Artificial Intelligence

Assignment 6

Matthias Horbach

mhorbach@uni-koblenz.de

Institute of Web Science and Technologies
Department of Computer Science
University of Koblenz-Landau

Submission until: 20.06.2023, 23:59

Tutorial on: 21.06.2023

Abhinav Ralhan (221202684), abhinavr8@uni-koblenz.de
Hammad Ahmed (221202832), hammadahmed@uni-koblenz.de
Muhammad Asad Chaudhry (221202659), maac@uni-koblenz.de
Vishal Vidhani (221202681), vvidhani@uni-koblenz.de

1 Answer Set Programming (10 Points)

Let P be the following extended logic program:

P:

a ← not c.
c ← b, not q.
c ← not d, not e.
e ← a, not d.
d ← q.
b.

a) Compute the reduct P S of state S = {a, b}. Justify why S is not an answer set for P.

b) Find all answer sets for P. Always mention the state S, its reduct PS and justify why S is an answer set.

c) Transform the program into an equivalent default theory T = (W, Δ) such that there is a one to one correspondence between extensions and answer sets.

Solution

a) P:

a ← not c.
c ← b, not q.
c ← not d, not e.
e ← a, not d.
d ← q.
b.

Rule a ← not c requires c to be false, but c is not in S.

Rule c ← b, not q requires q to be false, but q is not in S.

Rule c ← not d, not e requires d and e to be false, but both d and e are not in S.

Rule e ← a, not d requires d to be false, but d is not in S.

Rule d ← q requires q to be true, but q is not in S.

The minimal model { a, b, c, e} ≠ S
Therefore, S is not an answer set for P.

b)

S = {}
The reduct P^S remains the same as the original program P.
However, P^S contains unsatisfied rules:
a ← not c,
c ← b, not q,
c ← not d, not e,
e ← a, not d,
d ← q.

Therefore, {} is not an answer set for P.

c)

The Equivalent default theory says that
T = (W, Δ) , W = ∅
Δ= {d1, d2, d3, d4, d5, d6, d7}. Here,

d1 = T : ¬ c / a
d2 = b : ¬ q /c
d3 = ¬e : ¬d /c
d4 = ¬d : a /e
d5 = T : q / d
d6 = T : /b

2 Default Reasoning in Potassco (10 Points)

Let's try default reasoning in practice. We rely on Potassco1 for this. Make yourself familiar with the tool's possibilities by exploring the examples shown on their page.

Prolog and Potassco are both logic programming languages and therefore have a lot in common. The most important difference is that both classical negation (-) and default negation (not) are available in Potassco. Expressing the standard example that birds fly unless proven otherwise and that penguins are birds and do not fly looks like this:

```
fly(X) :- bird(X), not -fly(X).
bird(X) :- penguin(X).
-fly(X) :- penguin(X).
```

Furthermore, you can use constraints. A constraint is a rule without a head. It re- moves all answer sets for which the constraint is true. Assuming you want to remove all answer sets in which a planet is made of cheese, you could use the constraint :- planet(X), consistsOf(X,cheese).

We will use Potassco to solve a planning problem. Since the encoding can become a bit tedious, we restrict ourselves in this exercise to a rather simple problem. However, if you enjoy it, I encourage you to also try out the Jurassic Park problem from assignment 4.

The situation is as follows: Theo the turkey is walking down the road. Harry the hunter is carrying an unloaded gun, sees him and wants to shoot him.

To encode a planning problem in Potassco, we should first state which time frame we are interested in, so that the search space is bounded, e.g. to 100 points in time (but you can choose your own boundary):

```
time(0..99).
```

We will want to use walking, loaded, dead to describe states. These symbols may have different truth values at different points in time – they are fluent in time, so to say. To make upcoming rules simpler, it is good practice to wrap them as follows:

```
fluent(walking).
fluent(loaded).
fluent(dead).
literal(F)     :- fluent(F).
literal(neg(F)) :- fluent(F).
contrary(F, neg(F)) :- fluent(F).
contrary(neg(F), F) :- fluent(F).
```

Whether or not a literal holds at a point in time is described by a predicate holds/2. The initial state (time point 0) is given by:

```
holds(walking, 0).
holds(neg(loaded),0).
holds(neg(dead), 0).
```

1. Write a constraint stating a literal and its contrary cannot both hold at the same time.
2. Write a rule stating the closed world hypothesis: if a literal holds at a point in time and we cannot prove that its contrary literal holds at the next point in time, then the literal continues to hold.

Of course, we need actions to change the world:

```
action(load).
action(shoot).
```

3. Write rules for a predicate possible/2 that describes whether it is possible to perform each action at a point in time. This corresponds to the C set in STRIPS.
4. Of course, we do not perform an action if we cannot prove that it is possible to perform it:
   -performed(A,T) :- action(A), time(T), not possible(A,T).
   Write a rule for the predicate performed/2, stating the we will by default perform an action at a given point in time, if we cannot prove that it is not performed.
5. Write rules describing the effects of performing each action.
6. Execute the program. How can you see from the output what will happen?

Additional remarks:

- As always, make sure to submit a source code file that Potassco can actually interpret. If your program produces errors, you may not receive any points for the whole exercise.
- Wherever possible, use clauses with variables and not ground clauses. E.g. the rule given in question 4 uses variables. It is equivalent to a set of ground rules of the form -performed(load,0) :- not possible(load,0), but you should avoid that for rules that are valid for all literals, all actions, and/or all time points.
- If Potassco complains about ”unsafe variables” in a rule, make sure that you declare the type of each variable in the premise. In question 4, both A and T are typed by the premise action(A), time(T). If you run into any other error messages, there is no shame in asking the search engine of your choice for ideas.

Solution

1. :- walking, not walking.

2. holds(walking, 0) :- holds(walking, 0), not holds(walking, 0).

3. possible(shoot, 0) :- holds(loaded, -1).
   possible(load, 0) :- not holds(loaded, -1).

4. -performed(load, 0) :- action(load), time(0), not possible(load, 0).
   performed(load, 0) :- action(load), time(0), not possible(load, 0), not -performed(load, 0).

5. holds(loaded, 1) :- performed(shoot, 0), action(shoot, _, dead), member(loaded, dead).
   holds(loaded, 1) :- holds(loaded, 0), not deleted(loaded, 1).

   deleted(loaded, 1) :- performed(shoot, 0), action(shoot, _, dead), member(neg(loaded), dead).
   deleted(loaded, 1) :- deleted(loaded, 0), not holds(loaded, 1).