

CSE 4082 Artificial Intelligence – Project 2 Report

Group members:

Emirkan Karabulut 150118052

Ramazan Karkin 150119512

The function `game_board()` : Creates and returns the initial game board, which is filled with -1.

The function `move_piece(game_board, player, column)`: Allows a player to add a piece to a specific column of the board, if that column is not already full.

The function `win_condition(game_board, player)`: Checks whether the player has won by having four consecutive pieces either horizontally, vertically, diagonally or anti diagonally.

The function `print_board(board)`: This function after printing the Connect-Four game playfield to the console. According to the moves of the players, it will print the moves to the console by using the 'X' symbol for the first player and the 'O' symbol for the second player.

The function `is_playable((game_board, column)`: it will check whether the column passed as a parameter is a valid move or not.

The function `choice(board)`: It will check board for valid move and if there is a valid move then it will choose a random column.

The function `evaluate_heuristic1(board, player)`: First two if statement checks If player1 has won the function returns positive infinity, and if player2 has won returns negative infinity. Returning positive or negative infinity effectively assigns the highest possible or lowest possible score to the board state. It is used to evaluate the current board state and assign a score to it, based on sequences of length. The function iterates over each cell in the board and checks if the cell belongs to the current player. If the cell belongs to the current player, the function checks the horizontal, vertical, diagonal, and anti-diagonal sequences of length 3 in the board and adds 1 to the score for each sequence that can be completed by the current player, by comparing the next cells with current cell and current player. The minimax function uses the scores returned by `evaluate_heuristic1` to make decisions on the best move to make, by maximizing the score for the current player and minimizing the score for the opposing player.

The function `evaluate_heuristic2(board, player)`: This heuristic function uses a weight matrix to assign a score to each cell on the board. We have given weight to each position according to the positions we think are more important, with higher weights given to cells in the center of the board. This way, the AI player will prioritize moves that are made in the center of the board.

The function `evaluate_heuristic3(board, player)`: This evaluate function uses `heuristic2` and the number of occurrences of two adjacent same values (either the current player or the opponent) horizontally, vertically, or diagonally on the board. After `heuristic2` assigned a score to each cell on the board according to the weight matrix, then for each occurrence of two adjacent same values, we will triple the score. This function is used in the minimax algorithm as a way to evaluate the potential outcomes of a given board state and determine the best move to make.

The function `minimax(board, player, depth=4, alpha=-float("inf"), beta=float("inf"), eval_func=evaluate_heuristic1)`: It is an implementation of the Minimax algorithm. The algorithm uses this score to determine the best move to make by maximizing the score for the current player and minimizing the score for the opposing player. The implementation approach which we applied is by recursively calling the minimax function for each playable column on the board, and finding the α (best score for the maximizer) and β (best score for the minimizer) values which are used to prune the search tree to avoid unnecessary work. The function takes the current state of the board, the player that is making the move, α , β , and evaluation function as inputs.

The function starts by checking if the search has reached the maximum search depth specified by the `depth` parameter, or if the current player has won the game. If either of these conditions is true, the function calls the evaluation function passed in `eval_func` parameter, which in this case is `evaluate_heuristic1`, to return the heuristic score of the current board state for the given player.

If the search is not at the maximum depth or the game is not over, the function iterates through all columns of the board, and for each playable column, it performs the following steps:

it creates a copy of the board. Then the `move_piece(temp_board, player, column)` function to make a move in that column. After that it recursively calls the minimax function with the modified copy of the board and the other player as the active player. it negates the returned score of the recursive call. if the current score is greater than the current alpha the function updates the alpha and best_column with the current score and column. In case when alpha is greater than or equal to beta, the function breaks out of the loop return the alpha and best_column. In the case where no valid move is found, the function calls `choice(board)` which selects a random playable column.

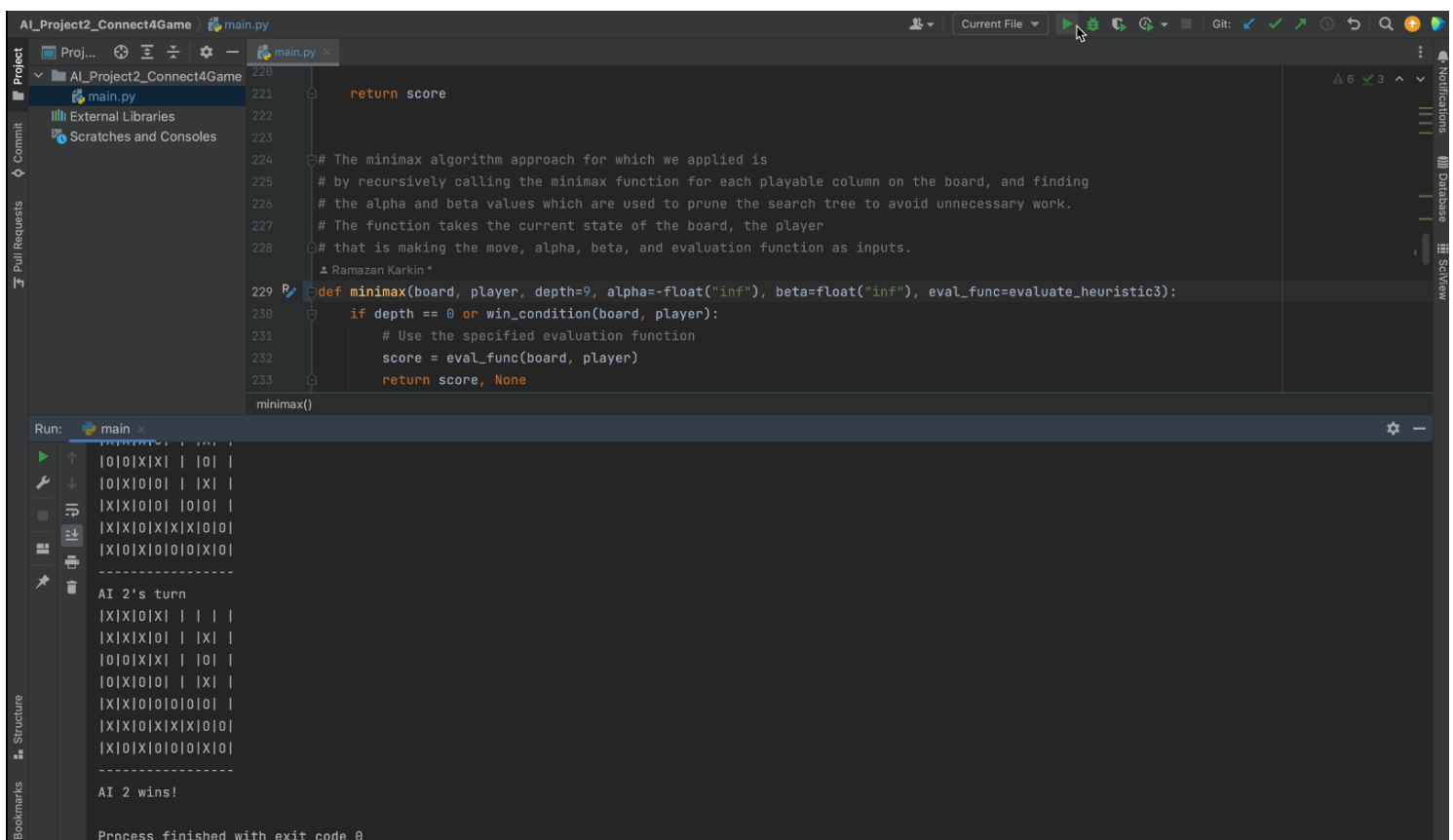
In summary, the function determine the best move to make by simulating all possible moves to a certain depth and choosing the move that results in the best score according to the evaluation function passed.

Video link: <https://vimeo.com/789315930>

drive link:

<https://drive.google.com/drive/u/0/folders/1Mr42mv32CSYlqDVdMmwVrw4OaG5pH82c>

AI player using h1 vs AI player h2: Video duration time 0.00 - 8.34 you can see



```
AI_Project2_Connect4Game  main.py
Project
  AI_Project2_Connect4Game
    main.py
  External Libraries
  Scratches and Consoles
Commit
Full Requests
Bookmarks
Structure

220
221     return score
222
223
224     # The minimax algorithm approach for which we applied is
225     # by recursively calling the minimax function for each playable column on the board, and finding
226     # the alpha and beta values which are used to prune the search tree to avoid unnecessary work.
227     # The function takes the current state of the board, the player
228     # that is making the move, alpha, beta, and evaluation function as inputs.
229     # Ramazan Karkin *
229     def minimax(board, player, depth=9, alpha=-float("inf"), beta=float("inf"), eval_func=evaluate_heuristic3):
230         if depth == 0 or win_condition(board, player):
231             # Use the specified evaluation function
232             score = eval_func(board, player)
233             return score, None
234
235     minimax()

Run: main
|0|0|X|X|X| |0| |
|0|X|0|0| |X| |
|X|X|0|0| |0|0|
|X|X|0|X|X|X|0|0|
|X|0|X|0|0|0|X|0|
-----
AI 2's turn
|X|X|0|X|X| | | | |
|X|X|X|0| |X| |
|0|0|X|X|X| |0| |
|0|X|0|0| |X| |
|X|X|0|0|0|0|0| |
|X|X|0|X|X|X|X|0|0|
|X|0|X|0|0|0|X|0|
-----
AI 2 wins!
Process finished with exit code 0
```

AI player using h2 vs AI player using h3: Video duration time 8.53 - 30.00

The screenshot displays the Microsoft Visual Studio Code interface. On the left, the Project Explorer shows a folder named 'AI_Project2_Connect4Game' containing a file 'main.py'. The main.py file is open in the editor, showing a Python script implementing a minimax algorithm for Connect Four. The code includes comments explaining the minimax approach and recursive calls. A def minimax function is defined with parameters board, player, depth, alpha, beta, and eval_func. It checks for win conditions or if the game is over (depth == 0). If not, it recursively calls itself for both players to find the best move. The function returns the score and None. Below the code editor, the Run console shows the output of the program, displaying the initial board state, AI's turn, and the final result: 'AI 2 wins!'

```

220
221     return score
222
223
224 # The minimax algorithm approach for which we applied is
225 # by recursively calling the minimax function for each playable column on the board, and finding
226 # the alpha and beta values which are used to prune the search tree to avoid unnecessary work.
227 # The function takes the current state of the board, the player
228 # that is making the move, alpha, beta, and evaluation function as inputs.
    Ramazan Karkin *
229 def minimax(board, player, depth=9, alpha=-float("inf"), beta=float("inf"), eval_func=evaluate_heuristic3):
230
231     if depth == 0 or win_condition(board, player):
232         # Use the specified evaluation function
233         score = eval_func(board, player)
234         return score, None
235
236     minimax()

```

Run: main ×

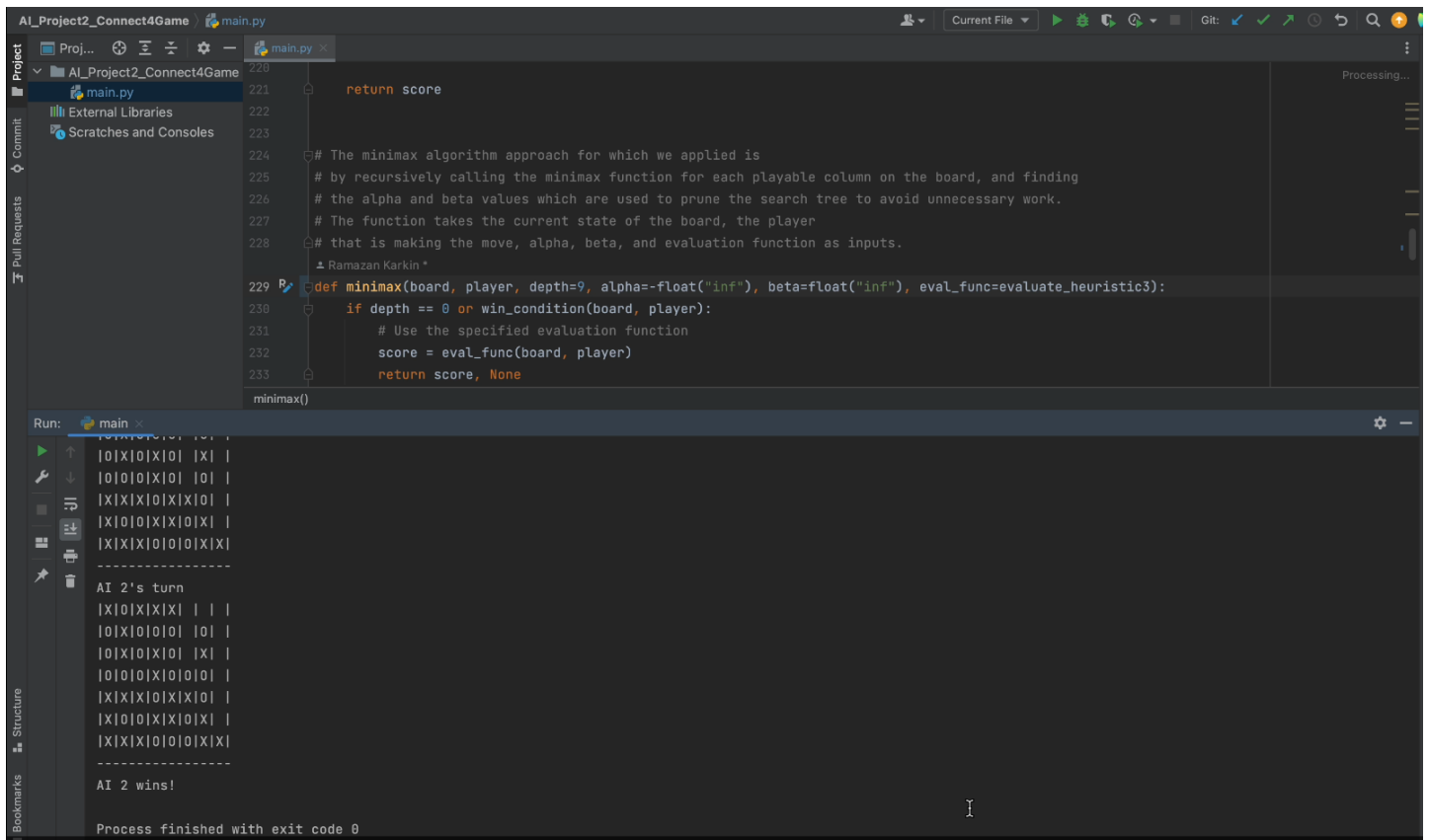
```

|-----| |X| |X| | | |
|O|O|O| |X| |X| |
|O|X|X| |X| |X| |
|X|O|X|X|O| |O| |
|X|X|O|X|X| |O|O|
|X|X|O|X|O|O|X|O|
|-----|
AI 2's turn
|X|O|O| |O| | | |
|O|O|X| |X| | |
|O|O|O| |X| |X| |
|O|X|X|O|X| |X| |
|X|O|X|X|O| |O| |
|X|X|O|X|X| |O|O|
|X|X|O|X|O|O|X|O|
|-----|
AI 2 wins!

Process finished with exit code 0

```

AI player using h1 vs AI player using h3: Video duration time 30.15 - 37.37



```
AI_Project2_Connect4Game main.py
Project
  AI_Project2_Connect4Game
    main.py
  External Libraries
  Scratches and Consoles
Commit
Pull Requests
Run: main
  101X101X101 |X| |
  1010101X101 |0| |
  1X1X1X101X1X101 |
  1X10101X1X101X1 |
  1X1X1X1010101X1X1
  -----
  AI 2's turn
  1X101X1X1X1 | | |
  101X1010101 |0| |
  101X101X101 |X| |
  1010101X1010101 |
  1X1X1X101X1X101 |
  1X10101X1X101X1 |
  1X1X1X1010101X1X1
  -----
  AI 2 wins!
Process finished with exit code 0
```

The screenshot shows a PyCharm IDE with a project named 'AI_Project2_Connect4Game'. The main.py file is open, showing a minimax function and a heuristic evaluation function. The Run console shows the game state and the result: AI 2 wins!

Human Player vs the Best AI Player Configuration:Video duration time 37.53 - 39.30

The screenshot displays an IDE with a Python script for a Connect4 game. The script is named 'main.py' and is part of a project 'AI_Project2_Connect4Game'. The code includes a minimax function and a heuristic evaluation function. The output window shows the game state and the AI's move.

```

220
221     return score
222
223
224     # The minimax algorithm approach for which we applied is
225     # by recursively calling the minimax function for each playable column on the board, and finding
226     # the alpha and beta values which are used to prune the search tree to avoid unnecessary work.
227     # The function takes the current state of the board, the player
228     # that is making the move, alpha, beta, and evaluation function as inputs.
229     # @Ramazan Karkin *
230
231     def minimax(board, player, depth=9, alpha=-float("inf"), beta=float("inf"), eval_func=evaluate_heuristic3):
232         if depth == 0 or win_condition(board, player):
233             # Use the specified evaluation function
234             score = eval_func(board, player)
235             return score, None
236
237     minimax()

```

The output window shows the game state and the AI's move:

```

Run: main
| | | |X| | | | |
| | | |O|X| | | |
| | |X|O|O|X| |
| | |O|O|X|X|X| |
| | |O|X|X|X|O| |
-----
AI move: 5
| | | | | | | | |
| | | | | | |
| | | | | | |
| | |X| | | | |
| | |O|O|X|O| | |
| | |X|O|O|O|X| |
| | |O|O|X|X|X| |
| | |O|X|X|X|O| |
-----
AI wins!

Process finished with exit code 0

```