## INTRODUCTION

This project is about creation of a Tic-Tac-Toe game featuring a graphical interface using Python and Pygame, complemented by an AI opponent that utilizes the Minimax algorithm to ensure optimal play. The project aims to offer hands-on experience with game development and adversarial search algorithms, focusing on engaging user interaction through a GUI, strategic AI gameplay, and comprehensive documentation to explain the logic and functionality of the implementation. From setting up the development environment to final testing, this project encapsulates the challenges and learnings of integrating AI into a classic game, providing an accessible yet complex platform for understanding the intricacies of AI-driven game design.

## ENVIRONMENT SETUP and INITIAL CONFIGURATION

**Python and Pygame Installation**:

Ensured Python was installed and then installed Pygame using `**pip install pygame**` to manage game graphics and interactions.

**Project Structure Setup**:

Created a folder `Project2_CIS597_W2024` and placed the provided distribution code within it after unzipping.

## IMPLEMENTATION STEPS

**Game Logic and Flow in tictactoe.py and Minimax Algorithm:**

The code represents a Tic-Tac-Toe AI that uses the minimax algorithm to determine the best move. The game is played on a 3x3 board, where two players take turns marking a cell in an attempt to place three of their marks in a horizontal, vertical, or diagonal line. Here's an explanation of each function and the overall logic flow:

**`initial_state()`**

This function initializes the game board as a 3x3 matrix filled with `None` values, indicating that all cells are empty at the start of the game.

**`player(board)`**

Determines whose turn it is based on the current state of the board by counting the number of `X` and `O` marks. Since `X` always goes first, if there are more `X` marks than `O` marks, it's `O`'s turn, and vice versa.

**`actions(board)`**

Returns a set of all possible moves (as `(i, j)` tuples) that can be made given the current board state. A move is possible if the corresponding cell is empty (`None`).

**`result(board, action)`**

Given a board and an action (a move), returns a new board state resulting from making that move. It deep copies the board to avoid mutating the original, then applies the move if it's valid (i.e., the targeted cell is empty). It raises an exception if the move is invalid.

**`winner(board)`**

Checks all possible winning combinations (three rows, three columns, two diagonals) for either player. If a line is found where all three cells are the same (and not empty), that player (`X` or `O`) is returned as the winner. Otherwise, if no winner is found, it returns `None`.

**`terminal(board)`**

Determines if the game is over. A game is over if there is a winner or if the board is full (no empty cells). Returns `True` if the game has ended, otherwise `False`.

**`utility(board)`**

Evaluates the utility of the terminal board state from the perspective of `X`. Returns `1` if `X` has won, `-1` if `O` has won, and `0` for a tie.

**`minimax(board)`**

The main function that implements the minimax algorithm. It checks if the board is in a terminal state; if so, there's no further move to make, and it returns `None`. Otherwise, it calls `max_value` or `min_value` depending on which player's turn it is (`X` or `O`, respectively), to explore all possible future game states and choose the move that leads to the best outcome.

**`max_value(board)`**

Recursively explores all possible moves for the `X` player (maximizing player) and returns the highest utility value from those moves, along with the move that leads to that utility. If the board is in a terminal state, it returns the utility of the board and `None` for the move.

**`min_value(board)`**

Similar to `max_value`, but for the `O` player (minimizing player). It explores all possible moves for `O` and returns the lowest utility value, along with the move that leads to that utility.

**Minimax Algorithm**

The Minimax algorithm is a decision-making tool used in game theory and AI to determine the optimal move for a player, assuming the opponent also plays optimally. It simulates all possible moves in a game, forecasting their outcomes to minimize the possible loss for a worst-case scenario or maximize the player's chance of winning.

Here, Minimax is utilized to enable the AI to choose the best possible move against the human player. It recursively examines all possible future moves and their outcomes (win, lose, draw), evaluating them based on a utility value assigned to terminal states (1 for a win, -1 for a loss, 0 for a draw). The AI predicts potential responses from the human player to these moves, aiming to maximize its own utility while minimizing the opponent's utility. This ensures the AI plays optimally, making it a challenging opponent.

**'minimax' Function**: Decides whether to use the max_value or min_value function based on the current player. It aims to maximize the utility for X and minimize it for O.

**Logic/Game Flow**

1. Start: Initialize the board with `initial_state()`.

2. Player's Turn: Determine whose turn it is with `player(board)`.

3. Possible Actions: Get a list of possible moves with `actions(board)`.

4. Minimax Decision: Use `minimax(board)` to decide the best move. This involves recursively evaluating all future game states using `max_value` and `min_value` to find the move that maximizes the player's chances of winning while minimizing the opponent's chances.

5. Resulting State: Apply the chosen move to the board using `result(board, action)`.

6. Check Terminal: Continuously check if the game has reached a terminal state with `terminal(board)`. If the game is over, use `utility(board)` to determine the outcome.

7. Repeat: Steps 2-6 are repeated until the game ends.

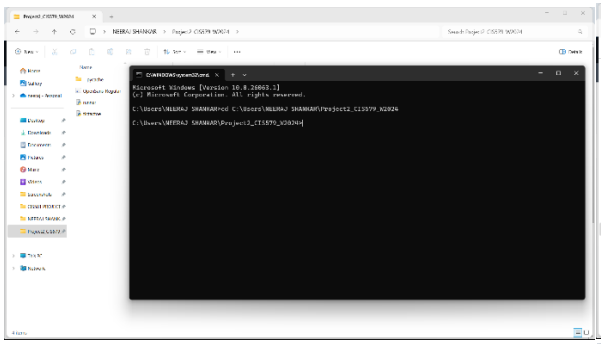
This implementation ensures the AI always makes the optimal move, considering all possible future game states.

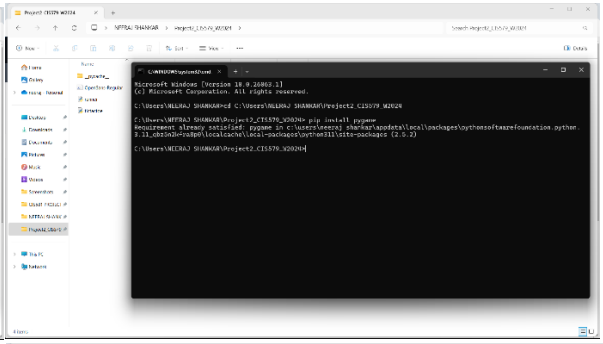
## TESTING AND VALIDATION

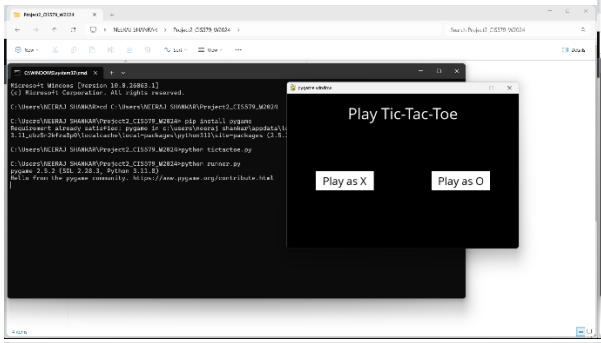
Extensive testing was conducted to ensure correctness and optimal AI behavior. Edge cases, such as attempting to play on a filled board or clicking outside the valid game area, were handled gracefully. The AI was tested to verify it never loses, only ties or wins against suboptimal human moves.

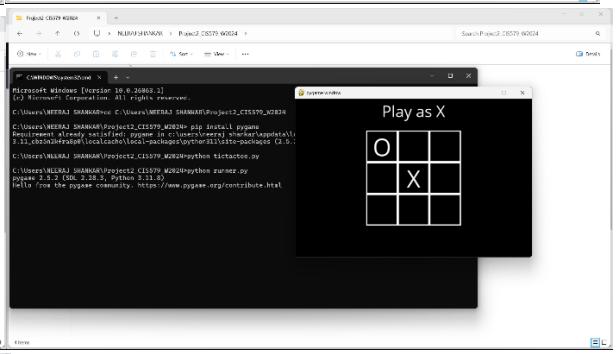
## RESULTS AND OBERSVATIONS

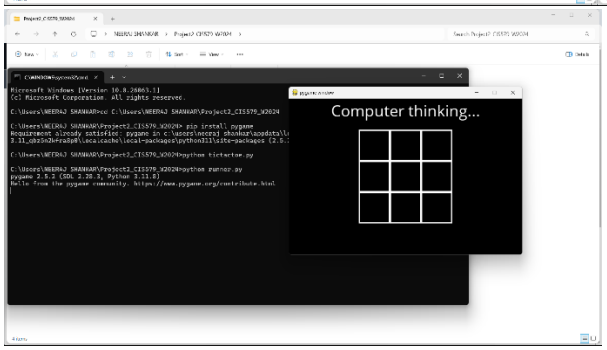
Following are the results of initial setup and GUI interface of the game and AI(computer) responses:



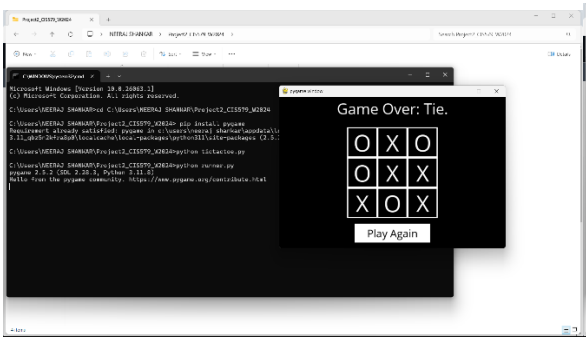


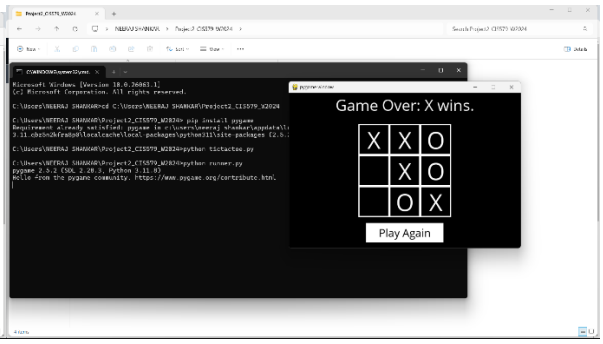


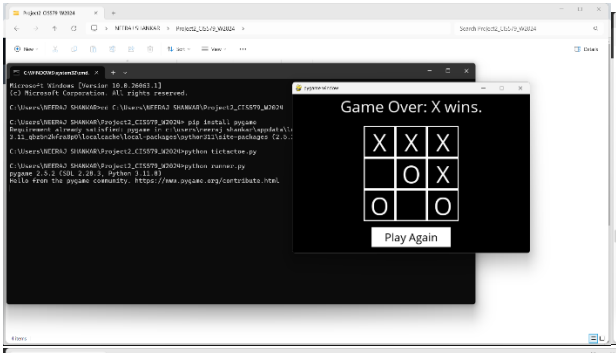



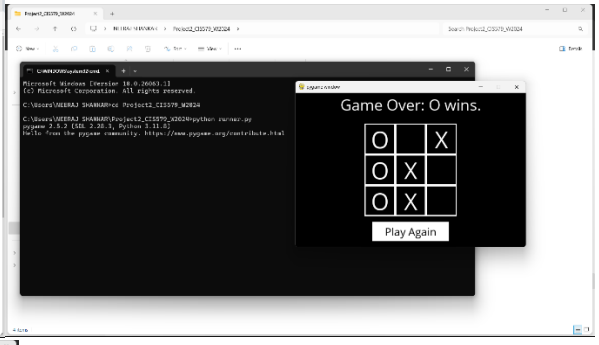

Following are the screenshots of 10 runs(games

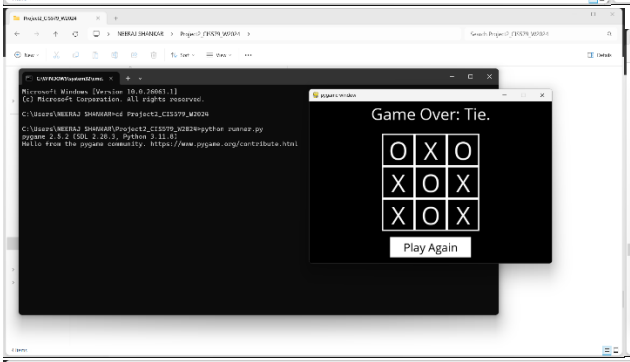


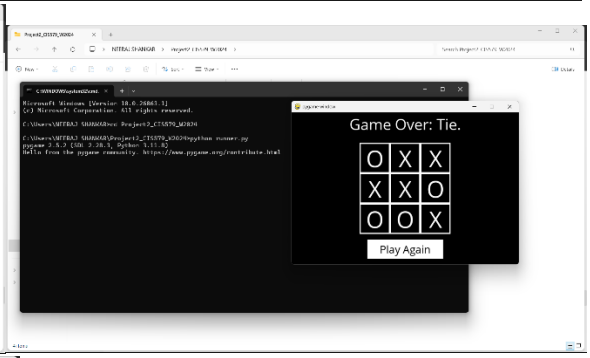



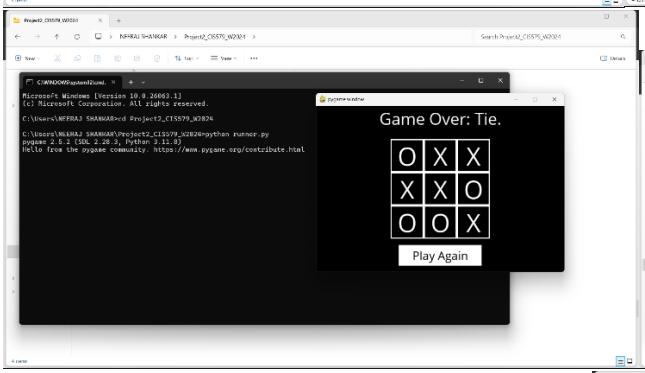

):

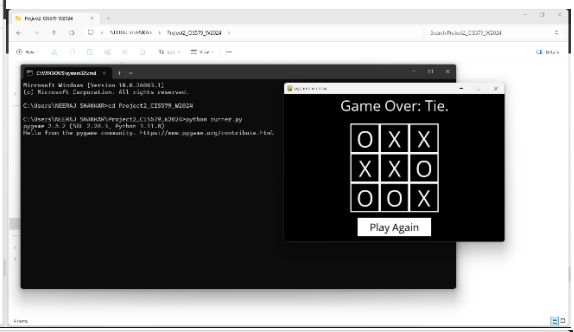

Game Over: X wins.
X X X
O X
O O
Play Again

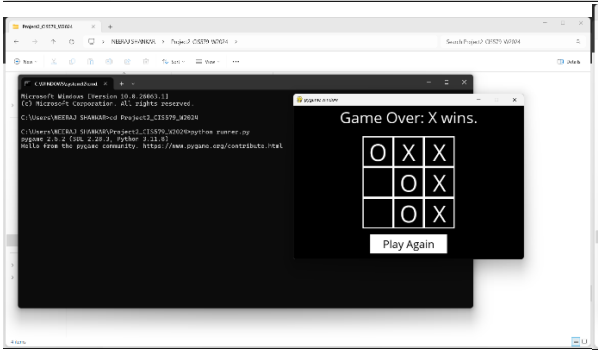

Game Over: O wins.
O X
O X
O X
Play Again

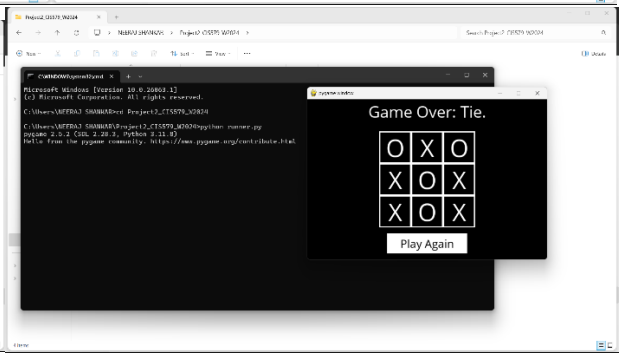

Game Over: Tie.
O X O
X O X
X O X
Play Again


Game Over: Tie.
O X X
X X O
O O X
Play Again


Game Over: Tie.
O X X
X X O
O O X
Play Again


Game Over: Tie.
O X X
X X O
O O X
Play Again


Game Over: X wins.
O X X
O X
O X
Play Again


Game Over: Tie.
O X O
X O X
X O X
Play Again

**OBSERVATIONS:**

| Game Number | Player Symbol | Computer Symbol | Result |
|---|---|---|---|
| 1 | X | O | Tie |
| 2 | O | X | Computer wins |
| 3 | O | X | Computer wins |
| 4 | X | O | Computer wins |
| 5 | O | X | Tie |
| 6 | X | O | Tie |
| 7 | O | X | Computer wins |
| 8 | O | X | Tie |
| 9 | X | O | Tie |
| 10 | O | X | Tie |

| Outcome | Count |
|---|---|
| Player Wins | 0 |
| Computer Wins | 4 |
| Ties | 6 |

The AI won four games, there were no wins by the player, and there were six ties. The occurrence of a significant number of ties suggests that in most cases, the game is resulting in the expected outcome for a well-played match of Tic-Tac-Toe, where both players are employing a good strategy. The absence of wins by the player could suggest that the AI is functioning effectively or that the player is not maximizing their chances of winning. If the AI is designed to play optimally, ties should be the most frequent outcome, which seems to be the case here, though the AI has also managed to win a few times, indicating it is leveraging the player's suboptimal moves. It may still be beneficial to review the AI's logic to ensure optimal play in all scenarios.

## CONCLUSION

The Tic-Tac-Toe project successfully integrates the Minimax algorithm into a graphical game using Python and Pygame, offering an engaging platform for understanding AI and game theory principles. By implementing a challenging AI opponent that plays optimally, the project not only enhances the gameplay experience but also provides valuable insights into adversarial search algorithms. This blend of theoretical concepts and practical application showcases the power of AI in game development, making it a compelling educational tool for students and enthusiasts alike.