EARIN
Exercise 2

Variant 1

Oleg Kim 01155526@pw.edu.pl
Seniv Volodymyr 01156240@pw.edu.pl

02.04.2023

Task description:

Write a program that playstic tac toewith the user on a 3×3board. The game continues until one of the players wins or it is a draw. The first player to get 3 of his/her marks in a row (up, down, across, or diagonally)wins.Algorithm: min-max with alpha-beta pruning. The game may beplayed on the consoleand the interface maybe as simple as possible(but of course itdoes not have to).

Launch instruction:

Only thing you need to do is just launch it throw the IDE you use or throw the console.

Then you choose the side you are going to play: X or O

Note: X goes first every game.

Then you just choosing the fields where you want to go (from 0 to 8).

Theoretical background:

The Minimax algorithm is a decision-making algorithm used in game theory and decision theory. It is a recursive algorithm that searches through the possible outcomes of a game or decision and evaluates the possible moves based on the assumption that the opponent will make the best possible move.

The algorithm works by constructing a game tree, where each node represents a possible state of the game and each edge represents a possible move. The algorithm traverses the tree recursively, evaluating each node based on whether it represents a winning or losing position for the player whose turn it is to move. The algorithm then chooses the move that leads to the best possible outcome for the player, assuming that the opponent will also make the best possible move.

Alpha-beta pruning is a technique used in game tree search algorithms to improve their efficiency. The idea behind alpha-beta pruning is to reduce the number of nodes that need to be explored by the search algorithm.

In the standard minimax algorithm, all possible moves are explored to the end of the game, and the best move is selected based on the score at the end of the game. Alpha-beta pruning uses a heuristic to estimate the value of a node without exploring all the way to the end of the game.

The alpha-beta pruning algorithm works by maintaining two values, alpha and beta, for each player. Alpha represents the best value found so far for the maximizing player, and beta represents the best value found so far for the minimizing player. The algorithm starts with alpha and beta set to negative and positive infinity, respectively.

As the search algorithm explores the game tree, it updates the alpha and beta values based on the values of the nodes it encounters. If the current node is a maximizing node, it updates the alpha value if the value of the node is greater than alpha. If the current node is a minimizing node, it updates the beta value if the value of the node is less than beta.

At each level of the tree, if beta becomes less than or equal to alpha, then the search algorithm can prune the rest of the nodes at that level, since they won't affect the final decision. This is because the maximizing player will never choose a node with a value less than or equal to alpha, and the minimizing player will never choose a node with a value greater than or equal to beta.

Solution:

Map:

In our solution map is just a list of strings of dashes and it prints line by line as 3x3 matrix. Our map is a global variable and printing is done inside "display_board" function.

Win condition:

We are checking if one of the players win. Firstly, we check the rows, secondly, we check the columns and after that we check diagonals.

Minimax:

This algorithm is done inside the "minimax(board , depth, alpha, beta, is_maximizing)" function where:

1. board – is the current state of the game board
2. depth – a number which represents the depth of the recursion
3. alpha, beta – parameters for the alpha beta pruning
4. is_maximizing – the variable which represents who will start the game

The function first checks if there is a winner, if the board is full, or if the depth limit has been reached. If any of these conditions are met, the function returns a score (-1, 0, or 1) depending on the outcome of the game.

If the game is not over, the function recursively calls itself with the board in the next possible state, and with the opposite value of is_maximizing (since it is the other player's turn now). It then updates the best_score variable based on the result of the recursive call, and updates the alpha and beta parameters for pruning.

If is_maximizing is True, it returns the best_score, which represents the best score that the maximizing player can achieve. If is_maximizing is False, it returns the best_score, which represents the best score that the minimizing player can achieve.

Moves of a computer:

This functionality is done inside the get_computer_move(board, mode), where:

1. board – is the current state of the game board which is the list
2. mode - a string representing the player the computer is playing as ('X' or 'O') or a continuation mode ('contX' or 'contO') indicating the player continues their previous move.

If the computer is playing as 'X' or in continuation mode 'contX', the function first initializes the best score to negative infinity and the best move to None. It then iterates through each available spot on the board and temporarily sets the spot to 'O', the opponent player's marker. The function then calls the minimax function with the updated board and other parameters. After the minimax function returns, the function sets the spot back to ' ' to undo the temporary change. If the score returned by minimax is greater than the current best score, the function updates the best score and best move variables accordingly. Finally, the function returns the best move found.

If the computer is playing as 'O' or in continuation mode 'contO', it does pretty the same thing but it sets the spot to “X”.

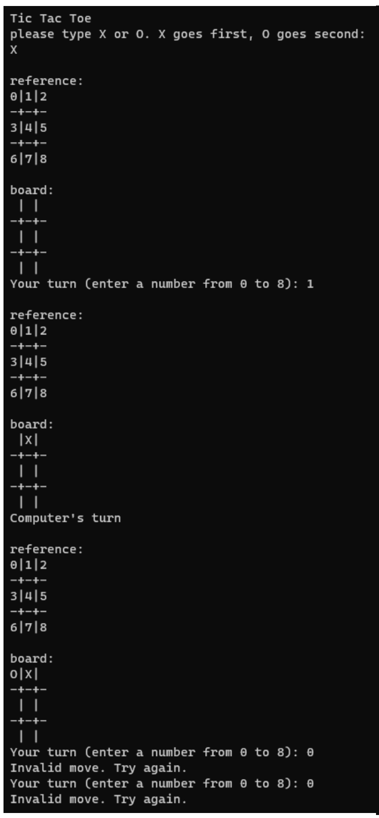
Game:

The game starts from this function play_game(mode), where mode is the variable which represents who starts first.

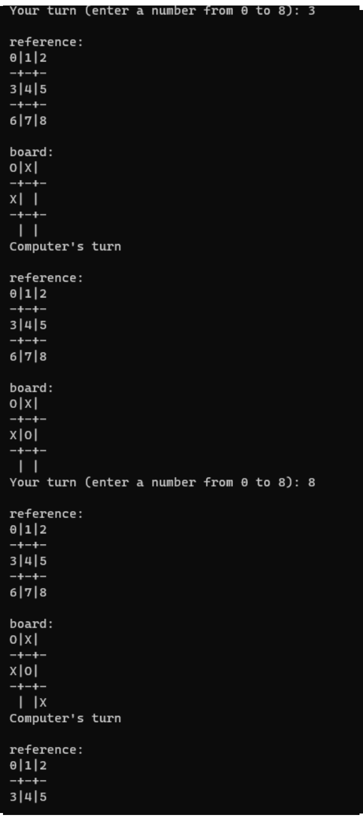
The function starts by displaying the Tic Tac Toe board using the display_board function. Then, it enters a while loop that continues until the board is full. If the current player is 'X' or in continuation mode 'contX', the function prompts the human player to enter their move using input() and checks if the move is valid by verifying that the spot on the board is empty. If the move is invalid, the function prints an error message and continues the loop. If the move is valid, the function sets the spot to 'X', updates the board using display_board, checks if the human player has won, and then sets mode to 'contX' so that the computer player knows it is their turn. If the current player is 'O' or in continuation mode 'contO', the function calls the get_computer_move function to determine the computer player's move. The function then sets the spot on the board to 'X', updates the board using display_board, checks if the computer player has won, and prompts the human player to enter their move using input(). The function checks if the move is valid, and if it is not, it prints an error message and continues the loop. If the move is valid, the function sets the spot to 'O', updates the board using display_board, and checks if the human player has won. Finally, the function sets mode to 'contO' so that the human player knows it is their turn. If the board is full and no one has won, the function prints a tie message.

Tests:

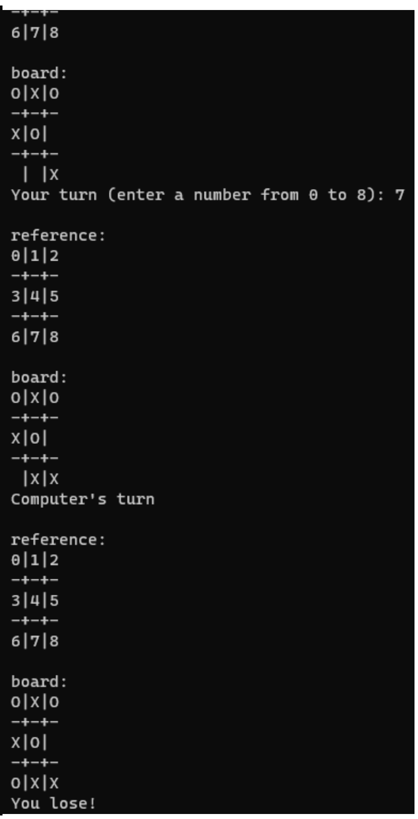
When the human starts the game:



```
Tic Tac Toe
please type X or O. X goes first, O goes second:
X

reference:
0|1|2
-+-+-
3|4|5
-+-+-
6|7|8

board:
 | |
-+-+-
 | |
-+-+-
 | |
Your turn (enter a number from 0 to 8): 1

reference:
0|1|2
-+-+-
3|4|5
-+-+-
6|7|8

board:
 |X|
-+-+-
 | |
-+-+-
 | |
Computer's turn

reference:
0|1|2
-+-+-
3|4|5
-+-+-
6|7|8

board:
O|X|
-+-+-
 | |
-+-+-
 | |
Your turn (enter a number from 0 to 8): 0
Invalid move. Try again.
Your turn (enter a number from 0 to 8): 0
Invalid move. Try again.
```

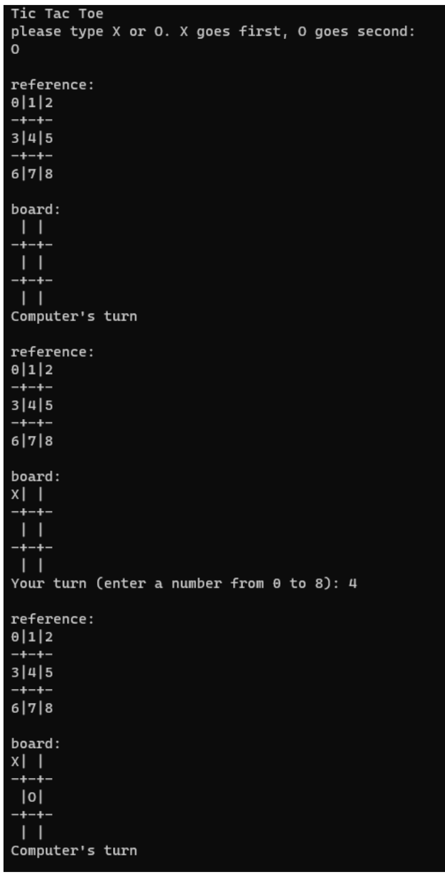


```
Your turn (enter a number from 0 to 8): 3

reference:
0|1|2
-+-+-
3|4|5
-+-+-
6|7|8

board:
O|X|
-+-+-
X| |
-+-+-
 | |
Computer's turn

reference:
0|1|2
-+-+-
3|4|5
-+-+-
6|7|8

board:
O|X|
-+-+-
X|O|
-+-+-
 | |
Your turn (enter a number from 0 to 8): 8

reference:
0|1|2
-+-+-
3|4|5
-+-+-
6|7|8

board:
O|X|
-+-+-
X|O|
-+-+-
 | |X
Computer's turn

reference:
0|1|2
-+-+-
3|4|5
```

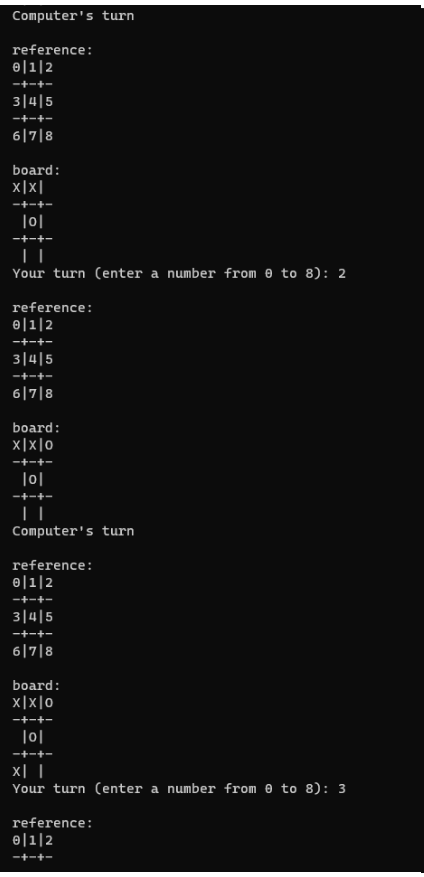


```
-+-+-
6|7|8

board:
O|X|O
-+-+-
X|O|
-+-+-
 | |X
Your turn (enter a number from 0 to 8): 7

reference:
0|1|2
-+-+-
3|4|5
-+-+-
6|7|8

board:
O|X|O
-+-+-
X|O|
-+-+-
 |X|X
Computer's turn

reference:
0|1|2
-+-+-
3|4|5
-+-+-
6|7|8

board:
O|X|O
-+-+-
X|O|
-+-+-
O|X|X
You lose!
```

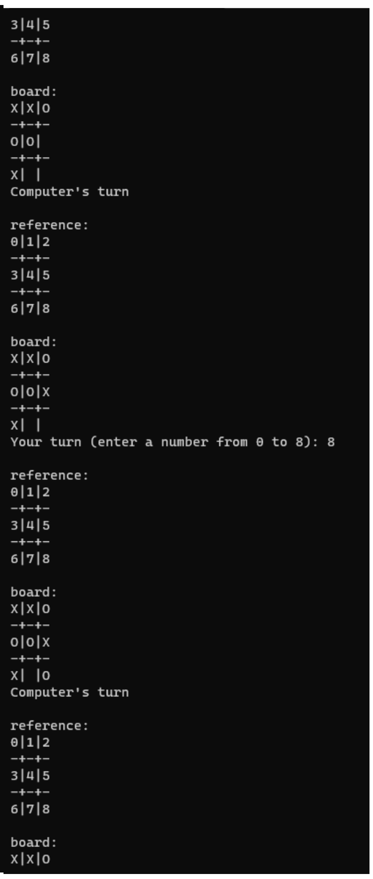
When the AI starts:



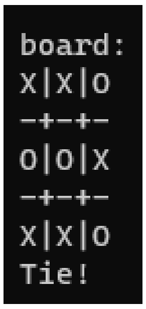
```
Tic Tac Toe
please type X or O. X goes first, O goes second:
O

reference:
0|1|2
-+-+-
3|4|5
-+-+-
6|7|8

board:
 | |
-+-+-
 | |
-+-+-
 | |
Computer's turn

reference:
0|1|2
-+-+-
3|4|5
-+-+-
6|7|8

board:
X| |
-+-+-
 | |
-+-+-
 | |
Your turn (enter a number from 0 to 8): 4

reference:
0|1|2
-+-+-
3|4|5
-+-+-
6|7|8

board:
X| |
-+-+-
 |O|
-+-+-
 | |
Computer's turn
```



```
Computer's turn

reference:
0|1|2
-+-+-
3|4|5
-+-+-
6|7|8

board:
X|X|
-+-+-
 |O|
-+-+-
 | |
Your turn (enter a number from 0 to 8): 2

reference:
0|1|2
-+-+-
3|4|5
-+-+-
6|7|8

board:
X|X|O
-+-+-
 |O|
-+-+-
 | |
Computer's turn

reference:
0|1|2
-+-+-
3|4|5
-+-+-
6|7|8

board:
X|X|O
-+-+-
 |O|
-+-+-
X| |
Your turn (enter a number from 0 to 8): 3

reference:
0|1|2
-+-+-
```



```
3|4|5
-+-+-
6|7|8

board:
X|X|O
-+-+-
O|O|
-+-+-
X| |
Computer's turn

reference:
0|1|2
-+-+-
3|4|5
-+-+-
6|7|8

board:
X|X|O
-+-+-
O|O|X
-+-+-
X| |
Your turn (enter a number from 0 to 8): 8

reference:
0|1|2
-+-+-
3|4|5
-+-+-
6|7|8

board:
X|X|O
-+-+-
O|O|X
-+-+-
X| |O
Computer's turn

reference:
0|1|2
-+-+-
3|4|5
-+-+-
6|7|8

board:
X|X|O
```



```
board:
X|X|O
-+-+-
O|O|X
-+-+-
X|X|O
Tie!
```

- final state of board

What can be improved:

Visualization could be added using pygame or tkinter libraries. Using this features we can interact with the board by clicking the field with the mouse but for now we have to provide the coordinates of the spot.

Also the winning path could be marked after the end of the game.

Conclusion:

In conclusion, the Minimax algorithm with Alpha-Beta pruning is an effective technique for playing Tic Tac Toe optimally. Using this algorithm, the computer player can evaluate all possible moves and choose the one that will maximize its chances of winning or minimize its chances of losing. With the help of Alpha-Beta pruning, the search tree can be pruned efficiently, allowing the computer to make its moves quickly and effectively. Overall, the Minimax algorithm with Alpha-Beta pruning provides a strong strategy for playing Tic Tac Toe and other similar games.

After the testing we observed that with the help of Alpha-Beta pruning the minimax algorithm is operating faster than without it since it neglects unnecessary moves.

As a result of our success we didn’t win a single game, it was a tie or lose!