Faculty of Engineering, KMITL

Software Engineering (International Program)

Artificial Intelligence

Member

Passawit	**Umrod**	**60090024**
Pollakit	**Ngamampornitthi**	**60090031**
Sarin	**Wanichwasin**	**60090034**

Project Name

The Intelligence Othello

Introduction

Othello (or reversi) is one of the most recognizable strategy board games. This classic game involves two players, whether it is human against human, or human against the computer. Each side is represented by two respective colors, black and white. The goal of the game is to have the most number of tokens of one's color on your board. Trapping your opponent's token between your own color will result in changing, or "reversi" of the in-between tokens to your side. Othello is surprisingly easy to learn, but very demanding to master. The outcome of each game can be changed on a whim, depending on the player's strategy, experience, intuition and foresight in the game.

The game can be produced by an artificial intelligence method. Many algorithms have been raised to solve this problem as following the minimax algorithm and alpha-beta pruning algorithm. In this project, We will be used both algorithms to implement in our program.

Game Rules

1. There are always two sides competing against each other, black and white
2. Each player must place their own respective color tokens once every turn, While also trying to "flip" (reverse the color of the other side's tokens)
3. If flipping can't be accomplished in a turn, that turn is skipped.
4. To flip an opponent's token, the play must surround their opponent's token with their own, in two adjacent directions (diagonal is not applicable)
5. When the board is filled, the color with the most amount of tokens win

AI Technique

Utilizing swi-prolog, minimax algorithm, and alpha beta pruning, we have managed to simulate a practical version of Othello, where the artificial intelligence is capable of “thinking” ahead more than a several turns to optimize its own decision, when trying to win against its opponent (whether it is AI vs AI, or AI vs human).

This is accomplished with the minimax algorithm, where, depending on the depth of the minimax tree, increases or decreases the difficulty (indicating how many turns ahead does the artificial intelligence calculate for optimal move. The more turns, the more complicated the tree become, and therefore, the more difficult and the more time consuming the decision become)

The minimax algorithm calculate for all the possible move the AI have on its own turn, deciding which move will result in the most profit it can gain for itself, along with how that move will result in the least profit it’s enemy can get on their turn.

When there are too many branches in the minimax decision tree, it may take too long for the algorithm to complete every single calculation, therefore we implement the alpha-beta pruning algorithm. Essentially, it ignores any branch that have a worse outcome than the previously found branch, cutting down calculation time into an acceptable rate for players. Alpha beta pruning works by initializing maximizer alpha as negative infinity, and initializing minimizer beta as positive infinity. Maximizer and minimizer along the root will update every time a better value is found, and whenever beta is lower or equal to alpha, all branches below it will be pruned (meaning that it will not be taken into consideration or calculated, saving a significant amount of time).

Othello.pl

```
init :-
    load_files([utilities,board,evaluator, alpha_beta], []).

rownum(8).
colnum(8).

% opponent_color
opponent_color(white, black).
opponent_color(black, white).

% play/1 : Starts the game
%   Depth: the maximum depth of the search
play(Depth):-
(
    init_board(0,[],Board),
    select_game_mode(Mode),
    game_loop(Board,Mode,Depth,black)
),!.

% gameLoop/4 : Loops the game, until the game ends
%   Board: the current board(state)
%   Color: the player that going to play

game_loop(Board, Mode, Depth, Color):-
    print_board(Board),
    print_player(Color),
    is_board_full(Board,IsBoardFull),
    (
        IsBoardFull = yes -> show_score(Board)
        ;
        find_moves(Board, Color, MovesList),
        member(_, MovesList),
        opponent_color(Color,OpponentColor),
        (
            Mode = 1 ->
            (
                Color = black ->
                    human_select_move(Move, MovesList),!,
                    set_piece(Board, Move, Color, FinalBoard),
```

```
                    game_loop(FinalBoard, 1, Depth, OpponentColor),!
                ;
                Color = white ->
                    machine_select_move(Board, Depth, white, 
FinalBoard),!,
                    game_loop(FinalBoard, 1, Depth, OpponentColor),!
            )
            ;
            Mode = 2 ->
            (
                Color = black ->
                    machine_select_move(Board, Depth, Color, 
FinalBoard),!,
                    game_loop(FinalBoard, 2, Depth, OpponentColor),!
                ;
                Color = white ->
                    human_select_move(Move, MovesList),!,
                    set_piece(Board, Move, Color, FinalBoard),
                    game_loop(FinalBoard, 2, Depth, OpponentColor),!
            )
            ;
            Mode = 3 ->
                human_select_move(Move, MovesList),!,
                set_piece(Board, Move, Color, FinalBoard),
                game_loop(FinalBoard, 3, Depth, OpponentColor),!
            ;
            Mode = 4 ->
                machine_select_move(Board, Depth, Color, FinalBoard),!,
                game_loop(FinalBoard, 4, Depth, OpponentColor),!
        )
    ).

game_loop(Board, Mode, Depth, Color):-
    find_moves(Board, Color, MovesList),!,
    not(member(_,MovesList)),!,
    opponent_color(Color, OpponentColor),
    (

        (find_moves(Board, OpponentColor, RivalMovesList), 
member(_,RivalMovesList))->
```

```
            writeln('There\'s no valid move.'),
            game_loop(Board, Mode, Depth, OpponentColor),!
        ;
            writeln('There\'s no valid move for both players.'),
            show_score(Board)
    ).

print_player(white):-
    nl,
    writeln('White (O) player''s turn'),!.

print_player(black):-
    nl,
    writeln('Black (X) player''s turn'),!.

show_score(Board):-
    nl,
    count_pieces(black, Board, NumBlack, NumWhite),
    writef('black: %d\t',[NumBlack]),
    writef('white: %d\n',[NumWhite]),
    (
        NumBlack > NumWhite -> write('black (X) win\n')
        ;
        NumBlack < NumWhite -> write('white (O) win\n')
        ;
        write('Tie game\n')
    ),
    nl.

% human_select_move/2
%   Move: Selected Move
%   MovesList: possible moves

human_select_move(Move, MovesList):-
    write('Enter the Row: '),
    read(SelectedRow),
    writeln('Enter the Column: '),
    read(SelectedColum),
    member(Move, MovesList),
    nth0(0, Move, SelectedRow),
```

```
    nth0(1, Move, SelectedColum).

human_select_move(Move, MovesList):-
    writeln('Not a valid move'),
    writeln(''),
    human_select_move(Move, MovesList).

machine_select_move(Board, Depth, Color, FinalBoard):-
    alpha_beta_pruning(Board, Depth, Color, FinalBoard, _).

select_game_mode(Mode):-
    writeln('Select a game mode'),
    writeln('1. human vs machine'),
    writeln('2. machine vs human'),
    writeln('3. human vs human'),
    writeln('4. machine vs machine'),
    write('Enter a number: '),
    read(SelectedMode),
    (
        SelectedMode is 1 ->
            Mode is SelectedMode,
            writeln('machine vs human selected'),
            writeln(''),!
        ;
        SelectedMode is 2 ->
            SelectedMode is 2,
            Mode is SelectedMode,
            writeln('human vs machine selected'),
            writeln(''),!
        ;
        SelectedMode is 3 ->
            Mode is SelectedMode,
            writeln('human vs human selected'),
            writeln(''),!
        ;
        SelectedMode is 4 ->
            Mode is SelectedMode,
            writeln('machine vs machine (black first) selected'),
            writeln(''),!
        ;
```

```
            writeln('Not a valid mode'),
            writeln(''),
            select_game_mode(Mode)
    ).
```

Utilities.pl

```
/*
 * for loop
 */

for(V,V,_,_) :- !.
% Action: eg. write('  ----')
for(Start,End,Inc,Action) :-
    End > Start,
    NewValue is Start+Inc,
    call(Action),
    for(NewValue,End,Inc,Action).

/**
 * first_elements
 */
first_elements([], BoardsList, BoardsList):-!.

first_elements([First|Rest], Temp, Boards):-
    nth0(0, First, Board),
    append(Temp, [Board], NTemp),
    first_elements(Rest, NTemp, Boards).

/**
 * first_n_elements
 */
first_n_elements(Number, List, NList):-
        length(List, N),
        N =< Number,
        List = NList,!.

first_n_elements(Number, List, NList):-
    first_n_elements_aux(Number, List, [], NList).
```

```
first_n_elements_aux(0, _, NList, NList):-!.

first_n_elements_aux(Number, [First|Rest], TempList, NList):-
    NNumber is Number - 1,
    append(TempList, [First], NTempList),
    first_n_elements_aux(NNumber, Rest, NTempList, NList).

/**
 * min_list
 */
min_list([First|Rest], Min):-
    min_list_aux(Rest, First, Min).

min_list_aux([], Min, Min):-!.

min_list_aux([First|Rest], CurrentMin, Min):-
    First < CurrentMin,
    min_list_aux(Rest, First, Min),!.

min_list_aux([_|Rest], CurrentMin, Min):-
    min_list_aux(Rest, CurrentMin, Min),!.

/**
 * max_list
 */
max_list([First|Rest], Max):-
    max_list_aux(Rest, First, Max).

max_list_aux([], Max, Max):-!.

max_list_aux([First|Rest], CurrentMax, Max):-
    First > CurrentMax,
    max_list_aux(Rest, First, Max),!.

max_list_aux([_|Rest], CurrentMax, Max):-
    max_list_aux(Rest, CurrentMax, Max),!.



max(A,B,Max):-
    A >= B->
```

```
        Max=A
    ;
    Max=B.
```

Evaluator.pl

```
/*
 * black heuristic
 */
evaluator(black,Board, Value):-
    count_pieces(black, Board, BlackPieces, WhitePieces),
    TotalPiece is BlackPieces +  WhitePieces,

    Piece_diff is BlackPieces - WhitePieces,
    getCorners(Corners),
    getXSquares(XSquares),
    positionCount(black,Board,Corners,BlackCorner,WhiteCorner),
    positionCount(black,Board,XSquares,BlackXSquares,WhiteXSquares),

    valid_positions(Board, black, BlackValidMoves),
    valid_positions(Board, white, WhiteValidMoves),
    Valid_diff is WhiteValidMoves-BlackValidMoves ,
    (
        (TotalPiece<36)->
            /*Piece_diff_Score is 0,*/
            Piece_diff_Score is Piece_diff,
            Valid_diff_score is Valid_diff
        ;
            Piece_diff_Score is Piece_diff,
            Valid_diff_score is 0
    ),

    CornersBonus is 10*(BlackCorner-WhiteCorner),
    XSquaresBonus is -10*(BlackXSquares-WhiteXSquares),
    Bonus is CornersBonus+XSquaresBonus,
    Value is Piece_diff_Score+Valid_diff_score+Bonus.

final(black,Board, Value):-
    full_board(Board),
```

```
    count_pieces(black, Board, BlackPieces, WhitePieces),
    Value is BlackPieces - WhitePieces.


/*count pieces at special position*/
positionCount(Color,Board,PositionList,Count,RivalCount):-
    positionCount(Color,Board,PositionList,0,0,Count,RivalCount).

positionCount(Color,Board,PositionList,CountBuf,RivalCountBuf,Count,Riv
alCount):-
    opponent_color(Color,OpponentColor),
    (
        PositionList=[]->
            Count=CountBuf,
            RivalCount=RivalCountBuf
        ;
        PositionList = [Position|PositionsRest],
        Position = [Rowi,Coli|CheckList],
        (
            CheckList\=[]->
                CheckList=[CheckRow,CheckCol],
                piece(Board,CheckRow,CheckCol,CheckPiece)
            ;
                CheckPiece=null
        ),
        piece(Board,Rowi,Coli,Piece),
        (
            (Piece=Color,CheckPiece\=Color)->
                NCountBuf is CountBuf+1,
                NRivalCountBuf is RivalCountBuf
            ;
            (Piece=OpponentColor,CheckPiece\=OpponentColor)->
                NCountBuf is CountBuf,
                NRivalCountBuf is RivalCountBuf +1
            ;
                NCountBuf is CountBuf,
                NRivalCountBuf is RivalCountBuf
        ),
```

```
positionCount(Color,Board,PositionsRest,NCountBuf,NRivalCountBuf,Count,
RivalCount)
    ).

%     0 1 2 3 4 5 6 7
%   ------------------
% 0 | X - - - - - - X
% 1 | - - - - - - - -
% 2 | - - - - - - - -
% 3 | - - - - - - - -
% 4 | - - - - - - - -
% 5 | - - - - - - - -
% 6 | - - - - - - - -
% 7 | X - - - - - - X

getCorners(Corners):-
    Corners=[[0,0],[0,7],
             [7,0],[7,7]].

%     0 1 2 3 4 5 6 7
%   ------------------
% 0 | O - - - - - - O
% 1 | - X - - - - X -
% 2 | - - - - - - - -
% 3 | - - - - - - - -
% 4 | - - - - - - - -
% 5 | - - - - - - - -
% 6 | - X - - - - X -
% 7 | O - - - - - - O

getXSquares(XSquares):-
    XSquares=[  [1,1, 0,0],
                [1,6, 0,7],
                [6,1, 7,0],
                [6,6, 7,7]
            ].

/*
 * white heuristic
```

```
 */
evaluator(white,Board, Value):-
    count_pieces(black, Board, BlackPieces, WhitePieces),
    PieceValue is (BlackPieces - WhitePieces),!,
    valid_positions(Board, black, BlackValidMoves),
    valid_positions(Board, white, WhiteValidMoves),
    MobilityValue is (BlackValidMoves - WhiteValidMoves),!,
    getCornerValue(Board, CornerValue),!,
    getEdgeValue(Board, EdgeValue),!,
    Value is (10* PieceValue + 10* MobilityValue + 10* CornerValue +
10* EdgeValue).

final(white,Board, Value):-
    valid_positions(Board, black, BlackValidMoves),
    BlackValidMoves is 0,!,
    valid_positions(Board, white, WhiteValidMoves),
    WhiteValidMoves is 0,!,
    count_pieces(black, Board, BlackPieces, WhitePieces),
    Value is 40*(BlackPieces - WhitePieces),!.

%      0 1 2 3 4 5 6 7
%    ------------------
% 0 | X X - - - - X X
% 1 | X X - - - - X X
% 2 | - - - - - - - -
% 3 | - - - - - - - -
% 4 | - - - - - - - -
% 5 | - - - - - - - -
% 6 | X X - - - - X X
% 7 | X X - - - - X X

getCornerSquares(CornerSquares):-
    CornerSquares= [[0,0,  0,1,  1,0,  1,1],
                    [7,0,  7,1,  6,0,  6,1],
                    [0,7,  0,6,  1,7,  1,6],
                    [7,7,  7,6,  6,7,  6,6]].

% getCornerValue
% Decide the effect of move around corners.
```

```
getCornerValue(Board, CornerValue):-
    getCornerSquares(CornerSquares),
    getCornerValue(Board, CornerSquares, CornerValue, 0),!.


%     0         1
%   ---------------
% 0 | R,C      R1,C1
% 1 | R2,C2    R3,C3


getCornerValue(Board, CornerSquares, CornerValue, CornerValueBuf):-
    (
        CornerSquares = [] ->
            CornerValue is CornerValueBuf
        ;
        CornerSquares = [CurrentSqure|CornerSquaresRest],
        CurrentSqure = [CornerR, CornerC|CurrentSqureR1],
        CurrentSqureR1 = [CornerR1, CornerC1|CurrentSqureR2],
        CurrentSqureR2 = [CornerR2, CornerC2|CurrentSqureR3],
        CurrentSqureR3 = [CornerR3, CornerC3],
        piece(Board, CornerR,  CornerC,  PieceCorner),
        piece(Board, CornerR1, CornerC1, PieceCorner1),
        piece(Board, CornerR2, CornerC2, PieceCorner2),
        piece(Board, CornerR3, CornerC3, PieceCorner3),
        (
            PieceCorner = empty ->
            (
                (PieceCorner1=black; PieceCorner2=black)->
                    Value = -3
                ;
                (PieceCorner3=black)->
                    Value = -4
                ;
                (PieceCorner1=white; PieceCorner2=white)->
                    Value = 3
                ;
                (PieceCorner3=white)->
                    Value = 4
                ;
```

```
                    Value = 0
            )
            ;
            PieceCorner = white ->
                Value = -4
            ;
            Value = 4
        ),!,
        NCornerValueBuf is CornerValueBuf + Value,
        getCornerValue(Board, CornerSquaresRest, CornerValue,
NCornerValueBuf)
    ).

/*
 * getEdgeValue
 * Decide the effect of move on edges.
 */
getEdgeValue(Board, EdgeValue):-
    getEdgeValue(Board, EdgeValue, 0, 0, 0, 0),!.

getEdgeValue(Board, EdgeValue, Rowi, Coli, BlackOnEdgeBuf,
WhiteOnEdgeBuf):-
    RowN is 8,
    ColN is 8,
    piece(Board, Rowi, Coli, PieceColor),
    (
        PieceColor = black ->
            NewBlackOnEdgeBuf is BlackOnEdgeBuf + 2,
            NewWhiteOnEdgeBuf is WhiteOnEdgeBuf
        ;
        PieceColor = white ->
            NewBlackOnEdgeBuf is BlackOnEdgeBuf,
            NewWhiteOnEdgeBuf is WhiteOnEdgeBuf + 2
        ;
            NewBlackOnEdgeBuf is BlackOnEdgeBuf,
            NewWhiteOnEdgeBuf is WhiteOnEdgeBuf
    ),
    (
        (Rowi is RowN-1, Coli is ColN-1) ->
            EdgeValue is NewBlackOnEdgeBuf - NewWhiteOnEdgeBuf
```

```
        ;
        (
            (Coli is ColN-1) ->
                NextColi is 0,
                NextRowi is Rowi+1
            ;
            (Rowi is RowN-1; Rowi is 0) ->
                NextColi is Coli+1,
                NextRowi is Rowi
            ;
            (Coli is 0, not(Rowi is RowN-1), not(Rowi is 0)) ->
                NextColi is ColN-1,
                NextRowi is Rowi
            ;
                writef('unexpected %w %w\n',[Rowi, Coli])
        ),
        getEdgeValue(Board, EdgeValue, NextRowi, NextColi,
NewBlackOnEdgeBuf, NewWhiteOnEdgeBuf)
    ),!.
```

Board.pl

```
/**
 * init_board
 */
init_board(Rowi,TempBoard,Board):-
    rownum(R),
    Rowi=R,
    Board=TempBoard,
    !.

init_board(Rowi,TempBoard,Board):-
    rownum(R),
    Rowi<R,
    init_row(Rowi,0,[],Row),
    append(TempBoard,[Row],NTempBoard),
    NRowi is Rowi+1,
    init_board(NRowi,NTempBoard,Board).

init_row(_,Coli,TempRow,Row):-
```

```
    colnum(C),
    Coli=C,
    append(TempRow,[],Row),
    !.
init_row(Rowi,Coli,TempRow,Row):-
    rownum(R),
    colnum(C),
    Coli<C,
    CenterR is C/2,
    CenterL is C/2-1,
    CenterD is R/2,
    CenterU is R/2-1,
    (
        ( (Rowi = CenterU , Coli = CenterL);
          (Rowi = CenterD , Coli = CenterR) ) ->
                Color=black
        ;
        ( (Rowi = CenterU , Coli = CenterR);
          (Rowi = CenterD , Coli = CenterL) ) ->
                    Color = white
        ;
        Color = empty
    ),
    append(TempRow,[Color],NTempRow),
    NColi is Coli+1,
    init_row(Rowi,NColi,NTempRow,Row).

/**
 * print_board
 */
print_board(Board):-
    tab(4),
    print_head(0),
    print_body(0,0,Board),
    nl.

print_head(Coli):-
    colnum(C),
    Coli=C,
    nl,
```

```
    write('  ----'),
    for(1,C,1,write('--')),
    nl,
    !.

print_head(Coli):-
    write(Coli),
    tab(1),
    NColi is Coli+1,
    print_head(NColi).


print_body(Rowi,Coli,_):-
    rownum(R),
    colnum(C),
    Rowi is R-1,
    Coli is C,
    !.

print_body(Rowi,Coli,Board):-
    colnum(C),
    (
        Coli=0 ->
            write(Rowi),
            write(' | '),
            NRowi is Rowi,
            NColi is Coli+1,
            print_piece(Rowi,Coli,Board)
        ;
        Coli=C ->
            NRowi is Rowi+1,
            NColi is 0,
            nl
        ;
            NRowi is Rowi,
            NColi is Coli+1,
            print_piece(Rowi,Coli,Board)
    ),
    print_body(NRowi,NColi,Board).
```

```
/*
 * piece
 */
print_piece(Rowi, Coli, Board):-
    piece(Board, Rowi, Coli, Piece),
    (
        Piece=black->
            write('X ')
        ;
        Piece=white ->
            write('O ')
        ;
        write('- ')
    ).

piece(Board, Rowi, Coli, Piece):-
    is_valid_index(Rowi,Coli),
    nth0(Rowi,Board,Row),
    nth0(Coli,Row,Piece).

/*
 * is_valid_index
 */
is_valid_index(Rowi,Coli):-
    rownum(R),
    colnum(C),
    Rowi>=0,
    Rowi<R,
    Coli>=0,
    Coli<C.

/*
 * is_board_full
 */
is_board_full(Board,IsBoardFull):-
    flatten(Board,PieceList),
    list_to_set(PieceList,PieceSet),
    (
        not(member(empty,PieceSet))->
            IsBoardFull = yes
```

```
        ;
        IsBoardFull = no
    ).
/*
 * empty_on_board
 */
empty_on_board(Board):-
    member(Row, Board),
    member(Piece, Row),
    Piece = empty,!.

/*
 * full_board
 */
full_board(Board):-
    flatten(Board, PiecesList),
    list_to_set(PiecesList, PiecesSet),
    not(member(empty, PiecesSet)).

/*
 * find_states
 */
find_states(Caller, State, Color, StatesList):-
    find_boards(Caller, State, Color, StatesList).

/*
 * find_boards
 */
find_boards(Caller, Board, Color, BoardsList):-
    find_moves(Board, Color, MovesList),
    find_boards(Caller, Board, Color, OrderedBoardsList, [],
MovesList),
    first_elements(OrderedBoardsList, [], BoardsList).

find_boards(_, Board,_, BoardsList, [], []):-
    append([], [[Board, 0]], BoardsList),!.

find_boards(_, _, _, BoardsList, BoardsList, []):-!.
```

```
find_boards(Caller, Board, Color, BoardsList, CurrentBoardsList,
[Move|RestMovesList]):-
    set_piece(Board, Move, Color, FinalBoard),
    order_boards(Caller, Color, CurrentBoardsList, FinalBoard,
NBoardsList),
    find_boards(Caller, Board, Color, BoardsList, NBoardsList,
RestMovesList),!.

/*
 * order_boards
 * arrange pruning order, better state first pruned
 */
order_boards(Caller, Color, CurrentBoardsList, FinalBoard,
NBoardsList):-
    /*opponent_color(Color, OpponentColor),*/
    /*valid_positions(FinalBoard, OpponentColor, Number),*/
    evaluator(Caller, FinalBoard, Number),
    order_boards_aux(Color, [FinalBoard, Number], CurrentBoardsList,
[], NBoardsList).

order_boards_aux(_, Board, [], CurrentList, FinalList):-
    append(CurrentList, [Board], FinalList),!.

order_boards_aux(Color, Board, [First|Rest], CurrentList, FinalList):-
    nth0(1, First, Value),
    nth0(1, Board, NewValue),
    (
       Color = black ->
           NewValue >= Value
       ;
           NewValue =< Value
    ),
    append(CurrentList, [Board], TempList),
    append(TempList, [First|Rest], FinalList),!.

order_boards_aux(Color, Board, [First|Rest], CurrentList, FinalList):-
    append(CurrentList, [First], NCurrentList),
    order_boards_aux(Color, Board, Rest, NCurrentList, FinalList),!.

/*
```

```
 * valid_positions
 */
valid_positions(Board, Color, Number):-
    valid_positions(Board, Color, 0, 0, 0, Number).

valid_positions(_, _, Rowi, Coli, Number, Number):-
    rownum(R),
    colnum(C),
    Rowi is R-1,
    Coli is C,!.

valid_positions(Board, Color, RowIndex, Coli, CurrentNumber,
FinalNumber):-
    colnum(C),
    Coli is C,
    NRowIndex is RowIndex + 1,
    valid_positions(Board, Color, NRowIndex, 0, CurrentNumber,
FinalNumber),!.

valid_positions(Board, Color, RowIndex, ColumnIndex, CurrentNumber,
FinalNumber):-
    single_valid_move(Board, RowIndex, ColumnIndex, Color),
    NCurrentNumber is CurrentNumber + 1,
    NColumnIndex is ColumnIndex + 1,
    valid_positions(Board, Color, RowIndex, NColumnIndex,
NCurrentNumber, FinalNumber),!.

valid_positions(Board, Color, RowIndex, ColumnIndex, CurrentNumber,
FinalNumber):-
    NColumnIndex is ColumnIndex + 1,
    valid_positions(Board, Color, RowIndex, NColumnIndex,
CurrentNumber, FinalNumber),!.

/*
 * single_valid_move
 */
single_valid_move(Board, RowIndex, ColumnIndex, Color) :-
    piece(Board, RowIndex, ColumnIndex, Piece),
    Piece = empty,
    direction_offsets(DirectionOffsets),
```

```
    member(DirectionOffset, DirectionOffsets),
    nth0(0, DirectionOffset, RowOffset),
    nth0(1, DirectionOffset, ColumnOffset),
    NeighborRow is RowIndex + RowOffset,
    NeighborColumn is ColumnIndex + ColumnOffset,
    opponent_color(Color, OpponentColor),
    piece(Board, NeighborRow, NeighborColumn, NeighborPiece),
    NeighborPiece = OpponentColor,
    find_color(Board, NeighborRow, NeighborColumn, RowOffset,
ColumnOffset, Color),!.

/*
 * find_moves
 */
find_moves(Board, Color, MovesList):-
    find_moves(Board, Color, 0, 0, [], MovesList).

find_moves(_, _, Rowi, Coli, MovesList, MovesList):-
    rownum(R),
    colnum(C),
    Rowi is R-1,
    Coli is C,!.

find_moves(Board, Color, RowIndex, Coli, MovesList, FinalList):-
    colnum(C),
    Coli is C,
    NRowIndex is RowIndex + 1,
    find_moves(Board, Color, NRowIndex, 0, MovesList, FinalList),!.

find_moves(Board, Color, RowIndex, ColumnIndex, MovesList, FinalList):-
    valid_move(Board, RowIndex, ColumnIndex, Color,
ValidDirectionOffsets),
    append(MovesList,[[RowIndex, ColumnIndex, ValidDirectionOffsets]],
NMovesList),
    NColumnIndex is ColumnIndex + 1,
    find_moves(Board, Color, RowIndex, NColumnIndex, NMovesList,
FinalList),!.

find_moves(Board, Color, RowIndex, ColumnIndex, MovesList, FinalList):-
    NColumnIndex is ColumnIndex + 1,
```

```
    find_moves(Board, Color, RowIndex, NColumnIndex, MovesList,
FinalList),!.

/*
 * valid_move
 */
valid_move(Board, RowIndex, ColumnIndex, Color,
ValidDirectionOffsets):-
    piece(Board, RowIndex, ColumnIndex, Piece),
    Piece = empty,
    direction_offsets(DirectionOffsets),
    valid_move(Board, RowIndex, ColumnIndex, Color, DirectionOffsets,
[], ValidDirectionOffsets).

valid_move(_, _, _, _, [], CurrentValidDirectionOffsets,
ValidDirectionOffsets):-
    CurrentValidDirectionOffsets \= [],
    CurrentValidDirectionOffsets = ValidDirectionOffsets.

valid_move(Board, RowIndex, ColumnIndex, Color, DirectionOffsets,
CurrentValidDirectionOffsets, ValidDirectionOffsets):-
    DirectionOffsets = [DirectionOffset|DirectionOffsetsRest],
    valid_move_offset(Board, RowIndex, ColumnIndex, Color,
DirectionOffset),
    append(CurrentValidDirectionOffsets, [DirectionOffset],
NCurrentValidDirectionOffsets),
    valid_move(Board, RowIndex, ColumnIndex, Color,
DirectionOffsetsRest, NCurrentValidDirectionOffsets,
ValidDirectionOffsets).

valid_move(Board, RowIndex, ColumnIndex, Color, DirectionOffsets,
CurrentValidDirectionOffsets, ValidDirectionOffsets):-
    DirectionOffsets = [_|DirectionOffsetsRest],
    valid_move(Board, RowIndex, ColumnIndex, Color,
DirectionOffsetsRest, CurrentValidDirectionOffsets,
ValidDirectionOffsets).

valid_move_offset(Board, RowIndex, ColumnIndex, Color,
DirectionOffset):-
    piece(Board, RowIndex, ColumnIndex, Piece),
```

```
    Piece = empty,
    nth0(0, DirectionOffset, RowOffset),
    nth0(1, DirectionOffset, ColumnOffset),
    NeighborRow is RowIndex + RowOffset,
    NeighborColumn is ColumnIndex + ColumnOffset,
    opponent_color(Color, OpponentColor),
    piece(Board, NeighborRow, NeighborColumn, NeighborPiece),
    NeighborPiece = OpponentColor,
    find_color(Board, NeighborRow, NeighborColumn, RowOffset,
ColumnOffset, Color).

/*
 * find_color
 */
find_color(Board, RowIndex, ColumnIndex, RowOffset, ColumnOffset,
Color) :-
    NRowOffset is RowIndex + RowOffset,
    NColumnOffset is ColumnIndex + ColumnOffset,
    piece(Board, NRowOffset, NColumnOffset, Piece),
    Piece = Color.

find_color(Board, RowIndex, ColumnIndex, RowOffset, ColumnOffset,
Color) :-
    NRowIndex is RowIndex + RowOffset,
    NColumnIndex is ColumnIndex + ColumnOffset,
    piece(Board, NRowIndex, NColumnIndex, Piece),
    opponent_color(Color, OpponentColor),
    Piece = OpponentColor,
    find_color(Board, NRowIndex, NColumnIndex, RowOffset, ColumnOffset,
Color).

/*
 * set_piece
 */
set_piece(Board, Move, Color, FinalBoard):-
    nth0(0, Move, Row),
    nth0(1, Move, Column),
    nth0(2, Move, ValidDirectionOffsets),
    set_single_piece(Board, Row, Column, Color, BoardWithPiece),
```

```
    set_pieces_on_offsets(BoardWithPiece, Row, Column, Color,
ValidDirectionOffsets, FinalBoard).

set_piece(Board, PieceRowIndex, PieceColumnIndex, Color, FinalBoard):-
    valid_move(Board, PieceRowIndex, PieceColumnIndex, Color,
ValidDirectionOffsets),
    set_single_piece(Board, PieceRowIndex, PieceColumnIndex, Color,
BoardWithPiece),
    set_pieces_on_offsets(BoardWithPiece, PieceRowIndex,
PieceColumnIndex, Color, ValidDirectionOffsets, FinalBoard).

/*
 * set_pieces_on_offsets
 */
set_pieces_on_offsets(FinalBoard, _, _, _, [], FinalBoard):-!.

set_pieces_on_offsets(Board, PieceRowIndex, PieceColumnIndex, Color,
ValidDirectionOffsets, FinalBoard):-
    ValidDirectionOffsets =
[ValidDirectionOffset|ValidDirectionOffsetsRest],
    set_pieces_on_offset(Board, PieceRowIndex, PieceColumnIndex, Color,
ValidDirectionOffset, TempBoard),
    set_pieces_on_offsets(TempBoard, PieceRowIndex, PieceColumnIndex,
Color, ValidDirectionOffsetsRest, FinalBoard).

set_pieces_on_offset(Board, PieceRowIndex, PieceColumnIndex, Color,
ValidDirectionOffset, FinalBoard):-
    nth0(0, ValidDirectionOffset, RowOffset),
    nth0(1, ValidDirectionOffset, ColumnOffset),
    NRowOffset is PieceRowIndex + RowOffset,
    NColumnOffset is PieceColumnIndex + ColumnOffset,
    piece(Board, NRowOffset, NColumnOffset, Piece),
    Piece = Color,
    Board = FinalBoard,!.

set_pieces_on_offset(Board, PieceRowIndex, PieceColumnIndex, Color,
ValidDirectionOffset, FinalBoard):-
    nth0(0, ValidDirectionOffset, RowOffset),
    nth0(1, ValidDirectionOffset, ColumnOffset),
    NRowOffset is PieceRowIndex + RowOffset,
```

```
    NColumnOffset is PieceColumnIndex + ColumnOffset,
    piece(Board, NRowOffset, NColumnOffset, Piece),
    opponent_color(Color, OpponentColor),
    Piece = OpponentColor,
    set_single_piece(Board, NRowOffset, NColumnOffset, Color,
TempBoard),
    set_pieces_on_offset(TempBoard, NRowOffset, NColumnOffset, Color,
ValidDirectionOffset, FinalBoard).

/*
 * set_single_piece
 */
set_single_piece(Board, PieceRowIndex, PieceColumnIndex, Color,
FinalBoard):-
    set_single_piece(Board, PieceRowIndex, PieceColumnIndex, 0, 0,
Color, [], FinalBoard, []).

set_single_piece(_, PieceRowIndex, _, Rowi, Coli, _, ResultingBoard,
FinalBoard, PieceRow):-
    rownum(R),
    colnum(C),
    PieceRowIndex is R-1,
    Rowi is R-1,
    Coli is C,
    append(ResultingBoard, [PieceRow], FinalBoard),!.

set_single_piece(_, _, _, Rowi, 0, _, FinalBoard, FinalBoard, _):-
    rownum(R),
    Rowi is R,!.

set_single_piece(Board, PieceRowIndex, ColumnRowIndex, PieceRowIndex,
Coli, Color, ResultingBoard, FinalBoard, RowIndex):-
    rownum(R),
    colnum(C),
    Coli is C,
    PieceRowIndex \= (R-1),
    NCurrentRowIndex is PieceRowIndex + 1,
    append(ResultingBoard, [RowIndex], NResultingBoard),
    set_single_piece(Board, PieceRowIndex, ColumnRowIndex,
NCurrentRowIndex, 0, Color, NResultingBoard, FinalBoard, []).
```

```
set_single_piece(Board, PieceRowIndex, PieceColumnIndex, PieceRowIndex,
PieceColumnIndex, Color, ResultingBoard, FinalBoard, PieceRow):-
    append(PieceRow, [Color], NPieceRow),
    NCurrentColumnIndex is PieceColumnIndex + 1,
    set_single_piece(Board, PieceRowIndex, PieceColumnIndex,
PieceRowIndex, NCurrentColumnIndex, Color, ResultingBoard, FinalBoard,
NPieceRow).

set_single_piece(Board, PieceRowIndex, PieceColumnIndex, PieceRowIndex,
CurrentColumnIndex, Color, ResultingBoard, FinalBoard, PieceRow):-
    CurrentColumnIndex \= PieceColumnIndex,
    piece(Board, PieceRowIndex, CurrentColumnIndex, Piece),
    append(PieceRow, [Piece], NPieceRow),
    NCurrentColumnIndex is CurrentColumnIndex + 1,
    set_single_piece(Board, PieceRowIndex, PieceColumnIndex,
PieceRowIndex, NCurrentColumnIndex, Color, ResultingBoard, FinalBoard,
NPieceRow).

set_single_piece(Board, PieceRowIndex, PieceColumnIndex,
CurrentRowIndex, _, Color, ResultingBoard, FinalBoard, PieceRow):-
    PieceRowIndex \= CurrentRowIndex,
    nth0(CurrentRowIndex, Board, CurrentRow),
    append(ResultingBoard, [CurrentRow], NResultingBoard),
    NCurrentRowIndex is CurrentRowIndex + 1,
    set_single_piece(Board, PieceRowIndex, PieceColumnIndex,
NCurrentRowIndex, 0, Color, NResultingBoard, FinalBoard, PieceRow).

/*
 * count_pieces
 */
count_pieces(Color,Board,Pieces,RivalPieces):-
    count_pieces(0,0,Color,Board,0,0,Pieces,RivalPieces).

count_pieces(Rowi,Coli,_,_,PiecesBuf,RivalPiecesBuf,Pieces,RivalPieces)
:-
    rownum(R),
    colnum(C),
    Rowi is R-1,
    Coli is C,
```

```
    Pieces = PiecesBuf,
    RivalPieces is RivalPiecesBuf,
    !.

count_pieces(Rowi,Coli,Color,Board,PiecesBuf,RivalPiecesBuf,Pieces,Riva
lPieces):-
    colnum(C),
    opponent_color(Color,OpponentColor),
    (
        Coli=C ->
            NRowi is Rowi+1,
            NColi is 0,
            NPiecesBuf is PiecesBuf,
            NRivalPiecesBuf is RivalPiecesBuf
        ;
        NRowi is Rowi,
        NColi is Coli+1,
        piece(Board, Rowi, Coli, Piece),
        (
            Piece = Color ->
                NPiecesBuf is PiecesBuf+1,
                NRivalPiecesBuf is RivalPiecesBuf
            ;
            Piece = OpponentColor ->
                NPiecesBuf is PiecesBuf,
                NRivalPiecesBuf is RivalPiecesBuf +1
            ;
            NPiecesBuf is PiecesBuf,
            NRivalPiecesBuf is RivalPiecesBuf
        )
    ),

count_pieces(NRowi,NColi,Color,Board,NPiecesBuf,NRivalPiecesBuf,Pieces,
RivalPieces).
/*
 * direction_offsets
 */
direction_offsets(OffsetsList) :-
    OffsetsList = [[-1, 0],
            [-1, 1],
```

```
            [0, 1],
            [1, 1],
            [1, 0],
            [1, -1],
            [0, -1],
            [-1,-1]].

getRowCol(R,C):-
    rownum(R),
    colnum(C).
```

Alpha_beta.pl

```
% alpha_beta_pruning/5: Searches for a move using the alpha-beta
pruning algorithm
%   State:current board
%   New state: edited board
%   Value: heuristic value of the move

alpha_beta_pruning(State, Depth, Color, NewState, Value):-
alpha_beta_pruning(Color,Depth, State, Color, NewState, Value, -100000,
100000).

% alpha_beta_pruning/7: with an alpha and a beta value
alpha_beta_pruning(Caller,_, State, _, State, Value, _, _) :-
final(Caller, State, Value),!.

alpha_beta_pruning(Caller,0, State, _, State, Value, _, _) :-
evaluator(Caller, State, Value),!.

alpha_beta_pruning(Caller,Depth, State, Color, NewState, Value, Alpha,
Beta) :-
Depth > 0,
find_states(Caller, State, Color, StatesList),
opponent_color(Color, OpponentColor),
NDepth is Depth - 1,
alpha_beta_pruning(Caller,StatesList, NDepth, Color, OpponentColor,
NewState, Value, Alpha, Beta).

% alpha_beta_pruning/8: with an alpha and a beta value
```

```
alpha_beta_pruning(Caller,[State], Depth, _, OpponentColor, State,
Value, Alpha, Beta):- !,
    alpha_beta_pruning(Caller,Depth, State, OpponentColor, _, Value,
Alpha, Beta).

alpha_beta_pruning(Caller,[State|Rest], Depth, Color, OpponentColor,
NewState, Value, Alpha, Beta) :-
    alpha_beta_pruning(Caller,Depth, State, OpponentColor, _, X, Alpha,
Beta),
    (
        prune(Color, X, Alpha, Beta) ->
        (
            NewState = State,
            Value is X
        );
        (
            recalc(Color, X, Alpha, Beta, Nalpha, NBeta),
            alpha_beta_pruning(Caller,Rest, Depth, Color,
OpponentColor, B, Y, Nalpha, NBeta),
            best(Color, X, Y, State, B, NewState, Value)
        )

    ).

% prune/4: prune has to be done
%       1: Color - the color of the player that moves next
%       2: Value - the value of the move
%       3: Alpha - the current best value for the player that tries to
minimize the game value
%       4: Beta - the current best value for the player that tries to
maximize the game value
prune(black, Value, _, Beta):-
Value >= Beta.

prune(white, Value, Alpha, _):-
Value =< Alpha.

% recalc/6: Recalculates alpha and beta values
%       1: Color - the color of the player that moves next
%       2: Value - the value of the move
```

```
%        3: Alpha - the current best value for the player that tries to
minimize the game value
%        4: Beta - the current best value for the player that tries to
maximize the game value
%        5: Nalpha - the new alpha
%        6: Nbeta - the new beta
recalc(black, Value, Alpha, Beta, Nalpha, Beta):-
max_list([Alpha, Value], Nalpha).

recalc(white, Value, Alpha, Beta, Alpha, NBeta):-
min_list([Beta, Value], NBeta).

% best/7: Calculates the best value depending on the color that moves
next
best(black, X, Y, A, _, A, X):- X>=Y,!.
best(black, _, Y, _, B, B, Y).
best(white, X, Y, A, _, A, X):- X=<Y, !.
best(white, _, Y, _, B, B, Y).
```

```
?- play(3).
Select a game mode
1. human vs machine
2. machine vs human
3. human vs human
4. machine vs machine
4.
Enter a number: machine vs machine (black first) selected
```

```
   0 1 2 3 4 5 6 7
 -----------------
0 | - - - - - - - -
1 | - - - - - - - -
2 | - - - - - - - -
3 | - - - X O - - -
4 | - - - O X - - -
5 | - - - - - - - -
6 | - - - - - - - -
7 | - - - - - - - -
```

```
   0 1 2 3 4 5 6 7
 -----------------
0 | O O O O O O O O
1 | O O O O O O O O
2 | O O O O X O O O
3 | O O O X O O O O
4 | O O O O O X X O
5 | O X O O X X X O
6 | O X O O O O O O
7 | O X O O O O O O

White (O) player's turn

black: 10        white: 54
white (O) win

true.

?- 
```

```
   0 1 2 3 4 5 6 7
 -----------------
0 | - - - - - - - -
1 | - - - - - - - -
2 | - - - - - - - -
3 | - - - X O - - -
4 | - - - O X - - -
5 | - - - - - - - -
6 | - - - - - - - -
7 | - - - - - - - -

Black (X) player's turn
|: 5.
Enter the Row: Enter the Column:
|: 3.
```