

The `Domain` heuristic assigns to each atom a `level`, and it decides first upon atoms of the highest level. The default value for each atom is `0`, and both positive and negative integers are valid.

Dynamic heuristic modifications

Example 10.4. In the next program, the heuristic directive for `c` depends on `b`:

```
#heuristic a. [ 1, sign]
#heuristic b. [ 1, sign]
#heuristic a. [10,level]
{a;b}.
:- a, b.
{c}.
#heuristic c :     b. [ 1,sign]
#heuristic c : not b. [-1,sign]
```

Priorities among heuristic modifications

The `Domain` heuristic allows for representing priorities between different heuristic directives that refer to the same atom. The priority is optionally represented by a positive integer p in '@p'. The higher the integer, the higher the priority of the heuristic atom. For example, the following are valid heuristic directives:

```
#heuristic c. [ 1@10,sign]
#heuristic c. [-1@20,sign]
```

With both, the `sign` assigned to `c` is -1 (because priority `20` overrules `10`).

Remark 10.3. If the priority is omitted then it defaults to 0, as with the priorities of weak constraints (Section 3.1.13). For example, with these directives

```
#heuristic c. [ 10,level]
#heuristic c. [5@2,level]
```

the level of `c` is `5` because `2` $>$ `0`.

Heuristic modifier: **true** and **false**

The modifiers `true` and `false` allow us to refer at the same time to the `level` and the `sign` of an atom. Internally, a heuristic directive with the form

#heuristic A : B. [w@p,true]

#heuristic A : B. [w@p,level]
#heuristic A : B. [1@p,sign]

#heuristic A : B. [w@p,false]

#heuristic A : B. [w@p,level]
#heuristic A : B. [-1@p,sign]

Heuristic modifier: **init** and **factor**

The modifiers `init` and `factor` allow for modifying the scores assigned to atoms by the underlying `Vsids` heuristic. Unlike the `level` modifier, `init` and `factor` allow us to bias the search without establishing a strict ranking among the atoms.

With `init`, we can add a value to the initial heuristic score of an atom. For example, with

```
#heuristic a. [2,init]
```

a value of `2` is added to the initial score that the heuristic assigns to atom `a`. Note that as the search proceeds, the initial score of an atom decays, so `init` only affects the beginning of the search.

To bias the whole search, we can use the `factor` modifier that multiplies the heuristic score of an atom by a given value. For example, with

```
#heuristic a. [2,factor]
```

the heuristic score for atom `a` is multiplied by `2`.

Heuristics for BlocksWorld Planning

```
time(1..lasttime).
location(table).
location(X) :- block(X).
holds(F,0) :- init(F).

% Generate
{ move(X,Y,T) : block(X), location(Y), X != Y } = 1
                                         :- time(T).
% Test
:- move(X,Y,T), holds(on(A,X),T-1).
:- move(X,Y,T), holds(on(B,Y),T-1), B != X, Y != table.
% Define
moved(X,T) :- move(X,Y,T).
holds(on(X,Y),T) :- move(X,Y,T).
holds(on(X,Z),T) :- holds(on(X,Z),T-1), not moved(X,T).

% Test
:- goal(F), not holds(F,lasttime).

% Display
#show move/3.
```

Heuristic Strategies:

1. Prioritize decisions on move/3

#heuristic move(B,L,T) : block(B),location(L),time(T). [1,level]

Force the solver to decide on move/3 atoms first

```
Answer: 1 (Time: 0.003s)
move(b2,table,1) move(b1,b0,2) move(b2,b1,3)
SATISFIABLE

Models       : 1
Calls        : 4
Time         : 0.003s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.016s
```

```
Answer: 1 (Time: 0.002s)
move(b2,table,1) move(b1,b0,2) move(b2,b1,3)
SATISFIABLE

Models       : 1
Calls        : 4
Time         : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

Heuristics for BlocksWorld Planning

```
time(1..lasttime).
location(table).
location(X) :- block(X).
holds(F,0) :- init(F).

% Generate
{ move(X,Y,T) : block(X), location(Y), X != Y } = 1
                                             :- time(T).
% Test
:- move(X,Y,T), holds(on(A,X),T-1).
:- move(X,Y,T), holds(on(B,Y),T-1), B != X, Y != table.
% Define
moved(X,T) :- move(X,Y,T).
holds(on(X,Y),T) :- move(X,Y,T).
holds(on(X,Z),T) :- holds(on(X,Z),T-1), not moved(X,T).

% Test
:- goal(F), not holds(F,lasttime).

% Display
#show move/3.
```

Heuristic Strategies:

2. Soft bias towards move/3:

#heuristic move(B,L,T) : ... [2,init]
#heuristic move(B,L,T) : ... [2,factor]

Increases their score (initially or cumulatively) without strict priority.

Heuristics for BlocksWorld Planning

Learning And Mining from DatA

```
time(1..lasttime).
location(table).
location(X) :- block(X).
holds(F,0) :- init(F).

% Generate
{ move(X,Y,T) : block(X), location(Y), X != Y } = 1
                                          :- time(T).
% Test
:- move(X,Y,T), holds(on(A,X),T-1).
:- move(X,Y,T), holds(on(B,Y),T-1), B != X, Y != table.
% Define
moved(X,T) :- move(X,Y,T).
holds(on(X,Y),T) :- move(X,Y,T).
holds(on(X,Z),T) :- holds(on(X,Z),T-1), not moved(X,T).

% Test
:- goal(F), not holds(F,lasttime).

% Display
#show move/3.
```

Heuristic Strategies:

3. Prefer move/3 to be true:

#heuristic move(B,L,T) : ... [1,true]
#heuristic move(B,L,T) : ... [1,sign]

Encourages one move/3 per time step to be selected quickly.

Heuristics for BlocksWorld Planning

```
time(1..lasttime).
location(table).
location(X) :- block(X).
holds(F,0) :- init(F).

% Generate
{ move(X,Y,T) : block(X), location(Y), X != Y } = 1
                                       :- time(T).
% Test
:- move(X,Y,T), holds(on(A,X),T-1).
:- move(X,Y,T), holds(on(B,Y),T-1), B != X, Y != table.
% Define
moved(X,T) :- move(X,Y,T).
holds(on(X,Y),T) :- move(X,Y,T).
holds(on(X,Z),T) :- holds(on(X,Z),T-1), not moved(X,T).

% Test
:- goal(F), not holds(F,lasttime).

% Display
#show move/3.
```

Search Direction Control:

1. Forward Search:

#heuristic move(B,L,T) : ... [lasttime - T + 1, true]

Gives higher priority to earlier moves

Heuristics for BlocksWorld Planning

```
time(1..lasttime).
location(table).
location(X) :- block(X).
holds(F,0) :- init(F).

% Generate
{ move(X,Y,T) : block(X), location(Y), X != Y } = 1
                                            :- time(T).
% Test
:- move(X,Y,T), holds(on(A,X),T-1).
:- move(X,Y,T), holds(on(B,Y),T-1), B != X, Y != table.
% Define
moved(X,T) :- move(X,Y,T).
holds(on(X,Y),T) :- move(X,Y,T).
holds(on(X,Z),T) :- holds(on(X,Z),T-1), not moved(X,T).

% Test
:- goal(F), not holds(F,lasttime).

% Display
#show move/3.
```

Search Direction Control:

2. Backward Search from Goals:

#heuristic move(B,L,T) : holds(on(B,L),T). [true, T]

Prefers actions that directly support current goals (holds/2)

Heuristics for BlocksWorld Planning

```
time(1..lasttime).
location(table).
location(X) :- block(X).
holds(F,0) :- init(F).

% Generate
{ move(X,Y,T) : block(X), location(Y), X != Y } = 1
                                             :- time(T).
% Test
:- move(X,Y,T), holds(on(A,X),T-1).
:- move(X,Y,T), holds(on(B,Y),T-1), B != X, Y != table.
% Define
moved(X,T) :- move(X,Y,T).
holds(on(X,Y),T) :- move(X,Y,T).
holds(on(X,Z),T) :- holds(on(X,Z),T-1), not moved(X,T).

% Test
:- goal(F), not holds(F,lasttime).

% Display
#show move/3.
```

Search Direction Control:

3. Promote State Persistence (Inertia):

#heuristic holds(on(B,L),T-1) : holds(on(B,L),T). [lasttime-T+1,true]

Encourages state continuity backward in time.

Command Line Structure-oriented Heuristics

The `Domain` heuristic also allows us to modify the heuristic of the solver from the command line. For this, it is also activated with option `--heuristic=Domain`, but now the heuristic modifications are specified by option:

`--dom-mod=<mod>[,<pick>]`

where `<mod>` ranges from `0` to `5` and specifies the modifier:

`<mod>`	Modifier	`<mod>`	Modifier
0	None	1	`level`
2	`sign` (positive)	3	`true`
4	`sign` (negative)	5	`false`

`<pick>` specifies bit-wisely the atoms to which the modification is applied:

0	Atoms only
1	Atoms that belong to strongly connected components
2	Atoms that belong to head cycle components
4	Atoms that appear in disjunctions
8	Atoms that appear in optimization statements
16	Atoms that are shown

Computing Subset Minimal Answer Sets with Heuristics

Find answer sets where the set of selected atoms is subset-minimal

1. Apply heuristic to atoms you want to minimize

```
#heuristic a(1..3). [1,false]
```

2. command line

```
--heuristic=Domain --dom-mod=5,16     % apply 'false' to atoms shown by #show
```

Optimization and Preference Handling

Multi-objective Optimization using max(min)imize

```
{ hotel(1..5) } = 1.
star(1,5). cost(1,170).
star(2,4). cost(2,140).
star(3,3). cost(3,90).
star(4,3). cost(4,75).  main_street(4).
star(5,2). cost(5,60).
noisy :- hotel(X), main_street(X).
#maximize { Y@1,X : hotel(X), star(X,Y) }.
#minimize { Y/Z@2,X : hotel(X), cost(X,Y), star(X,Z) }.
:~ noisy. [ 1@3 ]
```

Preference Handling with *asprin*

The system *asprin* provides a general framework for optimizing qualitative and quantitative preferences in ASP. It allows for computing optimal answer sets of logic programs with preferences. While *asprin* comes with a library of predefined preference types (`subset`, `pareto`, etc.), it is readily extensible by new customized preference types. For a formal description of *asprin*, please consult [11].

The following description conforms with *asprin* 3.1, which uses *clingo* 5.

https://github.com/potassco/asprin

Brewka, Gerhard, et al. "asprin: Customizing answer set preferences without a headache." Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 29. No. 1. 2015.

Input language of asprin

A *preference statement* is of the form

$$\texttt{\#preference(s,t)}\{e_1\texttt{;}\ldots\texttt{;}e_n\}\ \texttt{:}\ B\texttt{.}$$

where `s` is a term giving the preference name, `t` is a term providing the preference type, and each e_j is a preference element. The rule body B has the same form and purpose as above. That is, the body B of a preference statement is used to instantiate the variables of s, t and each e_i. For safety, all variables appearing in `s` and `t` must also appear in a positive literal in B.

Preference statements are accompanied by *optimization directives* such as

$$\texttt{\#optimize(s)}\ \texttt{:}\ B\texttt{.}$$

where B is as above, telling *asprin* to restrict its reasoning mode to the preference relation declared by `s`.

Input language of asprin

```
dom(1..3).

{ m(1..3) } = 1.

a(1)    :- m(1).  a(1..2) :- m(2).  a(3)    :- m(3).
b(1..3) :- m(1).  b(1)    :- m(2).  b(2..3) :- m(3).

#show m/1. #show a/1. #show b/1.
```

```
clingo version 5.4.0
Reading from base.lp
Solving...
Answer: 1
b(2) m(3) b(3) a(3)
Answer: 2
b(1) m(2) a(1) a(2)
Answer: 3
b(2) b(3) b(1) m(1) a(1)
SATISFIABLE

Models       : 3
Calls        : 1
Time         : 0.005s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

```
1 #preference(p1,subset){ a(X) : dom(X) }.
2 #optimize(p1).
```

Line 1 contains a preference statement of name p1 and type subset that contains a single (non-ground) preference element. Intuitively, the preference statement p1 defines a preference of type subset over atoms of predicate a/1. Line 2 contains an optimization directive that instructs *asprin* to compute answer sets that are optimal with respect to p1.

```
asprin version 3.1.1
Reading from base.lp ...
Solving...
Answer: 1
m(3) a(3) b(2) b(3)
OPTIMUM FOUND
Answer: 2
m(2) a(1) a(2) b(1)
Answer: 3
m(1) a(1) b(2) b(3) b(1)
OPTIMUM FOUND

Models       : 3
  Optimum    : yes
  Optimal    : 2
Calls        : 8
Time         : 0.074s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.078s
```

Input language of asprin

Example 11.4. Consider a preference specification about leisure activities (without base program).

```
#preference(costs,less(weight)){
  C :: sauna : cost(sauna,C);
  C ::  dive : cost(dive,C)
}.
#preference(fun,superset){ sauna; dive; hike; not bunji }.
#preference(temps,aso){
  dive >> sauna ||     hot;
 sauna >> dive   || not hot
}.
#preference(all,pareto){**costs; **fun; **temps}.

#optimize(all).
```

- 类型：`less(weight)` → 越“便宜”越好；
- 格式：`C :: atom : condition`
    - 例如：如果 `cost(sauna, 5)`，那选 `sauna` 会产生成本 5；
- 目标是：最小化所选活动的总成本，类似于 `#minimize` 的行为。

- 类型：`superset`
- 偏好定义了：我们更喜欢包含这些活动的解；
- 解中包含越多（或不包含 `bunji`），就越优。

- 类型：`aso`（Answer Set Optimization）；
- 定义了一种条件优先关系（`>>`）：
    - 如果天气是热的（`hot`），则 `dive` 优于 `sauna`；
    - 如果天气不热，则 `sauna` 优于 `dive`；
- 这是一种上下文敏感的偏好表达方式，非常强大。

- 类型：`pareto`（帕累托最优）；
- `**` 表示引用之前定义的其他偏好（不是原子，是 preference 名称）；
- 意思是：一个答案集是 Pareto 最优的，如果没有其他解在所有三个偏好上都更优。

Preference relations and preference types

```
#preference(p1,subset){ a(X) : dom(X) }.
#optimize(p1).
```

Example 11.5. The preference statement of Example 11.1 stands for the following ground preference statement:

```
#preference(p1,subset){ a(1); a(2); a(3) }.
```

It declares the following preference relation:

$$
\begin{aligned}
X \succ_{\mathtt{p1}} Y \quad \text{iff} \quad & \{e \in \{\mathtt{a(1)},\mathtt{a(2)},\mathtt{a(3)}\} \mid X \models e\} \\
\subset \quad & \{e \in \{\mathtt{a(1)},\mathtt{a(2)},\mathtt{a(3)}\} \mid Y \models e\}
\end{aligned}
$$

In Example 11.1, we get $X_1 \succ_{\mathtt{p1}} X_2$ because $\{\mathtt{a(1)}\} \subset \{\mathtt{a(1)},\mathtt{a(2)},\mathtt{a(3)}\}$ and $X_3 \succ_{\mathtt{p1}} X_2$ given that $\{\mathtt{a(2)},\mathtt{a(3)}\} \subset \{\mathtt{a(1)},\mathtt{a(2)},\mathtt{a(3)}\}$; however, we have $X_1 \not\succ_{\mathtt{p1}} X_3$ since $\{\mathtt{a(1)}\} \not\subset \{\mathtt{a(2)},\mathtt{a(3)}\}$. ■

Example 11.6. In Example 11.1, the preference statement `p1` is admissible because `a(1)`, `a(2)`, and `a(3)` are Boolean formulas and thus belong to the domain of `subset`. If we added the preference elements `1::a(1)` or `**p2`, the statement would not be admissible any more. ■

asprin library

The preference library of *asprin* implements the following basic preference types:

- `subset` and `superset`
- `less(cardinality)` and `more(cardinality)`
- `less(weight)` and `more(weight)`
- `minmax` and `maxmin`
- `aso` (Answer Set Optimization, [13])
- `poset` (Qualitative Preferences, [18])
- `cp` (CP nets, [10])

- 基于集合包含关系的偏好
- 基于集合大小的偏好
- 基于加权求和的偏好
- 多目标中最坏（好）值的最小化（最大化）
- Answer Set Optimization
- 类似于aso，但引入了严格的偏序结构
- 条件性偏好网络，相较于aso增强了条件功能

The library of *asprin* implements furthermore the following composite preference types, which amount to the ones defined in [79]:

- `neg`
- `and`
- `pareto`
- `lexico`

Example 11.12. Consider the following preference specification, where `p1` and `p` are defined as before:

```
#preference(p1,subset){                a(X)  : dom(X)  }.
#preference(p3,less(cardinality)){ b(X)  : dom(X)  }.
#preference(p8,neg){                   **p1            }. % 反转 p1 的偏好
#preference(p9,and){                   **p1;     **p3  }. % p1 与 p3 都优于，才算优于
#preference(p10,pareto){               **p1;     **p3  }. % p1 或 p3 提升，但不在其他方面更差
#preference(p11,lexico){            1::**p1;  2::**p3  }. % p1 优先级高于 p3 的字典序比较
```

Implementing preference types

Example 11.13. Recall the preference statement `p1` of Example 11.1:

```
#preference(p1,subset){ a(X) : dom(X) }.
```

This is translated into:[33]

```
1  preference(p1,subset).
2  preference(p1,(1,1,(X)),1,for(atom(a(X))),()) :- dom(X).
```

Line 1 states the name and the type of the preference statement. Line 2 can be read as follows: the preference statement `p1`, appearing as the first preference statement of the program, in the first element has variables {X}, and in the first position of the element there is a Boolean formula `a(X)` that has an empty list of associated weights.

Implementing preference types

A preference program implementing a preference type `t` compares two answer sets X and Y given a preference statement `s` of type `t`. To allow for this comparison, *asprin* provides for every term `for(`t_F`)` appearing in the translation of `s` the fact `holds(`t_F`)` whenever X satisfies the Boolean formula F. Analogously, *asprin* provides the fact `holds'(`t_F`)`, if Y satisfies F.

Example 11.17. The preference type `subset` can be implemented as follows (see file `subset.lp`).

```
#program preference(subset).
better(S) :- preference(S,subset),
  not holds(A),       holds'(A), preference(S,_,_,for(A),_),
  not holds(B) : not holds'(B), preference(S,_,_,for(B),_).
```

In addition, the following integrity constraint enforces the optimization with respect to a given optimization directive: (included in file `basic.lp`):

```
#program preference.
:- not better(P), optimize(P).
```

Instead of using *asprin*'s library, viz. `asprin_lib.lp`, we can now directly use the above preference program as follows:

```
asprin --no-asprin-lib \
       base.lp preference1.lp subset.lp basic.lp 0
```

Thank You

Question?
Suggest?