Master in Artificial Intelligence

CONNECT 4: A PROLOG IMPLEMENTATION WITH MINIMAX ALGORITHM, ALPHA-BETA PRUNING AND MAX-DEPTH PARAMETER

Project for the Languages and Algorithms for Artificial Intelligence course - Module 1

Alessandro Lombardini & Giacomo Melacini

February 2022

Contents

1 Introduction

As project for the Languages and Algorithms for Artificial Intelligence course, it is proposed a Prolog implementation of the famous game Connect 4 (Figure 1). It has been developed an Intelligent Agent (Alex), that is able to play against a skilled human player. To decide its moves, Alex uses the minimax algorithm with Alpha-Beta pruning and an heuristic specifically designed for this project. It is possible to play against the agent using the command line.

Figure 1: Connect 4

The code is available on GitHub at the following link: `https://github.com/Chavelanda/connect_four`

2 Connect 4

2.1 History

It seems that for as long as humans have been playing games, they have been racing to see who could line things up the fastest. There is evidence of alignment games being played as far back as 3500 years ago in ancient Egypt. These games were scratched into simple boards, and likely into dirt before that, and pieces would be placed in an effort to get three in a row before the opponent. Fast forward a few thousand years and while Captain James Cook was sailing around the world, he was spending much of his down time playing an early wooden version of Connect 4. It was only in 1974 that the Milton Bradley Company brought the plastic version of Connect 4 to the masses creating a cultural phenomenon. To Howard Wexler, a toy inventor, is given credit for creating the Connect 4 we all grew up playing.

2.2 How to play

Connect 4 (also known as 4 Up, Plot 4, Find 4, Captain's Mistress, 4 in a Row, Drop 4, and Gravitrips in the Soviet Union) is a two-player board game, in which the players choose a color and then take turns dropping colored discs into a seven-columns, six-rows grid. The pieces fall straight down, occupying the lowest available space within the column. The objective of the game is to be the first to form an horizontal, vertical, or diagonal line of four (or more) of one's own discs. The game ends in a draw when the grid has been entirely filled without any straight connection of four discs from a single player.

3 Two-persons, complete-information games

3.1 Games as trees

In this chapter will be considered techniques for playing two-person, complete information games, such as chess, go and Connect 4. In such games there are two players that make moves alternatively, and both players have complete information of the current situation in the game. The game is over when is reached a position that is qualified as terminal by the rules of the game - for example, mate in chess. The rules also determine what is the outcome of the game that has ended in this terminal position.
Such games can be represented by a tree. The nodes in such trees correspond to game situations, while the arcs correspond to valid moves. The initial situation of the game is the root node; leaves of the tree correspond to terminal positions. In most games of this type the outcome of the game can be win, loss or draw.
It is possible to calculate the worst case complexity of the tree of a Connect 4 game. At each turn, there are a maximum of seven valid moves and in a game there are a maximum of 42 turns. As shown in figure 2, there are $7^{42} = 3.11973482 \times 10^{35}$ leaf nodes in the tree (a very big number :)).

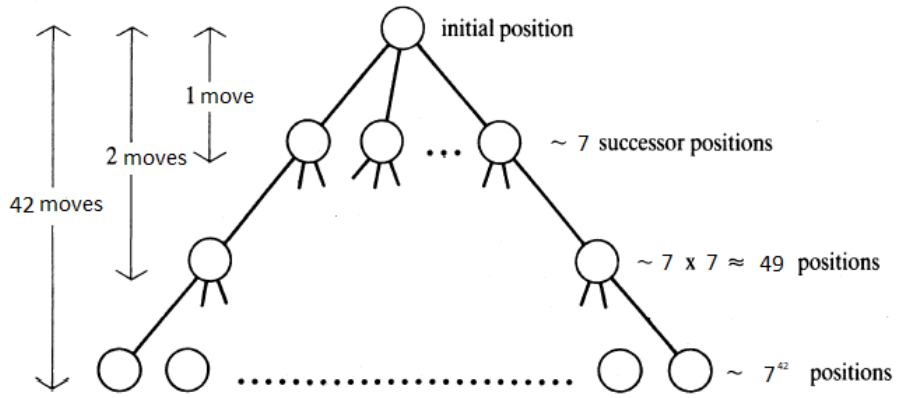


Figure 2: Complexity of the tree of Connect 4.

3.2 Minimax algorithm

Minimax is a decision-making algorithm, typically used in turn-based, two player games. The goal of the algorithm is to find the optimal next move. In the algorithm, one player is called the maximizer, while the other player is a minimizer. If we assign an evaluation score to the game board, one player tries to choose a game state with the maximum score, while the other chooses a state with the minimum score. In other words, the maximizer works to get the highest score, while the minimizer tries get the lowest score by trying to counter move. The algorithm starts with the root node and chooses the best possible node. Evaluation functions can assign scores only to terminal nodes (leaves). Therefore, leaves are recursively reached and their scores are back propagated. The algorithm is complete, meaning that in a finite search tree, a solution will be certainly found. It is also optimal if both the players are playing optimally, e.g. when both players use minimax.
Since the algorithm systematically visits all the positions in the search tree, the search becomes unfeasible when minimax is used in very large trees. Usually not all this work is necessary in order to correctly compute the minimax value of the root position. In the next section, a technique to make the algorithm more efficient will be shown.

3.3 Alpha-Beta pruning

Alpha–Beta pruning is a search algorithm that aims at decreasing the number of nodes that are evaluated by the minimax algorithm in its search tree. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.
The algorithm maintains two values, alpha and beta, which respectively represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of. Initially, alpha is negative infinity and beta is positive infinity, i.e. both players start with their worst possible score. Whenever the maximum score that the minimizing player (i.e. the "beta" player) is assured of becomes less than the minimum score that the maximizing player (i.e., the "alpha" player) is assured of (i.e. beta $<$ alpha), the maximizing player does not need to consider further descendants of this node, as they will never be reached in the actual play. For example, we can use this principle to reduce the search in the tree of Figure 3. The search process proceeds as follows:

1. Start with position *a*.
2. Move down to *b*.
3. Move down to *d*.
4. Take the maximum of *d*'s successors yelding 4.
5. Backtrack to *b* and move down to *e*.
6. Consider the first successor of e whose value is 5.
7. At this point MAX (who is to move in *e*) is guaranteed at least the value of 5 in position e regardless of other (possibly better) alternatives from *e*.
8. This is sufficient for MIN to realize that, at node *b*, the alternative *e* is worse w.r.t. d, even without knowing the exact value of *e*.
9. On these grounds, we can neglect the seconds successor of *e* and simply assign to e an approximate value 5. This approximation will, however, have no effect on the value of *b* and, hence, on *a*.

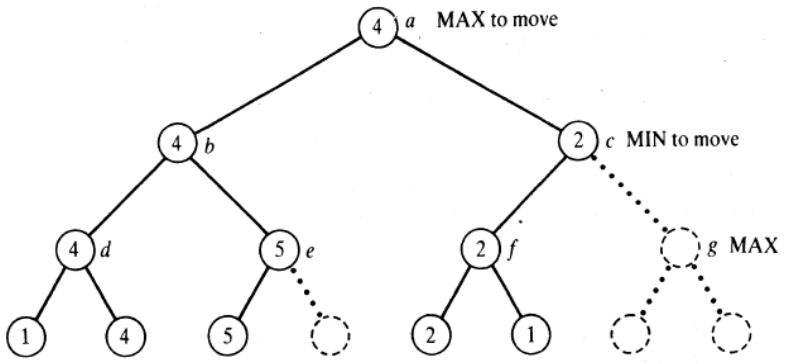


Figure 3: Alpha-Beta cuts example

3.4 Heuristic function

Even if the Alpha-Beta pruning reduces the complexity of the search, the process remains still highly demanding in term of computational cost. In order to further reduce the cost of the search, it is possible to add a maximum-depth parameter. This parameter is used to stop the expansion of the tree after a certain depth. The obvious flaw in this idea is that it is possible not to encounter a terminal node before reaching the maximum depth and therefore it is possible not to have a static value to back propagate. The solution to this problem is to use a function, called heuristic, to evaluate non terminal nodes. The value returned by this function will tell how good or bad a position is for a player. Hence, when during the expansion of the tree the maximum depth is reached, the back propagated value will be the heuristic of the reached node.
An advantage of this technique is that it is possible to choose the max-depth parameter based on the computing power of the machine running the algorithm.

4 Prolog implementation

To develop an Intelligent Agent (Alex) that is able to play Connect 4 against a skilled human player, it has been used the minimax algorithm with Alpha-Beta pruning and an heuristic that enables to limit the depth of the expansion of the tree.

4.1 Game Implementation

In this section we introduce the most important functions of our game implementation, which are the following ones:

- *game()*,
- *moves(...)*,
- *end_of_game(...)*,
- *alphabeta(...)*,
- *heuristic(...)*.

4.1.1 Function *game()*

It is the entry point of the game. Using this function, the user can play against Alex. At the beginning, the human player writes his move using the command line. After that, Alex chooses its move using the minimax algorithm with Alpha-Beta pruning and $max - depth = 4$. The game continues with the two players playing alternately. At each turn, the board is printed in the command line and it is checked whether the game is over. When one of the player wins or the board has been filled up, the result is printed out and the program terminates.

4.1.2 Function *moves()*

The function *moves(Pos, PosList)* is true when *PosList* is the list of all the positions that can be reached via valid moves from the position *Pos*.
A new position is reachable when the new disc is inserted in an empty slot and the slot below it is not empty, i.e. that in the lower slot there is another disc or that the new disc is placed in the first row.

4.1.3 Function *end_of_game()*

The function *end_of_game(..., Board, DiscsInLine, GameEnded)* is true when *GameEnded* is 1 and *Board* is a terminal position, meaning that there are four or more discs in line ($DiscsInLine \geq 4$) or that the board has been entirely filled. This represents the situation where the game is over. The function is also true when *GameEnded* is 0 and no player has put four or more discs in a line ($DiscsInLine < 4$) and the board has not been entirely filled. In this case the game is not over yet.

4.1.4 Function *alphabeta()*

It has been taken the implementation of this function and of the minimax algorithm (with the Alpha-Beta pruning) from the book *Prolog Programming For Artificial Intelligence*, by Ivan Bratko. Because of the high computational requirements to compute the complete tree, it has been added to this implementation the *max-depth* strategy, which allows to stop the expansion of the tree after a certain depth by using an heuristic.

4.1.5 Function *staticval()*

The function *staticval(Pos, Val)* is true when *Val* is the heuristic of the position *Pos*.

The heuristic of a position is the subtraction of the evaluation of MAX's discs minus the evaluation of MIN's discs. The evaluation of a player's discs is the sum of the scores associated to each sequence of discs in any direction (row, column, right diagonal, left diagonal). The score of a line of discs of length l is equal to 2^l.

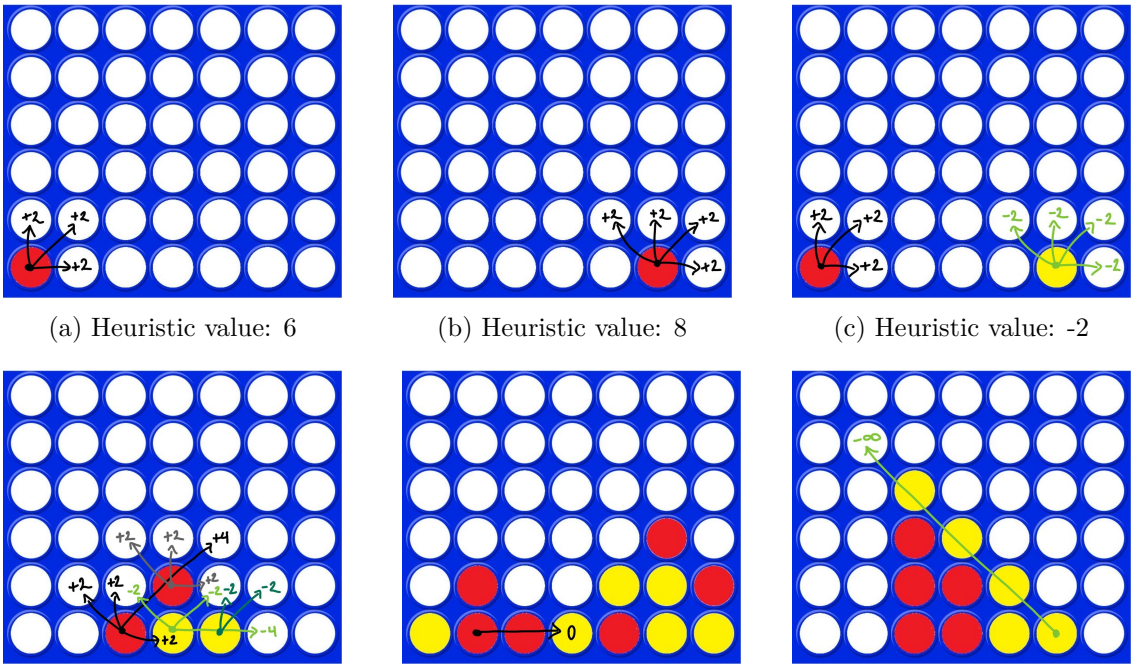
One exception to this is when the slots after and before a sequence are occupied by discs of the other player or when the sequence is limited by borders. In this case the heuristic function evaluates the sequence as 0. This happens because the sequence is not anymore expandable, hence it is useless to make the player win, and can therefore not be considered.
The other exception is when there are 4 contiguous discs in a row or column or diagonal. In this case the heuristic function returns a very high value (infinite, to be precise).

To sum up, the heuristic can be formalized with the following formula:

$$heuristic(Pos) = eval(Max) - eval(Min)$$
$$eval(Player) = \sum_{i=0}^{n} score(contiguous\_line_i) \text{ where } n \text{ is the number of contiguous lines}$$

$$score(contiguous\_line) = \begin{cases} \infty \text{ if } l \geq 4, \text{ where } l \text{ is the lenght of the contiguous line} \\ 0 \text{ if the line is blocked both at the beginning and at the end} \\ 2^l \text{ otherwise} \end{cases} \tag{1}$$

The heuristic of some game positions can be seen in Figure 4.



(a) Heuristic value: 6 (b) Heuristic value: 8 (c) Heuristic value: -2

(d) Heuristic value: 4 (e) Heuristic value: -18 (f) Heuristic value: -inf

Figure 4: Examples of game states with their evaluation.

5 Suggestions

In order to play against Alex, the world's best player of Connect 4, you have to consult the file *game.pl* and call the *game()* function. It is suggested to adopt a monospaced font in the GUI (e.g. fixedsys), to be able to see the grid well aligned. The player must write his move using the command line by inserting the column in which he would like to drop his disc. After that, Alex chooses its move. When one of the player wins or the board has been filled up, the result is printed out and the program terminates. Connect 4 is a solved game, i.e. the starting player can always win by playing the right moves.

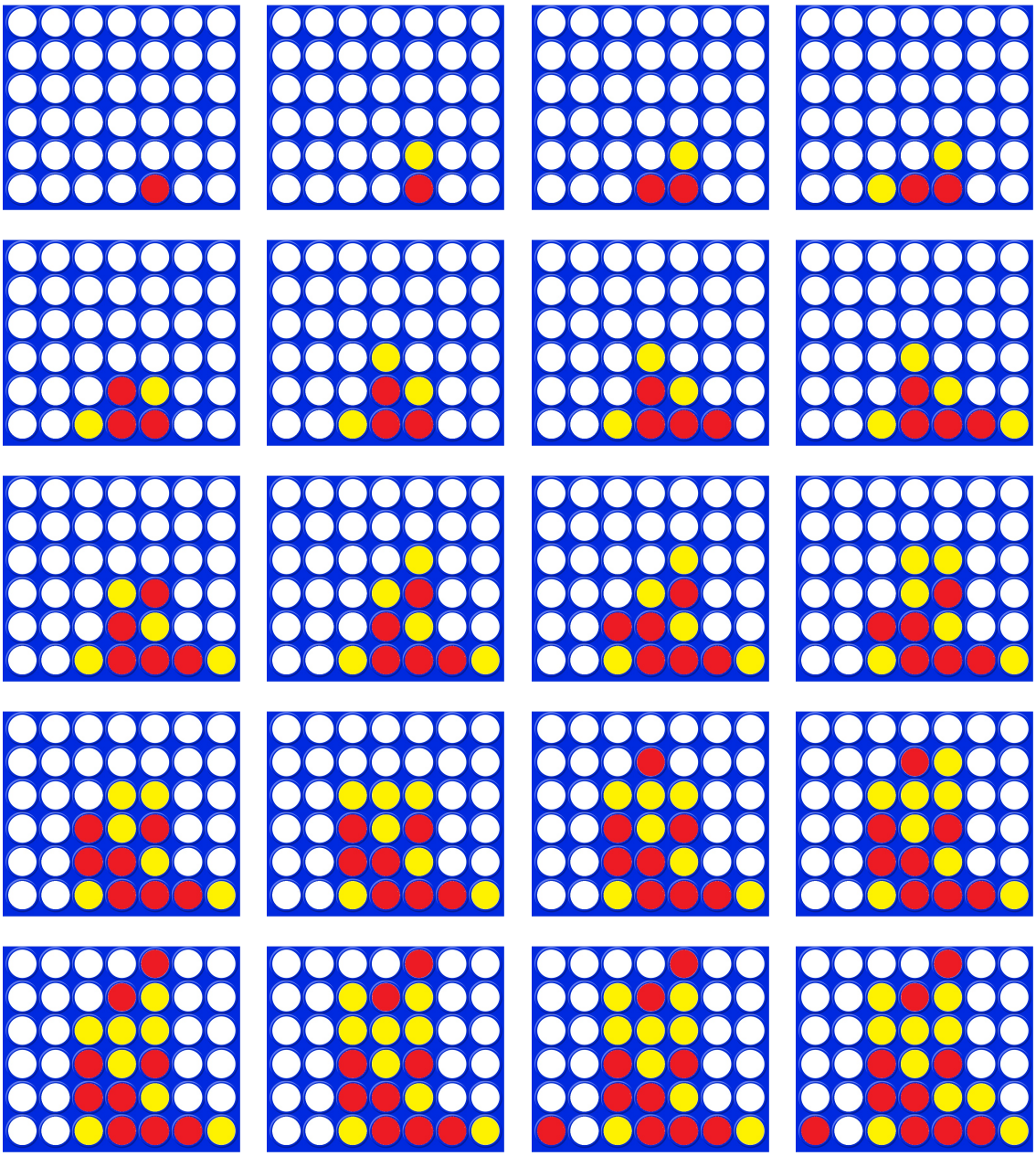
```
Write the column in which you want to put your disc (index s
tarts at 0)|: 5.
5|O_OOX_O|
4|X_OXO_X|
3|O_OOO_O|
2|X_XOXXX|
1|XOXXOOX|
0|XXOXXXO|
  0123456

Alex, the world's best player, has done its move
5|O_OOX_O|
4|X_OXO_X|
3|O_OOOOO|
2|X_XOXXX|
1|XOXXOOX|
0|XXOXXXO|
  0123456

Auch, you lose :(
true .

?-
```

Figure 5: GUI interface.

We challenge you to beat Alex, good luck!

A Game example

In this appendix it is shown an example of a played game. Alex uses the yellow discs. Red starts.



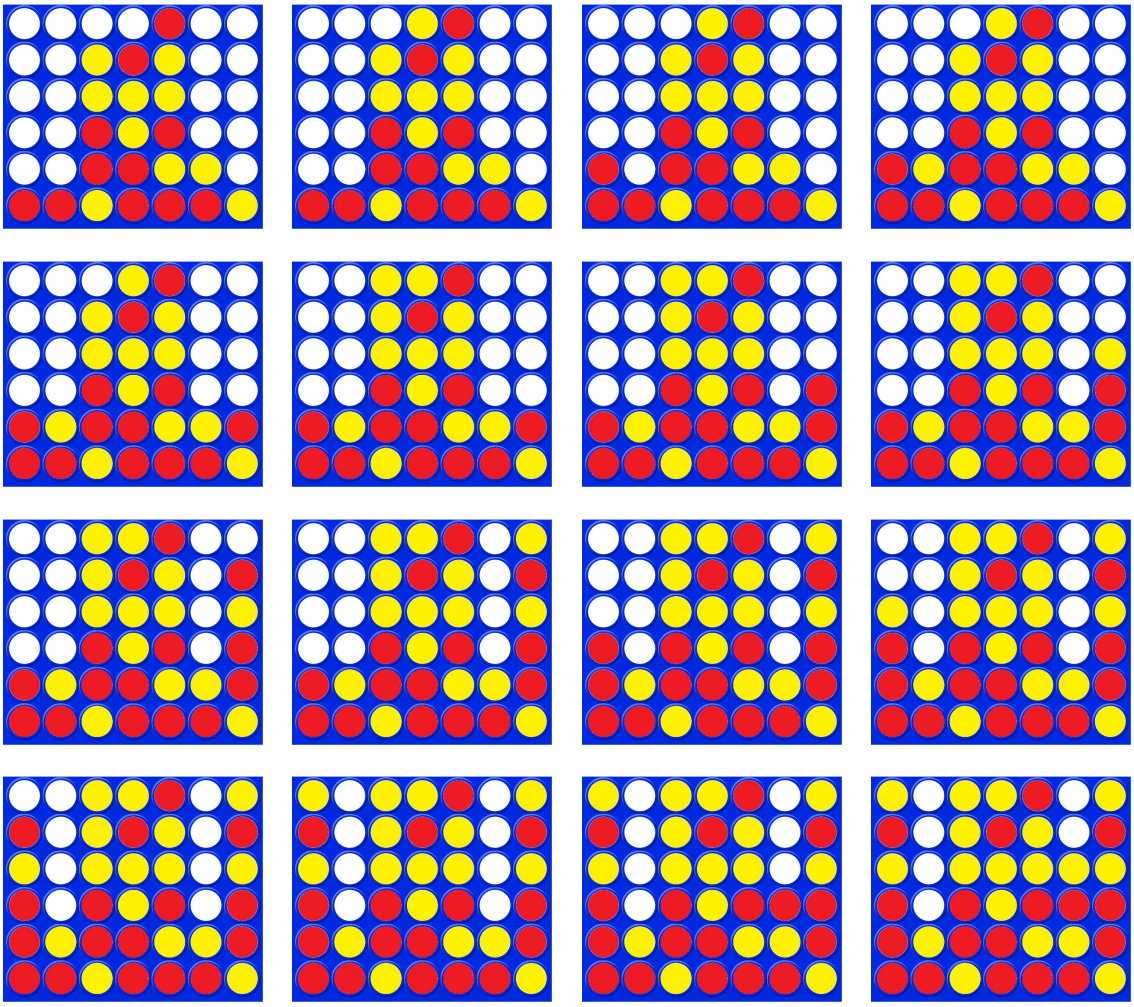


Figure 7: Game example: after an intriguing game Alex wins.

B Code

In this appendix it is shown the Prolog code of the implementation of Connect 4. It is divided in five different files, each one with a specific purpose. In order:

- *game.pl*: it is the entry point of our game. It manages the interface and allows the player to play against Alex, the world's best player;
- *connect_four.pl*: it defines the rules of the game.
- *alphabeta.pl*: it defines the minimax algorithm with alpha-beta pruning and max-depth;
- *heuristic.pl*: it allows to evaluate the heuristic of given game state;
- *utils.pl*: it contains useful generic functions;
- *test.pl*: it contains different test cases which have been used to check the code.

B.1 game.pl

```
% Game

:- consult(alphabeta).
:- consult(connect_four).
:- consult(utils).

% Entry point
game() :-
  game([[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]], 1, 0).

% Base case
% Always draws the board
% If the game is over say who won
% If not, let's make a move!
game(Pos, Player, 0) :-
  draw_board(Pos, 0),
  write('  0123456\n\n'),
  OpponentPlayer is Player * -1,
  end_of_game([], Pos, 0, 0, 0, OpponentPlayer, GameEnded),
  ((GameEnded is 1, game(Pos, 0, 1));
  (GameEnded is 0, game(Pos, Player, 1))).

% The game is over
game(Pos, 0, 1) :-
  staticval(Pos, Val),
  ((Val is 0, write('The game ended with a draw :| '));
  (Val is 1.0Inf, write('You woooooon, sbamm! '));
  (Val is -1.0Inf, write('Auch, you lose :( '))).

% Human plays
game(Pos, 1, 1) :-
  write('Write the column in which you want to put your disc
```

```
  (index starts at 0)'),
  read(Column),
  check_valid_move(Pos, Column),
  generate_pos(Pos, Column, NewPos),
  game(NewPos, -1, 0).

% CPU plays
game(Pos, -1, 1) :-
  write('Alex, the world\'s best player, has done its move\n'),
  min_to_move(Pos),
  alphabeta(Pos, -1.0Inf, 1.0Inf, NewPos, _, 0, 4),
  game(NewPos, 1, 0).



% Function to draw the Connect Four board
draw_board([Row|[]], _) :-
  write('5|'),
  draw_row(Row),
  write('|\n').

draw_board([Row|UpperBoard], I) :-
  NewI is I + 1,
  draw_board(UpperBoard, NewI),
  write(I),
  write('|'),
  draw_row(Row),
  write('|\n').



% Function to draw a Connect Four row
draw_row([]).

draw_row([Disc|Row]) :-
  ((Disc is 1, write('X'));
  (Disc is 0, write('_'));
  (Disc is -1, write('O'))),
  draw_row(Row).



% Functions that checks if the input of the human player is valid
check_valid_move(Pos, Column) :-
  integer(Column),
  Column >= 0,
  Column < 7,
  matrix(Pos, 5, Column, Disc),
  Disc is 0.

check_valid_move(Pos, _) :-
  write('\n!Warning! Your column is not a valid option. Please try
  with another one.\n'),
  game(Pos, 1, 1).
```

```
% Function that relates a position and a column in which the new
% disc must be inserted to the new position
generate_pos(Pos, Column, NewPos) :-
  generate_pos([], Pos, 0, Column, NewPos).

% Base case
% The first free slot in the right column has been reached
% and the new position is found.
generate_pos(LowerBoard, [Row|UpperBoard], _, Column, NewPos) :-
  nth0(Column, Row, 0),
  replace(Row, 0, Column, 1, NewRow),
  append(LowerBoard, [NewRow|UpperBoard], NewPos).

generate_pos(LowerBoard, [Row|UpperBoard], I, Column, NewPos) :-
  \+ nth0(Column, Row, 0),
  append(LowerBoard, [Row], NewLowerBoard),
  NewI is I + 1,
  generate_pos(NewLowerBoard, UpperBoard, NewI, Column, NewPos).
```

B.2 connect_four.pl

```
% Connect 4

:- consult(utils).

% [[0,0,1],[0,0,0],[0,0,0]] This is an example of a mini board
% [0,0,1]  is the lowest row of the mini board

end_of_row(J) :-
  J is 7.

end_of_row(J) :-
  J is -1.

end_of_board(I) :-
  I is 6.




% Base case
% Summation ended, max is to move if the sum is even
max_to_move([PosH|[]], S) :-
  sum_list(PosH, S1),
  SFinal is S + S1,
  even(SFinal).

% Sum up all the rows of the board
% S is the sum accumulator
max_to_move([PosH|PosT], S) :-
  sum_list(PosH, S1),
  S2 is S + S1,
  max_to_move(PosT, S2).
```

```
% Checks if max has to move in Pos.
% Max is to move if the sum of all the discs is even
max_to_move(Pos) :-
  max_to_move(Pos, 0).

% Min moves when max does not move
min_to_move(Pos) :-
  \+ max_to_move(Pos).



% Moves relates a position (Pos) to all the reachable
% positions (PosList) by executing valid moves
% If the game is over, no move is allowed
% The game is over if the opponent has won in the previous turn
% or if the board has been filled entirely
% Max must move
moves(Pos, PosList) :-
  max_to_move(Pos),
  end_of_game([], Pos, 0, 0, 0, -1, GameEnded),
  ((GameEnded is 0, moves([], Pos, 0, 0, PosList, 1)); PosList = []).

% Min must move
moves(Pos, PosList) :-
  min_to_move(Pos),
  end_of_game([], Pos, 0, 0, 0, 1, GameEnded),
  ((GameEnded is 0, moves([], Pos, 0, 0, PosList, -1)); PosList = []).



% Base case
% Stop when all the rows have been examined
% There are no more reachable positions
moves(_, _, I, _, [], _) :-
  end_of_board(I).

% Base case
% If a row has been completely examined, then examine next row.
moves(LowerBoard, [Row|UpperBoard], I, J, PosList, Player) :-
  end_of_row(J),
  append(LowerBoard, [Row], NewLowerBoard),
  NewI is I + 1,
  moves(NewLowerBoard, UpperBoard, NewI, 0, PosList, Player).

% If the examined position is empty (no disc => 0) and
% we are in the first row, or if the examined position
% is empty and the lower position is not empty,
% then we can play a disc in the examined position.
% The parameters are: Lower part the board, [Row examined|Upper part of the board],
% Row index, Column index, Possible new positions list, Player
moves(LowerBoard, [Row|UpperBoard], I, J, [PosListH|PosListT], Player) :-
  ((I is 0, nth0(J, Row, 0));
  (nth0(J, Row, 0), ICheck is I - 1,
  matrix(LowerBoard, ICheck, J, LowerVal),
  LowerVal =\= 0)),
```

```
  replace(Row, 0, J, Player, PosListHRow),
  append(LowerBoard, [PosListHRow|UpperBoard], PosListH),
  NewJ is J + 1,
  moves(LowerBoard, [Row|UpperBoard], I, NewJ, PosListT, Player).

% If the examined position is not playable, we go to the next one
moves(LowerBoard, UpperBoard, I, J, PosList, Player) :-
  NewJ is J + 1,
  moves(LowerBoard, UpperBoard, I, NewJ, PosList, Player).




% Base case
% If four discs of the right player are found in a line,
% then the game is over
end_of_game(_, _, _, _, DiscsInLine, _, 1) :-
  DiscsInLine >= 4.

% Base case
% The whole board has been examined.
% If a column is still free, then the game is not over
% If all columns are full, then the game is over
end_of_game(LowerRow, _, I, _, _, _, GameEnded) :-
  end_of_board(I),
  ((member(0, LowerRow), GameEnded is 0);
  (\+ member(0, LowerRow), GameEnded is 1)).

% When an entire row has been scanned,
% we continue the search in the upper row
end_of_game(_, [Row|UpperBoard], I, J, _, Player, GameEnded) :-
  end_of_row(J),
  NewI is I + 1,
  end_of_game(Row, UpperBoard, NewI, 0, 0, Player, GameEnded).

% If the disc is not of the right player, then we check the next column
end_of_game(LowerRow, [Row|UpperBoard], I, J, _, Player, GameEnded) :-
  \+ nth0(J, Row, Player),
  NewJ is J + 1,
  end_of_game(LowerRow, [Row|UpperBoard], I, NewJ, 0, Player, GameEnded).

% If the disc is of the right player, then we count the number of
% contiguous discs in all valid directions and we keep the highest.
end_of_game(LowerRow, [Row|UpperBoard], I, J, _, Player, GameEnded) :-
  PreviousJ is J - 1,
  NewJ is J + 1,

  ((nth0(PreviousJ, Row, Player), DiscsInRow is 0);
  contiguous_row(Row, J, Player, 0, DiscsInRow, _)),
  ((nth0(J, LowerRow, Player), DiscsInColumn is 0);
  contiguous_column([Row|UpperBoard], J, Player, 0, DiscsInColumn, _)),
  ((nth0(PreviousJ, LowerRow, Player), DiscsInRightDiagonal is 0);
  contiguous_diagonal([Row|UpperBoard], J, Player, 0, DiscsInRightDiagonal, _, 1)),
  ((nth0(NewJ, LowerRow, Player), DiscsInLeftDiagonal is 0);
  contiguous_diagonal([Row|UpperBoard], J, Player, 0, DiscsInLeftDiagonal, _, -1)),

  % We keep the longest contiguous line
  ((DiscsInColumn > DiscsInRow, Max1 is DiscsInColumn);
```

```
  Max1 is DiscsInRow),

  ((DiscsInRightDiagonal > Max1, Max2 is DiscsInRightDiagonal);
  Max2 is Max1),

  ((DiscsInLeftDiagonal > Max2, DiscsInLine is DiscsInLeftDiagonal);
  DiscsInLine is Max2),

  end_of_game(LowerRow, [Row|UpperBoard], I, NewJ, DiscsInLine, Player, GameEnded).



% Base case
% If we reach the end of the board (side) or if we find an opponent's disc,
% then the contiguous line is interrupted and it is blocked at the end.
contiguous_row(Row, J, Player, DiscsInLine, DiscsInLine, 1) :-
  OpponentPlayer is Player * -1,
  (end_of_row(J); nth0(J, Row, OpponentPlayer)).

% Base case
% If we find a free slot, the contiguous row is over and the end is free
contiguous_row(Row, J, _, DiscsInLine, DiscsInLine, 0) :-
  nth0(J, Row, 0).

contiguous_row(Row, J, Player, DiscsInLineAcc, DiscsInLine, EndBlocked) :-
  nth0(J, Row, Player),
  NewDiscsInLineAcc is DiscsInLineAcc + 1,
  NewJ is J + 1,
  contiguous_row(Row, NewJ, Player, NewDiscsInLineAcc, DiscsInLine, EndBlocked).



% Base case
% If we reach the end of the board (up)
% then the contiguous line is interrupted and it is blocked at the end.
contiguous_column([], _, _, DiscsInLine, DiscsInLine, 1).
  % Base case

% If we find an opponent's disc,
% then the contiguous line is interrupted and it is blocked at the end.
contiguous_column([Row|_], J, Player, DiscsInLine, DiscsInLine, 1) :-
  OpponentPlayer is Player * -1,
  nth0(J, Row, OpponentPlayer).

% Base case
% If we find a free slot, the contiguous column is over and the end is free
contiguous_column([Row|_], J, _, DiscsInLine, DiscsInLine, 0) :-
  nth0(J, Row, 0).

contiguous_column([Row|UpperBoard], J, Player,
                  DiscsInLineAcc, DiscsInLine, EndBlocked) :-
  nth0(J, Row, Player),
  NewDiscsInLineAcc is DiscsInLineAcc + 1,
  contiguous_column(UpperBoard, J, Player, NewDiscsInLineAcc,
                    DiscsInLine, EndBlocked).
```

```
% Base case
% If we reach the end of the board (up)
% then the contiguous line is interrupted and it is blocked at the end.
contiguous_diagonal([], _, _, DiscsInLine, DiscsInLine, 1, _).

% Base case
% If we reach the end of the board (up or side) or if we find an opponent's disc,
% then the contiguous line is interrupted and it is blocked at the end.
contiguous_diagonal([Row|_], J, Player, DiscsInLine, DiscsInLine, 1, _) :-
  OpponentPlayer is Player * -1,
  (nth0(J, Row, OpponentPlayer); end_of_row(J)).

% Base case
% If we find a free slot, the contiguous diagonal is over and the end is free
contiguous_diagonal([Row|_], J, _, DiscsInLine, DiscsInLine, 0, _) :-
  nth0(J, Row, 0).

% Direction stands for the direction of the diagonal (right=+1 or left=-1)
contiguous_diagonal([Row|UpperBoard], J, Player, DiscsInLineAcc,
                    DiscsInLine, EndBlocked, Direction) :-
  nth0(J, Row, Player),
  NewDiscsInLineAcc is DiscsInLineAcc + 1,
  NewJ is J + Direction,
  contiguous_diagonal(UpperBoard, NewJ, Player, NewDiscsInLineAcc,
                      DiscsInLine, EndBlocked, Direction).
```

B.3 alphabeta.pl

```
% Alpha beta

:- consult(connect_four).
:- consult(heuristic).

% GoodPos is a possible next position that satisfies the alpha beta
% values. First finds the possible moves from Pos, finds a good enough
% position if Pos is not a leaf otherwise it takes the value of the leaf node.
% alphabeta is executed up to a certain depth. If max depth is reached,
% then the position is evaluated using the heuristic.
alphabeta(Pos, Alpha, Beta, GoodPos, Val, Depth, MaxDepth) :-
  Depth < MaxDepth,
  moves(Pos, PosList),
  NewDepth is Depth + 1,
  !,
  (boundedbest(PosList, Alpha, Beta, GoodPos, Val, NewDepth, MaxDepth);
  staticval(Pos, Val)).

% Max depth has been reached
alphabeta(Pos, _, _, _, Val, _, _) :-
  staticval(Pos, Val).
```

```
% Finds a good enough position (known as GoodPos) in the list Poslist so
% that the backed-up value Val of GoodPos is a good enough approximation
% with respect to Alpha and Beta
boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal, Depth, MaxDepth) :-
  alphabeta(Pos, Alpha, Beta, _, Val, Depth, MaxDepth),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal, Depth, MaxDepth).



% no other candidates
goodenough([], _, _, Pos, Val, Pos, Val, _, _) :- !.

goodenough(_, Alpha, Beta, Pos, Val, Pos, Val, _, _) :-
  min_to_move(Pos), Val > Beta, !; % maximum reached
  max_to_move(Pos), Val < Alpha, !. % minimum reached

goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal, Depth, MaxDepth) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1, Depth, MaxDepth),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).



% min_to_move(Pos) is true if and only if the "minimizing" player
% must move in position Pos
newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha, !.

% max_to_move(Pos) is true if and only if the "maximizing" player
% must move in position Pos
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.

newbounds(Alpha, Beta, _, _, Alpha, Beta).



betterof(Pos, Val, _, Val1, Pos, Val) :-
  min_to_move(Pos), Val > Val1, !.

betterof(Pos, Val, _, Val1, Pos, Val) :-
  max_to_move(Pos), Val < Val1, !.

betterof(_, _, Pos1, Val1, Pos1, Val1).
```

B.4 heuristic.pl

```
% Heuristic

:- consult(connect_four).

 % Calculates the heuristic of the position by evaluating max and min positions
 % and subtracting them
staticval(Pos, Val) :-
```

```
  heuristic([], Pos, 0, 0, 0, MaxVal, 1),
  heuristic([], Pos, 0, 0, 0, MinVal, -1),
  ((MaxVal =:= +1.0Inf, Val is +1.0Inf);
  (MinVal =:= +1.0Inf, Val is -1.0Inf);
  (Val is MaxVal - MinVal)).



% When the heuristic is infinite
heuristic(_, _, _, _, Val, Val, _) :-
  Val =:= +1.0Inf;
  Val =:= -1.0Inf.

% When the whole board has been examined, we completed the calculation of
% the heuristic
heuristic(_, _, I, _, Val, Val, _) :-
  end_of_board(I).

% When an entire row has been scanned, we continue the search in the upper row
heuristic(_, [Row|UpperBoard], I, J, ValAcc, Val, Player) :-
  end_of_row(J),
  NewI is I + 1,
  heuristic(Row, UpperBoard, NewI, 0, ValAcc, Val, Player).

% If the disc is not of the right player, then we check the next column
heuristic(LowerRow, [Row|UpperBoard], I, J, ValAcc, Val, Player) :-
  \+ nth0(J, Row, Player),
  NewJ is J + 1,
  heuristic(LowerRow, [Row|UpperBoard], I, NewJ, ValAcc, Val, Player).

% If there is a disc of the right player, then the heuristic of the
% contiguous lines (in all directions) is calculated (only if it had
% not been calculated before)
% Parameters: LowerBoard, UpperBoard, Row index, Column index,
% Heuristic accumulator, Heuristic, Player
heuristic(LowerRow, [Row|UpperBoard], I, J, ValAcc, Val, Player) :-
  OpponentPlayer is Player * -1,
  PreviousJ is J - 1,
  NewJ is J + 1,

  heuristic_row(Row, J, PreviousJ, Player, OpponentPlayer, ValRow),
  heuristic_column([Row|UpperBoard], LowerRow, I, J, Player,
                   OpponentPlayer, ValColumn),
  heuristic_diagonal([Row|UpperBoard], LowerRow, I, J, PreviousJ, 1,
                     Player, OpponentPlayer, ValRightDiagonal),
  heuristic_diagonal([Row|UpperBoard], LowerRow, I, J, NewJ, -1,
                     Player, OpponentPlayer, ValLeftDiagonal),

  % Updating accumulator and checking if winning position
  ((ValRow =:= +1.0Inf, NewValAcc is ValRow);
  (ValColumn =:= +1.0Inf, NewValAcc is ValColumn);
  (ValRightDiagonal =:= +1.0Inf, NewValAcc is ValRightDiagonal);
  (ValLeftDiagonal =:= +1.0Inf, NewValAcc is ValLeftDiagonal);
  (NewValAcc is ValAcc + ValRow + ValColumn + ValRightDiagonal + ValLeftDiagonal)),

  heuristic(LowerRow, [Row|UpperBoard], I, NewJ, NewValAcc, Val, Player).
```

```
% Heuristic for the contiguous row
heuristic_row(Row, J, PreviousJ, Player, OpponentPlayer, ValRow) :-
  ((J is 0,
  StartBlocked is 1,
  contiguous_row(Row, J, Player, 0, DiscsInLine, EndBlocked));

  (nth0(PreviousJ, Row, OpponentPlayer),
  StartBlocked is 1,
  contiguous_row(Row, J, Player, 0, DiscsInLine, EndBlocked));

  (nth0(PreviousJ, Row, 0),
  StartBlocked is 0,
  contiguous_row(Row, J, Player, 0, DiscsInLine, EndBlocked));

  (nth0(PreviousJ, Row, Player),
  DiscsInLine is 0,
  StartBlocked is 1,
  EndBlocked is 1)),

  Blocked is StartBlocked + EndBlocked,

  ((DiscsInLine >= 4, ValRow is +1.0Inf);
  (Blocked is 2, ValRow is 0);
%  (Blocked is 1, ValRow is 2**(DiscsInLine + 1));
%  (Blocked is 0, ValRow is 2**(DiscsInLine + 2))).
  (Blocked is 1, ValRow is 2**(DiscsInLine));
  (Blocked is 0, ValRow is 2**(DiscsInLine))).



% Heuristic for the contiguous column
heuristic_column(UpperBoard, LowerRow, I, J, Player, OpponentPlayer, ValColumn) :-
  ((I is 0,
  contiguous_column(UpperBoard, J, Player, 0, DiscsInLine, EndBlocked));

  (nth0(J, LowerRow, OpponentPlayer),
  contiguous_column(UpperBoard, J, Player, 0, DiscsInLine, EndBlocked));

  (nth0(J, LowerRow, 0),
  contiguous_column(UpperBoard, J, Player, 0, DiscsInLine, EndBlocked));

  (DiscsInLine is 0,
  EndBlocked is 1)),

  ((DiscsInLine >= 4, ValColumn is +1.0Inf);
  (EndBlocked is 1, ValColumn is 0);
%  (EndBlocked is 0, ValColumn is 2**(DiscsInLine + 1))).
  (EndBlocked is 0, ValColumn is 2**(DiscsInLine))).



% Heuristic for the contiguous diagonal (left or right - defined by
% Direction together with CheckJ)
```

```
heuristic_diagonal(UpperBoard, LowerRow, I, J, CheckJ, Direction,
                   Player, OpponentPlayer, ValDiagonal) :-
  ((I is 0,
  StartBlocked is 1,
  contiguous_diagonal(UpperBoard, J, Player, 0, DiscsInLine, EndBlocked, Direction));

  (end_of_row(CheckJ),
  StartBlocked is 1,
  contiguous_diagonal(UpperBoard, J, Player, 0, DiscsInLine, EndBlocked, Direction));

  (nth0(CheckJ, LowerRow, OpponentPlayer),
  StartBlocked is 1,
  contiguous_diagonal(UpperBoard, J, Player, 0, DiscsInLine, EndBlocked, Direction));

  (nth0(CheckJ, LowerRow, 0),
  StartBlocked is 0,
  contiguous_diagonal(UpperBoard, J, Player, 0, DiscsInLine, EndBlocked, Direction));

  (DiscsInLine is 0,
  StartBlocked is 1,
  EndBlocked is 1)),

  Blocked is StartBlocked + EndBlocked,

  ((DiscsInLine >= 4, ValDiagonal is +1.0Inf);
  (Blocked is 2, ValDiagonal is 0);
%  (Blocked is 1, ValDiagonal is 2**(DiscsInLine + 1));
%  (Blocked is 0, ValDiagonal is 2**(DiscsInLine + 2))).
  (Blocked is 1, ValDiagonal is 2**(DiscsInLine));
  (Blocked is 0, ValDiagonal is 2**(DiscsInLine))).
```

B.5 utils.pl

```
% Utils

even(N) :- mod(N,2) =:= 0.
odd(N) :- mod(N,2) =:= 1.




% Defines a matrix, can be used for matrix indexing
matrix(Matrix, I, J, Value) :-
  nth0(I, Matrix, Row),
  nth0(J, Row, Value).




% Base case
% The right index has been reached, the value is changed
replace([_|T], J, J, Val, [Val|T]).

% Replace with value Val in list at position J
replace([H|T], I, J, Val, [H|OtherT]) :-
  NewI is I + 1,
```

```
  replace(T, NewI, J, Val, OtherT).
```

B.6 test.pl

```
% Test

:- consult(alphabeta).

test_heuristic() :-
  staticval([[1,0,0,0,0,-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]], Case1),
  Case1 is -8,
  staticval([[0,0,1,0,0,-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]], Case2),
  Case2 is 0,
  staticval([[0,0,1,-1,-1,0,0],[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]], Case3),
  Case3 is 12,
  staticval([[1,0,0,0,0,0,-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]], Case4),
  Case4 is 0,
  staticval([[1,1,0,0,0,-1,-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]], Case5),
  Case5 is 0,
  staticval([[1,-1,0,0,0,1,-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]], Case6),
  Case6 is 0,
  staticval([[1,-1,0,0,0,0,0],[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]], Case7),
  Case7 is 20,
  staticval([[0,0,0,0,1,1,-1],[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]], Case8),
  Case8 is -4,
  staticval([[0,0,1,-1,0,0,0],[0,0,1,-1,0,0,0],[0,0,1,-1,0,0,0],[0,0,1,0,0,0,0],
             [0,0,0,0,0,0,0],[0,0,0,0,0,0,0]], Case9),
  Case9 is +1.0Inf,
  staticval([[0,0,1,1,-1,-1,0],[0,0,1,1,-1,0,0],[0,0,1,-1,0,0,0],[0,0,-1,0,0,0,0],
             [0,0,0,0,0,0,0],[0,0,0,0,0,0,0]], Case10),
  Case10 is -1.0Inf,
  staticval([[1,-1,1,-1,1,-1,1],[1,-1,1,-1,1,-1,1],[1,-1,1,-1,1,-1,1],
             [-1,1,-1,1,-1,1,-1],[-1,1,-1,1,-1,1,-1],[-1,1,-1,1,-1,1,-1]], Case11),
  Case11 is 0.

test_end_game() :-
  moves([[1,0,0,0,0,-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]], PosList1),
  \+ PosList1 = [],
  moves([[0,0,1,0,0,-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]], PosList2),
  \+ PosList2 = [],
  moves([[0,0,1,-1,-1,0,0],[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]], PosList3),
  \+ PosList3 = [],
  moves([[1,0,0,0,0,0,-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]], PosList4),
  \+ PosList4 = [],
```

```
  moves([[1,1,0,0,0,-1,-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]], PosList5),
  \+ PosList5 = [],
  moves([[1,-1,0,0,0,1,-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]], PosList6),
  \+ PosList6 = [],
  moves([[1,-1,0,0,0,0,0],[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]], PosList7),
  \+ PosList7 = [],
  moves([[0,0,0,0,1,1,-1],[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]], PosList8),
  \+ PosList8 = [],
  moves([[0,0,1,-1,0,0,0],[0,0,1,-1,0,0,0],[0,0,1,-1,0,0,0],[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]], PosList9),
  PosList9 = [],
  moves([[0,0,1,1,-1,-1,0],[0,0,1,1,-1,0,0],[0,0,1,-1,0,0,0],[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]], PosList10),
  PosList10 = [],
  moves([[-1,1,1,-1,1,-1,-1],[0,1,0,0,-1,-1,0],[0,1,0,0,0,1,0],[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]], PosList11),
  PosList11 = [],
  moves([[1,-1,1,-1,1,-1,1],[1,-1,1,-1,1,-1,1],[1,-1,1,-1,1,-1,1],
        [-1,1,-1,1,-1,1,-1],[-1,1,-1,1,-1,1,-1],[-1,1,-1,1,-1,1,-1]], PosList12),
  PosList12 = [].

test_alpha_beta1() :-
  alphabeta([[-1,1,1,-1,1,-1,-1],[0,1,0,0,-1,-1,0],[0,1,0,0,0,1,0],
             [0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0]], -1.0Inf, +1.0Inf,
             BestPos, Val, 0, 1),
  BestPos = [[-1,1,1,-1,1,-1,-1],[0,1,0,0,-1,-1,0],[0,1,0,0,0,1,0],
             [0,1,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0]],
  Val is +1.0Inf,
  write(BestPos),
  write(Val).

test_alpha_beta2() :-
  alphabeta([[-1,1,1,-1,1,-1,-1],[0,1,0,0,-1,-1,0],[0,1,0,0,0,1,0],
             [0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0]], -1.0Inf, +1.0Inf,
             BestPos, Val, 0, 7),
  BestPos = [[-1,1,1,-1,1,-1,-1],[0,1,0,0,-1,-1,0],[0,1,0,0,0,1,0],
             [0,1,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0]],
  Val is +1.0Inf,
  write(BestPos),
  write(Val).

test_alpha_beta3() :-
  alphabeta([[1,-1,1,-1,1,-1,1],[1,-1,1,-1,1,-1,1],[1,-1,1,-1,1,-1,1],
             [-1,1,-1,1,-1,1,-1],[-1,1,-1,1,-1,1,-1],[0,1,-1,0,-1,1,0]], -1.0Inf,
             1.0Inf, BestPos, Val, 0, 7),
  write(BestPos),
  write(Val).
```