Cage

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo Cage_1:
Rui Gonçalves - 201201775
Bruno Santos - 201402962

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

13 de Novembro de 2016

Resumo

O problema que nos foi proposto foi o desenvolvimento de um jogo de tabuleiro, nomeadamente o jogo *Cage*, com o objetivo de consolidar os temas abordados nas aulas práticas. No final deste trabalho era expectável que nós ficássemos com um conhecimento mais aprofundado da linguagem Prolog e do paradigma lógico.

Começámos por dividir o problema em problemas mais pequenos, fazendo primeiro os predicados mais básicos que o jogo iria ter, como o menu inicial, o tabuleiro inicial e a impressão deste no ecrã. Posteriormente, focámo-nos o modo de jogo Humano contra Humano, onde implementámos as movimentações das peças por parte dos jogadores e a verificação de movimentos, com toda a lógica do jogo inerente. De seguida implementámos o Bot que iria estar presente nos modos de Jogo Humano contra Computador e Computador contra Computador. O modo de jogo Humano contra Computador tem apenas um nível de dificuldade, em que o Bot toma uma decisão aleatória.

Conteúdo

1 Introdução

No âmbito da Unidade Curricular de Programação em Lógica, do 3º ano do Mestrado Integrado de Engenharia Informática e Computação da Faculdade de Engenharia da Universidade do Porto, foi-nos pedido para desenvolvermos um jogo de tabuleiro, *Cage*, usando para isso a linguagem de programação Prolog, com o objetivo de nos familiarizarmos com os princípios básicos desta mesma linguagem, tais como o backtracking, uso de listas e do operador cut.

O relatório está estruturado em 4 pontos fundamentais. O primeiro destes pontos visa descrever sucintamente o jogo. O segundo aborda toda a parte lógica do jogo, a representação do tabuleiro, as jogadas possíveis e como funciona o jogo, do início ao fim. O terceiro tem por objetivo descrever o módulo de interface com o utilizador em modo de texto. Por fim, no quarto ponto iremos fazer as conclusões do projeto, o como achamos que poderíamos ter melhorado o trabalho desenvolvido.

2 O Jogo *Cage*

História

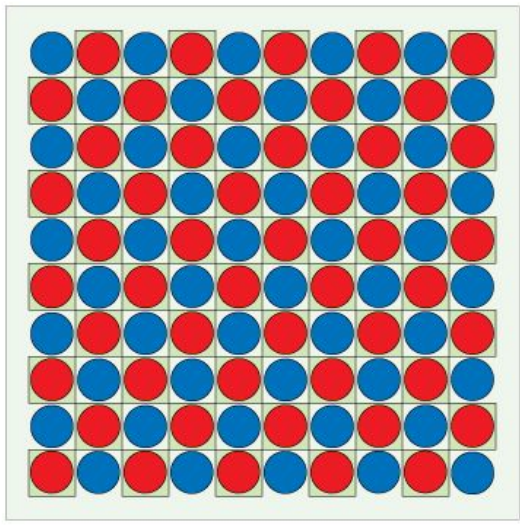
Cage é um jogo de estratégia abstrato para dois jogadores. Criado por Mark Steere a Maio de 2010, cage é um jogo de tabuleiro finito e omnidirecional com uma duração média de 30 minutos e idade mínima recomendada de 12 anos.

Objectivo

O jogo é composto por um tabuleiro 10x10 e 100 peças (50 peças azuis; 50 peças vermelhas) de forma cilíndrica, como nas Damas. O objetivo do jogo é capturar todas as peças do adversário, de forma a que no tabuleiro apenas restem peças da nossa cor, evitando ao mesmo tempo que o adversário faça o mesmo, sendo por isso considerado um jogo de aniquilação. O jogo acaba quando um dos jogadores o fizer, sendo considerado o vencedor.

Regras

A imagem à direita representa o estado inicial do tabuleiro. O jogador Vermelho começa o jogo, movimentando uma das suas peças, seguido do jogador Azul, jogando assim por turnos. Se um jogador tiver uma jogada possível, ele tem de jogar, mas caso não tenha, este deve esperar até poder efectuar uma jogada, consoante as jogadas do adversário, sendo que garantidamente pelo menos um dos jogadores pode efetuar uma jogada.



Existem 4 tipos de movimentos possíveis no *Cage*, sendo que os jogadores apenas podem movimentar as suas peças de acordo com as regras impostas por esses movimentos. Esses movimentos são *restricted*, *centering*, *adjoining*, *jump*.

Restricted

No movimento *Restricted* existem duas restrições aos movimentos do jogador:

- Não se pode mover uma peça de modo a que esta fique ortogonal a outra peça da mesma cor, nem mesmo momentaneamente, aquando de uma jogada múltipla.
- Não se pode mover uma peça que tenha uma peça adversária ortogonal para uma casa onde não tenha, a não ser que esse movimento seja um Jump.

Centering

No movimento *Centering* uma peça pode ser movimentada para uma casa desocupada que seja adjacente (verticalmente, horizontalmente ou diagonalmente) que leve a que a peça movimentada fique mais perto do centro do tabuleiro, isto é, que a distância em linha reta entre a peça e o ponto central do tabuleiro diminua.

Adjoing

No movimento Adjoining uma peça pode, caso não tenha nenhuma peça adversária ortogonal, ir para uma casa adjacente em que passe a ter uma peça adversária ortogonal.

Jump

Por fim, no movimento *Jump* pode-se mover uma peça da nossa cor para capturar uma peça adversária se esta estiver ortogonal para uma casa ortogonal à peça adversária do lado oposto, deste que esta casa esteja desocupada. Assim que isto acontecer, a peça adversária deve ser retirada do tabuleiro.

É também possível efetuar este movimento para capturar uma peça adversária que esteja no limite do tabuleiro com uma peça da sua cor que esteja numa casa ortogonal à peça adversária do lado oposto ao limite do tabuleiro. Desta forma, ambas as peças são removidas do tabuleiro.

A jogada de capturar uma peça adversária com *Jump* não é obrigatória, a não ser que esta seja a única jogada possível naquele turno.

Se for possível usar a peça que usámos para efetuar um movimento Jump para realizar outro movimento Jump, somos obrigados a fazê-lo, as vezes que forem precisas, pelo que a jogada acaba quando não for possível efetuar mais jogadas *Jump*.

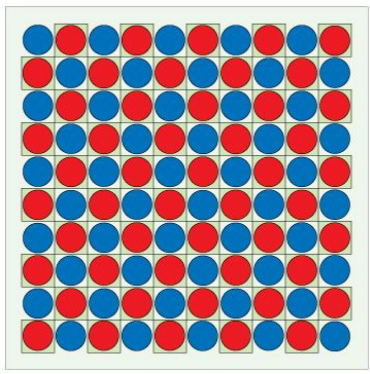
Se um jogador efetuar uma jogada *Jump* com a sua última peça para capturar a(s) última(s) peça(s) do adversário e é também removida do tabuleiro, ficando deste modo o tabuleiro sem peças, esse jogador ganha o jogo.

3 Lógica do Jogo

3.1 Representação do Estado do Jogo

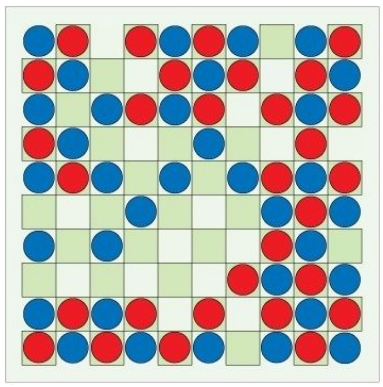
Inicialmente o tabuleiro terá todas as peças dispostas como se pode ver na figura anterior. Em qualquer estado do jogo, as casas apenas poderão ter uma peça azul, uma peça vermelha ou não ter peça nenhuma (casa vazia). Sendo assim, a representação inicial do tabuleiro em Prolog, utilizando uma lista de listas, será:

```
board(B) :- B=[[2,1,2,1,2,1,2,1,2,1],
              [1,2,1,2,1,2,1,2,1,2],
              [2,1,2,1,2,1,2,1,2,1],
              [1,2,1,2,1,2,1,2,1,2],
              [2,1,2,1,2,1,2,1,2,1],
              [1,2,1,2,1,2,1,2,1,2],
              [2,1,2,1,2,1,2,1,2,1],
              [1,2,1,2,1,2,1,2,1,2],
              [2,1,2,1,2,1,2,1,2,1],
              [1,2,1,2,1,2,1,2,1,2]].
```



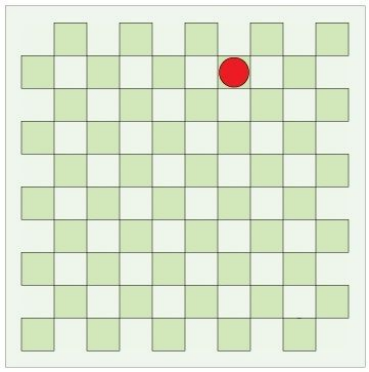
Uma representação intermédia do tabuleiro, tendo sempre em conta a legalidade das jogadas, poderá ser:

```
board(B) :- B=[[2,1,0,1,2,1,2,0,2,1],
              [1,2,0,0,1,2,1,0,1,2],
              [2,0,2,1,2,1,0,1,2,1],
              [1,2,0,0,0,2,0,0,1,0],
              [2,1,2,0,2,0,2,1,2,1],
              [0,0,0,2,0,0,0,2,1,2],
              [2,0,2,0,0,0,0,1,2,0],
              [0,0,0,0,0,0,1,2,1,2],
              [2,1,2,1,0,1,0,1,2,1],
              [1,2,1,2,1,2,0,2,1,2]].
```



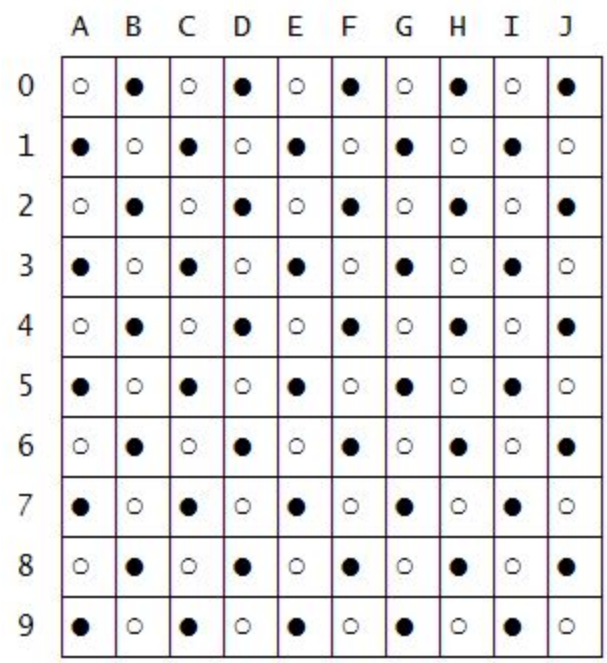
Por fim, uma representação final do tabuleiro, logo após a última jogada, poderá ser:

```
board(B) :- B=[[0,0,0,0,0,0,0,0,0,0],
              [0,0,0,0,0,0,1,0,0,0],
              [0,0,0,0,0,0,0,0,0,0],
              [0,0,0,0,0,0,0,0,0,0],
              [0,0,0,0,0,0,0,0,0,0],
              [0,0,0,0,0,0,0,0,0,0],
              [0,0,0,0,0,0,0,0,0,0],
              [0,0,0,0,0,0,0,0,0,0],
              [0,0,0,0,0,0,0,0,0,0],
              [0,0,0,0,0,0,0,0,0,0]].
```



3.2 Visualização do Tabuleiro

O tabuleiro de jogo será criado usando Unicode (através do predicado put_code) para representar tanto as casas como as peças. Recorre-se ao uso de letras e número para a representação de colunas e linhas, respectivamente. Exemplo de tabuleiro no princípio do jogo:



Para a construção do tabuleiro usou-se o predicado *print_board* (que permite a criação de tabuleiros com tamanho variável), e que recorre aos predicados *print_letters, print_top_lines*, *print_squares* e *print_bottom_lines* para a impressão da estrutura do tabuleiro, bem como as peças e as letras e número de identificação das casas.

3.3 Lista de Jogadas Válidas

Uma vez que existem três tipos de jogadas (Centering, Adjoin e Jump), na implementação achámos que seria mais simples de dividir a lista de jogadas possíveis em 3 listas diferentes, uma vez que cada movimento tem a sua lógica específica. Cada uma destas listas contém as jogadas possíveis para cada um dos 3 tipos de de jogadas existentes.

Para ir buscar a lista de jogadas possíveis da jogada Jump, implementámos o predicado *get_jump_positions*, que guarda em *AvailableMoves* a lista com todas as jogadas válidas para um jogador que se queira movimentar de uma posição inicial em que existe uma peça sua. Este predicado chama, por sua vez, 4 predicados, *check_jump_top*, *check_jump_left*, *check_jump_right* e *check_jump_bottom*, que guardam uma lista com os movimentos Jump possíveis, respetivamente, para cima, esquerda, direita e baixo.

```
get_jump_positions(Player, Board, Column, Line, AvailableMoves)
```

De seguida, para ir buscar a lista de jogadas possíveis da jogada Adjoin, implementámos o predicado *get_adjoin_positions*, que guarda em *AvailableMoves* a lista com todas as jogadas válidas para um jogador que queira efetuar uma jogada do tipo Adjoin.

```
get_adjoin_positions(Player, Board, Column, Line, AvailableMoves)
```

Por fim, para ir buscar a lista de jogadas possíveis da jogada Centering, implementámos o predicado *get_center_positions*, que guarda, também em *AvailableMoves*, a lista com todas as jogadas válidas para um jogador que queira efetuar uma jogada do tipo Centering.

```
get_center_positions(Player, Board, Column, Line, AvailableMoves)
```

3.4 Execução de Jogadas

A execução de uma jogada por um jogador está dividida em 3 grandes etapas. Primeiramente, pede-se ao jogador para introduzir o ID da casa inicial, através da utilização do predicado *read_position_from*, no qual são feitas as verificações necessárias (se a casa existe; se a peça que está na casa pertence ao jogador) e que vai buscar as listas com as casas para onde poderá ir.

```
read_position_from(Player, Board, Column, Line, JumpMoves, AdjoinMoves, CenterMoves)
```

Posteriormente, é pedido ao jogador que introduza o ID da casa para onde se quer dirigir, no caso de o movimento ser Centering ou Adjoin, ou o ID da casa da peça adversária que terá de "comer", com o recurso ao predicado *read_position_to*, que verifica se a casa introduzida existe e verifica se a casa introduzida pertence a uma das listas com as jogadas possíveis. Em ambos os casos, se o jogador introduzir uma casa inválida, será pedido que reintroduza uma casa válida.

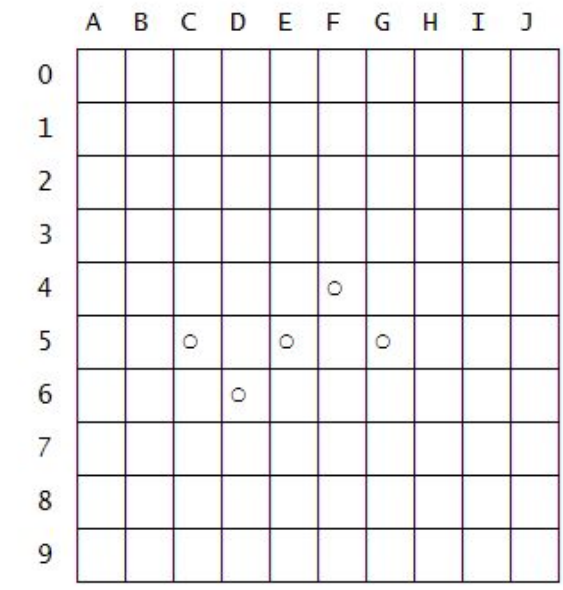
```
read_position_to(Player, Board, Column, Line, JumpMoves, AdjoinMoves, CenterMoves, Move)
```

Por fim, se tanto a casa inicial como a final introduzidas forem válidas, será efetuado o movimento em si, usando o predicado *move*, que recebe a posição inicial e final da peça a mover no tabuleiro, trocando a posição inicial por uma casa vazia e a final por uma casa com a peça selecionada.

```
move(Board, NewBoard, InitialColumn, InitialLine, FinalColumn, FinalLine)
```

3.5 Final do Jogo

A verificação de que se chegou ao final do jogo é feita sempre que se efetua um movimento. Assim que um jogador acaba de efetuar um movimento, é feita uma pesquisa para saber se existem peças do adversário ou nossas no tabuleiro. Se existirem peças de ambos os jogadores, o jogo continua com a chamada do predicado de jogo novamente. Se existir de apenas um dos jogadores, o jogo acaba e é impresso no ecrã o jogador vencedor.




```
Player2 won!
yes
```

Um exemplo desta situação em que o modo de jogo é Player vs. Player e a última jogada foi Centering:

```
(member2d(Other, NewBoard) ->
    (member2d(Player, NewBoard) ->
        play_pvp(Other, NewBoard);
        (write('Player'), write(Other), write(' won!'), nl));
    (write('Player'), write(Player), write(' won!'), nl)));
```

Em que *member2d* verifica a existência de uma peça do jogador no tabuleiro.

3.6 Jogada do Computador

Para escolher a jogada pelo computador são utilizados os predicados *random_position_from* e *random_position_to*, que fazem a escolha de peças aleatórias no tabuleiro iniciais e finais. O primeiro destes predicados irá fazer uma escolha aleatória de uma casa que contenha uma peça sua.

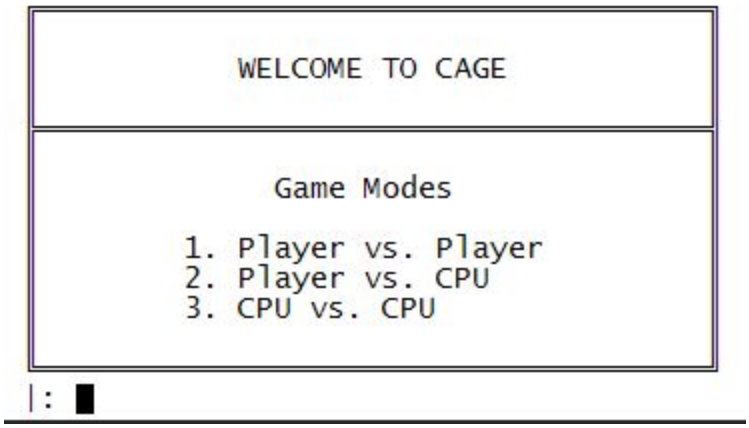
```
random_position_from(Player, Board, Column, Line, JumpMoves, AdjoinMoves, CenterMoves)
```

O segundo, *random_position_to*, irá fazer uma escolha, também aleatória, de uma posição final do movimento do computador, de entre as jogadas válidas presentes nas listas *JumpMoves*, *AdjoinMoves* e *CenterMoves* que, como já foi descrito anteriormente, contém no seu todo todas as jogadas possíveis para um jogador a partir da posição inicial, gerada em *random_position_form*.

```
random_position_to(Player, Board, Column, Line, JumpMoves, AdjoinMoves, CenterMoves, Move)
```

4 Interface com o Utilizador

Neste trabalho recorremos diversas vezes ao predicado *put_code* para utilizar caracteres Unicode, tendo em vista criar uma interface mais apelativa para o utilizador. No menu inicial do jogo é possível selecionar uma de 3 opções de jogo (Player vs. Player; Player vs. CPU; CPU vs. CPU):

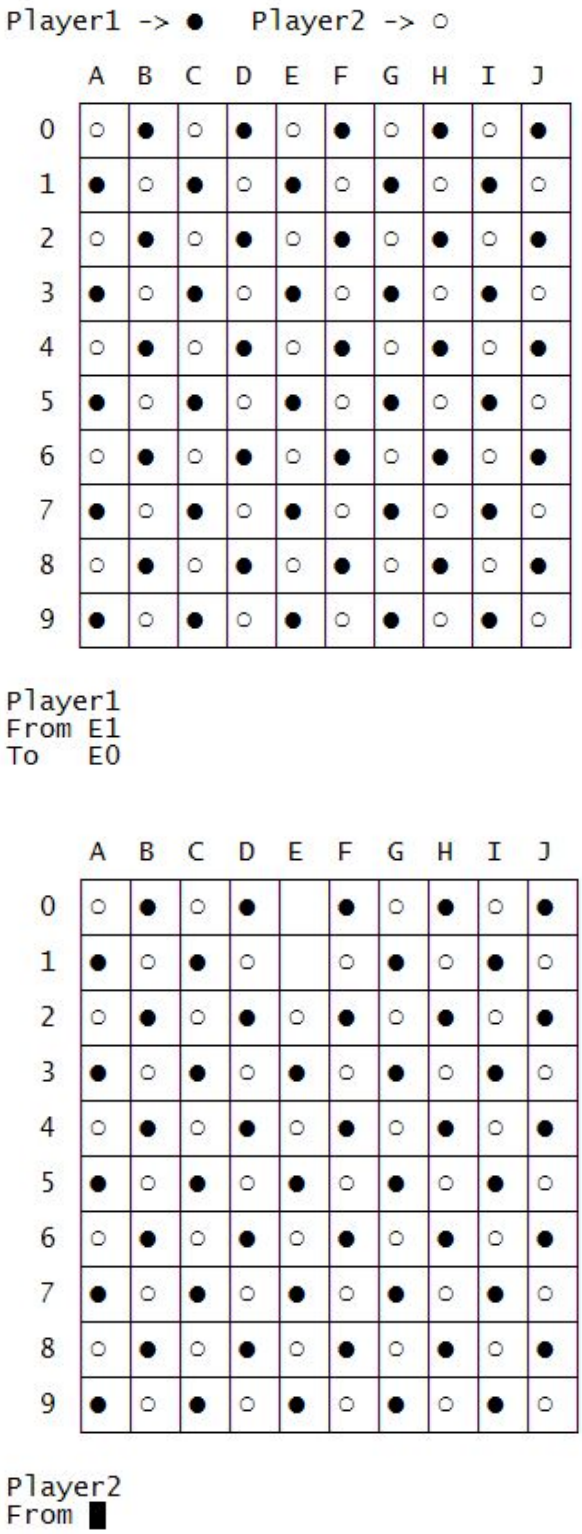


Escolhendo, por exemplo, a primeira opção, Player vs. Player, começará de imediato o jogo. As regras de *Cage* ditam que o jogador que começa o jogo é o jogador 1, no caso da nossa implementação, o jogador com as peças pretas. No início do jogo, aparece uma curta legenda a mostrar a que Jogadores pertencem cada uma das peças, seguida da impressão do tabuleiro em modo texto. Depois será pedido ao jogador que introduza as casas inicial e final para efetuar a jogada, de acordo com a lógica previamente descrita.

Será pedido para o jogador reintroduzir as casas, como já dito anteriormente, caso estas não sejam válidas. Segue um exemplo dessa interface no início do jogo:

```
Player1
From F1
From K3
From E1
To   D7
To   T9
To   E0
```

Segue um exemplo de um movimento Jump por parte do jogador 1 no início do jogo.



5 Conclusões

Consideramos que o projeto desenvolvido foi uma boa forma de adquirirmos conhecimentos aprofundados da linguagem Prolog, o que não seria possível apenas com os exercícios das aulas e, de uma forma mais abrangente, com o paradigma de Programação em Lógica. A linguagem de programação Prolog é muito diferente das linguagens Imperativas e Orientadas a Objetos a que estamos habituados, como C++ ou Java, mas ainda assim conseguimos ter sucesso no decorrer do desenvolvimento.

O trabalho poderia ter sido melhorado com a implementação de vários níveis de dificuldade do computador, como foi pedido, mas devido à falta de tempo não conseguimos implementar as funcionalidades todas.

Bibliografia

1. SICStus Prolog User's Manual, Mats Carlsson et al., Junho 2016
2. SWI-Prolog Documentation (http://www.swi-prolog.org/pldoc/doc_for?object=manual)
3. Cage, Rules of the Game, Mark Steere (http://www.marksteeregames.com/Cage_rules.html)

Anexo A (Código Prolog)

board_states.pl

```
* BOARD */

/* Declaration of the initial board */
board(B) :- B=[[2,1,2,1,2,1,2,1,2,1],
           [1,2,1,2,1,2,1,2,1,2],
           [2,1,2,1,2,1,2,1,2,1],
           [1,2,1,2,1,2,1,2,1,2],
           [2,1,2,1,2,1,2,1,2,1],
           [1,2,1,2,1,2,1,2,1,2],
           [2,1,2,1,2,1,2,1,2,1],
           [1,2,1,2,1,2,1,2,1,2],
           [2,1,2,1,2,1,2,1,2,1],
           [1,2,1,2,1,2,1,2,1,2]].
```

graphics.pl

```
/* GRAPHICS */

/* Print board */
/* parameters: Size - of list; Board - list that represents the
board */
print_board(Size, Board) :-
      nl,
      print_letters(Size, Size),
      print_top_lines(Size),
      print_squares(0, Size, Board),
      print_bottom_lines(Size).


/* Print letters on top of the board */
print_letters(Size, Size) :-
      Size > 0,
      write('    '),
      print_letter(Size, Size).

print_letter(1, Size) :-
      write(' '),
      C is 65+Size-1,
      put_code(C),
      write(' '), nl.

print_letter(Line, Size) :-
      write(' '),
      C is 65+Size-Line,
      put_code(C),
      write(' '),
      Nextline is Line-1,
      print_letter(Nextline,Size).


/* Print the top line of the board */
print_top_lines(Column) :-
      Column > 0,
```

```
      write('    '),
      lt_corner,
      print_top_line(Column).

print_top_line(1) :-
      horiz,
      horiz,
      rt_corner, nl.

print_top_line(Column) :-
      horiz,
      horiz,
      top_con,
      Nextcolumn is Column-1,
      print_top_line(Nextcolumn).


/* Print the bottom line line of the board */
print_bottom_lines(Column) :-
      Column > 0,
      write('    '),
      lb_corner,
      print_bottom_line(Column).

print_bottom_line(1) :-
      horiz,
      horiz,
      rb_corner, nl.

print_bottom_line(Column) :-
      horiz,
      horiz,
      bottom_con,
      Nextcolumn is Column-1,
      print_bottom_line(Nextcolumn).


/* Print the middle of the board */
print_squares(_, _, []).

print_squares(0, Size, [Line|Board]) :-
      print_pieces(0, Size, Line),
```

```
      print_squares(1, Size, Board).

print_squares(Currentline, Size, [Line|Board]) :-
      print_middle_lines(Size),
      print_pieces(Currentline, Size, Line),
      Nextline is Currentline+1,
      print_squares(Nextline, Size, Board).


/* Print the horizontal lines and connectors of the board */
print_middle_lines(Size) :-
      Size > 0,
      write('    '),
      left_con,
      print_middle_line(Size).

print_middle_line(1) :-
      horiz,
      horiz,
      right_con, nl.

print_middle_line(Size) :-
      horiz,
      horiz,
      middle,
      Nextsize is Size-1,
      print_middle_line(Nextsize).


/* Print the pieces and the vertical lines */
print_pieces(Currline, _, Line) :-
      write('  '),
      write(Currline),
      write(' '),
      print_piece(Line),
      vert, nl.

print_piece([]).

print_piece([0|Line]) :-
      vert,
      write('  '),
```

```
      print_piece(Line).

print_piece([1|Line]) :-
      vert,
      black_circle,
      write(' '),
      print_piece(Line).

print_piece([2|Line]) :-
      vert,
      white_circle,
      write(' '),
      print_piece(Line).


/* Characteres */

lt_corner :- put_code(9484).
rt_corner :- put_code(9488).
lb_corner :- put_code(9492).
rb_corner :- put_code(9496).
horiz :- put_code(9472).
vert :- put_code(9474).
top_con :- put_code(9516).
bottom_con :- put_code(9524).
left_con :- put_code(9500).
right_con :- put_code(9508).
middle :- put_code(9532).

double_lt_corner :- put_code(9556).
double_rt_corner :- put_code(9559).
double_lb_corner :- put_code(9562).
double_rb_corner :- put_code(9565).
double_vert :- put_code(9553).
double_horiz :- put_code(9552).
double_left_con :- put_code(9568).
double_right_con :- put_code(9571).

black_circle :- put_code(11044).
white_circle :- put_code(11093).

double_horiz(0).
```

```
double_horiz(N) :- double_horiz, N1 is N-1, double_horiz(N1).

space(0).
space(N) :- write(' '), N1 is N-1, space(N1).
```

utilities.pl

```
/* UTILITIES */

/* Check if Elem is in a 2d List */
member2d(_, []) :- fail.

member2d(Elem, [Line|List2d]) :-
     (member(Elem, Line) -> true; member2d(Elem, List2d)).

/* Return the elem in a 2d list position; return 3 if out of list
position */
get_piece(Board, Line, Column, Piece) :-
     nth0(Line, Board, PieceLine), /* get the line of the piece */
     nth0(Column, PieceLine, Piece). /* get the piece ID to be
moved */

get_piece(_, _, _, Piece) :- Piece is 3.


/* On a 2d list, moves value at initial position to final position
*/

move(Board, NewBoard, InitialColumn, InitialLine, FinalColumn,
FinalLine) :-
     get_piece(Board, InitialLine, InitialColumn, Piece),
     change_board_position(Board, NewBoardTemp, 0, InitialColumn,
InitialLine, 0),
     change_board_position(NewBoardTemp, NewBoard, 0, FinalColumn,
FinalLine, Piece).


/* Changes the position ColumnNr, LineNr of a 2d list to Piece */
change_board_position([], [], _, _, _, _).

change_board_position([Line|Board], [Line|NewBoard], Count,
ColumnNr, LineNr, Piece) :-
     Count \= LineNr,
     NextCount is Count+1,
```

```
      change_board_position(Board, NewBoard, NextCount, ColumnNr,
LineNr, Piece).

change_board_position([Line|Board], NewBoard, Count, ColumnNr,
LineNr, Piece) :-
      change_line_position(Line, NewLine, 0, ColumnNr, Piece),
      NextCount is Count+1,
      change_board_position([NewLine|Board], NewBoard, NextCount,
ColumnNr, LineNr, Piece).


/* Changes the position ColumnNr of a list to Piece */
change_line_position([], [], _, _, _).

change_line_position([Position|Line], [Position|NewLine], Count,
ColumnNr, Piece) :-
      Count \= ColumnNr,
      NextCount is Count+1,
      change_line_position(Line, NewLine, NextCount, ColumnNr,
Piece).

change_line_position([_|Line], [Piece|NewLine], Count, ColumnNr,
Piece) :-
      NextCount is Count+1,
      change_line_position(Line, NewLine, NextCount, ColumnNr,
Piece).
```

rules.pl

```
/* RULES */

/*=====================================================================
=====================================================================
==*/

/* RESTRICTIONS */

% Two pieces with the same color can never be adjacent
restriction1(Player, Board, Column, Line) :-
      Right is Column+1, Left is Column-1, Top is Line-1, Bottom is
Line+1,
      get_piece(Board, Top, Column, Piece0) , !,
      Piece0 \= Player,
      get_piece(Board, Bottom, Column, Piece1), !,
      Piece1 \= Player,
      get_piece(Board, Line, Right, Piece2), !,
      Piece2 \= Player,
      get_piece(Board, Line, Left, Piece3), !,
      Piece3 \= Player.

restriction1(Player, Board, Column, Line, NewColumn, NewLine) :-
      Right is NewColumn+1, Left is NewColumn-1, Top is NewLine-1,
Bottom is NewLine-1,
      get_piece(Board, Top, NewColumn, Piece0) , !,
      ((Column == NewColumn, Line == Top) -> (Piece0 == Player);
(Piece0 \= Player)),
      get_piece(Board, Bottom, NewColumn, Piece1), !,
      ((Column == NewColumn, Line == Bottom) -> (Piece1 == Player);
(Piece1 \= Player)),
      get_piece(Board, NewLine, Right, Piece2), !,
      ((Column == Right, Line == NewLine) -> (Piece2 == Player);
(Piece2 \= Player)),
      get_piece(Board, NewLine, Left, Piece3), !,
      ((Column == Left, Line == NewLine) -> (Piece3 == Player);
(Piece3 \= Player)).
```

```
% Check if the piece is adjacent to one of the opponent's
check_adjacency(Player, Board, Column, Line, Adjacency) :-
      Right is Column+1, Left is Column-1, Top is Line-1, Bottom is
Line+1,
      Other is ((Player mod 2) + 1),
      get_piece(Board, Top, Column, Piece0) , !,
      get_piece(Board, Bottom, Column, Piece1), !,
      get_piece(Board, Line, Right, Piece2), !,
      get_piece(Board, Line, Left, Piece3), !,
      (((Piece0 == Other); (Piece1 == Other); (Piece2 == Other);
(Piece3 == Other)) -> (Adjacency='yes'); (Adjacency='no')).

% Check if the new position also as adjacencies
restriction2(_, _, _, _, 'no').

restriction2(Player, Board, Column, Line, 'yes') :-
      check_adjacency(Player, Board, Column, Line, Adjacency),
      Adjacency == 'yes'.


/*=================================================================
================================================================
==*/
/* JUMP */

get_jump_positions(Player, Board, Column, Line, AvailableMoves) :-
      Other is ((Player mod 2) + 1),
      Empty is 0,
      check_jump_top(Player, Board, Column, Line, Other, Empty,
Top),
      append(Top,[], Temp1),
      check_jump_left(Player, Board, Column, Line, Other, Empty,
Left),
      append(Left, Temp1, Temp2),
      check_jump_right(Player, Board, Column, Line, Other, Empty,
Right),
      append(Right, Temp2, Temp3),
      check_jump_bottom(Player, Board, Column, Line, Other, Empty,
Bottom),
      append(Bottom, Temp3, AvailableMoves).
```

```
% Top
% Jump out
check_jump_top(_, Board, Column, 1, Other, _, Top) :-
      get_piece(Board, 0, Column, Jumped),
      Jumped =:= Other,
      Top=[Column].

check_jump_top(Player, Board, Column, Line, Other, Empty, Top) :-
      L1 is Line-1, L2 is Line-2,
      restriction1(Player, Board, Column, L2),
      get_piece(Board, L1, Column, Jumped),
      Jumped =:= Other,
      get_piece(Board, L2, Column, Dest),
      Dest =:= Empty,
      T is (10*(Line-1)+Column),
      Top=[T].

check_jump_top(_, _, _, _, _, _, Top) :- Top=[].

% Bottom
% Jump out
check_jump_bottom(_, Board, Column, 8, Other, _, Bottom) :-
      get_piece(Board, 9, Column, Jumped),
      Jumped =:= Other,
      B is (90+Column),
      Bottom=[B].

check_jump_bottom(Player, Board, Column, Line, Other, Empty, Bottom)
:-
      L1 is Line+1, L2 is Line+2,
      restriction1(Player, Board, Column, L2),
      get_piece(Board, L1, Column, Jumped),
      Jumped =:= Other,
      get_piece(Board, L2, Column, Dest),
      Dest =:= Empty,
      B is (10*(Line+1)+Column),
      Bottom=[B].

check_jump_bottom(_, _, _, _, _, _, Bottom) :- Bottom=[].

% Left
```

```
% Jump out
check_jump_left(_, Board, 1, Line, Other, _, Left) :-
      get_piece(Board, Line, 0, Jumped),
      Jumped =:= Other,
      L is (Line*10),
      Left=[L].

check_jump_left(Player, Board, Column, Line, Other, Empty, Left) :-
      C1 is Column-1, C2 is Column-2,
      restriction1(Player, Board, C2, Line),
      get_piece(Board, Line, C1, Jumped),
      Jumped =:= Other,
      get_piece(Board, Line, C2, Dest),
      Dest =:= Empty,
      L is (10*Line+Column-1),
      Left=[L].

check_jump_left(_, _, _, _, _, _, Left) :- Left=[].

% Right
% Jump out
check_jump_right(_, Board, 8, Line, Other, _, Right) :-
      get_piece(Board, Line, 9, Jumped),
      Jumped =:= Other,
      R is (Line*10+9),
      Right=[R].

check_jump_right(Player, Board, Column, Line, Other, Empty, Right)
:-
      C1 is Column+1, C2 is Column+2,
      restriction1(Player, Board, C2, Line),
      get_piece(Board, Line, C1, Jumped),
      Jumped =:= Other,
      get_piece(Board, Line, C2, Dest),
      Dest =:= Empty,
      R is (10*Line+Column+1),
      Right=[R].

check_jump_right(_, _, _, _, _, _, Right) :- Right=[].
```

```
/*=================================================================
=================================================================
==*/

/* ADJOIN */

get_adjoin_positions(Player, Board, Column, Line, AvailableMoves) :-
      PL is Line-1, NL is Line+1, PC is Column-1, NC is Column+1,
      check_no_ortogonal(Board, Column, Line),
      get_adjacency(Player, Board, NC, NL, BR),
      get_adjacency(Player, Board, NC, Line, R),
      get_adjacency(Player, Board, NC, PL, TR),
      get_adjacency(Player, Board, Column, NL, B),
      get_adjacency(Player, Board, Column, PL, T),
      get_adjacency(Player, Board, PC, NL, BL),
      get_adjacency(Player, Board, PC, Line, L),
      get_adjacency(Player, Board, PC, PL, TL),
      Lists=[BR,R,TR,B,T,BL,L,TL],
      append(Lists, AvailableMoves).

get_adjoin_positions(_, _, _, _, AvailableMoves) :-
AvailableMoves=[].


check_no_ortogonal(Board, Column, Line) :-
      PL is Line-1, NL is Line+1, PC is Column-1, NC is Column+1,
      get_piece(Board, PL, Column, Piece1), !,
      (Piece1 =:= 0; Piece1 =:= 3),
      get_piece(Board, NL, Column, Piece2), !,
      (Piece2 =:= 0; Piece2 =:= 3),
      get_piece(Board, Line, NC, Piece3), !,
      (Piece3 =:= 0; Piece3 =:= 3),
      get_piece(Board, Line, PC, Piece4), !,
      (Piece4 =:= 0; Piece4 =:= 3).


get_adjacency(Player, Board, Column, Line, Position) :-
      get_piece(Board, Line, Column, Piece0),
      Piece0 =:= 0,
      Other is ((Player mod 2) + 1),
      PL is Line-1, NL is Line+1, PC is Column-1, NC is Column+1,
      get_piece(Board, PL, Column, Piece1),
```

```
      get_piece(Board, NL, Column, Piece2),
      get_piece(Board, Line, NC, Piece3),
      get_piece(Board, Line, PC, Piece4),
      (Piece1 =:= Other; Piece2 =:= Other; Piece3 =:= Other; Piece4
=:= Other),
      P is Line*10+Column,
      Position=[P].

get_adjacency(_, _, _, _, Position) :- Position=[].


/*=================================================================
================================================================
==*/

/* CENTERING */

get_center_positions(Player, Board, Column, Line, AvailableMoves) :-
      PL is Line-1, NL is Line+1, PC is Column-1, NC is Column+1,
      Dist is sqrt(abs(Column-4.5)^2+abs(Line-4.5)^2),
      check_adjacency(Player, Board, Column, Line, Adjacency),
      get_closer(Player, Board, Dist, Column, Line, NC, NL, BR,
Adjacency),
      get_closer(Player, Board, Dist, Column, Line, NC, Line, R,
Adjacency),
      get_closer(Player, Board, Dist, Column, Line, NC, PL, TR,
Adjacency),
      get_closer(Player, Board, Dist, Column, Line, Column, NL, B,
Adjacency),
      get_closer(Player, Board, Dist, Column, Line, Column, PL, T,
Adjacency),
      get_closer(Player, Board, Dist, Column, Line, PC, NL, BL,
Adjacency),
      get_closer(Player, Board, Dist, Column, Line, PC, Line, L,
Adjacency),
      get_closer(Player, Board, Dist, Column, Line, PC, PL, TL,
Adjacency),
      Lists=[BR,R,TR,B,T,BL,L,TL],
      append(Lists, AvailableMoves).

get_closer(Player, Board, Dist, Column, Line, NewColumn, NewLine,
Position, Adjacency) :-
```

```
        get_piece(Board, NewLine, NewColumn, Piece),
        Piece =:= 0,
        restriction1(Player, Board, Column, Line, NewColumn, NewLine),
        restriction2(Player, Board, NewColumn, NewLine, Adjacency),
        NewDist is sqrt(abs(NewColumn-4.5)^2+abs(NewLine-4.5)^2),
        NewDist < Dist,
        P is NewLine*10+NewColumn,
        Position=[P].

get_closer(_, _, _, _, _, _, _, Position, _) :- Position=[].
```

input.pl

```
/* READ GAME MODE */

get_menu_input(Option, Min, Max) :-
      get_code(O), skip_line,
      O > 47+Min, O < 49+Max,
      Option is O.

get_menu_input(Option, Min, Max) :- get_menu_input(Option, Min,
Max).


/* READ POSITION */

read_position_from(Player, Board, Column, Line, JumpMoves,
AdjoinMoves, CenterMoves) :-
      write('From '),
      get_code(C),
      get_code(L), skip_line,
      C > 64, C < 75,
      L > 47, L < 58,
      CNumber is C - 65,
      LNumber is L - 48,
      get_piece(Board, LNumber, CNumber, Piece),
      Piece =:= Player,
      get_jump_positions(Player, Board, CNumber, LNumber, JM),
      get_adjoin_positions(Player, Board, CNumber, LNumber, AM),
      get_center_positions(Player, Board, CNumber, LNumber, CM),
      (JM \= []; AM \= []; CM \= []),
      Column is CNumber,
      Line is LNumber,
      JumpMoves=JM,
      AdjoinMoves=AM,
      CenterMoves=CM.
```

```
read_position_from(Player, Board, Column, Line, JumpMoves,
AdjoinMoves, CenterMoves) :- read_position_from(Player, Board,
Column, Line, JumpMoves, AdjoinMoves, CenterMoves).


read_position_to(_, _, Column, Line, JumpMoves, AdjoinMoves,
CenterMoves, Move) :-
     write('To   '),
     get_code(C),
     get_code(L), skip_line,
     C > 64, C < 75,
     L > 47, L < 58,
     CNumber is C - 65,
     LNumber is L - 48,
     Pos is LNumber*10+CNumber,
     (member(Pos, AdjoinMoves) -> (Move='adjoin');
          member(Pos, CenterMoves) -> (Move='center');
               member(Pos, JumpMoves) -> (Move='jump')),
     Column is CNumber,
     Line is LNumber.

read_position_to(Player, Board, Column, Line, JumpMoves,
AdjoinMoves, CenterMoves, Move) :- read_position_to(Player, Board,
Column, Line, JumpMoves, AdjoinMoves, CenterMoves, Move).
```

play_pvp.pl

```
/* PLAY MODE PVP/

/* Play the game in Player vs Player mode */
play_pvp(Player, Board) :-
     write('Player'), write(Player), nl,
     read_position_from(Player, Board, InitialColumn, InitialLine,
JumpMoves, AdjoinMoves, CenterMoves),
     read_position_to(Player, Board, FinalColumn, FinalLine,
JumpMoves, AdjoinMoves, CenterMoves, Move),

     Other is ((Player mod 2) + 1),
     move(Board, NewBoard, InitialColumn, InitialLine, FinalColumn,
FinalLine),

     % Adjoining move -> player plays again
     ((Move=='adjoin') ->
          (print_board(10, NewBoard), nl,
          (member2d(Other, NewBoard) ->
               (member2d(Player, NewBoard) ->
                    play_pvp(Player, NewBoard);
                    (write('Player'), write(Other), write('
won!'), nl));
               (write('Player'), write(Player), write(' won!'),
nl)));

     % Centering move -> player passes the turn
     ((Move=='center') ->
          (print_board(10, NewBoard), nl,
          (member2d(Other, NewBoard) ->
               (member2d(Player, NewBoard) ->
                    play_pvp(Other, NewBoard);
                    (write('Player'), write(Other), write('
won!'), nl));
```

```
                (write('Player'), write(Player), write(' won!'),
nl)));

     % Jumping move -> player passes the turn if the selected piece
can't jump again
     ((Move=='jump') ->
          (DeltaLine is FinalLine-InitialLine,
          DeltaColumn is FinalColumn-InitialColumn,
          JumpLine is FinalLine+DeltaLine,
          JumpColumn is FinalColumn+DeltaColumn,
          move(NewBoard, JumpBoard, FinalColumn, FinalLine,
JumpColumn, JumpLine),
          print_board(10, JumpBoard), nl,
          (member2d(Other, JumpBoard) ->
               (member2d(Player, JumpBoard) ->
                    (get_jump_positions(Player, JumpBoard,
JumpColumn, JumpLine, DoubleJumpMoves),
                    ((DoubleJumpMoves == []) ->
                         play_pvp(Other, JumpBoard);
                         play_pvp_jump(Player, JumpBoard,
JumpColumn, JumpLine, DoubleJumpMoves)));
                    (write('Player'), write(Other), write('
won!'), nl));
               (write('Player'), write(Player), write(' won!'),
nl))))))).


play_pvp_jump(Player, Board, InitialColumn, InitialLine, JumpMoves)
:-
     write('Player'), write(Player), nl,
     read_position_to(Player, Board, FinalColumn, FinalLine,
JumpMoves, [], [], _),
     Other is ((Player mod 2) + 1),
     move(Board, NewBoard, InitialColumn, InitialLine, FinalColumn,
FinalLine),
     DeltaLine is FinalLine-InitialLine,
     DeltaColumn is FinalColumn-InitialColumn,
     JumpLine is FinalLine+DeltaLine,
     JumpColumn is FinalColumn+DeltaColumn,
     move(NewBoard, JumpBoard, FinalColumn, FinalLine, JumpColumn,
JumpLine),
     print_board(10, JumpBoard), nl,
```

```
      (member2d(Other, JumpBoard) ->
            (member2d(Player, JumpBoard) ->
                        (get_jump_positions(Player, JumpBoard,
JumpColumn, JumpLine, DoubleJumpMoves),
                        ((DoubleJumpMoves == []) ->
                              play_pvp(Other, JumpBoard);
                              play_pvp_jump(Player, JumpBoard,
JumpColumn, JumpLine, DoubleJumpMoves)));
                        (write('Player'), write(Other), write('
won!'), nl));
            (write('Player'), write(Player), write(' won!'), nl)).
```

play_pvc.pl

```
/* PLAY MODE PVC */

/* PLAYER */
play_pvc(Player, Board) :-
     (Player=:=1 ->
          (write('Player'), nl,
          read_position_from(Player, Board, InitialColumn,
InitialLine, JumpMoves, AdjoinMoves, CenterMoves),
          read_position_to(Player, Board, FinalColumn, FinalLine,
JumpMoves, AdjoinMoves, CenterMoves, Move));
          (write('CPU'), nl,
          sleep(1),
          random_position_from(Player, Board, InitialColumn,
InitialLine, JumpMoves, AdjoinMoves, CenterMoves),
          random_position_to(Player, Board, FinalColumn, FinalLine,
JumpMoves, AdjoinMoves, CenterMoves, Move))),

     Other is ((Player mod 2) + 1),
     move(Board, NewBoard, InitialColumn, InitialLine, FinalColumn,
FinalLine),

     % Adjoining move -> player plays again
     ((Move=='adjoin') ->
          (print_board(10, NewBoard), nl,
          (member2d(Other, NewBoard) ->
               (member2d(Player, NewBoard) ->
                    play_pvc(Player, NewBoard);
                    (write('Player'), write(Other), write('
won!'), nl));
               (write('Player'), write(Player), write(' won!'),
nl))));

     % Centering move -> player passes the turn
```

```
     ((Move=='center') ->
          (print_board(10, NewBoard), nl,
          (member2d(Other, NewBoard) ->
               (member2d(Player, NewBoard) ->
                    play_pvc(Other, NewBoard);
                    (write('Player'), write(Other), write('
won!'), nl));
               (write('Player'), write(Player), write(' won!'),
nl))));

     % Jumping move -> player passes the turn if the selected piece
can't jump again
     ((Move=='jump') ->
          (DeltaLine is FinalLine-InitialLine,
          DeltaColumn is FinalColumn-InitialColumn,
          JumpLine is FinalLine+DeltaLine,
          JumpColumn is FinalColumn+DeltaColumn,
          move(NewBoard, JumpBoard, FinalColumn, FinalLine,
JumpColumn, JumpLine),
          print_board(10, JumpBoard), nl,
          (member2d(Other, JumpBoard) ->
               (member2d(Player, JumpBoard) ->
                    (get_jump_positions(Player, JumpBoard,
JumpColumn, JumpLine, DoubleJumpMoves),
                    ((DoubleJumpMoves == []) ->
                         play_pvc(Other, JumpBoard);
                         play_pvc_jump(Player, JumpBoard,
JumpColumn, JumpLine, DoubleJumpMoves)));
                    (write('Player'), write(Other), write('
won!'), nl));
               (write('Player'), write(Player), write(' won!'),
nl))))))).


play_pvc_jump(Player, Board, InitialColumn, InitialLine, JumpMoves)
:-
     (Player=:=1 ->
          (write('Player'), nl,
          read_position_to(Player, Board, FinalColumn, FinalLine,
JumpMoves, [], [], Move));
          (write('CPU'), nl,
```

```
          random_position_to(Player, Board, FinalColumn, FinalLine,
JumpMoves, [], [], Move))),
     Other is ((Player mod 2) + 1),
     move(Board, NewBoard, InitialColumn, InitialLine, FinalColumn,
FinalLine),
     DeltaLine is FinalLine-InitialLine,
     DeltaColumn is FinalColumn-InitialColumn,
     JumpLine is FinalLine+DeltaLine,
     JumpColumn is FinalColumn+DeltaColumn,
     move(NewBoard, JumpBoard, FinalColumn, FinalLine, JumpColumn,
JumpLine),
     print_board(10, JumpBoard), nl,
     (member2d(Other, JumpBoard) ->
          (member2d(Player, JumpBoard) ->
                    (get_jump_positions(Player, JumpBoard,
JumpColumn, JumpLine, DoubleJumpMoves),
                    ((DoubleJumpMoves == []) ->
                         play_pvc(Other, JumpBoard);
                         play_pvc_jump(Player, JumpBoard,
JumpColumn, JumpLine, DoubleJumpMoves)));
                    (write('Player'), write(Other), write('
won!'), nl));
          (write('Player'), write(Player), write(' won!'), nl)).
```

play_cvc.pl

```
/* PLAY MODE cvc/

/* Play the game in Player vs Player mode */
play_cvc(Player, Board) :-
     write('CPU'), write(Player), nl,
     sleep(1),
     random_position_from(Player, Board, InitialColumn,
InitialLine, JumpMoves, AdjoinMoves, CenterMoves),
     random_position_to(Player, Board, FinalColumn, FinalLine,
JumpMoves, AdjoinMoves, CenterMoves, Move),

     Other is ((Player mod 2) + 1),
     move(Board, NewBoard, InitialColumn, InitialLine, FinalColumn,
FinalLine),

     % Adjoining move -> player plays again
     ((Move=='adjoin') ->
          (print_board(10, NewBoard), nl,
          (member2d(Other, NewBoard) ->
               (member2d(Player, NewBoard) ->
                    play_cvc(Player, NewBoard);
                    (write('CPU'), write(Other), write(' won!'),
nl));
               (write('CPU'), write(Player), write(' won!'), nl))));

     % Centering move -> player passes the turn
     ((Move=='center') ->
          (print_board(10, NewBoard), nl,
          (member2d(Other, NewBoard) ->
               (member2d(Player, NewBoard) ->
                    play_cvc(Other, NewBoard);
                    (write('CPU'), write(Other), write(' won!'),
nl));
```

```
                (write('CPU'), write(Player), write(' won!'), nl)));

     % Jumping move -> player passes the turn if the selected piece
can't jump again
     ((Move=='jump') ->
          (DeltaLine is FinalLine-InitialLine,
          DeltaColumn is FinalColumn-InitialColumn,
          JumpLine is FinalLine+DeltaLine,
          JumpColumn is FinalColumn+DeltaColumn,
          move(NewBoard, JumpBoard, FinalColumn, FinalLine,
JumpColumn, JumpLine),
          print_board(10, JumpBoard), nl,
          (member2d(Other, JumpBoard) ->
               (member2d(Player, JumpBoard) ->
                    (get_jump_positions(Player, JumpBoard,
JumpColumn, JumpLine, DoubleJumpMoves),
                    ((DoubleJumpMoves == []) ->
                         play_cvc(Other, JumpBoard);
                         play_cvc_jump(Player, JumpBoard,
JumpColumn, JumpLine, DoubleJumpMoves)));
                    (write('CPU'), write(Other), write(' won!'),
nl));
               (write('CPU'), write(Player), write(' won!'),
nl))))))).


play_cvc_jump(Player, Board, InitialColumn, InitialLine, JumpMoves)
:-
     write('CPU'), write(Player), nl,
     random_position_to(Player, Board, FinalColumn, FinalLine,
JumpMoves, [], [], _),
     Other is ((Player mod 2) + 1),
     move(Board, NewBoard, InitialColumn, InitialLine, FinalColumn,
FinalLine),
     DeltaLine is FinalLine-InitialLine,
     DeltaColumn is FinalColumn-InitialColumn,
     JumpLine is FinalLine+DeltaLine,
     JumpColumn is FinalColumn+DeltaColumn,
     move(NewBoard, JumpBoard, FinalColumn, FinalLine, JumpColumn,
JumpLine),
     print_board(10, JumpBoard), nl,
     (member2d(Other, JumpBoard) ->
```

```
           (member2d(Player, JumpBoard) ->
                     (get_jump_positions(Player, JumpBoard,
JumpColumn, JumpLine, DoubleJumpMoves),
                     ((DoubleJumpMoves == []) ->
                          play_cvc(Other, JumpBoard);
                          play_cvc_jump(Player, JumpBoard,
JumpColumn, JumpLine, DoubleJumpMoves)));
                     (write('CPU'), write(Other), write(' won!'),
nl));
           (write('CPU'), write(Player), write(' won!'), nl)).
```

bot.pl

```
/* BOT */

/* Choose move on the given board randomly */
random_position_from(Player, Board, Column, Line, JumpMoves,
AdjoinMoves, CenterMoves) :-
      random(0, 10, CNumber),
      random(0, 10, LNumber),
      get_piece(Board, LNumber, CNumber, Piece),
      Piece =:= Player,
      get_jump_positions(Player, Board, CNumber, LNumber, JM),
      get_adjoin_positions(Player, Board, CNumber, LNumber, AM),
      get_center_positions(Player, Board, CNumber, LNumber, CM),
      (JM \= []; AM \= []; CM \= []),
      Column is CNumber,
      Line is LNumber,
      JumpMoves=JM,
      AdjoinMoves=AM,
      CenterMoves=CM.


random_position_from(Player, Board, Column, Line, JumpMoves,
AdjoinMoves, CenterMoves) :- random_position_from(Player, Board,
Column, Line, JumpMoves, AdjoinMoves, CenterMoves).



random_position_to(_, _, Column, Line, JumpMoves, AdjoinMoves,
CenterMoves, Move) :-
      random(0, 10, CNumber),
      random(0, 10, LNumber),
      Pos is LNumber*10+CNumber,
      (member(Pos, AdjoinMoves) -> (Move='adjoin');
            member(Pos, CenterMoves) -> (Move='center');
                  member(Pos, JumpMoves) -> (Move='jump')),
```

```
      Column is CNumber,
      Line is LNumber.

random_position_to(Player, Board, Column, Line, JumpMoves,
AdjoinMoves, CenterMoves, Move) :- random_position_to(Player, Board,
Column, Line, JumpMoves, AdjoinMoves, CenterMoves, Move).
```

cage.pl

```
/* CAGE */

/* includes */
:- use_module(library(random)).
:- use_module(library(lists)).
:- use_module(library(system)).
:- ensure_loaded('board_states.pl').
:- ensure_loaded('graphics.pl').
:- ensure_loaded('utilities.pl').
:- ensure_loaded('rules.pl').
:- ensure_loaded('input.pl').
:- ensure_loaded('play_pvp.pl').
:- ensure_loaded('play_pvc.pl').
:- ensure_loaded('play_cvc.pl').
:- ensure_loaded('bot.pl').

cage :-
     nl,
     double_lt_corner, double_horiz(37), double_rt_corner, nl,
     double_vert, space(37), double_vert, nl,
     double_vert, space(11), write('WELCOME TO CAGE'), space(11),
double_vert, nl,
     double_vert, space(37), double_vert, nl,
     double_left_con, double_horiz(37), double_right_con, nl,
     main_menu.


/* MAIN MENU */
main_menu :-
     double_vert, space(37), double_vert, nl,
     double_vert, space(13), write('Game Modes'), space(14),
double_vert, nl,
     double_vert, space(37), double_vert, nl,
     double_vert, space(8), write('1. Player vs. Player'),
space(9), double_vert, nl,
```

```
      double_vert, space(8), write('2. Player vs. CPU'), space(12),
double_vert, nl,
      double_vert, space(8), write('3. CPU vs. CPU'), space(15),
double_vert, nl,
      double_vert, space(37), double_vert, nl,
      double_lb_corner, double_horiz(37), double_rb_corner, nl,
      get_menu_input(Option, 1, 3),
      NOption is Option-48,
      play_mode(NOption).

play_mode(1) :-
      board(B),
      nl, nl,
      write('Player1 -> '), black_circle, write('   '),
      write('Player2 -> '), white_circle, nl,
      print_board(10, B), nl,
      play_pvp(1, B).

play_mode(2) :-
      board(B),
      nl, nl,
      write('Player -> '), black_circle, write('   '),
      write('CPU -> '), white_circle, nl,
      print_board(10, B), nl,
      play_pvc(1, B).

play_mode(3) :-
      board(B),
      nl, nl,
      write('CPU1 -> '), black_circle, write('   '),
      write('CPU2 -> '), white_circle, nl,
      print_board(10, B), nl,
      play_cvc(1, B).
```