Second Answer Set Programming Modeling Competition

28 September 2015, Lexington, Kentucky, USA

A team consists of one to three members, using one computer. The number of correct submissions (the last one submitted for a particular problem) within 120 minutes is used as the first ranking criterion. Incorrect submissions cost nothing. For each correct submission, the time it took to submit it (starting from some arbitrary time point) is added up: this total is used as a tie breaker in case teams end up with the same number of correct solutions. If that doesn't break all ties, the organizers can use any other criterion like performance, space usage, coin tossing, etc. Note that it is advantageous to submit solutions one by one, as soon as it is deemed correct, rather than waiting to submit a bunch of solutions at once. Efficiency of your programs is not important, but if your program fails to finish in a reasonable time, it is considered incorrect.

The name of the file that contains your submitted solution must be the same as given in parentheses in the header of a task description: so, for the first problem, you make a file named `match.asp`. Programs must be written in ASP-Core-2 (version 2.03b[1]), pragmatically taken to be processible with *clingo* (version 4.5.3[2]). The input to your program is always in the form of facts; see examples in task descriptions. On any input, your program must specify (optimal) solutions, determined by the output predicates given in task descriptions. Arbitrary auxiliary predicates can be defined in addition but will be ignored in correctness evaluation.

Keep in mind:

Correctness matters, submission time matters, task order doesn't matter. Good luck!

[1] `https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03b.pdf`

[2] `http://sourceforge.net/projects/potassco/files/clingo/4.5.3/`

1 Revenge of the List (`match.asp`)

Given two lists, say $[a, b, c, b, d]$ and $[b]$, where the second one provides a pattern, we are interested in all sublists of the first list starting with that pattern. Prolog experts may do the following:

```
?- append(_,L,[a,b,c,b,d]), append([b],_,L).
L = [b,c,b,d] ;
L = [b,d] ;
false.
```

However, this is about Answer Set Programming, and we represent a list and a pattern as above in terms of facts like the following:

```
list(f(a,f(b,f(c,f(b,f(d,nil)))))). % [a,b,c,b,d]
find(f(b,nil)).                     % [b]
```

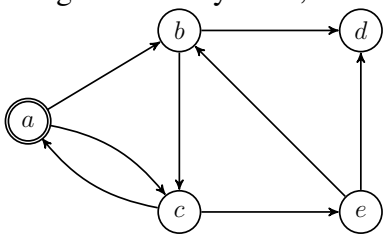
All sublists starting with the pattern shall be provided by atoms over the predicate `match/1` in an answer set. Along with the instruction '`#show match/1.`', a correct solution in file `match.asp` yields the following *clingo* behavior:

```
$ clingo instance01.asp match.asp
clingo version 4.5.3
Reading from instance01.asp ...
Solving...
Answer: 1
match(f(b,f(c,f(b,f(d,nil))))) match(f(b,f(d,nil)))
SATISFIABLE
```

You can assume that there are one non-empty list and one non-empty pattern (both longer than `nil`) per instance, while there is no length limitation. In particular, your solution must be able to cope with patterns consisting of two or more elements.

2 Dominators Reign (`dominator.asp`)

Consider a directed graph with a distinguished entry node, such as a below:



A node n is a dominator of another node n', if all paths from the entry node to n' go through n.[3] Basically by definition (when there are at least two nodes), the entry node dominates all other nodes. Moreover, c dominates e above as all paths from the entry node a to e go through c. One can check that no further node dominates another, so that a and c are the dominators of interest.

Put differently, a node n does not dominate another node n', if there is a path from the entry node to n' that does not include n. So one can cast identifying dominators to checking which nodes are not necessarily on a path to n'. For the entry node a above, it is clear that no other node is necessary. Propagating this information to its direct successors b and c, we find that no other node but a is necessary. Continuing further from b to d and from c to e, we see that c and e are unnecessary for d as well as that b and d are unnecessary for e. Finally, we can proceed from e to d to check that b is unnecessary for it either. In summary, we have identified the following sets of unnecessary nodes: $a : \{b, c, d, e\}, b : \{c, d, e\}, c : \{b, d, e\}, d : \{b, c, e\}, e : \{b, d\}$. To read off the dominators, it is now sufficient to check which nodes have some direct successor where they are not unnecessary, in turn saying that they dominate. This applies to the entry node a and its successors b and c as well as the successor e of c, thus reproducing the dominators identified above. OK, this lengthy explanation sketches an idea to potentially encode in ASP, but stop here.

An instance like the directed graph above consists of facts as follows:

```
entry(a). edge(a,b). edge(a,c). edge(b,c). edge(b,d).
          edge(c,a). edge(c,e). edge(e,b). edge(e,d).
```

All dominators in the given graph shall be provided by atoms over the predicate `dominator/1` in an answer set. Along with the instruction '`#show dominator/1.`', a correct solution in file `dominator.asp` yields the following *clingo* behavior:

```
$ clingo instance01.asp dominator.asp
clingo version 4.5.3
Reading from instance01.asp ...
Solving...
Answer: 1
dominator(a) dominator(c)
SATISFIABLE
```

You can assume that no instance includes self-loops for any node, while all nodes are reachable from the distinguished entry node.

[3] Also called strict dominator: `https://en.wikipedia.org/wiki/Dominator_(graph_theory)`

3 Almost Unsatisfied (`true.asp`)

The well-known Boolean Satisfiability problem applies to clauses like this:

$$a \vee \neg b \vee c, \neg a \vee c \vee \neg d, b \vee d, \neg c \vee \neg d$$

These clauses have four models, expressed by the true propositions: $\{a, b\}, \{a, b, c\}, \{b, c\}, \{d\}$. Observe that the first three have correspondents differing by one proposition only, which can be toggled to move from one to the other model. Unlike that, if we turn d to false or either of a, b, and c to true, the fourth model $\{d\}$ yields an assignment not satisfying the clauses anymore. Hence, we call $\{d\}$ an almost unsatisfied model, in which we are interested in the following.

Clauses like the above are given by an instance as follows:

```
clause(1). literal(1,pos,a). literal(1,neg,b). literal(1,pos,c).
clause(2). literal(2,neg,a). literal(2,pos,c). literal(2,neg,d).
clause(3). literal(3,pos,b). literal(3,pos,d).
clause(4). literal(4,neg,c). literal(4,neg,d).
```

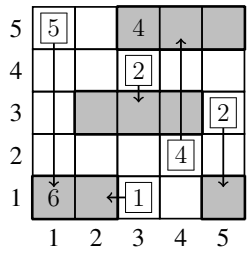
Note that the constants `pos` or `neg` are used to indicate whether a proposition occurs positively or negatively in a clause. Moreover, some (not necessarily unique) almost unsatisfied model shall be provided by atoms over the predicate `true/1` in an answer set. Along with the instruction '`#show true/1.`', a correct solution in file `true.asp` yields the following *clingo* behavior:

```
$ clingo instance01.asp true.asp
clingo version 4.5.3
Reading from instance01.asp ...
Solving...
Answer: 1
true(d)
SATISFIABLE
```

You can assume that no instance includes a tautological clause, i.e., a fact `literal(`c`,pos,`x`)` is never accompanied by `literal(`c`,neg,`x`)`, where c and x denote the same clause or proposition, respectively.

4 Yosenabe (`target.asp`)

A good modeling competition features some Japanese grid puzzle. Luckily there is Yosenabe[4]:



Given a grid as above,[5] initially without the arrows, the task is to move each number surrounded by a frame into one of the gray areas along a straight line, respecting the following conditions:

1. The ways of any two moved numbers must not cross or meet at any grid cell.
2. Each gray area must be populated with at least one moved number.
3. An area may be associated with a (positive) goal number, shown within it. If so, the numbers moved into the area must sum up exactly to the goal.

The (unique) solution indicated by arrows above fulfills these conditions. While a number can be moved through an area, such as 4 above, you may assume that a move stops at the first cell w.r.t. its direction of the area into which it leads. That is, regardless of number 5, it cannot help to move 1 to the cell in the lower left corner as the move enters the same area already before.

An instance like the grid above consists of facts as follows:

```
cell(1,1). cell(1,2).  ...  cell(5,4). cell(5,5). number(1,5,5).
area(1,1,1). area(2,1,1).              goal(1,6). number(3,1,1).
area(2,3,2). area(3,3,2). area(4,3,2).            number(3,4,2).
area(3,5,3). area(4,5,3). area(5,5,3). goal(3,4). number(4,2,4).
area(5,1,4).                                      number(5,3,2).
```

Note that the first two arguments in facts over `area/3` and `number/3` provide grid cells, and the third an area identifier or a number to move, respectively. If an area is associated with a goal number, its identifier is reused as first argument in a fact over `goal/2`, and the goal is given by the second. Then, the moves in a solution shall be provided by atoms over the predicate `target/4` in an answer set, expressing a move in terms of coordinates of the initial cell as well as the cell to which a number is moved. Along with the instruction '`#show target/4.`', a correct solution in file `target.asp` yields the following *clingo* behavior:

```
$ clingo instance01.asp target.asp
clingo version 4.5.3
Reading from instance01.asp ...
Solving...
Answer: 1
target(1,5,1,1) target(3,1,2,1) target(3,4,3,3) \
target(4,2,4,5) target(5,3,5,1)
SATISFIABLE
```

[4] `http://www.nikoli.co.jp/en/puzzles/yosenabe.html`

[5] Some Yosenabe puzzles to practice can be found here: `http://www.janko.at/Raetsel/Yosenabe/`

5 The Bishop and the Rook (`place.asp`)

In the final task, we consider the construction of an optimal placement of bishops and rooks, attacking one another according to the rules of chess, while admitting rectangular chessboards:

Given an empty board, bishops and rooks can be placed on it and attack in diagonal or straight directions, respectively, all following cells up to next chess piece (inclusive) or the end of the board, taking into account four conditions:

1. Two bishops must not be placed on adjacent cells (also diagonally), and likewise for rooks.
2. Two bishops must not attack each other, and likewise for rooks.
3. Each cell on which a bishop is placed, must be attacked by some rook, and vice versa.
4. Each cell of the board must be attacked by some chess piece.

The placements shown above fulfill these conditions. In particular, neither bishops nor rooks are placed adjacent to a chess piece of the same kind, while a bishop adjacent to a rook is alright. On the left, a bishop in between even keeps two rooks off from inadmissibly attacking each other.

We are interested in minimizing the total value of chess pieces placed on a board. The value of a bishop is 2, and a rook has the value 3. Thus, the four bishops and three rooks on the left sum up to a total value of 17, while the three bishops and two rooks on the right amount to 12 only. Indeed, the placement on the right is one among four optimal placements with total value 12.

A rectangular board is simply represented by facts providing its cells:

```
cell(1,1). cell(1,2). ...  cell(3,3). cell(3,4).
```

The placement of bishops and rooks shall be provided by atoms over the predicate `place/3` in an answer set, including the constant `bishop` or `rook` and cell coordinates as arguments. Along with the instruction '`#show place/3.`', a correct solution in file `place.asp` yields an optimal placement in a last answer set obtained like this:

```
$ clingo instance01.asp place.asp
clingo version 4.5.3
Reading from instance01.asp ...
Solving...
Answer: 1
place(bishop,1,2) place(bishop,1,4) place(bishop,3,1) \
place(bishop,3,3) place(rook,1,1) place(rook,1,3) place(rook,3,4)
Optimization: 17
Answer: 2
place(bishop,1,1) place(bishop,3,2) place(bishop,3,4) \
place(rook,2,1) place(rook,3,3)
Optimization: 12
OPTIMUM FOUND
```