EECS 3401 Winter 2017

Assignment 4

Family Name: Shen

First Name: Kevin

Student Number: 212298535

EECS login: kshen94

Preferred email address: kshen94@my.yorku.ca

```
/* -----------------------------------------------------------
    CSE 3401 F12 Assignment 4 file

% Surname: Shen
% First Name: Kevin
% Student Number: 212298535

  ------------------------------------------------------- */

%do not chagne the follwoing line!
:- ensure_loade('play.pl').
:- ensure_loade('testboards.pl').

% /* -------------------------------------------------------- */
%              IMPORTANT! PLEASE READ THIS SUMMARY:
%       This files gives you some useful helpers (set &get).
%       Your job is to implement several predicates using
%       these helpers (feel free to add your own helpers if needed,
%       MAKE SURE to write comments for all your helpers, marks will
%       be deducted for bad style!).
%
%       Implement the following predicates at their designated space
%       in this file (we suggest to have a look at file ttt.pl  to
%       see how the implementations is done for game tic-tac-toe.
%
%          * initialize(InitialState,InitialPlyr).
%          * winner(State,Plyr)
%          * tie(State)
%          * terminal(State)
%          * moves(Plyr,State,MvList)
%          * nextState(Plyr,Move,State,NewState,NextPlyr)
%          * validmove(Plyr,State,Proposed)
%          * h(State,Val)  (see question 2 in the handout)
%          * lowerBound(B)
%          * upperBound(B)
% /* -------------------------------------------------------- */







% /* -------------------------------------------------------- */

% We use the following State Representation:
% [Row0, Row1 ... Rown] (ours is 6x6 so n = 5 ).
% each Rowi is a LIST of 6 elements '.' or '1' or '2' as follows:
%    . means the position is  empty
%    1 means player one has a stone in this position
%    2 means player two has a stone in this position.



% given helper: Inital state of the board
initBoard([ [.,.,.,.,.,.],
            [.,.,.,.,.,.],
            [.,.,1,2,.,.],
            [.,.,2,1,.,.],
            [.,.,.,.,.,.],
            [.,.,.,.,.,.] ]).


%Utilities
%%%%%%%%%%%%%%%%%%%%

%%Opponent Facts
%Opponent of Player 1 is 2
opp(1,2).
%Opponent of Player 2 is 1
```

```
opp(2,1).

%%%%%%%% countPoints(State,Start1,Start2,Points1,Points2)
% Counts points by counting the amount of each player's tiles are on the
% board, S1 and S2 are initiated at 0
countPoints([],S1,S2,P1,P2):- P1 is S1, P2 is S2.
countPoints(State,S1,S2,P1,P2):- State = [H|T],  countRows(H,0,0,R1,R2),
    X is S1+R1, Y is S2+R2, countPoints(T,X,Y,P1,P2).

%!  %%%%%%% countRows(Row,Start1, Start2, RowPoint1,RowPoint2)
% Counts the amount of points in a row and adds it to the other sum of
% the other rows
countRows([],S1,S2,R1,R2):- R1 is S1,R2 is S2.
countRows(Row,S1,S2,R1,R2):- Row = [H|T], (H = 1 -> X is S1+1, Y is S2;
H = 2 -> Y is S2+1, X is S1; X is S1,Y is S2), countRows(T,X,Y,R1,R2).

%!  %%% search(State,X,Y, Return)
%Searches the board for a pair [X,Y] and returns its symbol
search(State,X,Y,Return):- nth0(Y,State,Row),nth0(X,Row,Return).

%!  %%%%% changeList(List,Index,Input,Return)
%Changes a element of a list at a given Index with the given input
%and returns a new List
changeList([_|T],0,X,[X|T]).
changeList([H|T],I,Player,[H|R]):- I >0, I1 is I-1, changeList(T,I1,Player,R)
,!.



%%%%%%%%%%%%%%%%%%% IMPLEMENT: initialize(...)%%%%%%%%%%%%%%%%%%%%%%%
%%% Using initBoard define initialize(InitialState,InitialPlyr).
%%%  holds iff InitialState is the initial state and
%%%  InitialPlyr is the player who moves first.
%

initialize(X,1):- initBoard(X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%winner(...)%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% define winner(State,Plyr) here.
%     - returns winning player if State is a terminal position and
%     Plyr has a higher score than the other player
%     Uses countPoints/5 to determine whose the winner

winner(State,Player):- terminal(State),countPoints(State,0,0,P1,P2),!,
    P1 =\= P2,
    (P1 @> P2 -> opp(2,Player); opp(1,Player)).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%tie(...)%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% define tie(State) here.
%    - true if terminal State is a "tie" (no winner)
%    Uses countPoints/5 to verify it is a tie

tie(State):- terminal(State),countPoints(State,0,0,P1,P2), P1 == P2.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%terminal(...)%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% define terminal(State).
%   - true if State is a terminal
%   If Player 1 and Player 2 has no valid moves then the game is in
%   terminal state

terminal(State):- moves(1,State,L1), moves(2,State,L2),!, L1==[], L2==[].
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%showState(State)%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% given helper. DO NOT  change this. It's used by play.pl
%%
showState( G ) :-
        printRows( G ).

printRows( [] ).
printRows( [H|L] ) :-
        printList(H),
        nl,
        printRows(L).
printList([]).
printList([H | L]) :-
        write(H),
        write(' '),
        printList(L).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%moves(Plyr,State,MvList)%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%%
%% define moves(Plyr,State,MvList).
%   - returns list MvList of all legal moves Plyr can make in State
%   finds all empty spots on the board and uses validmoves/4 to create a
%   list of all legal moves

moves(Player,State,List):- findall([X,Y],search(State,X,Y,'.'),L),
   validmoves(Player,State,L,L1),reverse(L1,L2),reverse(L2,List).



%%%%%%%%%%%% validmoves(Player,State,List,Return)%%%%%%%%%%%%%%%
% returns list of valid moves given a list of non-occupied places on
% the board
% If 'n' is given, check if there are legal moves

validmoves(Player,State,[],[]):-validmoves(Player,State,[],[n]).
validmoves(Player,State,[],X).
validmoves(Player,State,[H|T],X):- validmove(Player,State,H)-> member(H,X),
    validmoves(Player,State,T,X);
    validmoves(Player,State,T,X).


%%%%%%%%%%%%%nextState(Plyr,Move,State,NewState,NextPlyr)%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% define nextState(Plyr,Move,State,NewState,NextPlyr).
%   - given that Plyr makes Move in State, it determines NewState (i.e. the n
ext
%     state) and NextPlayer (i.e. the next player who will move).
%     Given a move, it will place it on the board using playmove/5 and
%     flips all legal tiles using flipX/5
%     if [n] is given as a move, NewState is State

nextState(Player,[n],State,NewState,NextPlayer):- opp(Player,NextPlayer),
    NewState = State.
nextState(Player,Move,State,NewState,NextPlayer):- opp(Player,NextPlayer),Mov
e = [X,Y],
    playmove(Player,X,Y,State,S1),
    flipN(Player,S1,X,Y,S2), flipS(Player,S2,X,Y,S3),flipE(Player,S3,X,Y,S4),
    flipW(Player,S4,X,Y,S5), flipNW(Player,S5,X,Y,S6), flipNE(Player,S6,X,Y,S
7),
    flipSW(Player,S7,X,Y,S8),flipSE(Player,S8,X,Y,S9), NewState = S9.

%!  %% playmove(Player,X,Y,State,NewState)
%Places the given move onto the board
%    playmove(Player,X,Y,State,NewState):- nth0(Y,State,Row),
%    append([Head,[Row],Tail],State),changeRow(Row,X,Player,NewRow),
%    append([Head,[NewRow],Tail],NewState).

playmove(Player,X,Y,State,NewState):- nth0(Y,State,Row),
```

```
    changeList(Row,X,Player,NewRow),changeList(State,Y,NewRow,NewState).

%!  %%% flipX(Player,State,X,Y,NewState)
% All the flipX/5 checks for valid tiles in its current State to flip in
% all directions of the placed tile and returns it as NewState
    flipN(Player,State,X,Y,NewState):- verifyN(Player,State,X,Y,0)-> B is Y-1
,
    nth0(B,State,Row),append([Head,[Row],Tail],State),changeList(Row,X,Player
,NewRow),
    append([Head,[NewRow],Tail],NS),flipN(Player,NS,X,B,NewState); NewState =
 State.

    flipS(Player,State,X,Y,NewState):- verifyS(Player,State,X,Y,0)-> B is Y+1
,
    nth0(B,State,Row),append([Head,[Row],Tail],State),changeList(Row,X,Player
,NewRow),
    append([Head,[NewRow],Tail],NS),flipS(Player,NS,X,B,NewState); NewState =
 State.

    flipE(Player,State,X,Y,NewState):- verifyE(Player,State,X,Y,0)-> A is X+1
,
    nth0(Y,State,Row),append([Head,[Row],Tail],State),changeList(Row,A,Player
,NewRow),
    append([Head,[NewRow],Tail],NS),flipE(Player,NS,A,Y,NewState); NewState =
 State.

    flipW(Player,State,X,Y,NewState):- verifyW(Player,State,X,Y,0)-> A is X-1
,
    nth0(Y,State,Row),append([Head,[Row],Tail],State),changeList(Row,A,Player
,NewRow),
    append([Head,[NewRow],Tail],NS),flipW(Player,NS,A,Y,NewState); NewState =
 State.

    flipNE(Player,State,X,Y,NewState):- verifyNE(Player,State,X,Y,0)-> A is X
+1, B is Y-1,
    nth0(B,State,Row),append([Head,[Row],Tail],State),changeList(Row,A,Player
,NewRow),
    append([Head,[NewRow],Tail],NS),!,flipNE(Player,NS,A,B,NewState); NewStat
e = State.


    flipNW(Player,State,X,Y,NewState):- verifyNW(Player,State,X,Y,0)-> A is X
-1, B is Y-1,
    nth0(B,State,Row),append([Head,[Row],Tail],State),changeList(Row,A,Player
,NewRow),
    append([Head,[NewRow],Tail],NS),flipNW(Player,NS,A,B,NewState); NewState
= State.

    flipSE(Player,State,X,Y,NewState):- verifySE(Player,State,X,Y,0)-> A is X
+1, B is Y+1,
    nth0(B,State,Row),append([Head,[Row],Tail],State),changeList(Row,A,Player
,NewRow),
    append([Head,[NewRow],Tail],NS),flipSE(Player,NS,A,B,NewState); NewState
= State.

    flipSW(Player,State,X,Y,NewState):- verifySW(Player,State,X,Y,0)-> A is X
-1, B is Y+1,
    nth0(B,State,Row),append([Head,[Row],Tail],State),changeList(Row,A,Player
,NewRow),
    append([Head,[NewRow],Tail],NS),flipSW(Player,NS,A,B,NewState); NewState
= State.


%%%%%%%%%%%%%%%%%%%%%%%%%%%validmove(Plyr,State,Proposed)%%%%%%%%%%%%%%%%%%%
%
%%
%% define validmove(Plyr,State,Proposed).
%   - true if Proposed move by Plyr is valid at State.
%   Checks whether the proposed move is legal by using verifyX/5 to
%   check in all directions
```

```
validmove(Player,State,[n]):- moves(Player,State,L),!, L == [].

validmove(Player,State,Proposed):- Proposed =[X,Y], search(State,X,Y,R),
 R == '.' ->
 (verifyN(Player,State,X,Y,0);verifyS(Player,State,X,Y,0);
 verifyE(Player,State,X,Y,0); verifyW(Player,State,X,Y,0);
 verifyNE(Player,State,X,Y,0);verifyNW(Player,State,X,Y,0);
 verifySW(Player,State,X,Y,0);verifySE(Player,State,X,Y,0)); fail.

%!  %%%% verifyX(Player,State,X,Y,C)
% All verifyX/5 checks whether the tile placed in its current State is
% valid in its corresponding direction
verifyN(Player,State,X,Y,C):- B is Y-1, opp(Player,Opp),
 search(State,X,B,R),(R == Opp -> C1 is C+1,verifyN(Player,State,X,B,C1);
 C > 0, R == Player -> true; fail).

verifyS(Player,State,X,Y,C):- B is Y+1, opp(Player,Opp),
 search(State,X,B,R) ,(R == Opp -> C1 is C+1,verifyS(Player,State,X,B,C1);
 C > 0, R == Player -> true; fail).

verifyE(Player,State,X,Y,C):- A is X+1, opp(Player,Opp),
    search(State,A,Y,R),(R == Opp -> C1 is C+1, verifyE(Player,State,A,Y,C1);
                       C > 0, R == Player -> true; fail).

verifyW(Player,State,X,Y,C):- A is X-1, opp(Player,Opp),
    search(State,A,Y,R),(R == Opp -> C1 is C+1, verifyW(Player,State,A,Y,C1);
                       C > 0, R == Player -> true; fail).

verifyNW(Player,State,X,Y,C):- B is Y-1, A is X-1, opp(Player,Opp),
    search(State,A,B,R), (R == Opp -> C1 is C+1, verifyNW(Player,State,A,B,C1
);
                        C> 0, R == Player -> true; fail).

verifyNE(Player,State,X,Y,C):- A is X+1, B is Y-1, opp(Player,Opp),
    search(State,A,B,R), (R == Opp -> C1 is C+1, verifyNE(Player,State,A,B,C1
);
                        C>0,R == Player -> true; fail).

verifySW(Player,State,X,Y,C):- A is X-1, B is Y+1, opp(Player,Opp),
    search(State,A,B,R), (R == Opp -> C1 is C+1, verifySW(Player,State,A,B,C1
);
                        C>0, R == Player -> true; fail).

verifySE(Player,State,X,Y,C):- A is X+1, B is Y+1, opp(Player,Opp),
    search(State,A,B,R), (R == Opp -> C1 is C+1, verifySE(Player, State,A,B,C
1);
                        C>0, R == Player -> true; fail).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%h(State,Val)%%%%%%%%%%%%%%%%%%%%%%%%
%
%%
%% define h(State,Val).
%   - given State, returns heuristic Val of that state
%   - larger values are good for Max, smaller values are good for Min
%   NOTE1. If State is terminal h should return its true value.
%   NOTE2. If State is not terminal h should be an estimate of
%          the value of state (see handout on ideas about
%          good heuristics.
%
%          This heuristics will calculate the
%current amount of points  + amount of corner tiles of each player*3 +
%the amount of tiles adjacent to corner tiles.
% h(v) = P1 - P2
% P1 = Tile Points + #corner*3 + adj to corners
% P2 = Tile Points + #corner*3 + adj to corners
%
% This puts more emphasis on the corner and stable tiles
% Winning and Losing is +100/-100
```

```
h(State,V):- countPoints(State,0,0,A1,B1),
    countcorner(State,A2,B2),countadjcorner(State,A3,B3),
    P1 is A1+A2+A3, P2 is B1+B2+B3, V is P1-P2.
h(State,100):- winner(State,1).
h(State,-100):- winner(State,2).

%!  %%%%%% countcorner(State,Player1,Player2)
%This calculates the h value from corner tiles
%
countcorner(State,P1,P2):- checkcorner(State,TL,TR,BL,BR), checktile(TL,A1,B1
),
    checktile(TR,A2,B2),checktile(BL,A3,B3),checktile(BR,A4,B4),
    P1 is A1+A2+A3+A4, P2 is B1+B2+B3+B4.

%!  %%%%% checkcorner(State,TL,TR,BL,BR)
%Obtains the corner tiles in a given states
checkcorner(State,TL,TR,BL,BR):- nth0(0,State,Top),nth0(0,Top,TL),nth0(5,Top,
TR), nth0(5,State,Bot),nth0(0,Bot,BL),nth0(5,Bot,BR).

%!  %%%%% checktile(Tile, Points1, Points2)
% gives h values depending on the tile
checktile(Tile,P1,P2):- Tile == '.'-> P1 is 0, P2 is 0;
Tile == 1 -> P1 is 3, P2 is 0; P2 is 3, P1 is 0.

%!  %%%%%countadjcorner(State,Point1, Point2)
%Calculates the h value of tiles that are adjacent to corner tiles
%
countadjcorner(State,P1,P2):- checkcorner(State,TL,TR,BL,BR),
countTL(State,TL,A1,B1),countTR(State,TR,A2,B2),countBL(State,BL,A3,B3),
    countBR(State,BR,A4,B4), P1 is A1+A2+A3+A4, P2 is B1+B2+B3+B4.

%!  %%%% countXX(State,XX,P1,P2)
%calculates the h value given from adjacent tiles of each corner
countTL(State,TL,P1,P2):- TL == '.' -> P1 is 0, P2 is 0;
countRight(State,0,0,TL,0,A1,B1),
    countDown(State,0,0,TL,0,A2,B2), P1 is A1+A2, P2 is B1+B2.
countTR(State,TR,P1,P2):- TR == '.' -> P1 is 0, P2 is 0;
countLeft(State,5,0,TR,0,A1,B1),
    countDown(State,5,0,TR,0,A2,B2), P1 is A1+A2, P2 is B1+B2.
countBL(State,BL,P1,P2):- BL == '.' -> P1 is 0, P2 is 0;
countRight(State,0,5,BL,0,A1,B1),
    countUp(State,0,5,BL,0,A2,B2), P1 is A1+A2, P2 is B1+B2.
countBR(State,BR,P1,P2):- BR == '.' -> P1 is 0, P2 is 0;
countLeft(State,5,5,BR,0,A1,B1),
    countUp(State,5,5,BR,0,A2,B2), P1 is A1+A2, P2 is B1+B2.

%!  %%%%% countX(State,X,Y,Player,C,Point1, Point2)
%counts the amount of consecutive tiles adjacent to the corner tile
%

countRight(State,4,Y,Player,C,P1,P2):- Player == 1 -> P1 is C,P2 is 0;
    P2 is C, P1 is 0.
countRight(State,X,Y,Player,C, P1, P2):- A is X+1, search(State,A,Y,R),
    R == Player -> C1 is C+1, countRight(State,A,Y,Player,C1,P1,P2);
    Player == 1 -> P1 is C, P2 is 0; P2 is C, P1 is 0.

countDown(State,X,4,Player,C,P1,P2):- Player == 1 -> P1 is C,P2 is 0;
    P2 is C, P1 is 0.
countDown(State,X,Y,Player,C, P1, P2):- B is Y+1, search(State,X,B,R),
    R == Player -> C1 is C+1, countDown(State,X,B,Player,C1,P1,P2);
    Player == 1 -> P1 is C, P2 is 0; P2 is C, P1 is 0.

countLeft(State,1,Y,Player,C,P1,P2):- Player == 1 -> P1 is C,P2 is 0;
    P2 is C, P1 is 0.
countLeft(State,X,Y,Player,C, P1, P2):- A is X-1, search(State,A,Y,R),
    R == Player -> C1 is C+1, countLeft(State,A,Y,Player,C1,P1,P2);
    Player == 1 -> P1 is C, P2 is 0; P2 is C, P1 is 0.

countUp(State,X,1,Player,C,P1,P2):- Player == 1 -> P1 is C,P2 is 0;
```

```
    P2 is C, P1 is 0.
countUp(State,X,Y,Player,C, P1, P2):- B is Y-1, search(State,X,B,R),
    R == Player -> C1 is C+1, countUp(State,X,B,Player,C1,P1,P2);
    Player == 1 -> P1 is C, P2 is 0; P2 is C, P1 is 0.




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%lowerBound(B)%%%%%%%%%%%%%%%%%%%%%%%
%%
%%
%% define lowerBound(B).
%   - returns a value B that is less than the actual or heuristic value
%     of all states.

lowerBound(-101).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%upperBound(B)%%%%%%%%%%%%%%%%%%%%%%%
%%
%%
%% define upperBound(B).
%   - returns a value B that is greater than the actual or heuristic value
%     of all states.

upperBound(101).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                          %%
%                                                                          %%
%                 Given   UTILITIES                                        %%
%                    do NOT change these!                                  %%
%                                                                          %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% get(Board, Point, Element)
%    : get the contents of the board at position column X and row Y
% set(Board, NewBoard, [X, Y], Value):
%    : set Value at column X row Y in Board and bind resulting grid to NewBoa
rd
%
% The origin of the board is in the upper left corner with an index of
% [0,0], the upper right hand corner has index [5,0], the lower left
% hand corner has index [0,5], the lower right hand corner has index
% [5,5] (on a 6x6 board).
%
% Example
% ?- initBoard(B), showState(B), get(B, [2,3], Value).
%. . . . . .
%. . . . . .
%. . 1 2 . .
%. . 2 1 . .
%. . . . . .
%. . . . . .
%
%B = [['.', '.', '.', '.', '.', '.'], ['.', '.', '.', '.', '.', '.'],
%     ['.', '.', 1, 2, '.', '.'], ['.', '.', 2, 1, '.'|...],
%     ['.', '.', '.', '.'|...], ['.', '.', '.'|...]]
%Value = 2
%Yes
%?-
%
% Setting values on the board
% ?- initBoard(B),  showState(B),set(B, NB1, [2,4], 1), set(NB1, NB2, [2 3],
1),  showState(NB2).
%
% . . . . . .
% . . . . . .
% . . 1 2 . .
% . . 2 1 . .
% . . . . . .
% . . . . . .
```

```
%
% . . . . . .
% . . . . . .
% . . 1 2 . .
% . . 1 1 . .
% . . 1 . . .
% . . . . . .
%
%B = [['.', '.', '.', '.', '.', '.'], ['.', '.', '.', '.', '.', '.'], [' ', '
.',
%1, 2, '.', '.'], ['.', '.', 2, 1, '.'|...], ['.', '.', '.', '.'|...], [ .',
'.',
% '.'|...]]
%NB1 = [['.', '.', '.', '.', '.', '.'], ['.', '.', '.', '.', '.', '.'], '.',
 '.'
%, 1, 2, '.', '.'], ['.', '.', 2, 1, '.'|...], ['.', '.', 1, '.'|...], [ .',
'.
%', '.'|...]]
%NB2 = [['.', '.', '.', '.', '.', '.'], ['.', '.', '.', '.', '.', '.'], '.',
 '.'
%, 1, 2, '.', '.'], ['.', '.', 1, 1, '.'|...], ['.', '.', 1, '.'|...], [ .',
%'.', '.'|...]]

% get(Board, Point, Element): get the value of the board at position
% column X and row Y (indexing starts at 0).
get( Board, [X, Y], Value) :-
        nth0( Y, Board, ListY),
        nth0( X, ListY, Value).

% set( Board, NewBoard, [X, Y], Value)

set( [Row|RestRows], [NewRow|RestRows], [X, 0], Value)
    :- setInList(Row, NewRow, X, Value).

set( [Row|RestRows], [Row|NewRestRows], [X, Y], Value) :-
        Y > 0,
        Y1 is Y-1,
        set( RestRows, NewRestRows, [X, Y1], Value).

% setInList( List, NewList, Index, Value)

setInList( [_|RestList], [Value|RestList], 0, Value).

setInList( [Element|RestList], [Element|NewRestList], Index, Value) :-
        Index > 0,
        Index1 is Index-1,
        setInList( RestList, NewRestList, Index1, Value).
```

Assignment 4: Tracing 3 simple runs of your code

Name:..... Student no:.....

Please submit these 2 pages together with your paper submission. Remember even if you decide not to do part II of the assignment, you still have to implement the mentioned simple heuristic (that always returns 0 except that it returns a high value for a state where 1 wins, a low value for a state where 2 wins).

1. **Testing your code with MiniMax**: Trace your code on test cases 1 to 3 (provided in `testboards.pl`) using the MiniMax algorithm when using the depth bound 5, by doing the following:

   ```
   testBoard1(St), mmeval(2,St,Val,BestMv,5,SeF)
   ```

   this will bind SeF to the number of states searched, and BestMv to the computed move for board1. Repeat this for testBoard2 and testBoard3. Write down the results in the following table.

Test Boards	*regular MiniMax*	
	# Expanded Nodes	*Applied Move [X,Y]*
Test Board 1		
Test Board 2		
Test Board 3		

2. **Testing alpha-beta prunning**: Trace your code on test cases 1 to 3 (provided in `testboards.pl`) using the alpha-beta algorithm when using the depth bound 5, by doing the following (assuming you have named alpha-beta predicate `abmmeval`):

   ```
   testBoard1(St), abmmeval(2,St,Val,BestMv,5,SeF)
   ```

   This will bind SeF to the number of states searched, and BestMv to the computed move for board1. Repeat this for testBoard2 and testBoard3. Write down the results in the following table.

Test Boards	α-β pruning	
	# Expanded Nodes	*Applied Move [X,Y]*
Test Board 1		
Test Board 2		
Test Board 3		

3. Write down one or two paragraphs comparing the results on the two tables.

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .