CSE 4082 / Project 2

Kevser İldeş - 150116048

Buse Batman – 150117011

Main:

Main method gets player informations (human or ai) and create players (with calling related classes) based on the given inputs.

And then it creates the game with Connect4 class.

- Options_AI(sign)

If AI will play the game, this method gets the AI options (depth, heuristic option) from the user.

- minimax_options_AI()

This method takes the minimax options from the user:

1- Normal minimax 2- Minimax with alpha beta prunning

Class: Connect4

- __init__(self, players, turn)

This method creates the board and initializes game operators.

- switch_players_turn_and_inc_round(*self*):

This method switch players turn and increments the round.

- nextMove(*self*)

This method calls related player's move method. And adds that move to the board.

- check_fours(*self*,*row*,*col*)

This method checks whether is there any 4 same signs in horizontal – diagonal or vertical lines.

- check_winner(*self*, *curr_row*, *curr_column*)

This method determines the winner according to game's turn value.

- vertical_check(*self*, *curr_row*, *curr_column*)

This method visits the board and checks whether is there any 4 same signs in same column.

- horizontal_check(*self*, *curr_row*, *curr_column*)

This method visits the board and checks whether is there any 4 same signs in same row.

- diagonal_check(*self*, *curr_row*, *curr_column*)

This method visits the board and checks whether is there any 4 same signs in diagonal order.

- printState(*self*)

This method prints the current state informations like: Round count , turn of the players, current state of the game board.

- printFinishState(*self*)

This method prints the final results when the game is finished.

Class: HumanPlayer

- __init__(*self*, *sign*)

This method initializes the human player.

- move(*self*, *state*)

This method gets the column choice from the user and checks whether the input is valid or not.

Class: AIPlayer(HumanPlayer)

- __init__(*self*, *sign*, *depth*,*heuristic*)

This method initializes the AI Player and it extends the Human Player class.

- move(*self*, *state*)

This method creates a Minimax object calling Minimax Class. And with alpha = - inf and beta = inf values, it calls Minimax's best move method for its move.

Class: Minimax_1

-> This minimax algorithm is normal one. (w/out alpha beta prunning)

- __init__(*self*, *game_board*, *sign*, *heuristic*)

This method copies the game board and initalizes the Minimax tree

- valid_moves(*self*,*board*)

This method visits the game board and searches empty locations.

- heuristic1(*self*,*board*,*sign*)

For our first heuristic, we calculated the opponent's possible four counts. If there is any possibility it returns -100000. If not we calculate current users possible four, three and two counts. And we calculate the score like: 100000 * pos. fours + 100 * pos. threes + pos. twos.

- heuristic2(*self*,*board*,*sign*)

For our second heuristic, we calculated the opponent's user possible four counts. If there is any possibility it returns -infinity. If not we calculate current users possible four, three and two counts and opponent's possible four, three and two counts.

And we calculate the score like: (100000 * pos. fours + 1000 * pos. threes + 100 * pos. twos) - (100000 * opp's pos. fours + 1000 * opp's pos. threes + 100 * opp's pos. twos).

- heuristic3(*self*, *board*, *sign*)

For our final heuristic, we calculated the opponent's user posible four counts. If there is any possibility it returns -infinity. If not we calculate current users possible four, three and two counts and opponent's possible four, three and two counts.

And we calculate the current user's score like: (100000 * pos. fours + 1000 * pos. threes + pos. twos).

And opp's score like: (100000 * opp's pos. fours + 1000 * opp's pos. threes + opp's pos. twos). If opponent's score is greater than current user's score, it returns – opponent's score. If not, it returns current user's score – opponent's score.

- evaluate(*self*, *board*, *sign*, *heuristic*)

This method simply orients to the related heuristic method.

- countSigns(*self*, *board*, *sign*, *count*)

This method controls whether the given state contains any diagonal – horizontal or vertical fours.

- vertical_check(*self*, *curr_row*, *curr_column*, *board*, *count*)

Like in Connect4 class, also this method controls whether the the given state contains any 4 same signs in the same column.

- horizontal_check(*self*, *curr_row*, *curr_column*, *board*, *count*)

Like in Connect4 class, also this method controls whether the the given state contains any 4 same signs in the same row.

- diagonal_check(*self*, *curr_row*, *curr_column*, *board*, *count*)

Like in Connect4 class, also this method controls whether the the given state contains any 4 same signs in diagonal order.

- gameOver(*self,board*)

This method checks whether the game is finished or not by calling countSigns method. If it returns a value greater than 1, this method returns true. If not, then it returns false.

- makeMove(*self*, *state*, *column*, *sign*)

This method simply fills every empty locations with current user's sign.

- bestMove(*self*, *board*, *depth*, *curr_sign*)

This method gets the valid moves with valid_moves method and shuffles them. This is a recursive method and when there is no possible moves or copy game board is full, it calculates the score with evaluate method and it finishes.

For the maximizing player, it starts the value as – infinity and for each valid moves, it makes a move and calculates the score with recursively calling itself. If final score is greater than – infinity, it replaces the value with that value.

For the minimizing player, it starts the value as infinity and for each valid moves, it makes a move and calculates the score with recursively calling itself. If final score is less than infinity, it replaces the value with that value.

Class: Minimax_2

-> This minimax algorithm uses alpha beta prunning. It is similar to Minimax_1 class, with this additional methods (bestMove uses this methods):

- MinimizeBeta (*self*, *board*, *depth*, *a*, *b*, *sign*, *opponent*):

This method simply minimizes beta with checking all valid moves in the board, making the move and calculating the score with maximizeAlpha metod. Initially board score is infinity, and when that score is less than beta, it updates beta with that board score value.

- MaximizeAlpha (*self*, *board*, *depth*, *a*, *b*, *sign*, *opponent*):

This method simply maximizes alpha with checking all valid moves in the board, making the move and calculating the score with minimizeAlpha metod. Initially board score is - infinity, and when that score is greater than beta, it updates beta with that board score value.