Warsaw University of Technology

FACULTY OF
ELECTRONICS AND INFORMATION TECHNOLOGY

EARIN

Laboratory 2
Group 21
Variant 1
Two-player deterministic games

Tymon Żarski 310992
Bartosz Peczyński 310703

WARSAW April 4, 2024

Contents

1. Introduction

Assigned task: Develop a Tic-Tac-Toe game program using the Minimax algorithm with Alpha-Beta Pruning. Users can choose to play as the first or second player, with "X" representing the user and "O" representing the computer on the 3x3 game board.

The program must provide a user-friendly interface, displaying the board after each move and offering clear instructions. Additionally, it should handle user inputs efficiently and undergo rigorous testing to ensure functionality and minimize the computer's loss percentage.

Two test cases with three empty spaces will be created to demonstrate the Minimax algorithm's decision-making process. The report will discuss the evaluation function's suitability for Tic-Tac-Toe and its potential applicability to other games.

Algorithms used while performing the task:

- **Minimax algorithm** - The Minimax algorithm is a decision-making technique commonly used in two-player games with alternating turns, such as Tic-Tac-Toe. It aims to determine the optimal strategy for a player by recursively exploring possible moves and their outcomes. The algorithm assumes that both players play optimally, with one player maximizing their score and the other minimizing it. In the context of Tic-Tac-Toe, the Minimax algorithm evaluates possible moves by simulating the game to its conclusion, assigning scores to terminal states (win, lose, draw), and backtracking to choose the move that maximizes the player's chances of winning.
- **Addition of alpha-beta pruning to Minimax** - Alpha-beta pruning is an optimization technique applied to the Minimax algorithm to reduce the number of nodes evaluated during the search, thereby improving its efficiency. By maintaining two additional parameters, alpha and beta, representing the minimum score the maximizing player is assured of and the maximum score the minimizing player is assured of, respectively, the algorithm can prune branches of the search tree that are guaranteed to be suboptimal. This pruning mechanism eliminates unnecessary exploration of certain nodes, significantly reducing the computational overhead, especially in games with large state spaces like Tic-Tac-Toe. As a result, the addition of alpha-beta pruning enhances the performance of the Minimax algorithm, making it more suitable for real-time gameplay.

2. Implementation

2.1. Architecture of the solution

Our solution is divided into four files: runner.py and tictactoe.py. In this section, we will review all the files focusing on the **tictactoe.py**, which implements the Minimax algorithm. File **runner.py** will explained based on the visual examples as well as the description of how Minimax was utilized in the game Runner.

2.2. Step-by-step solution analysis

File tictactoe.py: To understand the Tic-Tac-Toe AI implementation without delving deeply into the code, we'll focus on the conceptual flow and critical functions of the program. This explanation emphasizes how the components interact and the logic behind key operations, minimizing direct code references.

2.2.1. Game Setup and Progression

1. Game Board Initialization: The game begins with an empty 3x3 board. Each cell can be in one of three states: empty, occupied by player O, or occupied by player X.
2. Determining the Current Player: The game alternates turns between two players, O and X. The current player is decided based on the number of empty cells; if even, it's O's turn; otherwise, X's. This is a straightforward way to ensure players alternate turns, starting with O.
3. Making Moves: Players make moves by placing their marker (O or X) in an empty cell. The board updates accordingly. This action progresses the game towards completion (win, loss, or draw).

2.2.2. Core Game Logic

1. Identifying Possible Actions: At any game state, the AI identifies all empty cells where a move can be made. These represent the immediate choices available to the current player.

```
    def get_possible_actions(board):
        possible_actions = set()

        for i in range(3):
            for j in range(3):
                if board[i][j] == CellValues.EMPTY.value:
                    possible_actions.add((i, j))

        return possible_actions
```

2. Evaluating Game State: The AI continuously checks if the game has reached a terminal state: a win for either player or a draw (no empty cells left). Wins are detected by finding three identical markers in a row, column, or diagonal.

```
    def is_game_finished(board):
        if get_winner(board) != CellValues.EMPTY.value:
            return True

        for i in range(3):
            if CellValues.EMPTY.value in board[i]:
                return False

        return True
```

3. Decision Making with Minimax: The AI uses the Minimax algorithm to decide the best move. This involves simulating all possible future moves (both for itself and the opponent), anticipating the opponent's responses, and selecting the move that leads to the most favourable outcome.
    - Minimax Algorithm: A recursive strategy that plays out all possible moves from the current state to the end of the game. For each possible move, it evaluates the game state, assuming both players make the best possible moves henceforth. The goal is to maximize the AI's chance of winning while minimizing the opponent's chance.

```
        def minimax(board):
            current_player = get_current_player(board)
            optimal_move = CellValues.EMPTY.value

            if is_game_finished(board):
                return optimal_move

            if current_player == CellValues.X.value:
                best_score = -math.inf
                for action in get_possible_actions(board):
                    score = min_value(move(board, action), -math.inf, math.inf)
                    if score > best_score:
                        best_score = score
                        optimal_move = action
            else:
                best_score = math.inf
                for action in get_possible_actions(board):
                    score = max_value(move(board, action), -math.inf, math.inf)
                    if score < best_score:
                        best_score = score
                        optimal_move = action

            return optimal_move
```

    - Alpha-Beta Pruning: This optimization reduces the number of game states the algorithm needs to evaluate by discarding scenarios that won't influence the final decision. It significantly speeds up the decision-making process without affecting the outcome.

```
        def max_value(board, alpha, beta):
            if is_game_finished(board):
                return get_game_status(board)

            max_eval = -math.inf
            for action in get_possible_actions(board):
                eval = min_value(move(board, action), alpha, beta)
                max_eval = max(max_eval, eval)
                alpha = max(alpha, eval)
                if beta ≤ alpha:
                    break
            return max_eval

        def min_value(board, alpha, beta):
            if is_game_finished(board):
                return get_game_status(board)

            min_eval = math.inf
            for action in get_possible_actions(board):
                eval = max_value(move(board, action), alpha, beta)
                min_eval = min(min_eval, eval)
                beta = min(beta, eval)
                if beta ≤ alpha:
                    break
            return min_eval
```

2.2.3. Outcome Determination

Winning the Game: The game ends when one player aligns three markers in a row, column, or diagonal or when all cells are filled, resulting in a draw. The AI evaluates the board to declare a winner or a draw, concluding the game. This approach, which describes the game's flow and the logic behind its critical operations, provides a comprehensive understanding of how the Tic-Tac-Toe AI works. It involves setting up the game, making decisions through the minimax algorithm, and aiming to win or draw by strategically placing markers based on predicted outcomes.

File runner.py In a Python-scripted Tic-Tac-Toe game utilizing the **pygame** library, the game begins with initialization procedures that set up the pygame environment, define the dimensions of the game window, and establish essential game variables, including colours for UI elements and fonts for text rendering. The game state is initialized to an empty board, awaiting player moves. The main loop of the game handles user events, such as quitting the game or interacting with the UI to make moves.

Initially, players are given the option to choose between playing as X or O, after which the game board is rendered according to the current state of play. Empty slots are displayed alongside player and AI moves, with the AI decisions driven by a Minimax algorithm that simulates future moves to determine the most advantageous outcome. The game continuously checks the board to identify if there's a win, loss, or tie, thereby adjusting the flow of play—whether prompting the AI to make a move or concluding the game with a message indicating the game's outcome. Through these steps, the script not only facilitates interactive gameplay against an AI opponent but also demonstrates fundamental concepts in GUI development, event handling, and AI integration within a gaming context, offering a comprehensive and engaging user experience.

2.3. Results and search visualizations

Section describes a Python script using the **pygame** library to simulate Tic-Tac-Toe game versus Minimax algorithm.

Key features of simulation include:

- Initializing Pygame and setting up a window with a hello message and two buttons for *X* or *O* symbol selection.

Figure 2.1. Initial draw of the Tic-Tac-Toe game

- Initializing pygame and setting up a window board and awaiting for user move or Minimax move depending respectively on the previous selection.

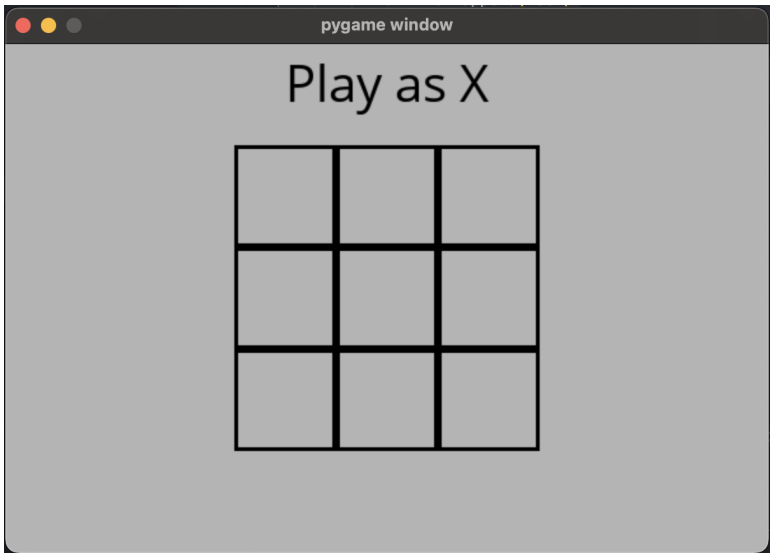

Figure 2.2. User move on the Tic-Tac-Toe board

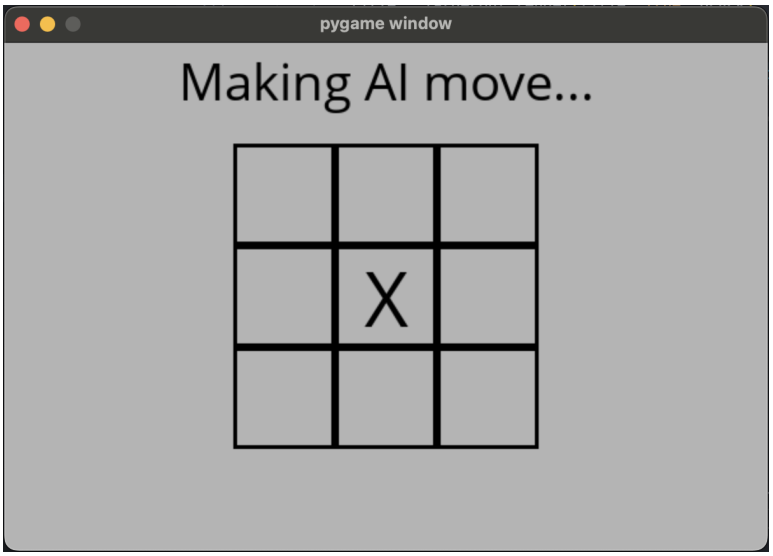

Figure 2.3. Minimax move on the Tic-Tac-Toe board

- Continually accepting valid moves and awaiting the terminal state of the game.
- After the terminal state is reached (no more moves are possible), the game's result is concluded where there could be a User, Minimax winner or a tie. In addition, we are given the option to start the game again, making a fill cycle to the initial state of the simulation.

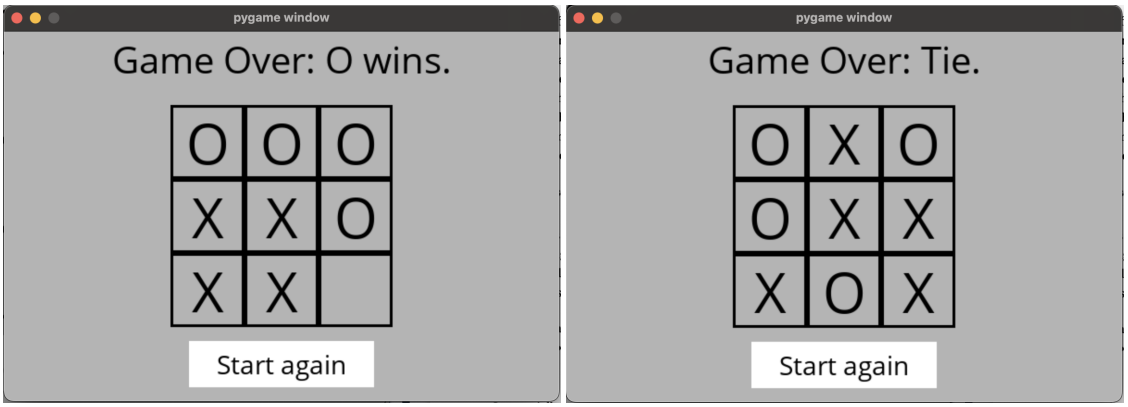

Figure 2.4. Final simulation step

The script is designed to run indefinitely until the user quits by closing the window or pressing the **q** key. This simulation provides an interactive and comparative way to take a challenge against the Minimax algorithm.

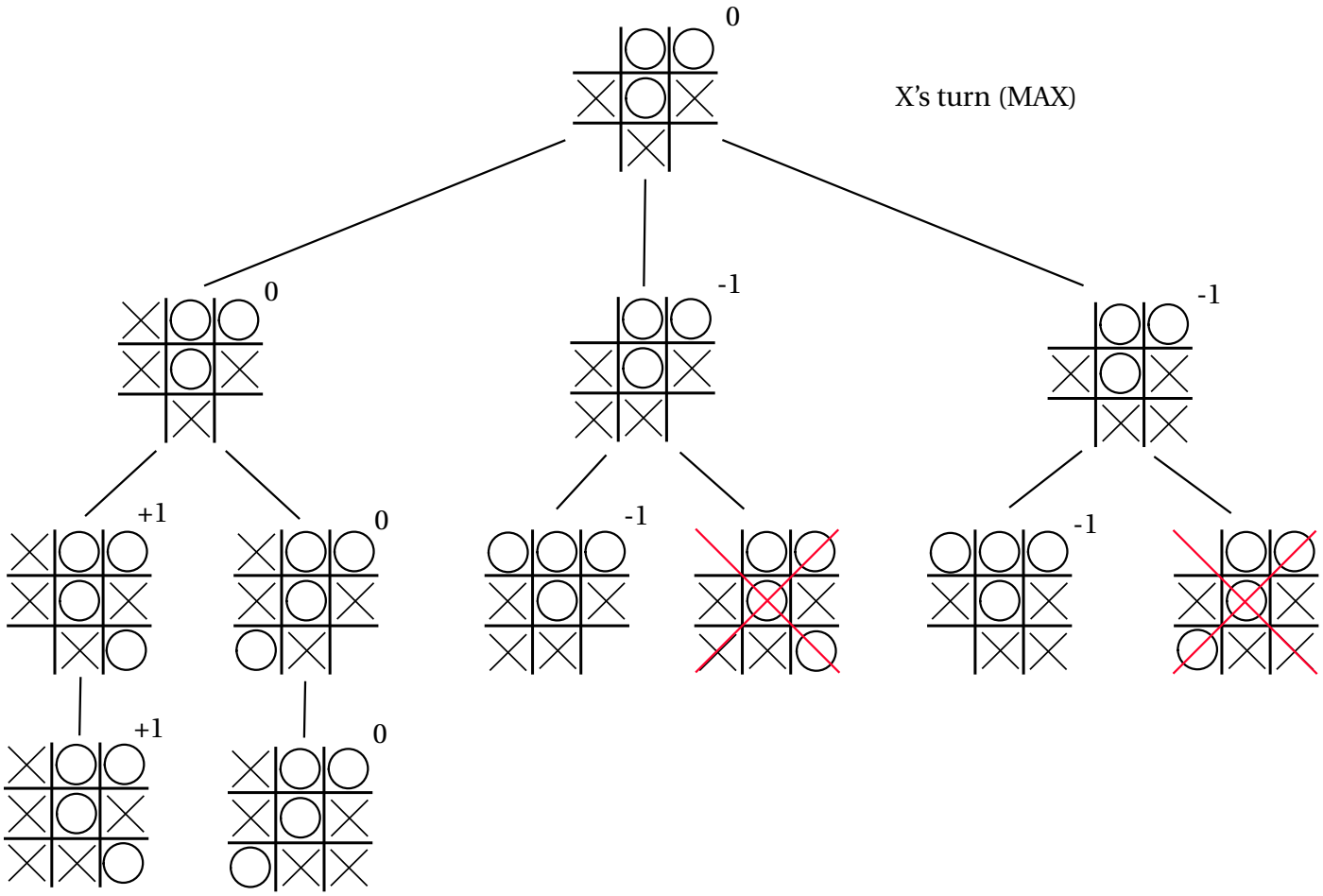

Figure 3.1. Test case 1

3. Discussion

Draw 2 test cases with 3 empty spaces left (so you will have three branches at the beginning). Draw the whole decision tree and cross-out branches that are filtered out by alpha-beta pruning. Near each of the left variants, write the return value of your evaluation function. Include the images in the report with explanations.

Test cases can be seen on Figure 3.1 and Figure 3.2. Nodes that got pruned are crossed out with a red cross. In the upper-right corner of every board that was not pruned, there is a value of the evaluation function.

Depending on the task instructions, discuss the implemented evaluation function. Answer the questions asked in the task.

3.1. Discuss the evaluation function you have used.

The game of tic-tac-toe is focused on the final result, which is win, draw, or loss. During the game, there is no metric suggesting how the players are performing and which one is most probable to win. As a result,

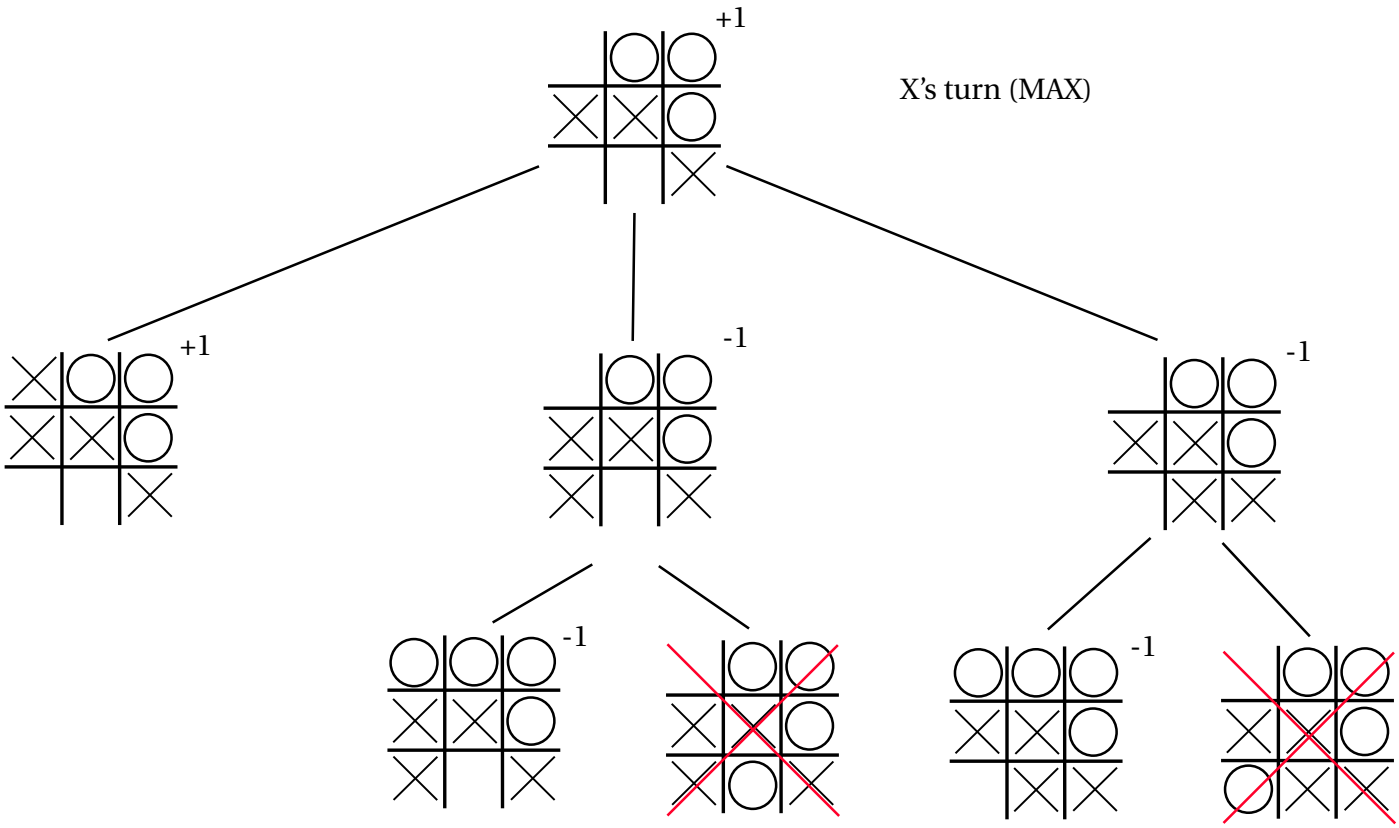

Figure 3.2. Test case 2

we have chosen a function that returns -1 if AI lost, 0 if the game ended with a draw, and 1 if AI won. This function works well for tic-tac-toe, as the board is small, and we may easily evaluate all moves until the end of the game.

3.2. Will it work well for checkers or chess? Why? If not, why not?

Although minimax and alpha-beta pruning algorithms are well-suited for checkers and chess, our evaluation function is not. Those two games are more complex and have bigger boards, which leads to an enormous number of possible moves until the end of the game. Additionally, it is possible to estimate the current probability of winning by considering the number of pieces or figures that a player has on the board. A further improvement would be to consider the type of the figure (the queen is better than the pawn).

4. Conclusion

Write conclusion including what you have learned, what was difficult and what can be done better

Thanks to this laboratory, we have learned how to design solutions that can find an optimal strategy. This knowledge might be helpful in areas like designing computer players (bots) for computer games, as well as engineering and finance. We used the Minimax algorithm and extended it with alfa-beta pruning to provide better performance. Pygame library helped us easily program this basic game and connect it with the algorithm's response. Solving this problem and playing in a game against the bot we implemented was quite exciting and fun.