Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Learning Strategies in Logic Programming

Author:
Luca Grillotti

Supervisor:
Krysia Broda

Submitted in partial fulfillment of the requirements for the MSc degree in Computing Science / Artificial Intelligence of Imperial College London

September 2017

Contents

Chapter 1

Background

1.1 Answer Set Programming (ASP)

1.1.1 Logic Program

We will work on finite Logic Programs with rules of the form:

- $\leftarrow b_1, b_2, ..., b_m,$ not $c_1,$ not $c_2, ...,$ not c_n (also called *constraint*).
- $a \leftarrow b_1, b_2, ..., b_m,$ not $c_1,$ not $c_2, ...,$ not c_n. This is a *default* rule.
- $l\{a_1, a_2, ..., a_p\}u \leftarrow b_1, b_2, ..., b_m,$ not $c_1,$ not $c_2, ...,$ not c_n, where l and u are numbers such that $l \leq u$. Also, $l\{a_1, a_2, ..., a_p\}u$ is true in an Herbrand Interpretation I iff $l \leq |\{a_1, a_2, ..., a_p\} \cap I| \leq u$. This type of rule is also called *choice rule*.

In all these rules, " not " refers to the *"negation as failure"*, and $a, a_1, a_2, ..., a_p, b_1, b_2, ..., b_m, c_1, c_2, ..., c_n$ are atoms.

a and $l\{a_1, a_2, ..., a_p\}u$ are called the *head* of the rule, and $b_1, b_2, ..., b_m,$ not $c_1,$ not $c_2, ...,$ not c_n is the *body* of the rule. If we call r the following rule: $\alpha \leftarrow b_1, b_2, ..., b_m,$ not $c_1,$ not $c_2, ...,$ not c_n where α is either an atom or an aggregate, then $body^+(r) = \{b_1, b_2, ..., b_m\}$ and $body^-(r) = \{c_1, c_2, ..., c_n\}$.

A *literal* is an atom or the negation (by failure) of an atom.

1.1.2 Stable Models (Answer Sets)

Let P be a logic program and X be an Herbrand Interpretation of P. From P and X, we can construct a new program P^X called the *reduct* of P by applying these methods [2, 6]:

- For every rule $r \in P$, if $X \cap body^-(r) = \emptyset$, then we remove every negative literal in r.
- Otherwise, if $X \cap body^-(r) \neq \emptyset$, then we delete the whole rule.

- We replace every constraint : $-body$ by $\perp$: $-body$. Here $\perp$ is an atom that does not appear in P. As a consequence, $\perp$ does not belong to any Answer Set.
- For every rule r with an aggregate in the head: $l\{a_1, a_2, ..., a_p\}u \leftarrow b_1, b_2, ..., b_m$ (we suppose that we have already removed the negative literals according to the first method)
  - if $l \leq |\{a_1, a_2, ..., a_p\} \cap X| \leq u$, we replace r by all the rules in the following set: $\{a_i \leftarrow b_1, b_2, ..., b_m | a_i \in X\}$.
  - otherwise, we replace r by $\perp \leftarrow b_1, b_2, ..., b_m$

Thus, the reduct P^X is a *definite* logic program (which means that it does not contain any negation as failure). Definite logic programs have a unique minimal Herbrand model [8], that is very easy to construct. We will write $M(P^X)$ the minimal Herbrand model of P_X.

We call *stable model* of P every Herbrand interpretation X that satisfies the relation: $X = M(P^X)$ [9]. Moreover, as we do not use classical negation $\neg$, we will not make a distinction between *answer sets* and stable models.

Stable models of P have more properties than its models in general: they also are *minimal* (for inclusion) and *supported* [9] (every atom in the stable model appears in a rule whose body evaluates to *true*).

1.1.3 Tools

For ASP problems, we will use some of the tools developed by the University of Potsdam: `gringo`, `clasp` and `clingo` [1]. `gringo` transforms the logic program in input into an equivalent variable-free program (it is a *grounder*). And `clasp` is a *solver* capable of solving ground logic programs. Finally, `clingo` only combines grounding (with `gringo`) and solving (with `clasp`).

We used the version 4.3.0 of `clingo` since this the one that ILASP uses.

1.2 Single-Player Games in ASP

It is possible to formalize and solve Single-Player Games in ASP [10]. To illustrate the predicates that can be used to describe these games, we will focus on a basic example: a simple graph game.

1.2.1 Rules of the game

The rules of the graph game on figure 1.1 are the following:

- The player starts in state a.
- If the player is in state a or in state b then he can go to the left or to the right.
- The player wins iff he goes to the state *"victory"*.

- The player cannot go back to state a once he has reached the state *"hole"*.

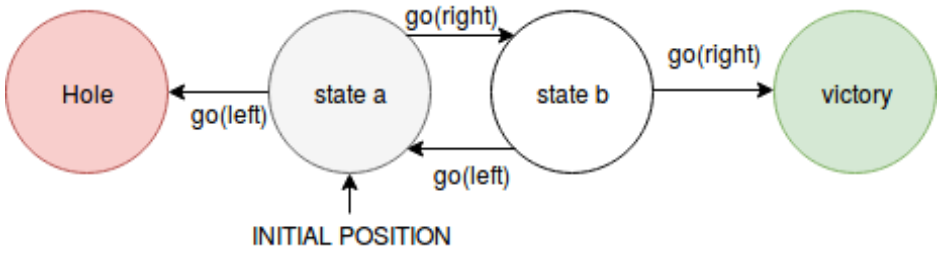

Figure 1.1: Simple graph game

1.2.2 Formalizing these rules

- We start by defining the different players in the game by using the predicate `role`. Here, there is only one player so we include the fact: `role(player).`

- Then, we define what are the initial states by using the predicate `holds` for the first time-step:
```
holds(hole, empty, 1).
holds(a, player, 1). % the player is in state a
holds(b, empty, 1).
holds(victory, empty, 1).
```

- We also say what actions are legal at time T, depending on the state the player is:
```
legal(player,go(left),T) :- holds(a,player,T) ; holds(b,player, T)
legal(player,go(right),T) :- holds(a,player,T) ; holds(b,player,T)
```

- And we explain the situation at time $T+1$ depends on what holds at time T and what the player does:
```
holds(hole,player,T+1) :- holds(a,player,T), does(player,go(left),T).
holds(b,player,T+1) :- holds(a,player,T), does(player,go(right),T).
...
```

- We can also define what is a `terminal` state, and we can evaluate each final state by using the predicate `goal`:
```
terminal(T) :- holds(victory, player, T).
terminal(T) :- holds(hole, player, T).
goal(player,100,T) :- holds(victory, player, T).
goal(player, 0,T) :- holds(hole, player, T).
```

We will follow the following Game Description Language (GDL) restriction: the predicate `does` does not appear in the definition of `terminal`, `goal` or `legal`. Also, `does` only appears in rule bodies. Just below does appears in a choice rule?

1.2.3 Finding a solution to the game

We have just defined a few predicates to translate the rules of the game in ASP. Now we want to simulate the game and to find winning sequences of moves.

- First of all, the player can do only one move at every time step (as long as the game is not finished).
  `1{does(player,go(left),T);does(player,go(right),T)}1 :- not terminated(T).`
  Where `terminated(T)` is true if and only if there is a T1 such that $T1 < T$ and `terminal(T1)` is true:
  ```
  terminated(T) :- terminal(T).
  terminated(T+1) :- terminated(T).
  ```

- Also, every move that is played must be legal:
  `:- does(player, M, T), not legal(player, M, T).`

- We want the game to terminate at some point:
  ```
  :- 0{terminated(T) : time_domain(T)}0.
  time_domain(1..10).  % We allow at most 10 moves
  ```

- Finally, we would like to find a sequence of moves in order to win the game:
  `:- terminal(T), not goal(player, 100, T).`

1.3 Inductive Logic Programming

In this part, B is the logic program that represents the *Background Knowledge*, S_M is the set of possible hypotheses, E^+ is the set of positive examples (all the elements that have been observed) and E^- is the set of negative examples (all the elements that can't appear in any result). Our task is to find an hypothesis $H \in S_M$ such that $B \cup H$ *explains* E^+ and E^-. There are different ways of defining the word "explains" used in the last sentence. We will have a look at three of them.

1.3.1 Brave Inductive Logic Programming

Brave Induction

A brave inductive task is a tuple $T_b =< B, E^+, E^- >$ (where E^+ and E^- are sets of atoms occurring in B) such that: an hypothesis H is solution of T_b iff there is an answer set A of $B \cup H$ verifying $E^+ \subseteq A$ and $E^- \cap A = \emptyset$

We call $ILP_b(B, E^+, E^-)$ the set of hypotheses that are solutions of T_b.

ASPAL encoding

By using ASPAL, we can find the optimal solutions to a brave inductive task. We illustrate how to use the ASPAL encoding with clingo in the following example.

First of all, we consider the background knowledge B:
`person(Nicolas).` `person(Pierre).`
`play_video_games(Nicolas).` `play_video_games(Pierre).` `works(Nicolas).`
And we take: $E^+ = \{$`good_marks(Nicolas)`$\}$ and $E^- = \{$`good_marks(Pierre)`$\}$

- We are studying the hypotheses that are following this mode: *modeh(good_marks(+X))*, *modeb(1,works(+X))* and *modeb(1,play_video_games(+X))*. So we write all the possible hypotheses, and we add the predicate `rule/1` that takes as an argument the *id* of the hypothesis.
```
good_marks(P) :- rule(1).
good_marks(P) :- works(P), rule(2).
good_marks(P) :- play_video_games(+P), rule(3).
good_marks(P) :- works(P), play_video_games(+P), rule(4).
```
- We will select some rules between those above:
`{rule(1..4).}`
- And we want to respect the examples:
```
goal :- good_marks(Nicolas), not good_marks(Pierre).
:- not goal.
```
- Finally, we want to find the optimal (the shortest) hypothesis:
`#minimize[rule(1)=1,rule(2)=2,rule(3)=2,rule(4)=3].`

1.3.2 Cautious Inductive Logic Programming

A cautious inductive task is a tuple $T_c =< B, E^+, E^- >$ (where E^+ and E^- are sets of atoms occurring in B) such that: an hypothesis H is solution of T_c iff

- $B \cup H$ has at least one answer set
- every answer set of $B \cup H$ verifies $E^+ \subseteq A$ and $E^- \cap A = \emptyset$

We call $ILP_c(B, E^+, E^-)$ the set of hypotheses that are solutions of T_c.

1.3.3 Inductive Learning From Answer Sets Programming

We present here a new type of inductive task which is more general than the cautious and the brave inductive tasks.

Partial Interpretations

We call *partial interpretation* [3, 4] a tuple $< e^+, e^- >$ where $e^+ = \{e_1^+, e_2^+, ..., e_m^+\}$ and $e^- = \{e_1^-, e_2^-, ..., e_n^-\}$ are two sets of atoms.
We say that an interpretation I of B extends a partial interpretation $< e^+, e^- >$ iff $e^+ \subseteq I$ and $e^- \cap I = \emptyset$.

Learning from Answer Sets Induction

A Learning from Answer Sets task is a tuple $T_{LAS} =< B, S_M, E^+, E^- >$ (where E^+ and E^- are sets of partial interpretations) [3] such that: an hypothesis H is solution of T_{LAS} iff

- For all $< e^+, e^- > \in E^+$, there is an answer set of $B \cup H$ that extends $< e^+, e^- >$
- For all $< e^+, e^- > \in E^-$, there are no answer sets of $B \cup H$ that extend $< e^+, e^- >$

We call $ILP_{LAS}(B, S_M, E^+, E^-)$ the set of hypotheses that are solutions of T_{LAS}.

ILASP

ILASP is a tool developed at Imperial College London that is capable of finding optimal hypotheses for Learning from Answer Sets tasks [3, 4]. We only need to give it: the background knowledge with the `clingo` syntax, and the mode declaration for the variables that can appear in the head or in the body of a rule in the hypothesis. We can also specify the positive and negative partial interpretations of the task.

Learning rules of games/environment ILASP has been used for learning the rules of Sudoku [4]. Also, a simulated agent in an unknown environment could learn progressively the rules of this environment (such that: "the agent cannot go through a wall", "the agent has to get the key before entering a locked cell"...) [3].

Learning weak constraints ILASP can also be used to learn weak constraints [5, 4], which are some kind of constraints used in `clingo` to rank the answer sets (we can find which solutions/answer sets optimize the problem).

Learning from Context Dependant Answer Sets Instead of containing partial interpretations, E^+ and E^- contain tuples $<< e^+, e^- >, C >$ where C is a *context* [7, 4].
We say that an hypothesis H is solution of $T^{Context}_{LAS} =< B, S_M, E^+, E^- >$ iff

- for all $<< e^+, e^- >, C > \in E^+$, there is an Answer Set A of $B \cup H \cup C$ such that A extends $< e^+, e^- >$
- for all $<< e^+, e^- >, C > \in E^-$, there are no Answer Set A of $B \cup H \cup C$ such that A extends $< e^+, e^- >$

Learning from Noisy Examples ILASP (v3) is also able to learn from noisy data [4]: it can learn an hypothesis that covers a majority (or the most important) examples. To do so, we need to specify which examples are in the noisy data, and we give a weight to them.
ILASP will minimize the sum of the length of the hypothesis and of the weights of the uncovered examples (the weight of an example reflects its importance).

Chapter 2

Progress

2.1 Two-Players games

2.1.1 Representing Two-Players games in ASP

We only need to change a few things to adapt the rules in single-player games for two-players games:

- First of all, we create two roles instead of one:
  ```
  role(player1).
  role(player2).
  ```
- Then we replace `player` in every rule by the variable `P`, and we add `role(P)` at the end of each rule.
- Besides, only one move is allowed at each time step:
  ```
  1{does(P,M,T):role(P),move_domain(M)}1 :- not terminated(T).
  move_domain(go(left)).
  move_domain(go(right)).
  ```
- Finally, a player cannot play twice:
  ```
  :- does(P,M1,T), does(P,M2,T+1).
  ```

2.1.2 Games under study

This project focuses on three games: *Five Field Kono*, *Nine Men's Morris* and *ASALTO*.

For the moment, we have only tried to study how to learn a strategy on a very simple game (with a small number of possible states): the *tic-tac-toe*. Afterwards, we will try to extend what we have done on *tic-tac-toe* to *Five Field Kono*.

Moreover, we have represented these four games in ASP.

2.2 Finding/Learning a strategy

We found several ways of thinking about how to learn a strategy:

- The first one consists in trying to find a move such that: whatever the second player does the next turn (or his 2 or 3 next turns), the first player can still win. This method seems very similar to the *minimax* algorithm.
- We could also study several games played by two players, and try to make hypotheses about what to do in which circumstances.
- It is possible to imagine a mix between the two previous methods: the second one tells us what are the good states, and the first method gives us a move so that we can reach a good state for sure (whatever the second player does).

For the moment, we have only studied the first method.

2.2.1 Planning moves

Basic method

In *tic-tac-toe*, we initialized the game with 4 specific moves already done. If we restrict the hypothesis space by not allowing negation by failure nor constraints, ILASP can find some (ground) hypotheses of the form:

```
does(player1,fill(cell(x)),5).  % at time T=5
does(player1,fill(cell(y)),7) :- does(player2,fill(cell(z)),6).
```

that make `player1` win.

For this, we only need to specify that the program has at least an answer set: `#pos({},{})`, and that `player1` wins in every answer set: `#neg({},{wins(player1)})`.

The problem of the negation by failure

On figure 2.1, we have represented two simplified player games:

- There are 3 turns, `player1` (the blue player) starts and can perform any of the 2 actions `a(1)` or `a(2)` for his first turn.
- The arrows determine what are the possible moves for `player2` (depending on what `player1` did the first turn). So if `player1` starts with `a(1)`, then `player2` can only play `b(1)` or `b(2)`.
- Only one move is possible at each turn.
- The green moves are winning moves.

The only difference between the two graphs in figure 2.1 is that on the second one, `player2` is also allowed to play `b(4)` after `a(1)` (but `player1` cannot win if `player2` plays `b(4)`).

It could be possible to learn definite programs as in tic-tac-toe, but it starts to be very slow if we add a few layers in the graph (it takes more than 3 minutes for graph games with 2 more layers with 9 possible moves instead of 2).

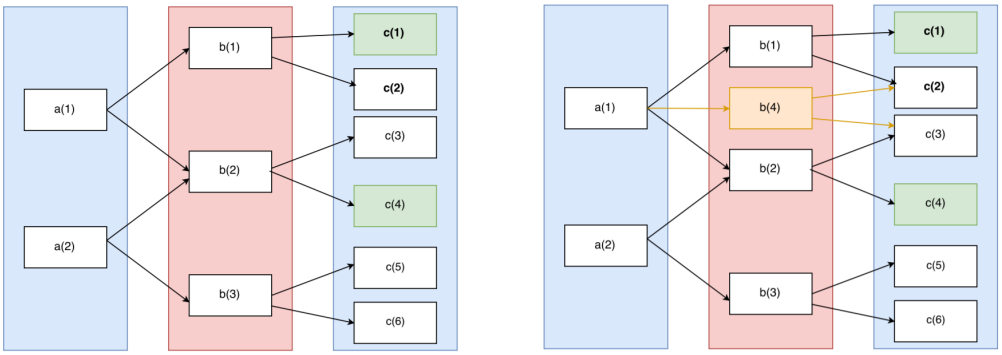

Figure 2.1: Two simple graph games for two players

With negation by failure, we can reduce the length of the optimal hypothesis, and we can generate exceptions structures of this form:
`does(player1,x) :- not does(player2,a)`
`does(player1,y) :- does(player2,a)`
The exceptions generated could also be more complicated.

But if we allow negation by failure in bodies, The hypothesis that may be given for the second graph (on figure 2.1) is
`does(player1,a(1)).`
`does(player1,c(1)) :- not does(player1,b(1)).`
`does(player1,c(4)) :- not does(player1,b(2)).`
and there is no answer set containing `b(4)` with this hypothesis (as it would imply 2 actions for `player1` at the same time, which is forbidden in the background knowledge).

The solution we have found consists in using context dependant positive examples. For each possible move `x` for `player2`, we add
$<<$ {`wins(player1)`}, {} $>$, {`does(player2,x) :- legal(player2,x)`} $>$ to the positive examples. But this solutions does not work anymore if we add some new layers in the graphs.

Very good so far.

Bibliography

[1] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Clingo* = ASP + control: Preliminary report. volume arXiv:1405.3694v1. pages 2

[2] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. pages 1070–1080. MIT Press, 1988. pages 1

[3] M. Law, A. Russo, and K. Broda. *Inductive Learning of Answer Set Programs*, pages 311–325. Springer International Publishing, Cham, 2014. pages 5, 6

[4] M. Law, A. Russo, and K. Broda. The ILASP system for learning answer set programs. `https://www.doc.ic.ac.uk/~ml1909/ILASP`, 2015. pages 5, 6

[5] M. Law, A. Russo, and K. Broda. Learning weak constraints in answer set programming. *CoRR*, abs/1507.06566, 2015. pages 6

[6] M. Law, A. Russo, and K. Broda. Simplified reduct for choice rules in asp. Technical report, Tech. Rep. DTR2015-2, Imperial College of Science, Technology and Medicine, Department of Computing, 2015. pages 1

[7] M. Law, A. Russo, and K. Broda. Iterative learning of answer set programs from context dependent examples. *CoRR*, abs/1608.01946, 2016. pages 6

[8] M. Sergot. Minimal models and fixpoint semantics for definite logic programs. `https://www.doc.ic.ac.uk/~mjs/teaching/KnowledgeRep491Fixpoint_Definite_491-2x1.pdf`, 2005. pages 2

[9] M. Sergot. Stable models (answer sets). `https://www.doc.ic.ac.uk/~mjs/teaching/KnowledgeRep491/StableModels-2x1.pdf`, 2006. pages 2

[10] M. Thielscher. *Answer Set Programming for Single-Player Games in General Game Playing*, pages 327–341. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. pages 2