ULTIMATE TIC TAC TOE

Francesco Zangrillo, Simone Mele

Project structure

Four decoupled modules:

- Game
- Ultimate Tic Tac Toe
- AlphaBeta
- Evaluation

GAME

start_game

Initialize the game - called by the user:

- *start_game(ultimate, ai, human)*

```
start_game(Game, Player1Type, Player2Type):-
        game(Game),
        set_player_type(x, Player1Type),
        set_player_type(o, Player2Type),

        explain_game_if_human(Game, Player1Type, Player2Type),      % show explaination for human
        play_game(Game), !.

play_game(Game):-
        init_board(Game, Board),
        Player = x,                                                 % initial player
        play(Game, Board, Player), !.                               % start the game
```

play

The core of the game play: manages the turns and executes the moves

```
% if game is over
play(Game, Board, _):-
        terminal_state(Board, Winner),
        print_board(Game, Board), nl,
        show_win_message(Winner), !.
```

```
% otherwise
play(Game, Board, Player):-
        print_board(Game, Board), nl,
        write('Player\'s turn: '), write(Player), nl,
        set_current_player(Player),
        get_player_type(Player, PlayerType),

        % get the move - play as human or ai
        play_as(PlayerType, Game, Board, Move), !,
        % perform the move
        move(Board, Player, Move, NewBoard),

        other(Player, OtherPlayer),              % change player
        play(Game, NewBoard, OtherPlayer).       % next turn
```

play_as

Get the move according to the player type (ai or human).

```
% play as AI - run alphabeta
play_as(ai, Game, Board, Move):-
        max_depth(Game, MaxDepth), !,
        alphabeta(MaxDepth, Board, Move).

% play as human - read move
play_as(human, _, Board, Move):-
        read_valid_move(Board, Move).
```

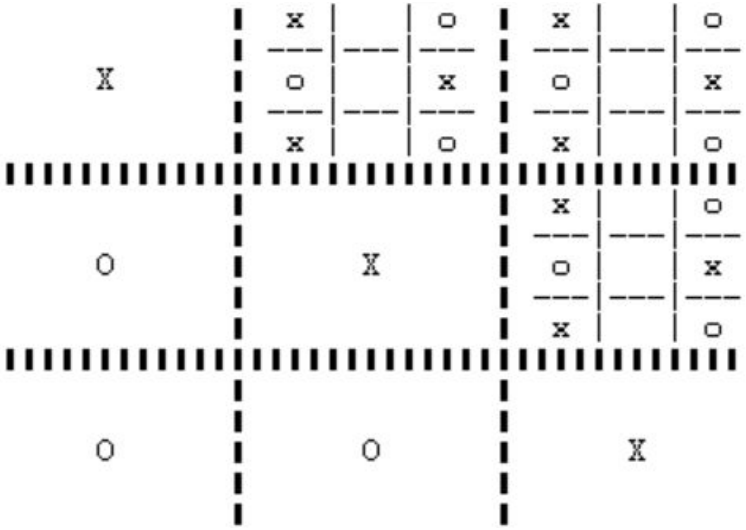
```
% read until a valid move is entered
read_valid_move(Board, ValidMove):-
        read(ValidMove),
        move(Board, _, ValidMove, _),!.
read_valid_move(Board, ValidMove):-
        write('Illegal move !'),
        read_valid_move(Board, ValidMove), !.
```

Ultimate Tic Tac Toe

Game field

GameField = [B1, …, B9, Global]

Board = [A, B, C, D, E, F, G, H, I].



B1, …, B9 : miniboards
Global : global board

Each cell i of the global board is mapped with the result of the correspondent miniboard Bi

From Simple tic tac toe … - move

Put ***X*** or ***O*** on the specified cell of the miniboard

```
nomoves(Board):- not(member(b, Board)).

% 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,C,D,E,F,G,H,b], Player, 9, [A,B,C,D,E,F,G,H,Player]).
```

A LocalMove on the Board is valid if Board[LocalMove] = b.

… to Ultimate Tic Tac Toe - move

Put ***X*** or ***O*** on the specified cell of the miniboard

```
% move(+Board, +Player, +Move, -NewBoard)
move(GameField, Player, [GlobalPos,LocalPos], NewGameField):-
        nth1(GlobalPos, GameField, Miniboard),                      % get Miniboard
        nth1(10, GameField, Globalboard),                           % get Global board
        nth1(GlobalPos, Globalboard, b),                            % if 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),    % update the game field
        replace(GameField1, 10, NewGlobalboard, NewGameField).
```

win

Retrieve the winner of the board (if exist)

```
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,_,_].
```

A board is won by a player, if he wins either a row, a column or a diagonal

terminal_state

Check if the game is over

```
% terminal_state(+Board, -Winner)
terminal_state(Board, Player):-
        board(Board),
        player(Player),
        win(Board, Player), !.

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

% terminal_state(+GameField, -Winner)
terminal_state(GameField, Winner):-
        nth1(10, GameField, Globalboard),
        terminal_state(Globalboard, Winner), !.
```

A board is over if there is a winner or there are no more legal moves

Game is over if the global board is over

Alpha Beta

alphabeta

Choose the best possible move for the given board / game field by exploring the game space.
The search space is reduced according to the *alpha* and *beta* values.

```
alphabeta(MaxDepth, Board, BestMove):-
    % alphabeta_step(+MinMax, +MaxDepth, +Node, +Alpha, +Beta, -BestNode, -BestValue)
    alphabeta_step(max, MaxDepth, [nil, Board], -999999, 999999, [BestMove, _], _), !.
```

It adopts the Node variable to decouple the algorithm from the games,
where ***Node = [Move, Board]***

alphabeta

Choose the best possible move for the given board / game field by exploring the game space.
The search space is reduced according to the *alpha* and *beta* values.

```
% 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),                % get legal moves
        NextDepth is Depth -1,
        bounded_best_node(MinMax, NextDepth, Children, Alpha, Beta, BestNode, BestValue), !.
```

If the algorithm reaches either a terminal state or the max depth, the node is evaluated

bounded_best_node

Explore the descendants

```
% 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),

        % explore current child node
        alphabeta_step(Other, MaxDepth, Node, Alpha, Beta, _, BottomBestV),

        % go to the next node, if pruning doesn't occurr
        next_if_good(MinMax, MaxDepth, NodeList, Alpha, Beta, Node, BottomBestV, BestNode, BestValue).
```

next_if_good

Stops the exploration if cut off conditions hold

```
% 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 and take the best
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).
```

best_of

Take the best node according to the minimax principle

```
%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).
```

if MAX, the best is the greatest
if MIN, the best is the lowest

Evaluation

Scores and weights

```
%terminal_state_value(+Player, +Winner, -Value)
terminal_state_value(Player, Winner, 500):- Player = Winner, !.
terminal_state_value(_, draw, 0):- !.
terminal_state_value(_, _, -500).
```

If the game is over, the outcome probability is 100%

→ an high score is necessary…

```
% eval(+Player, +Node, -Val)
eval(Player, [_, GameField], Value):-
        eval_game_field(Player, GameField, Value), !.
```

… otherwise, the node should be evaluated

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

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

eval_game_field

Evaluate the whole game field

```
% 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), !.
```

The game field score is the weighted sum of the global cell values

eval_game_field

Evaluate the whole game field

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

If the global cell is blank, the score is given by the miniboard score

```
eval_global_cell(Player, GlobaCell, _, Value):-
	eval_cell(Player, GlobaCell, V),
	Value is 12 * V , !.
```

Otherwise, the score is determined by a value based on the outcome of the miniboard.

Since a won miniboard is better than the best unfinished one, the former should have a slightly higher value (e.g. 12).

eval_miniboard

Evaluate a single miniboard

```
%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), !.
```

```
% eval_cell(+Player, +CellValue, -Value)
eval_cell(_, b, 0):- !.
eval_cell(_, draw, 0):- !.
eval_cell(Player, Player, 1):- !.
eval_cell(_, V, -1):- cell_value(V), !.
```

1, if cell is occupied by MAX
-1, if cell is occupied by MIN
0, if cell is blank (or draw)

THANKS FOR YOUR ATTENTION