8. Answer Set Programming with MVPF

Answer Set Programming

Beyond the study of proofs in mathematics, logic has been applied to designing and reasoning about computer hardware and software. One of the prominent applications of logic in computer science is the use of logic as a programming language. *Logic programming* specifies *what* is to be computed, but not necessarily *how* it is computed.

> A logic program is a set of axioms, or rules, defining relations between objects. A computation of a logic program is a deduction of consequences of the program. A program defines a set of consequences, which is its meaning. The art of logic programming is constructing concise and elegant programs that have the desired meaning. (From *The Art of Prolog*, page 9.)

In this handout, we will study "Answer Set Programming (ASP)," a form of declarative programming oriented towards difficult combinatorial search problems. It has been applied, for instance, to knowledge representation, planning, and product configuration problems in artificial intelligence and to graph-theoretic problems arising in VLSI design, historical linguistics, and bioinformatics.

The idea of ASP is to represent the search problem we are interested in as the problem of finding the stable models (a.k.a. answer sets) of a formula, and then find the solutions using *answer set solvers*, such as the systems SMODELS, CLINGO, DLV, ASSAT, CMODELS, and CLASP.

System `mvsm`

System `mvsm` is an implementation of multi-valued propositional formulas under the stable model semantics. It is in fact a script that invokes several software, such as MVPF2LPCOMPILER, F2LP, GRINGO, CLASPD, and AS2TRANSITION. For this class, you'll be using the one that is already installed in a server maintained by the Automated Reasoning Group. You can log into your account in the server by typing

```
ssh -l [asurit-id] reasoning-server.fulton.ad.asu.edu
```

and type in your asurite password.

Multi-valued propositional formulas can be encoded in the language of MVSM using the following ASCII characters.

Symbol	$\neg$	$\wedge$	$\vee$	$\bot$	$\top$	$\rightarrow$	$\leftarrow$
ASCII	`-`	`&`	`|`	`false`	`true`	`->`	`<-`

A very simple example. The example program in Problem 6.6 can be encoded in the language of MVSM as follows:

```
1  % File 'ex1'
2
3  :- sorts
4    num.
5
6  :- objects
7    1..3      :: num.
8
9  :- constants
10   c         :: num.
11
12 {c=1}.
13 {c=2}.
```

Any line starting with `%` is a comment. Lines 3–4 declare a domain named `num`. The domain consists of three numbers as declared in lines 6–7. Lines 9–10 declare a constant whose domain is `num`. Lines 12–13 are default formulas, which can be also written as

```
c=1 | -(c=1).
c=2 | -(c=2).
```

Let's call the file `ex1`. We invoke MVSM as follows:

```
$ mvsm ex1 0
```

The zero at the end indicates that we want to compute all stable models; a positive number k would tell MVSM to terminate after computing k stable models. The default value of k is 1.

For the command, the MVSM output is as follows.

```
claspD version 1.1.1. Reading...done
Solution: 1

c=2

Solution: 2

c=1

Models       : 2
Time         : 0.000  (Parsing: 0.000)
```

Schematic variables. A group of rules that follow a pattern can be often described concisely in the input language of MVSM using schematic variables, which must be capitalized.

```
1  % File 'ex2'
2
3  #const max=3.
4
5  :- sorts
6    num.
7
8  :- objects
9    1..max          :: num.
10
11 :- variables
12   X,X1,Y          :: num.
13
14 :- constants
15   c(num)          :: num.
16
17 {c(X)=Y}.
18 <- c(X)=Y & c(X1)=Y & X!=X1.
```

Line 3 declares a symbolic name for a specific value. Lines 11–12 introduce three variables of sort NUM. Line 17 is a schematic formula where `X` and `Y` range over all values of NUM; it expresses that `c` is a function that maps NUM to NUM. The last line expresses that this function is 1–1.

8.1^e How many stable models are there?

Model checking with propositional logic. Recall that propositional formulas are a special case of multi-valued formulas. According to Problem 6.7, we can compute models of propositional logic using MVSM. The following program represents the train example we discussed earlier. `boolean` is a built-in sort containing `true` and `false` as the elements.

```
1  % File 'train'
2
3  :- constants
4    tl, taxi, jl      :: boolean.
5
6  :- variables
7    B                 :: boolean.
8
9  {tl=B}.
10 {taxi=B}.
11 {jl=B}.
12
13 tl & -taxi  -> jl.
14 -jl.
15 tl.
```

To check whether lines 13–15 entail `taxi` under propositional logic, we compute all stable models of lines 9–15.

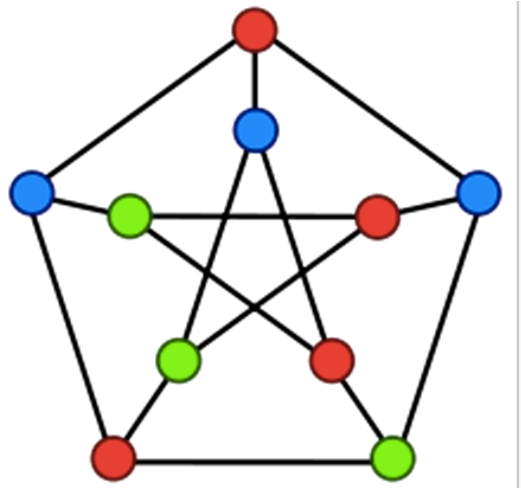
```
$ mvsm train 0
claspD version 1.1.1. Reading...done
Solution: 1

taxi
tl

Models       : 1
Time         : 0.000  (Parsing: 0.000)
```

System MVSM, by default, does not show Boolean constants that are mapped to false. So this output means that there is only one model of propositional formulas in lines 13–15 where `taxi` and `tl` are mapped true and `jl` is mapped to false.

Another way to check the entailment is to add `-taxi` at the end of the file, and check whether the program has no stable models in accordance with Problem 3.17.

Reasoning about States

Example: Graph Coloring. An *n-coloring* of a graph G is a function f from its set of vertices to $\{1, \ldots, n\}$ such that $f(x) \neq f(y)$ for every pair of adjacent vertices x, y. The stable models of the following program are in a 1–1 correspondence with the n-colorings of G.



A proper vertex coloring of the Petersen graph with 3 colors, the minimum number possible.

The following file `color` is an MVSM encoding of the n coloring problem.

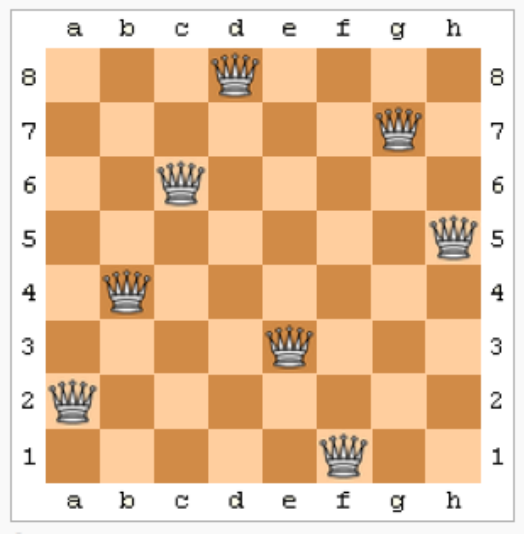
```
% File 'color'
% Find a 3-coloring of the given graph

:- sorts
  node;
  color.

:- objects
  0..4                  :: node;
  r,g,b                 :: color.

:- constants
  col(node)             :: color.

:- variables
```

```
  X,Y                    :: node;
  C                      :: color.

:- constants
  e(node,node)           :: boolean;
  col(node)              :: color.

e(0,1).  e(1,2).  e(2,3).  e(3,4).  e(4,0).  e(0,2).

{col(X)=C}.
{e(X,Y)=false}.

<- col(X)=C & col(Y)=C & e(X,Y).
```

One can find all 3-colorings using the following command:

```
$ mvsm color 0
```

N Queens. The goal is to place n queens on an $n \times n$ chessboard so that no two queens would be placed on the same row, column or diagonal. A solution can be described by a set of atoms of the form $q(i, j)$ $(1 \leq i, j \leq n)$; including $q(i, j)$ in the set indicates that there is a queen at position (i, j).



The following is a representation of 8-Queens puzzle in the input language of MVSM:

```
1  #const n=8.
2
3  :- sorts
4    num.
5
6  :- objects
7    1..n                :: num.
8
9  :- variables
10   I, I1, J, J1        :: num;
11   B                   :: boolean.
12
13 :- constants
14   non_empty_row(num) :: boolean;
15   q(num,num)          :: boolean.
16
17 {q(I,J)=B}.
18 <- q(I,J) & q(I1,J) & I!=I1.
19 <- q(I,J) & q(I,J1) & J!=J1.
20 <- q(I,J) & q(I1,J1) & I!=I1 & #abs(I1-I)==J1-J.
21 non_empty_row(I) <- q(I,J).
22 <- not non_empty_row(I).
```

Line 17 expresses that q is a function that maps each position to $\{\mathbf{t}, \mathbf{f}\}$. Lines 18–20 express that no two queens are on the same column, row, and diagonal. Lines 21–22 express that every row has at least one queen placed.

Reasoning about Dynamic Systems

Simple Transition System. The simple transition system in Handout 7 can be represented in the input language of MVSM as follows.

```
1  % File 'sd'
2
3  :- sorts
4    step;
5    astep.
6
7  :- objects
8    0..maxstep     :: step;
9    0..maxstep-1   :: astep.
10
```

```
11 :- variables
12   B              :: boolean;
13   ST             :: step;
14   T              :: astep.
15
16 :- constants
17   p(step)        :: boolean;
18   a(astep)       :: boolean.
19
20 % direct effect
21 p(T+1) <- a(T).
22
23 % initial states are exogenous
24 {p(0)=B}.
25
26 % actions are exogenous
27 {a(T)=B}.
28
29 % p is inertial
30 {p(T+1)=B} <- p(T)=B.
```

Lines 3–5 declare two sorts required for describing transition systems, and lines 7–9 introduce the elements that belong to the each sort. Lines 11–14 declare schematic variables for each sort. Line 17 declares boolean constants `p(0), p(1), ..., p(maxstep)` and line 18 declares boolean constants `a(0), a(1), ..., a(maxstep-1)`. Lines 20–30 represent formulas (5) in Handout 7.

Let's call this file `sd`. One can compute all states of the transition system by the command:

```
mvsm sd 0 0
```

The first 0 instructs the system to find all states and the second 0 sets `maxstep` to 0.[1]

Similarly, the following command instructs the system to find all transitions.

```
$ mvsm sd 0 1
claspD version 1.1.1. Reading...done
Solution: 1
```

[1] You cannot specify maxstep without specifying the number of stable models to be returned.

```
Solution: 2

a(0)
p(1)

Solution: 3

p(0)
p(1)

Solution: 4

a(0)
p(0)
p(1)

Models       : 4
Time         : 0.000  (Parsing: 0.000)
```

MB in the Language of ASP

The following file describes the monkey and bananas domain in the language of MVSM.

```
% File 'mb'

:- sorts
  step;
  astep;
  thing;
  location.


:- objects
  0..maxstep             :: step;
  0..maxstep-1           :: astep;
  monkey,bananas,box     :: thing;
  l1,l2,l3               :: location.

:- variables
  ST                     :: step;
  T                      :: astep;
  B                      :: boolean;
  Th                     :: thing;
```

```
21    L                       :: location.
22
23  :- constants
24    loc(thing,step)                 :: location;
25    hasBananas(step),onBox(step)    :: boolean;
26
27    walk(location,astep),
28    pushBox(location,astep),
29    climbOn(astep),
30    climbOff(astep),
31    graspBananas(astep)             :: boolean.
32
33  {loc(Th,T+1)=L} <- loc(Th,T)=L.
34  {hasBananas(T+1)=B} <- hasBananas(T)=B.
35  {onBox(T+1)=B} <- onBox(T)=B.
36
37  {walk(L,T)=B}.
38  {pushBox(L,T)=B}.
39  {climbOn(T)=B}.
40  {climbOff(T)=B}.
41  {graspBananas(T)=B}.
42
43  loc(bananas,ST)=L <- hasBananas(ST) & loc(monkey,ST)=L.
44  loc(monkey,ST)=L <- onBox(ST) & loc(box,ST)=L.
45
46  loc(monkey,T+1)=L <- walk(L,T).
47  <- walk(L,T) & loc(monkey,T)=L.
48  <- walk(L,T) & onBox(T).
49
50  loc(box,T+1)=L <- pushBox(L,T).
51  loc(monkey,T+1)=L <- pushBox(L,T).
52  <- pushBox(L,T) & loc(monkey,T)=L.
53  <- pushBox(L,T) & onBox(T).
54  <- pushBox(L,T) & loc(monkey,T)=L1 & loc(box,T)=L2 & L1 != L2.
55
56  onBox(T+1) <- climbOn(T).
57  <- climbOn(T) & onBox(T).
58  <- climbOn(T) & loc(monkey,T)=L1 & loc(box,T)=L2 & L1 != L2.
59
60  onBox(T+1) = false <- climbOff(T).
61  <- climbOff(T) & -onBox(T).
62
63  hasBananas(T+1) <- graspBananas(T).
64  <- graspBananas(T) & hasBananas(T).
65  <- graspBananas(T) & -onBox(T).
```

```
66
67 <- graspBananas(T) & loc(monkey,T)=L1 & loc(bananas,T)=L2 & L1 != L2.
68
69 <- walk(L,T) & pushBox(L,T).
70 <- walk(L,T) & climbOn(T).
71 <- pushBox(L,T) & climbOn(T).
72 <- climbOff(T) & graspBananas(T).
73
74 % initial states are exogenous
75 {loc(Th,0)=L}.
76 {onBox(0)=B}.
77 {hasBananas(0)=B}.
```

In order to solve the planning problem, one can append the following lines.

```
% planning query
--(loc(monkey,0)=l1 & loc(bananas,0)=l2 & loc(box,0)=l3
   & hasBananas(0)=false & onBox(0)=false).

--(hasBananas(maxstep)).
```

The following execution shows that the shortest step solution takes 4 steps.

```
$ mvsm mb 1 3
claspD version 1.1.1. Reading...done
Models       : 0
Time         : 0.000  (Parsing: 0.000)
No solution.

jlee89@reasoning-server:~/459-f13$ mvsm mb 1 4
claspD version 1.1.1. Reading...done
Solution: 1

climbOn(2)
graspBananas(3)
hasBananas(4)
loc(bananas,0)=l2
loc(bananas,1)=l2
loc(bananas,2)=l2
loc(bananas,3)=l2
loc(bananas,4)=l2
```

```
loc(box,0)=l3
loc(box,1)=l3
loc(box,2)=l2
loc(box,3)=l2
loc(box,4)=l2
loc(monkey,0)=l1
loc(monkey,1)=l3
loc(monkey,2)=l2
loc(monkey,3)=l2
loc(monkey,4)=l2
onBox(3)
onBox(4)
pushBox(l2,1)
walk(l3,0)

Models       : 1
Time         : 0.000  (Parsing: 0.000)
```