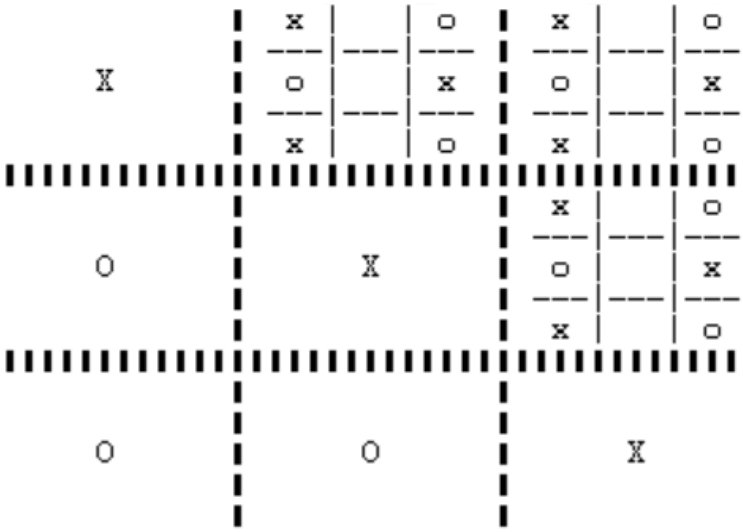
Traditional Project: Alpha-beta project from Bratko book (pag. 368)



Students: Francesco Zangrillo (0001031021), Simone Mele (0001037557)

Table of Contents

User manual

Start the game with the following command:

start_game(<Game>, <player_x_type>, <player_o_type>).

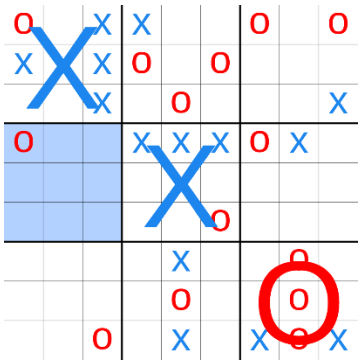
start_game(tic_tac_toe|ultimate, ai|human, ai|human).

e.g. start_game(ultimate, ai, human).

In this project we implemented the Minimax algorithm with alpha-beta cuts for playing two-person, perfect information games. For this purpose, we implemented a variant of the Tic Tac Toe game: Ultimate Tic Tac Toe.

1. Ultimate Tic Tac Toe

Ultimate tic-tac-toe is a board game composed of nine tic-tac-toe boards arranged in a 3 × 3 grid. Players take turns playing in the smaller tic-tac-toe boards until one of them wins in the larger tic-tac-toe board.



The game starts with X playing wherever they want in any of the 81 empty spots. If a move is played so as to win a local board according to normal tic-tac-toe rules, then the entire local board is marked as a victory for the player in the global board.

Each small 3 × 3 tic-tac-toe board is referred to as a miniboard, and the larger 3 × 3 board is referred to as the global board. Therefore, we can represent the game field as an array of ten boards:

GameField = [B1, …, B9, GlobalBoard]

where each board is itself an array of nine cells: *Board = [A1, …, A9]*.

A cell can have the following values: x, o, b (*blank*), draw. The last one is used only in the global board cells, to represent a draw in the corresponding miniboards.

A move on the game field is represented as a tuple: *[Global board cell, Miniboard cell]*.
A move *y* on the miniboard can be performed if *Miniboard[y] = b*.
A move *[x,y]* on the game field can be performed if *GlobalBoard[x] = b* and *y* is a valid move on the selected miniboard #x (i.e. *GameField[x]*).

```
% move(+Board, +Player, +Move, -NewBoard)
move(GameField, Player, [GlobalPos,LocalPos], NewGameField):-
    nth1(GlobalPos, GameField, Miniboard), %get Miniboard
    nth1(10, GameField, Globalboard), %get Globalboard
    nth1(GlobalPos, Globalboard, b),  % global cell is empty
    move(Miniboard, Player, LocalPos, NewMiniboard), % move on miniboard

    board_state(NewMiniboard, State), % x, o, b, draw
    move(Globalboard, State, GlobalPos, NewGlobalboard), %move on global board,

    % replace the boards with the new ones in the GameField
    replace(GameField, GlobalPos, NewMiniboard, GameField1),
    replace(GameField1, 10, NewGlobalboard, NewGameField).
```

```
% move(+Board, +Player, +LocalMove, -NewBoard)
move([b,B,C,D,E,F,G,H,I], Player, 1, [Player,B,C,D,E,F,G,H,I]).
move([A,b,C,D,E,F,G,H,I], Player, 2, [A,Player,C,D,E,F,G,H,I]).
move([A,B,b,D,E,F,G,H,I], Player, 3, [A,B,Player,D,E,F,G,H,I]).
move([A,B,C,b,E,F,G,H,I], Player, 4, [A,B,C,Player,E,F,G,H,I]).
move([A,B,C,D,b,F,G,H,I], Player, 5, [A,B,C,D,Player,F,G,H,I]).
move([A,B,C,D,E,b,G,H,I], Player, 6, [A,B,C,D,E,Player,G,H,I]).
move([A,B,C,D,E,F,b,H,I], Player, 7, [A,B,C,D,E,F,Player,H,I]).
move([A,B,C,D,E,F,G,b,I], Player, 8, [A,B,C,D,E,F,G,Player,I]).
move([A,B,C,D,E,F,G,H,b], Player, 9, [A,B,C,D,E,F,G,H,Player]).
```

Game play ends when either a player wins the global board or there are no legal moves remaining, in which case the game is a draw (see ***terminal_state***).

```
% Check if the miniboard is over
terminal_state(Board, Player):-
    board(Board),
    player(Player),
    win(Board, Player), !.

terminal_state(Board, draw):-
    board(Board),
    nomoves(Board), !.

% Check if the game is over.
% The game is over if the global board is over
terminal_state(GameField, Winner):-
    nth1(10, GameField, Globalboard),
    terminal_state(Globalboard, Winner), !.
```

```
win(Board, Player) :- rowwin(Board, Player), !.
win(Board, Player) :- colwin(Board, Player), !.
win(Board, Player) :- diagwin(Board, Player), !.

rowwin(Board, Player) :- Board = [Player,Player,Player,_,_,_,_,_,_].
rowwin(Board, Player) :- Board = [_,_,_,Player,Player,Player,_,_,_].
rowwin(Board, Player) :- Board = [_,_,_,_,_,_,Player,Player,Player].

colwin(Board, Player) :- Board = [Player,_,_,Player,_,_,Player,_,_].
colwin(Board, Player) :- Board = [_,Player,_,_,Player,_,_,Player,_].
colwin(Board, Player) :- Board = [_,_,Player,_,_,Player,_,_,Player].

diagwin(Board, Player) :- Board = [Player,_,_,_,Player,_,_,_,Player].
diagwin(Board, Player) :- Board = [_,_,Player,_,Player,_,Player,_,_].
```

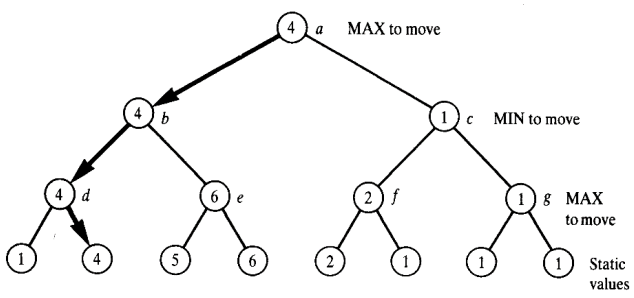
The win predicate check if a single board is won. The winner of the game is the winner of the global board.

2. Minimax with AlphaBeta cuts

The Minimax principle is a standard technique on which computer game playing is based on.

A game tree is only searched up to a certain depth, typically a few moves, and then the tip nodes of the search tree are evaluated by some evaluation function. The estimated value then propagates up the search tree according to the minimax principle.

The higher the value the higher the player's chances are to win, and the lower the value the higher the opponent's chances are to win. The two players will be called respectively MAX and MIN: MAX will choose a move that maximizes the value; on the contrary, MIN will choose a move that minimizes the value.




```
%best_of(+MinMax, +NodeA, +ValueA, +NodeB, +ValueB, -BestNode, -BestValue)
best_of(MinMax, NodeA, ValueA, _, ValueB, NodeA, ValueA) :-
    MinMax = max, ValueA >= ValueB, !;
    MinMax = min, ValueA =< ValueB, !.
best_of(_, _, _, NodeB, ValueB, NodeB, ValueB).
```

In order to reduce the search space, the algorithm holds two values, alpha and beta, representing respectively the minimum score that MAX ensures and the maximum score that MIN ensures.

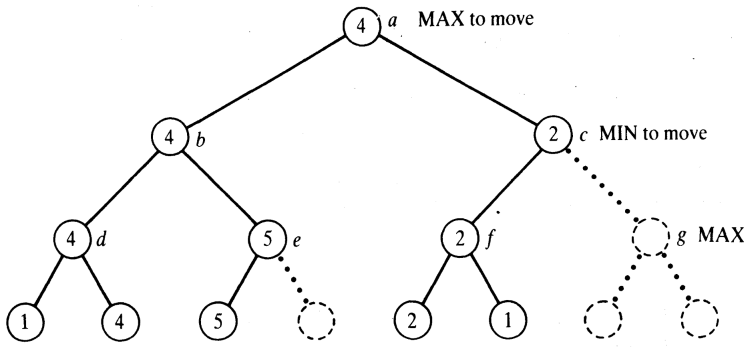
```
% alphabeta(+MaxDepth, +Board, -BestMove)
% Chooses the best possible move for the current board.
alphabeta(MaxDepth, Board, BestMove):-
    alphabeta_step(max, MaxDepth, [nil, Board], -999999, 999999, [BestMove, _], _), !.

% alphabeta_step(+MinMax, +MaxDepth, +Node, +Alpha, +Beta, -BestNode, -BestValue)
alphabeta_step(_, Depth, [_, Board], _, _, _, BestValue) :-
    terminal_state(Board, Winner),
    current_player(MaxPlayer),
    terminal_state_value(MaxPlayer, Winner, Value),
    BestValue is Depth * Value, !.

alphabeta_step(_, 0, Node, _, _, _, BestValue) :-
    current_player(MaxPlayer),
    eval(MaxPlayer, Node, BestValue), !.

alphabeta_step(MinMax, Depth, Node, Alpha, Beta, BestNode, BestValue) :-
    player_color(MinMax, Player),
    children(Node, Player, Children),
    NextDepth is Depth -1,
    bounded_best_node(MinMax, NextDepth, Children, Alpha, Beta, BestNode, BestValue), !.

% bounded_best_node(+MinMax, +MaxDepth, +NodeList, +Alpha, +Beta, -BestNode, -BestValue)
bounded_best_node(MinMax, MaxDepth, [Node | NodeList], Alpha, Beta, BestNode, BestValue) :-
    swap_max_min(MinMax, Other),
    alphabeta_step(Other, MaxDepth, Node, Alpha, Beta, _, BottomBestV),
    next_if_good(MinMax, MaxDepth, NodeList, Alpha, Beta, Node, BottomBestV, BestNode, BestValue).
```

The node is evaluated if the algorithm reaches either a terminal state or the max depth; otherwise the descendants are explored. The exploration is done by the bounded_best_node

Whenever the temporary score of a MIN (MAX) node becomes lesser (greater) than the minimum (maximum) score of MAX (MIN), the MAX (MIN) player should not consider further descendants of this node, as they will never be reached in the actual play.

The cutoff is applied on descendants by the algorithm if the following statement holds:

$$V(n_{min}) \leq Alpha \qquad \text{or} \qquad V(n_{max}) \geq Beta$$



The ***next_if_good*** predicate has been implemented for this purpose. The exploration stops if the cut off conditions hold, otherwise it explores the next node and takes the best one (according to the minimax principle).

```
% next_if_good(+MinMax, +MaxDepth, +NodeList, +Alpha, +Beta, +Node, +Value, -BestNode, -BestValue)
next_if_good(_, _, [], _, _, Node, Value, Node, Value):- !.
% if not good enough -> cutoff
next_if_good(MinMax, _, _, Alpha, Beta, Node, Value, Node, Value):-
    MinMax = max, Value >= Beta, !;
    MinMax = min, Value =< Alpha, !.
% otherwise go to the next node
next_if_good(MinMax, MaxDepth, NodeList, Alpha, Beta, Node, Value, BestNode, BestValue):-
    update_bounds(MinMax, Alpha, Beta, Value, NewAlpha, NewBeta),
    bounded_best_node(MinMax, MaxDepth, NodeList, NewAlpha, NewBeta, CurrentBestN, CurrentBestV),
    best_of(MinMax, Node, Value, CurrentBestN, CurrentBestV, BestNode, BestValue).
```

3. Node evaluation

The aim of the evaluation function is to determine the quality of the state, in the sense of the winning probability. The outcome is known if the state represents a closed game, otherwise it should be estimated by an heuristic function.

The idea is to assign a score to each miniboard and calculate the game field score as a weighted sum (see ***eval_game_field***). The score of each miniboard is itself calculated as a weighted sum of the values assigned to each cell, which are 1 if the cell is occupied by the max , -1 otherwise (see ***eval_miniboard***).
Once a local board is won by a player or it is filled completely, a fixed score is assigned to the miniboard (see ***eval_global_cell***): it has to be large enough to prefer a local win to any other move within the miniboard, but small enough to make it possible to play on another miniboard.

```
% eval_game_field(+Player, +GameField, -Value)
eval_game_field(Player, GameField, Value):-
    global_board_weights(GlobalWeights),
    nth1(10, GameField, GlobalBoard),
    eval_game_field(Player, GameField, GlobalBoard, Values),
    weighted_sum(Values, GlobalWeights, Value).

eval_game_field(_, _, [], []):- !.
eval_game_field(Player, [Miniboard | Boards], [GlobalCell|Cells], [Value|Values]):-
    eval_global_cell(Player, GlobalCell, Miniboard, Value),
    eval_game_field(Player, Boards, Cells, Values), !.

% eval_global_cell(+Player, +GlobaCell, +Miniboard, -Value)
eval_global_cell(Player, b, Miniboard, Value):-
    eval_miniboard(Player, Miniboard, Value), !.

% Since max miniboard's value is 11, a won global cell should be a bit
% greater than 11, e.g. 12.
eval_global_cell(Player, GlobaCell, _, Value):-
    eval_cell(Player, GlobaCell, V),
    Value is 12 * V , !.
```

```
%eval_miniboard(+Player, +Board, -Value)
eval_miniboard(Player, Board, Value):-
    miniboard_weights(Weights),
    map_cell_values(Player, Board, MappedValues),
    weighted_sum(MappedValues, Weights, Value).

map_cell_values(_, [], []):- !.
map_cell_values(Player, [Cell|Cells], [NewCell|NewCells]):-
    eval_cell(Player, Cell, NewCell),
    map_cell_values(Player, Cells, NewCells), !.
```

The cells in the miniboards and the global board are weighted according to their winning probability. Therefore, the central one has a greater weight than the corners which, at the same time, have a greater weight than the remaining positions.

miniboard_weights*([3, 1, 3, 1, 5, 1, 3, 1, 3]).*

global_board_weights*([3, 1, 3, 1, 5, 1, 3, 1, 3]).*