MOIA — Quarto

Vincent Hugot, Benjamin Dreux

May 14, 2009

/** Quarto AI : file ai.pl

Vincent Hugot and Benjamin Dreux

*/

/** **MODULES** */

```
:-use_module(library(lists)).
:-use_module(library(assoc)).
```

/** **All possible pieces.**
The game supports four characteristics: **W**: white; **S**: square; **F**: full; **B**: big.
If the set of characteristics is $C = \{w, s, f, b\}$, then the set of pieces P is given by $P = \mathcal{P}(C)$.

Note: Generated Prolog Code: An external program (written in `Caml`) was used to generate the list of pieces, — each denoted by `piece_w?s?f?b?` — and in general all parts of the Prolog code describing game rules, where there is a lot of repetitive typing to do. The idea was simply to avoid making mistakes or typos which might be quite hard to track down later on, and avoid any superfluous computation at runtime. */

```
all_pieces([piece_w, piece_wb, piece_wfb, piece_wf, piece_wsf,
  piece_wsfb, piece_wsb, piece_ws, piece_s,
  piece_sb, piece_sfb, piece_sf, piece_f,
  piece_fb, piece_b, piece_]).

piece(P) :-
  all_pieces(L), member(P, L).
```

/** **List representations of the pieces.**
For each piece, we need a representation telling us which characteristics the piece has, but also which ones it does not have, since *not* having a characteristic is a characteristic in itself. In what follows, any piece P is assimilated to the set of its characteristics and negative-characteristics as per this representation. Again, the code of this predicate has been generated by an external program. */

```
representation(piece_w   , [car_w, nocar_s, nocar_f, nocar_b]  ).
representation(piece_wb  , [car_b, car_w, nocar_s, nocar_f]    ).
representation(piece_wfb , [car_b, car_f, car_w, nocar_s]      ).
representation(piece_wf  , [car_f, car_w, nocar_s, nocar_b]    ).
representation(piece_wsf , [car_f, car_s, car_w, nocar_b]      ).
representation(piece_wsfb, [car_b, car_f, car_s, car_w]        ).
representation(piece_wsb , [car_b, car_s, car_w, nocar_f]      ).
representation(piece_ws  , [car_s, car_w, nocar_f, nocar_b]    ).
representation(piece_s   , [car_s, nocar_w, nocar_f, nocar_b]  ).
representation(piece_sb  , [car_b, car_s, nocar_w, nocar_f]    ).
representation(piece_sfb , [car_b, car_f, car_s, nocar_w]      ).
representation(piece_sf  , [car_f, car_s, nocar_w, nocar_b]    ).
representation(piece_f   , [car_f, nocar_w, nocar_s, nocar_b]  ).
representation(piece_fb  , [car_b, car_f, nocar_w, nocar_s]    ).
representation(piece_b   , [car_b, nocar_w, nocar_s, nocar_f]  ).
representation(piece_    , [nocar_w, nocar_s, nocar_f, nocar_b]).

% does this piece have that characteristic ?
has_car(Piece, Car) :-
  representation(Piece, R), member(Car, R).
```

/** **Representation of the game board**
A square is represented as a couple $(x, y) \in [\![1, 4]\!]^2$, where x is the line and y the column.

A board is a list `[ASS, REMP, REMS]` where `ASS` is an association list which to a square associates the piece on this square, if there is one, `REMP` is the list of all remaining (available) pieces, and `REMS` is the list of all remaining (free) squares.

`REMP` and `REMS` could be deduced from `ASS`, but it is convenient to have easy access to that information. */

```
% all possible squares on the board
squares([1-1, 1-2, 1-3, 1-4, 2-1, 2-2, 2-3, 2-4,
  3-1, 3-2, 3-3, 3-4, 4-1, 4-2, 4-3, 4-4]).

% true if S is a valid square
square(S) :-
  squares(Ss),
  member(S, Ss).
```

/** **All possible winning figures on the board**
We call *figure* a set of four squares arranged in line, diagonal, column, straight or rotated square. The list of all possible figures, associated with their nature, has been generated by an external program. */

```
% figure(T, F) is true iff F is a figure of type T
figure(fig_small_square, [1-1, 1-2, 2-1, 2-2]).
figure(fig_small_square, [1-2, 1-3, 2-2, 2-3]).
figure(fig_small_square, [1-3, 1-4, 2-3, 2-4]).
```

```
figure(fig_small_square, [2-1, 2-2, 3-1, 3-2]).
figure(fig_small_square, [2-2, 2-3, 3-2, 3-3]).
figure(fig_small_square, [2-3, 2-4, 3-3, 3-4]).
figure(fig_small_square, [3-1, 3-2, 4-1, 4-2]).
figure(fig_small_square, [3-2, 3-3, 4-2, 4-3]).
figure(fig_small_square, [3-3, 3-4, 4-3, 4-4]).
figure(fig_align       , [1-1, 1-2, 1-3, 1-4]).
figure(fig_align       , [1-1, 2-1, 3-1, 4-1]).
figure(fig_align       , [1-1, 2-2, 3-3, 4-4]).
figure(fig_align       , [1-2, 2-2, 3-2, 4-2]).
figure(fig_align       , [1-3, 2-3, 3-3, 4-3]).
figure(fig_align       , [1-4, 2-3, 3-2, 4-1]).
figure(fig_align       , [1-4, 2-4, 3-4, 4-4]).
figure(fig_align       , [2-1, 2-2, 2-3, 2-4]).
figure(fig_align       , [3-1, 3-2, 3-3, 3-4]).
figure(fig_align       , [4-1, 4-2, 4-3, 4-4]).
figure(fig_bigsquare   , [1-1, 1-3, 3-1, 3-3]).
figure(fig_bigsquare   , [1-1, 1-4, 4-1, 4-4]).
figure(fig_bigsquare   , [1-2, 1-4, 3-2, 3-4]).
figure(fig_bigsquare   , [2-1, 2-3, 4-1, 4-3]).
figure(fig_bigsquare   , [2-2, 2-4, 4-2, 4-4]).
figure(fig_rotated     , [1-2, 2-1, 2-3, 3-2]).
figure(fig_rotated     , [1-3, 2-2, 2-4, 3-3]).
figure(fig_rotated     , [2-2, 3-1, 3-3, 4-2]).
figure(fig_rotated     , [2-3, 3-2, 3-4, 4-3]).
figure(fig_rotated     , [1-2, 2-4, 3-1, 4-3]).
figure(fig_rotated     , [1-3, 2-1, 3-4, 4-2]).
```

% get an initial, empty board.

```
empty_board([ASS, REMP, REMS]) :-
  empty_assoc(ASS), all_pieces(REMP), squares(REMS).
```

% place a piece on the board and get a new board.

```
place_piece([ASS, REMP, REMS], Piece, Position, NewB) :-
  put_assoc(Position, ASS, Piece, NASS),
  select(Piece, REMP, NREMP),
  select(Position, REMS, NREMS),
  NewB = [NASS, NREMP, NREMS].
```

/** **Example game**
This is just a "saved" game, very useful to try things out... */
% http://quarto.freehostia.com/fr/choix-piece.php?N=3&C=12&P=3&B0=-1&B1=5&B2=-1&B3=-1&B4=-1&B5=0&B6=-1&B7=2&B8=-1&B9=-1&B10=6&B11=-1&B12=-1&B13=-1&B14=-1&B15=-1

```
some_board([ASS, REMP, REMS]) :-
  list_to_assoc([
    1-2-piece_wb, 2-2-piece_wsfb, 2-4-piece_wsf,
    3-3-piece_wf, 4-1-piece_ws], ASS),
  REMP = [piece_w, piece_wfb,
```

```
  piece_wsb, piece_s,
  piece_sb, piece_sfb, piece_sf, piece_f,
  piece_fb, piece_b, piece_],
  REMS = [1-1, 1-3, 1-4, 2-1, 2-3,
  3-1, 3-2, 3-4, 4-2, 4-3, 4-4].
```

/** **Manipulation of the game** */

```
% true if a piece P has been set on square S
piece_on_square(Ass, S, P) :-
  get_assoc(S, Ass, P).

% true if a piece P has been set somewhere on figure F
piece_on_figure(Ass, P, F) :-
  figure(_, F),
  member(Square, F),
  piece_on_square(Ass, Square, P).
```

/** **Groups**: we name *group* a set of pieces placed on a figure on the board. A group therefore has at most four elements */

```
% true if G is the group of pieces currently placed on figure F
group_of_figure(Ass, F, G) :-
  setof(P, piece_on_figure(Ass, P, F), G).
```

/** true if `Car` is a shared characteristic of group G. That is, `Car` $\in \bigcap G$. It follows that the empty group has all characteristics in C – which might seem odd at first glance. */

```
shared_carac([], _).
shared_carac(Group, Car) :-
  Group = [P|Ps], has_car(P, Car),
  shared_carac(Ps, Car).

% true if Cars is the list of characteristics common to the group G
% the empty group has no characteristics in common though.
% UNUSED
shared_caracs([], []).
shared_caracs(G, Cars) :-
  setof(C, shared_carac(G, C), Cars).
```

/** **Critical groups and completing pieces**:
A group G is called *critical* if $\bigcap G \neq \emptyset$ and $\operatorname{card} G = 3$.
A *completing piece* to a group G is a piece P such that $\bigcap (\{P\} \cup G) \neq \emptyset$.
Adding a completing piece to a critical group wins the game. Unsurprisingly, the piece is then called a *critical piece*. */

```
% true if P completes the group, that is to say
% has a characteristic which is shared by the group
completing_piece(P, G) :-
```

```
  shared_carac(G, C), has_car(P, C).


% true if G is a critical group on figure F,
% which can be completed by an available piece P
critical_group(Board, F, G, P) :-
  Board = [ASS, REMP, _],
  group_of_figure(ASS, F, G),
  length(G, 3),
  completing_piece(P, G),
  member(P, REMP).

% true if S is a square which is free on figure F
free_square(Ass, Fig, S) :-
  member(S, Fig),
  \+piece_on_square(Ass, S, _).
```

/** **The minimal algorithm**:
The approach chosen here is the simplest and most primitive possible, with no lookahead. We have two different kinds of problems to deal with:

- ◇ Choose a piece P, not knowing where it will go.
  We need to choose P non-critical, since failing that our opponent shall surely win on next move. So we simply select a non-critical remaining piece, if there exists one. If not, we have lost or the game is a draw. This is done by `choose_piece`.

- ◇ Choose a square for a given piece P.
  We have two options: either P is critical, in which case we must not miss the opportunity to win, or it is not, in which case we are careful to choose a square such that not all remaining pieces become critical, which would mean loosing on next move. This is done by `choose_square`.

Verification of the validity of our opponent's moves is made using the same predicates.
*/

```
% choose a piece for our opponent. Don't hand over a critical piece
choose_piece(Board, P) :-
  Board = [_,REMP,_],
  member(P, REMP),
  \+critical_group(Board, _, _, P).

% there is no good piece, we have lost, hand over a bad piece and weep...
choose_piece(Board, P) :-
  Board = [_,REMP,_],
  member(P, REMP).

% is this piece available ?
is_available_piece([_,REMP,_], P) :-
  member(P, REMP).
```

```
% is this piece available ?
is_available_square([_,_,REMS], S) :-
  member(S, REMS).

% choose a square for a piece in case it is a winning move
choose_square(Board, P, S, Move, NBoard) :-
  Board = [ASS,_,REMS],
  critical_group(Board, F, _, P),
  free_square(ASS, F, S),
  member(S, REMS),
  figure(Move, F),
  place_piece(Board, P, S, NBoard).

% choose a square for a piece in case there is no winning move
% the square is chosen such that there are still available noncritical
% pieces to hand over to the opponent on next move.
choose_square(Board, P, S, fig_none, NBoard) :-
  place_piece(Board, P, S, NBoard),
  choose_piece(NBoard, _).
```

/** A draw happens iff we are choosing a square where to put a piece, but are unable to find anyplace where it wins the game for us. This predicate is only called if `choose_square` fails in which case, if it was the last piece, it's a draw */

```
is_last_piece(Board, P) :-
  Board = [_,[P],_].
```

/** **TESTING** */

```
% some_board(Board), choose_square(Board, P, S, F, NBoard).
```