IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Learning Rules of Board Games using Answer Set 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

Abstract

Your abstract.

Contents

Chapter 1

Introduction

Chapter 2

Background

2.1 Answer Set Programming (ASP)

2.1.1 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.

Reducts apply to ground programs remember -
also perhaps say all your programs will be "safe", so it is easy to ground even with negated body literals.

2.1.2 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:

Some definitions, such as evaluating a choice rule and this simplified reduct are Mark's simplified verison, so reference his notes.

- 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, 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: $X = M(P^X)$. 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* (every atom in the stable model appears in a rule whose body evaluates to *true*).

2.1.3 Tools

For ASP problems, we will use some of the tools developed by the University of Potsdam: `gringo`, `clasp` and `clingo`. `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.

2.1.4 Optimization in clingo

In `clingo`, it is possible to explain what kind of answer sets are optimal by using *weak constraints*. A weak constraint has the following form:

Are the id constants? w can be a variable from the ti and I guess the id can be too

$$:\sim t_1, t_2, t_3, ..., t_n.[w@p, id_1, id_2, ..., id_m]$$

where `w` is called the *weight*, and `p` is called the *priority* of the weak constraint. We will also call *ID* of the weak constraint the list $[w, p, id_1, id_2, ..., id_m]$. We say that a weak constraint is satisfied by an answer set A if there is at least one atom among t_1, t_2, t_3,..., and t_n that is not in A.

We call cost of an answer set A for a given priority `p` (written $cost_p(A)$) the sum of the weights `w` of the different lists $[w, p, id_1, id_2, ..., id_m]$ such that: there is a weak constraint whose ID is $[w, p, id_1, id_2, ..., id_m]$ that is not satisfied by A.

or perhaps violated?

As a consequence:

- if we prefer the answer sets that satisfy a constraint `:- t`$_1$`, t`$_2$`, t`$_3$`,..., t`$_n$`.`, then we can use a weak constraint of the form `:~ t`$_1$`, t`$_2$`, t`$_3$`,..., t`$_n$`.[w@p, id`$_1$`, id`$_2$`, ..., id`$_m$`]` where the weight `w` is **positive**.
- if we prefer the answer sets that satisfy all the atoms `t`$_1$`, t`$_2$`, t`$_3$`,...,` and `t`$_n$ at the same time, then we can use a weak constraint of the form `:~ t`$_1$`, t`$_2$`, t`$_3$`,..., t`$_n$`.[w@p, id`$_1$`, id`$_2$`, ..., id`$_m$`]` where the weight `w` is **negative**.

For `clingo`, an answer set A_1 is more optimal than A_2 if and only if $cost_{\mathtt{p}}(A_1) < cost_{\mathtt{p}}(A_2)$ where `p` is the lowest priority for which $cost_{\mathtt{p}}(A_1) \neq cost_{\mathtt{p}}(A_2)$.

Example 2.1. We consider the following ASP program:

```
1{go_on_holidays ; work_at_imperial}1.
bad_mark :- go_on_holidays.
good_mark :- work_at_imperial.
```

To this program we add the two following weak constraints:

```
:~ go_on_holidays.[-1@1] % we enjoy holidays (priority 1)
:~ good_mark.[-1@2] % we like good marks (priority 2)
```

This ASP program has two answer sets: $A_1 = \{\mathtt{go\_on\_holidays}, \mathtt{bad\_mark}\}$ and $A_2 = \{\mathtt{work\_at\_imperial}, \mathtt{good\_mark}\}$.

- A_1 satisfies only the second weak constraint. So its scores are $cost_1(A_1) = -1$ and $cost_2(A_1) = 0$.
- A_2 satisfies only the first weak constraint. So its scores are $cost_1(A_2) = 0$ and $cost_2(A_2) = -1$.

Is there a discrepancy here - is it lowest or highest?

The highest priority is 2 and $cost_2(A_2) < cost_2(A_1)$. Thus, $A_2 = \{\mathtt{work\_at\_imperial}, \mathtt{good\_mark}\}$ is optimal.

cost_1 is different too for the two models?

2.2 Single-Player Games in ASP

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

2.2.1 Rules of the game

The rules of the graph game on figure 2.1 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”*.

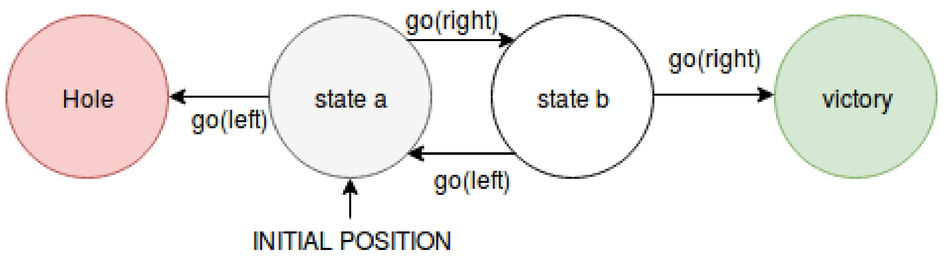

Figure 2.1: Simple graph game

2.2.2 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).
  ```

Is this the first time you mention GDL? Shouldn't it go earlier in this section? Perhaps with a ref or two?

We will follow the following Game Description Language (GDL) restriction: the predicate `does` does not appear in the definition of `terminal`, `goal` or `legal`.

2.2.3 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)
  ```

2.3 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.

2.3.1 Brave Inductive Logic Programming

Brave Induction

Need to be a bit more precise: Aren't example atoms in the Herbrand Base of the program?

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].
  ```

You could give the result?

2.3.2 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.

2.3.3 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.

In all 3 cases give references! (I know you said you hadn't included them yet, so this is a reminder.

Partial Interpretations

We call *partial interpretation* 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) 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. 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. 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...).

Learning weak constraints ILASP can also be used to learn weak constraints, 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*.
We say that an hypothesis H is solution of $T_{LAS}^{Context} =< B, S_M, E^+, E^- >$ 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: 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).

Can you adapt your ASPAL example to illustrate some of these things?

Chapter 3

From Single Player Games to Two Player Games

3.1 Games under study

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

3.1.1 Five Field Kono

Five Field Kono is a Korean strategy game. It is played on a 5×5 grid. Its initial configuration is as presented in figure 3.1.

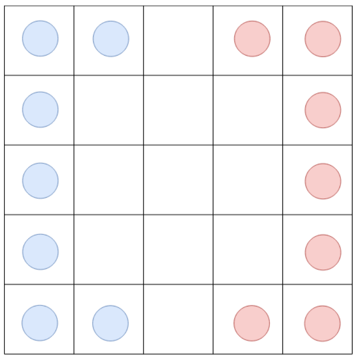

Figure 3.1: Initial configuration of Five Field Kono. The two players (in blue and red) have seven pawns each.

The two opponents move one of their pawns in turn knowing that a pawn can only be moved diagonally to an adjacent square (forwards or backwards). A player wins if all his pawns have taken the positions initially held by his opponent's pawns (see figure 3.2).

3.1.2 Nine Men's Morris

Nine Men's Morris is also a Korean strategy game [1]. At first the two opponents place in turn their pawns on the board (figure 3.3). When three pawns are aligned

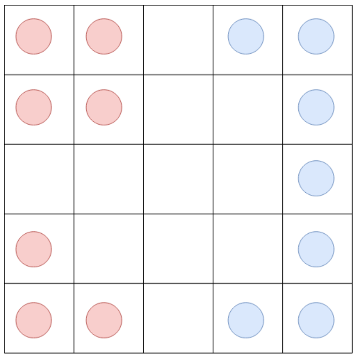

Figure 3.2: Situation where the blue player wins at Five Field Kono. Every blue pawn is in a square initially occupied by a red pawn (see figure 3.1)

makes?

on the board, we say that they form a **mill**. Each time a player does a mill, he can remove one of his opponent's pawns on the board (that is not already in a mill if possible). Once both players have placed all their pawns, they move in turn one of them to an adjacent empty position. In this second phase, creating a meal has the same effect as before. A player wins if his opponent cannot move or has only two pawns left.

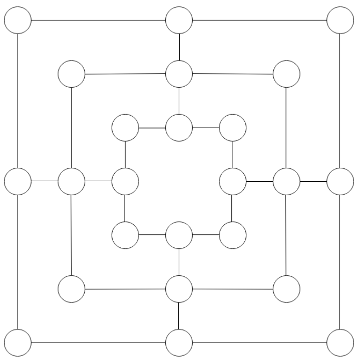

Figure 3.3: Board for Nine Men's Morris. The circles are the positions where the pawns can be placed or moved. Two positions are adjacent if there is an edge that links them.

3.1.3 ASALTO

3.2 Representing Two-Players games in ASP

3.2.1 Similarities and Differences with Single-Player games

Representing Two-Players games in ASP follows the same scheme as for Single-Player games [2]. However, the predicate `role/1` needs to be modified to include the *color* of each player. And the predicate `legal/3` has to take into account the fact that this is a turn-based game: a player has the right to play if and only if it is his turn.

Need to make clear this is your extension. eg Say (before 3.2.1) In this section we define our extension of GDL to 2 player games (or some such)

Also, we will consider that each *cell* has a set of coordinates (represented by the function `coord`), and a *state*. For example, if the red player has a red pawn on the cell of coordinates $(1, 3)$ at time 4, then its translation in ASP is: `holds(cell(coord(1,3),red),4).`

For the moment, we only have studied single-phase game (in the background chapter, and in [2]). But *Nine Men's Morris* and *ASALTO* are two-phases games: in both of them, players place their pawns before moving them. So we need to define a new predicate `phase/2` that will appear in the bodies of the rules that define `legal/3`

How to describe a two-players-game in ASP

1. First, we describe the two players, by using `role/2`, that takes the name of the player as first argument, and his color for its second argument. From now on, we will suppose the two players are respectively called *player1* and *player2*.

2. Also, we add the following rule:
   `opponent(P1, P2) :- role(P1, C1), role(P2, C2), P1 != P2.`

   Isn't opponent(player1,player2) and opponent(player2,player1) simpler?

3. Then we describe the game as if it was a single player game with only one phase [2] (we forget that this is a turn based game with one or more phases). In particular, we will have to define the predicates `holds/3`, `legal/3`, `terminal/1`, and `wins/2`. Moreover, `legal(P,M,T)` should only rely on `holds(C,T)` (at time T *only*), `role/2`, and `opponent/2` for the moment. That way, `legal/3` does not depend on the previous actions or on the previous states, but only on the current states of the game.

4. In each rule that defines `legal/3` (of the form `legal(Player, a(X), T)`), we add the following predicate in the body: `can_play(Player, a, T)`. This predicate is true if an only if `Player` has the right to perform an action of type `a` at time `T`.

5. We define the predicate `can_play/3`. For instance, in the case of the single-phase turn-based game, `can_play/3` can be defined in the following way:
   ```
   turn(player1, 1).
   turn(P1, T+1) :- time(T), turn(P2, T), opponent(P1, P2).
   can_play(Player, Action, T) :- turn(Player, T).
   ```

6. If the game has more than one phase, we need to describe the mechanism of the phase. At time $T = 1$, we are in the first phase:
   ```
   phase(phase(1), 1).
   ```
   We also want a phase to continue before it has finished:
   ```
   phase(phase(N), T+1) :- time(T), phase(phase(N), T), not finished(phase(N),
   T+1).
   ```
   And if it has finished at time T, we go to the next phase:
   ```
   phase(phase(N+1), T) :- time(T), finished(phase(N),T).
   ```

   Then, we add `phase/2` in the definition of `can_play/3`. For example, if the two players can only play an action of type a in the first phase, and of type b in the

second one, then it is equivalent to:
```
can_play(Player, a, T) :- turn(Player, T), phase(phase(1),T).
can_play(Player, b, T) :- turn(Player, T), phase(phase(2),T).
```

7. Finally, we need to write the rules that define `finished/2`. These rules mainly depend on the board game: for *Nine Men's Morris*, we move to the second phase once both players have placed all their pawns, but in *ASALTO*, the soldiers are already positioned at time $T = 1$, and only one player has to place the officers.

3.2.2 Example: Nine Men's Morris in ASP

Coordinate System

To represent *Nine Men's Morris*, ASP, we first need to define what are the coordinates of the different cells. It seems that the system has three layers with the second one inside of the first one, and the third one inside the first one. This will be our second coordinate. Besides, there are eight cells in each layer. We will suppose the cell at the top-left corner of the layer has its first coordinate equal to 1, and we increment this coordinate as we go through the layer clockwise (Figure 3.4).

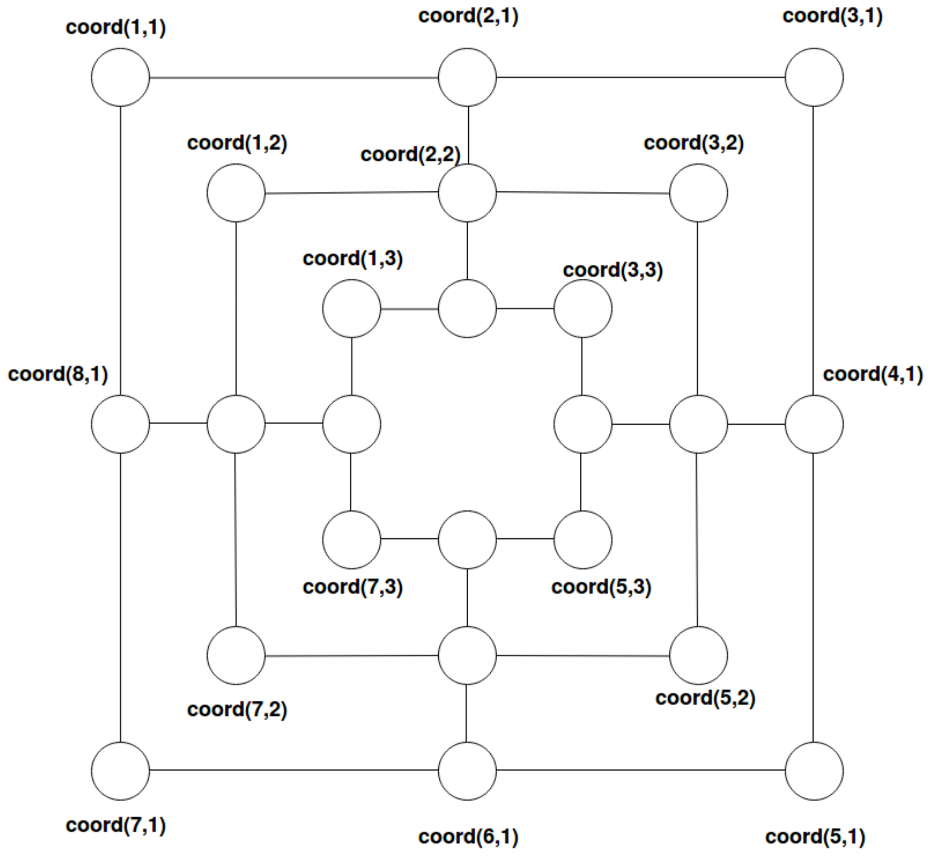

Figure 3.4: Coordinates of the cells used in Nine Men's Morris

Nine Men's Morris described in ASP

Here, each item refers to the last subsection.

1. First, we define the players and their roles
   `role(player1, red).`
   `role(player2, blue).`

2. We also add the following rule:
   `opponent(P1, P2) :- role(P1, C1), role(P2, C2), P1 != P2.`

3. Here, we describe the game as in [2]: `legal/3` only depends on the state of the game (`holds/2`) and the player that is playing. In other words, we do not take the turn-based aspect into account.

   ?? briefly? or remove the word!

   We will only introduce quickly the predicates that were defined in the ASP Program. You may refer to the complete ASP program (in Appendix ??) for further information.

predicate	description
`holds/3`	Describes the state of the game: which cells are empty, and which ones contain a pawn of color red or blue.
`legal/3`	Tells which action a player has the right to perform at a specific time. For instance, `legal(P,remove_pawn(C),T)` is true if and only if there is a pawn of the opponent in `C` and this pawn is not in a mill (but if all his pawns form mills, the player `P` can still remove it)
`adjacent/2`	Two pairs of coordinates are adjacent if there is an edge that links them (Figure 3.4)
`is_in_mill/2`	`is_in_mill(Coord,mill(Coord_1, Coord_2, Coord_3),T)` is true if a player owns the mill `mill(Coord_1, Coord_2, Coord_3)` and if `Coord` is one of its three coordinates
`has_mill/3`	`has_mill(P,Mill,T)` is true if Mill is a mill owned by P at time T. It helps to define `is_in_mill/2`
`all_in_mill/2`	Tells if a player has all his pawns in mills
`terminal/1`	Represents the end of the game (when a player has won). Its only argument is the time `T`.
`wins/2`	Tells which player has won and when. A player wins if the other has 2 or less pawns on the board or if the other player is not able to perform any move.
`pawns_on_board/3`	Counts the number of pawns for each player on the board at each time.
`able_to_play/2`	Tells if a player is able to perform any action at a specific time.

4. After having defined `legal/3` in the previous step, we add:

   - `can_play(Player, place_pawn, T)` in the body of `legal(Player, place_pawn(Coord), T)`

- `can_play(Player, move, T)` in the body of `legal(Player, move(Coord_1, Coord_2), T)`
- `can_play(Player, remove_pawn, T)` in the body of `legal(Player, remove_pawn(Coord), T)`

5. We will consider that there are two phases in this game. In the first phase, the players place their respective pawns on the board. And in the second, they have finished to place their pawns and they only move them.

   We suppose that `player1` starts: `can_play(player1, place_pawn, 1).`

   To define `can_play/3` at time `T+1` we need to have a closer look at the rules of the game:

   - A player can remove a pawn at time `T` if he created a new mill with his action at time `T-1` (in both phases):
     `can_play(P1, remove_pawn, T+1) :- does(P1, Action, T), time(T), has_new_mill(P1, T+1), role(P1, C1), phase(phase(1..2),T+1).`
   - Thus, a player can place a pawn if his opponent did not get a new mill with his last action, and if the game is in its first phase:
     `can_play(P1, place_pawn, T+1) :- does(P2, Action, T), time(T), not has_new_mill(P2, T+1), opponent_player(P1, P2), phase(phase(1),T+1).`
   - Also, a player can move a pawn at time `T+1` if his opponent did not get a new mill at the same time, and if the game is in its second phase:
     `can_play(P1, move, T+1) :- does(P2, Action, T), time(T), not has_new_mill(P2, T+1), opponent_player(P1, P2), phase(phase(2),T+1).`

   The definition of `has_new_meal/2` can be found in appendix ??

6. As there are more than one phase, we add the following definition of `phase/2` in the program (as explained in the last subsection):
   ```
   phase(phase(1), 1).
   phase(phase(N), T+1) :- time(T), phase(phase(N), T),
   not finished(phase(N), T+1).
   phase(phase(N+1), T) :- time(T), finished(phase(N),T).
   ```

7. Finally we define the predicate `finished/2`:

   - The first phase is finished when both players do not have any pawn in their hands:
     `finished(phase(1), T) :- time(T), has_pawns(player1, 0, T), has_pawns(player2, 0, T).`
     Where `has_pawns/3` counts the number of pawns that each player still has to place. Its complete definition may be found in appendix ??
   - We suppose that the second phase never finishes (as there is no third phase). So we do not need to define `finished(phase(2), T)`.

Maybe worth showing an example play using the rules? i.e. one of the many answer sets.

Chapter 4

Learning rules for board Games

4.1 General Idea

We observe people playing the game, and
if they do a valid action - positive example
if they do an action that is not legal - negative example

goal : learning legal/3 predicate (according to the structure of GDL)

source: has been made in ILASP (paper reference)

Learning predicate wins/2 ?

4.2 Evaluation

Evaluation at least for Five Field Kono.

added example	hypothesis found	time (s)

Chapter 5

Learning strategies

Be positive!

For the moment, we only have implemented the *tic-tac-toe* and the *Field Field Kono* in python by using `pygame` (python library for video games).

The idea would be to learn from the actions performed by a player to train our system by using ILASP. Then we could make him play against this trained system and continue to train it that way.

5.1 First Method: maximize the chances of winning

5.1.1 Learning Exception Structure

Supposing we are at time $T = t$, we want the computer to induce a set of rules H such that:

- There is a move m such that for every answer set A of $B \cup H$, $does(player1, m, t) \in A$
- For every answer set A of $B \cup H$, for all move m_1 different from m, we have: $does(player1, m_1, t) \notin A$
- H also tries to take into account the possible future moves.

To reduce the hypothesis space, it would be interesting to focus only on *exception structures*, which means we want H to be of this form:

$$H = \left\{ \begin{array}{c} \texttt{does(player1, move\_p1\_t0\_1, t).} \\ \texttt{does(player1, move\_p1\_t2\_1, t+2) :- does(player2, move\_p2\_t1\_1, t+1).} \\ \texttt{does(player1, move\_p1\_t2\_2, t+2) :- not does(player2, move\_p2\_t1\_1, t+1).} \end{array} \right\}$$

In the set above, we will call "*exception rule*" the second rule, and "*general rule*" the third.

Remark 5.1. The hypothesis space could also be extended so that it accepts two or more exception rules.

$$H = \left\{ \begin{array}{ll} \quad\texttt{does(player1, move\_p1\_t0\_1, t).} & \\ \texttt{does(player1, move\_p1\_t2\_1, t+2) :-} & \quad\texttt{does(player2, move\_p2\_t1\_1, t+1).} \\ \texttt{does(player1, move\_p1\_t2\_2, t+2) :-} & \quad\texttt{does(player2, move\_p2\_t1\_2, t+1).} \\ \texttt{does(player1, move\_p1\_t2\_3, t+2) :-} & \texttt{not does(player2, move\_p2\_t1\_1, t+1),} \\ & \texttt{not does(player2, move\_p2\_t1\_2, t+1).} \end{array} \right\}$$

Moreover, the atoms in the exception rule must appear in the general rule (see the red and blue atoms in the hypotheses above).

With ILASP 3.1.0 it is possible add "bias constraints" to specify what kind of rule we want to learn. However it is not possible to add constraints on the whole set of rules induced. As a consequence, we cannot say that for every solution H in $ILP_{LAS}(B, S_M, E^+, E^-)$, if there are $move\_p1\_t2\_1$ and $move\_p2\_t1\_1$ such that:
`"does(player1,move_p1_t2_1,t+2) :- does(player2,move_p2_t1_1,t+1)."` $\in H$,
then there is $move\_p1\_t2\_2$ such that:
`"does(player1,move_p1_t2_2,t+2):- not does(player2,move_p2_t1_1,t+1)."` $\in H$

The solution we used consists in using the ASPAL encoding in ILASP. With this encoding, we can add more flexible constraints for the hypothesis space.

Not sure I fully follow this ....

5.1.2 Example

Current State of the Game Consider a tic-tac-toe game after the 4 following moves (represented in 5.1):

```
does(player1, fill(coord(2,2)),1).
does(player2, fill(coord(1,2)),2).
does(player1, fill(coord(1,1)),3).
does(player2, fill(coord(3,3)),4).
```

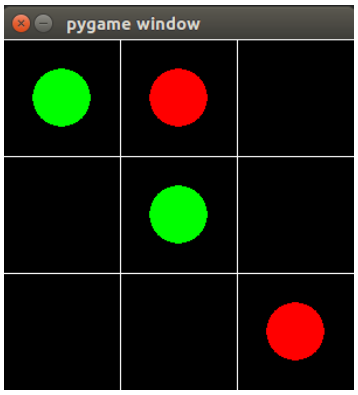

Figure 5.1: State of a tic-tac-toe game after 4 moves. The green player plays first (he is `player1`).

Here, we are the green player and we want to find a move such that we are sure to win in three moves.

Hypothesis space We use the following mode declaration:

$$M = \left\{\begin{array}{rl} m1: & \texttt{modeh(does(player1,\#move,5)).} \\ m2: & \texttt{modeh(does(player1,\#move,7)).} \\ m3: & \texttt{modeb(does(player2,\#move,6)).} \\ m4: & \texttt{modeb(not does(player2,\#move,6)).} \end{array}\right\}$$

Moreover, we take the following hypothesis space:

$$R_M = \left\{\begin{array}{l} \texttt{does(player1, X, 5).} \\ \texttt{does(player1, Z, 7) :- does(player2, Y, 6).} \\ \texttt{does(player1, X, 7) :- not does(player2, Y, 6).} \end{array}\right\}$$

So the top theory that we use for this example is:

```
     does(player1, X, 5) :-      legal(player1,X,5), rule((m1),(X)).
     does(player1, Z, 7) :-   legal(player1, Z, 7), legal(player2, Y, 6),
T =                            does(player2, Y, 6), rule((m2,m3,2),(Z,Y)).
     does(player1, X, 7) :-   legal(player1, X, 7), legal(player2, Y, 6),
                            not does(player2, Y, 6), rule((m2,m4,2),(X,Y)).
```

where we added the predicate `legal/3` to make the variables safe.

So now the hypothesis we want to learn with ILASP has the following form:

$$H = \left\{\begin{array}{c} \texttt{rule((m1),(\#move\_p1\_t0\_1)).} \\ \texttt{rule((m2,m3,2),(\#move\_p1\_t2\_1, \#move\_p2\_t1\_1)).} \\ \texttt{rule((m2,m4,2),(\#move\_p1\_t2\_2, \#move\_p2\_t1\_1)).} \end{array}\right\}$$

We want H to contain only one rule of each type: only one rule of the form `rule((m1),(#move_p1_t0_1))` and so on... So we add some rules to make them unique:

```
rule1 :- rule((m1),(fill(coord(X,Y)))).
:- not rule1.
:- rule((m1),(fill(coord(X1,Y1)))), rule((m1),(fill(coord(X2,Y2)))), g(X1,
Y1)< g(X2, Y2).
```

The two first rules make `rule((m1),(#move_p1_t0_1))` appear at least once, and the last rule makes it appear at most once. We add similar rules for `rule((m2,m3,2), (#move_p1_t2_1, #move_p2_t1_1))` and `rule((m2,m4,2), (#move_p1_t2_2, #move_p2_t1_1))`.

Also, we added some constraints to reduce the computation time. For instance, if `rule((m1),(fill(coord(X,Y))))` is true, then `legal(player1,fill(coord(X,Y)),5)` must be true. So we add the following constraint:

```
:- rule((m1),(fill(coord(X,Y)))), not legal(player1,fill(coord(X,Y)),5).
```

And if `rule((m2,m4,2),(fill(coord(X,Y)),fill(coord(Z,T))))` and `not does(player2, fill(coord(Z,T)), 6)` are true, then `legal(player1, fill(coord(X,Y)),7)` must

be true unless `player2` plays in `coord(X,Y)` at time 6. Thus, we add the following constraint:
`:- rule((m2,m4,2),(fill(coord(X,Y)),fill(coord(Z,T)))), not does(player2, fill(coord(Z,T)), 6), not legal(player1, fill(coord(X,Y)),7).`
and we add this context-dependent example to the set of positive examples :
$<< \{$`wins(player2,8)`$\}, \emptyset >,$
$\{$`:- rule((m2,m4,2),(fill(coord(X,Y)),fill(coord(Z,T)))), not does(player2, fill(coord(X,Y)), 6).`$\} >$
With this example, we are sure that there will be at least one answer set where `player2` plays in `coord(X,Y)`.

Declaring the examples First of all, we want the first player to win in two moves all the time. So there is no answer set that does not contain `wins(player1,8)`. Thus, we take : $E^- = \{< \emptyset, \{$`wins(player1,8)`$\} >\}$.

For the positive examples, we only need to take the one given in the previous paragraph. The `wins(player2,8)` in it is optional but it reduces the computation time from 11 seconds to 1 second. Maybe since the results are better, be very clear about the examples.

Evaluation

5.1.3 Finding a move in the middle of a game

We are only able to look two moves ahead, so we cannot predict how to win a game if we are not very close to the end. It would be a good idea to define what a good state is (or to make ILASP learn it). And after that, it could be possible to choose a state at time t that maximizes the chances of reaching a good state at time $t + 2$.

For instance, if we consider the game represented in figure 5.2, where `player1` plays at times t and $t + 2$ and `player2` plays at time $t + 1$.
We want to maximize the probability of reaching a good state at $t + 2$. So we choose $a(x)$ such that $\#\{b(y)|legal(b(y)) \wedge (b(y) \implies \exists\, c(z)\ ;\ legal(c(z)) \wedge good\_state(state\_c(z)))\}$ is maximal, which is $a(1)$ in this example. What is x? And a/b/c?

5.2 Second Method: recognize preferred moves and states

Goal: learn the preferences of the player by using Weak Constraints.

5.2.1 Learning what kind of action the player prefers

When we have the choice of the kind of action like in Five Field Kono : Learning direction, up_right, down_right...

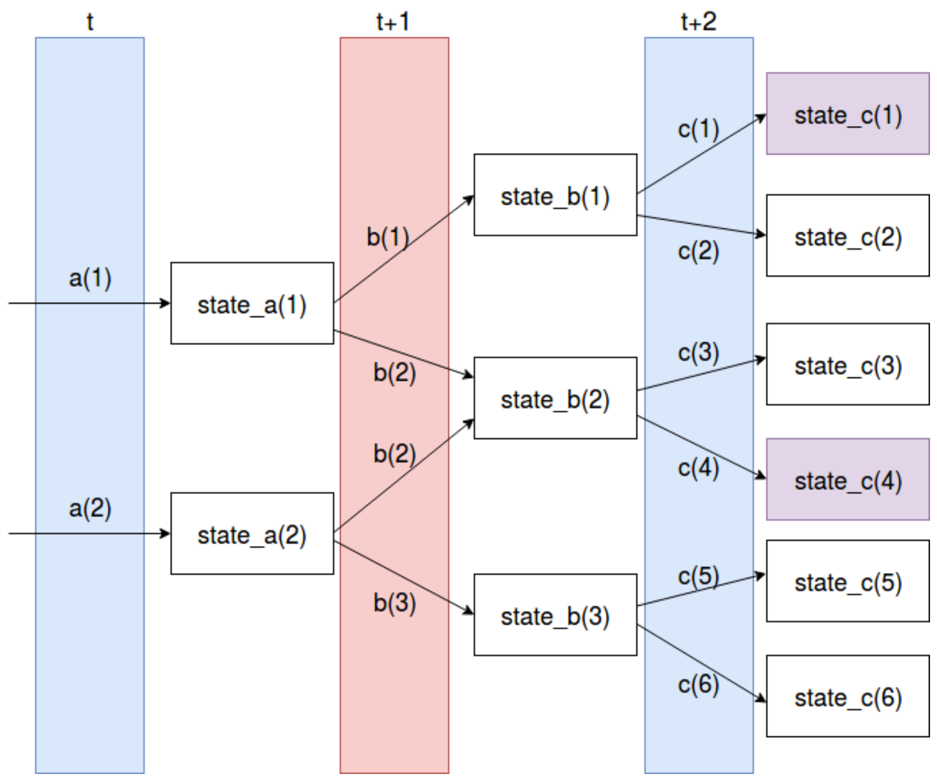

Figure 5.2: Desription of a graph game with its states and the actions to perform to go from one state to another. The blue player (that plays at times t and $t + 2$) is `player1`. The states in purple are the good states.

5.2.2 Discover the sates of the board that the player chooses

Talk about the the drawbacks of last subsection: does not give enough information. So now we learn what kind of configuration the player prefers in a game.

How to represent patterns

How to represent patterns, how to combine them. Example of Five Field Kono

Learning preferred patterns

Test results with program, with time results.
Problem: one simple pattern can cover many more complex patterns

Chapter 6

Evaluation

TODO : or maybe a part evaluation in each of the previous chapters

Chapter 7

Conclusions and Future work

Bibliography

[1] R. Bell. *Board and Table Games from Many Civilizations*, volume 1 to 2 of *Board and Table Games from Many Civilizations*, chapter 3, pages 93–95. Dover Publications, 1979. pages 9

[2] M. Thielscher. Answer set programming for single-player games in general game playing. In *ICLP*, pages 327–341. Springer, 2009. pages 10, 11, 13

Appendix A

Nine Men’s Morris described in ASP

Nine Men’s Morris

```
% initialize the two players
role(player1, red).
role(player2, blue).

% Opponent relations
opponent_player(P1, P2) :- role(P1, C1), role(P2, C2), P1 != P2.
opponent_color(C1, C2) :- role(P1, C1), role(P2, C2), C1 != C2.

% describe the state of the game before any player plays
holds(cell(coord(1..8,1..3), empty),1).

range0_x(1..8).
range1_x(1..7).

range0_y(1..3).
range1_y(1..2).

% even and odd numbers
even(2).
even(4).
even(6).
even(8).

% explain how an action at time T by a player influences the state of the game at time T+1
holds(cell(Coord, Color), T+1) :- time(T), role(Player, Color),
                                  does(Player, place_pawn(Coord), T).
holds(cell(Coord_2, Color), T+1) :- time(T), role(Player, Color),
                                  does(Player, move(Coord_1, Coord_2), T).
holds(cell(Coord_1, empty), T+1) :- time(T), role(Player, Color),
                                  does(Player, move(Coord_1, Coord_2), T).

holds(cell(Coord, empty), T+1) :- time(T), role(Player, Color),
                                  does(Player, remove_pawn(Coord), T).

holds(cell(Coord, empty), T+1) :- holds(cell(Coord, empty), T), time(T),
                        not holds(cell(Coord, red), T+1), not holds(cell(Coord, blue), T+1).
holds(cell(Coord, red), T+1) :- holds(cell(Coord, red), T), time(T),
                        not holds(cell(Coord, empty), T+1), not holds(cell(Coord, blue), T+1).
holds(cell(Coord, blue), T+1) :- holds(cell(Coord, blue), T), time(T),
                        not holds(cell(Coord, red), T+1), not holds(cell(Coord, empty), T+1).

% describe what actions are legal or not.
legal(Player, place_pawn(Coord), T) :- holds(cell(Coord,empty),T), role(Player, Color),
                                        time(T), can_play(Player, place_pawn, T).


legal(Player, move(Coord_1, Coord_2), T) :- time(T), holds(cell(Coord_1, Color),T),
                                        holds(cell(Coord_2,empty),T), role(Player, Color),
                                        adjacent(Coord_1,Coord_2), can_play(Player, move, T).


legal(Player, remove_pawn(Coord), T) :- time(T), role(Player, Color),
                                holds(cell(Coord,Color_2),T), opponent_color(Color, Color_2),
                                not is_in_mill(Coord, T), can_play(Player, remove_pawn, T).

legal(Player, remove_pawn(Coord), T) :- time(T), role(Player, Color),
                        holds(cell(Coord,Color_2),T), opponent_player(Player, Player_2),
                        role(Player_2, Color_2), all_in_mill(Player_2, T),
                        can_play(Player, remove_pawn, T).

% adjacent/2 tells when two cells are linked
adjacent(coord(X1,Y1),coord(X1+1,Y1)) :- range1_x(X1), range0_y(Y1).
adjacent(coord(X1+1,Y1),coord(X1,Y1)) :- range1_x(X1), range0_y(Y1).
adjacent(coord(8,Y1),coord(1,Y1)) :- range0_y(Y1).
adjacent(coord(1,Y1),coord(8,Y1)) :- range0_y(Y1).

adjacent(coord(X1,Y1),coord(X1,Y1+1)) :- even(X1), range1_y(Y1).
adjacent(coord(X1,Y1+1),coord(X1,Y1)) :- even(X1), range1_y(Y1).

% is_in_mill(Coord, T) is true iff the cell of coordinates Coord is in the mill of a player.
is_in_mill(Coord_1, T) :- has_mill(Player, mill(Coord_1, Coord_2, Coord_3), T),
                        role(Player, Color), time(T).
is_in_mill(Coord_2, T) :- has_mill(Player, mill(Coord_1, Coord_2, Coord_3), T),
                        role(Player, Color), time(T).
is_in_mill(Coord_3, T) :- has_mill(Player, mill(Coord_1, Coord_2, Coord_3), T),
                        role(Player, Color), time(T).

% has_mill(Player, mill(Coord_1, Coord_2, Coord_3), T) is true iff Player has a mill of his
% own color and the mill is formed by the following coordinates: Coord_1, Coord_2, Coord_3
has_mill(Player, mill(coord(1,Y), coord(2,Y), coord(3,Y)), T) :- time(T), role(Player, Color),
                        range0_y(Y), holds(cell(coord(1,Y),Color),T),
                        holds(cell(coord(2,Y),Color),T), holds(cell(coord(3,Y),Color),T).

has_mill(Player, mill(coord(3,Y), coord(4,Y), coord(5,Y)), T) :- time(T), role(Player, Color),
                        range0_y(Y), holds(cell(coord(3,Y),Color),T),
                        holds(cell(coord(4,Y),Color),T), holds(cell(coord(5,Y),Color),T).

has_mill(Player, mill(coord(5,Y), coord(6,Y), coord(7,Y)), T) :- time(T), role(Player, Color),
                        range0_y(Y), holds(cell(coord(5,Y),Color),T),
                        holds(cell(coord(6,Y),Color),T), holds(cell(coord(7,Y),Color),T).

has_mill(Player, mill(coord(7,Y), coord(8,Y), coord(1,Y)), T) :- time(T), role(Player, Color),
                        range0_y(Y), holds(cell(coord(7,Y),Color),T),
                        holds(cell(coord(8,Y),Color),T), holds(cell(coord(1,Y),Color),T).

has_mill(Player, mill(coord(X,1), coord(X,2), coord(X,3)), T) :- time(T),
                        role(Player, Color), even(X), holds(cell(coord(X,1),Color),T),
                        holds(cell(coord(X,2),Color),T), holds(cell(coord(X,3),Color),T).

% all_in_mill/2 tells if a player has all his pawns in mills at time T
all_in_mill(Player, T) :- time(T), role(Player, Color), not not_all_in_mill(Player, T).

not_all_in_mill(Player, T) :- role(Player, Color), holds(cell(Coord,Color),T),
                                not is_in_mill(Coord,T), time(T).



% can_play/3 describe what kind of action a player can perform at each time step
can_play(player1, place_pawn, 1).

can_play(P1, place_pawn, T+1) :- does(P2, Action, T), time(T), not has_new_mill(P2, T+1),
                                opponent_player(P1, P2), phase(phase(1),T+1).

can_play(P1, move, T+1) :- does(P2, Action, T), time(T), not has_new_mill(P2, T+1),
                                opponent_player(P1, P2), phase(phase(2),T+1).

can_play(P1, remove_pawn, T+1) :- does(P1, Action, T), time(T), has_new_mill(P1, T+1),
                                role(P1, C1),  phase(phase(1..2),T+1).

% has_new_mill/2 tells if a player has a mill at time T that he did not have at time T-1.
has_new_mill(Player, T) :- time(T), has_mill(Player, Mill, T), not has_mill(Player, Mill, T-1).

% phase/2 defines the phase of the game in which we are. phase(Phase, T) is
% true iff the system is in the phase "Phase" at time T.
phase(phase(1), 1).
phase(phase(N), T+1) :- time(T), phase(phase(N), T), not finished(phase(N), T+1).
phase(phase(N+1), T) :- time(T), finished(phase(N),T).

% finished(Phase, T) is true iff the phase "Phase" is finished at time T.
finished(phase(1), T) :- time(T), has_pawns(player1, 0, T), has_pawns(player2, 0, T).

% has_pawns(Player,N,T) is true iff Player has still N pawns to place at time T
has_pawns(Player,9,1) :- role(Player, Color).

has_pawns(Player, N-1, T+1) :- time(T), role(Player, Color), has_pawns(Player, N, T),
                does(Player, place_pawn(coord(X,Y)), T), range0_x(X), range0_y(Y).
has_pawns(Player, N,   T+1) :- time(T), role(Player, Color), has_pawns(Player, N, T),
                not has_pawns(Player, N-1, T+1).





% conditions to stop the game
terminal(T) :- wins(Player,T), role(Player,Color).

% A player wins if his opponent cannot move or if his opponent has only than
% 2 pawns left on the board.
wins(P1, T) :- pawns_on_board(P2, N, T), opponent_player(P1, P2), phase(phase(2),T),
                N<=2, time(T).
wins(P1, T) :- can_play(P2, move, T), opponent_player(P1, P2), time(T),
                not able_to_play(P2, T).

% pawns_on_board(Player, N, T) is true iff Player has exactly
% N pawns on the board at time T.
pawns_on_board(Player,0,1) :- role(Player, Color).

pawns_on_board(Player,N+1,T+1) :- role(Player, Color), time(T), pawns_on_board(Player,N,T),
                                does(Player, place_pawn(Coord), T).
pawns_on_board(Player,N-1,T+1) :- role(Player, Color), time(T), pawns_on_board(Player,N,T),
                does(Player_2, remove_pawn(Coord), T), opponent_player(Player, Player_2).
pawns_on_board(Player,N  ,T+1) :- role(Player, Color), time(T), pawns_on_board(Player,N,T),
                not pawns_on_board(Player,N+1,T+1), not pawns_on_board(Player,N-1,T+1).

% A player is able to play if he can perform at least one action.
able_to_play(Player, T) :- legal(Player, Action, T).




% only one action is performed at every step
1{does(Player, M, T) : legal(Player, M, T)}1 :- time(T), not terminated(T).

terminated(T) :- terminal(T).
terminated(T+1) :- terminated(T), time(T).

% everything that is done must be legal
:- does(P,M,T), not legal(P,M,T).
```