T.C.

MARMARA UNIVERSITY

FACULTY of ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

CSE4082 AI PROJECT-2

SAMET ENES ÖRSDEMİR

150119661

Classes:

There are two classes which are State class and Player class. These classes store State information, board view and instant information of the player. The game progresses using these classes. Every node created in the tree is a State Object. It also contains the Player object as a feature.

Functions:

- **checkmove(state,move):** This function checks the suitability of the column, whether there is enough space for the move command.
- **check_win(state):** This function, which has two, checks whether the game is over or not. Returns the number 1 or 2 depending on the ending state of the game. The reason for the existence of two is that

one of them proceeds through the class and the other through the map.

- **check_win2(map): (same)** This function, which has two, checks whether the game is over or not. Returns the number 1 or 2 depending on the ending state of the game. The reason for the existence of two is that one of them proceeds through the class and the other through the map.
- **getAllmoves(map):** This function is a function that adds all possible moves to the list by scanning the map and returns the list.
- **get_children(node,player):** This is a function that returns all the next playable positions (i.e. children) of the sent node by adding it to a list.
- **is_column_full(board, col):** This function returns whether the column is full.
- **possible_wins(board,player):** This function returns all possible winning positions, counting both for the player and for the opponent.
- **consecutive_pieces(board,player):** This function returns consecutive pieces of the same color, counting for both the player and the opponent.
- **symmetry(board,player):** This function counts the pieces that are symmetrically on the board. Used by heuristics.
- **h1(map,player):** This function scores for both the opponent and the player in all possible winning situations. Then it scores the blocking positions against the opponent. Finally, he gets the heuristic score by subtracting the opponent score from the player score.
- **h2(map,player):** This function calculates and returns the difference between player and opponent by scoring all possible wins, consecutive pieces, symmetric status and 4 pieces.
- **h3(map,player):** This function scores all possible win situations, pieces in the middle column, blocked opponent win moves and 4 pieces. Finally, the player-oppenent returns the score difference.
- **count_block_moves(board,player):** This function counts all possible opponent blocking moves. Used by heuristics.
- **count_pieces(board,player, col):** This function counts the pieces of

the requested player in the desired column.

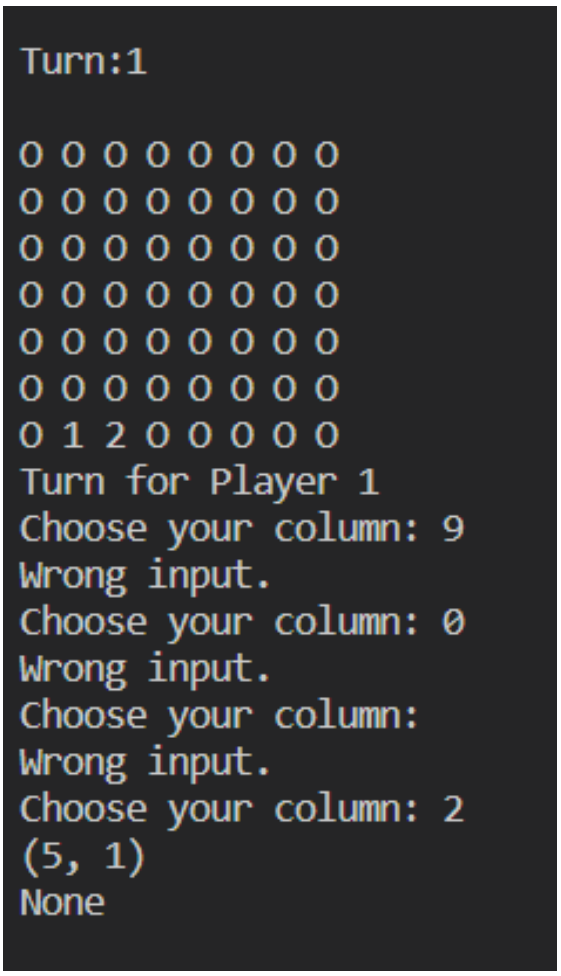
- **alphabeta_aivsai(player,node,depth,h,alpha,beta, maximizingPlayer, best_move = None):** This function implements the minimax algorithm using the selected heuristic. While creating the tree by determining the depth limit, it recursively calculates the score and returns the last score and the best move by returning to the node started with the minimax algorithm. With the alpha-beta method, it does not open unnecessary parts and increases performance.
- **alphabeta(node,depth,h,alpha,beta, maximizingPlayer, best_move = None):** Same as other alpha beta minimax. The only difference is that there is no player information.
- **player_vs_player(states, player1, player2):** This function starts with a while loop that counts 56 parts. It then continues the loop by pulling the instant state from the end of the states list. In the loop, the 1st player or the 2nd player plays depending on the number of turns. It updates the board by receiving column input from the player and continues the loop. At the end of the loop, it checks the win status with the check_win() function. The cycle continues until the game is over or there is a draw.
- **player_vs_bestai(depth_limit,heuristic1):** This function is the function that manages the game to be played between the player and the AI. The only difference from the player_vs_player() function is that one of the players is ai. When it's ai's turn, alpha-beta minimax is applied and the best move returned is played, then it's the player's turn.
- **ai_vs_ai(depth_limit, depth_limit2, heuristic1, heuristic2):** This function works with the same logic as other game management functions. The only difference is that both sides are AI. After the minimax is applied on both sides, the AI makes the next move. If any score could not be calculated (ie the endgame is not clear), it performs a random move.

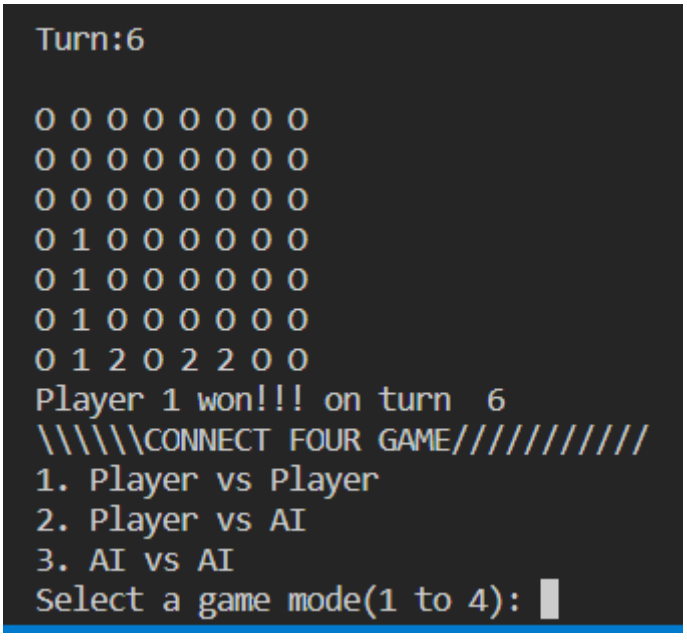
Performance: Finding the best move took at most 12 seconds with 8 depth for h3.

Screenshots:

```
\\\\\\CONNECT FOUR GAME//////////
1. Player vs Player
2. Player vs AI
3. AI vs AI
Select a game mode(1 to 4):
```

```
Select a game mode(1 to 4): 1


Turn:0

0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
Turn for Player 1
Choose your column: 2
(6, 1)
None


Turn:0

0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
Turn for Player 2
Choose your column: 3
(6, 2)
None
```


Turn:1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 1 2 0 0 0 0 0
Turn for Player 1
Choose your column: 9
Wrong input.
Choose your column: 0
Wrong input.
Choose your column:
Wrong input.
Choose your column: 2
(5, 1)
None


Turn:6
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 1 2 0 2 2 0 0
Player 1 won!!! on turn 6
\\\\\\CONNECT FOUR GAME//////////
1. Player vs Player
2. Player vs AI
3. AI vs AI
Select a game mode(1 to 4):