

**Introduction to Artificial Intelligence**

**COSC-330**

**Course Project**

**The Tic-Tac-Toe game**



**Submitted to: Dr. Naoufel Werghi**

**Date: 27 November 2024**

**Students:**

Nawwaf Safwan Alsebaiha 100058607

Mohamed Aldhanhani 100058766

Badr Bin Rabaa 100059071

Khalifa Almansoori 100058813

**Table of Contents**

[**1.Introduction: 3**](#_Toc183391433)

[**2.Problem Statement: 3**](#_Toc183391434)

[**3.Methodology: 3**](#_Toc183391435)

[**3.1 Requirement Analysis: 4**](#_Toc183391436)

[**3.2 Design: 4**](#_Toc183391437)

[**3.3 Development: 5**](#_Toc183391438)

[**4. Outcomes: 12**](#_Toc183391439)

[**4.1 Design and Implementation of a Tree Structure: 12**](#_Toc183391440)

[**4.2 Tree Search Techniques: 13**](#_Toc183391441)

[**4.3 predicting winning strategy: 13**](#_Toc183391442)

[**4.4 Advantages and Limitation: 14**](#_Toc183391443)

[**4.5 Key Takeaways 14**](#_Toc183391444)

[**5. Conclusion: 14**](#_Toc183391445)

[**6. CODE: 15**](#_Toc183391446)

# 1.Introduction:

This report details the development of an interactive Tic-Tac-Toe game designed for the Windows platform using Python. The primary objective was to create a user-friendly and engaging application suitable for players of all ages, especially younger users, by emphasizing simplicity and accessibility.

Tic-Tac-Toe was chosen due to its straightforward yet captivating gameplay, making it an ideal foundation for showcasing artificial intelligence (AI) techniques and user interface (UI) design. The game features two AI difficulty levels:

* Easy Mode: Random moves made by the computer opponent.
* Hard Mode: An unbeatable AI powered by the Minimax algorithm.

Additionally, the project incorporates a database system to manage game data efficiently, ensuring robust gameplay mechanics and persistence.

# 2.Problem Statement:

The project aimed to address the following requirements provided by a leading game software company:

* Interactive Gameplay: Enable players to compete against a computer-controlled AI opponent.
* Simple UI: Ensure a visually appealing and intuitive interface tailored for children.
* AI Difficulty Levels: Provide distinct AI levels, ensuring enjoyment for players of varying skills.
* Data Management: Incorporate a database system to store game records and configurations.

Challenges

* Designing a game accessible for children with minimal guidance.
* Balancing AI strategy and simplicity.
* Ensuring efficient game state management using a database.

# 3.Methodology:

This section outlines the structured methodology used to develop the interactive Tic-Tac-Toe game, integrating AI-powered gameplay, a leaderboard system, and a user-friendly interface designed for younger players.

## 3.1 Requirement Analysis:

**Requirement Analysis:**

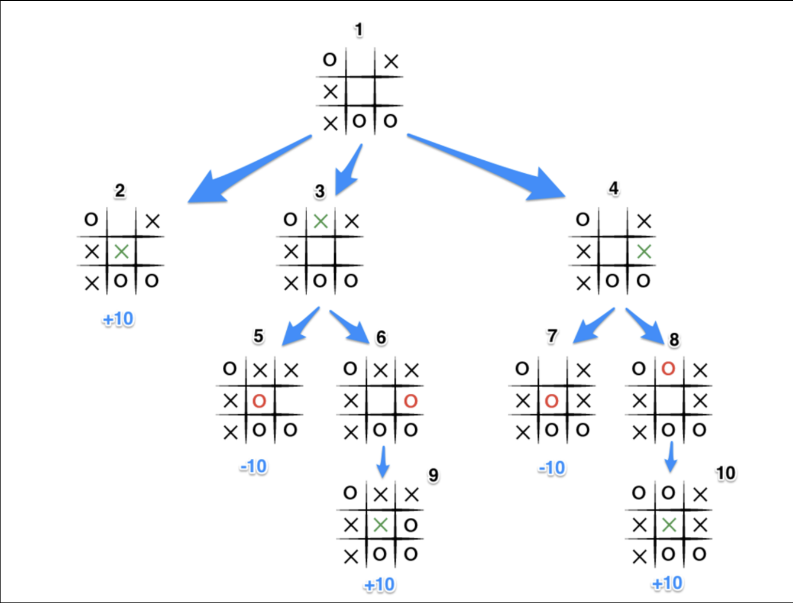
The requirements were divided into three primary components:

1. **Game Logic and AI Integration**:
   * Develop AI opponents with two difficulty levels:
     + **Easy Mode**: A simple random move generator.
     + **Hard Mode**: A sophisticated AI using the **Minimax algorithm** to ensure unbeatable gameplay.
2. **Leaderboard System**:
   * Design a database schema to track and store player performance, focusing on:
     + **Games Played**: Total games completed by each player.
     + **Scores**: Cumulative points earned across games.
     + **Time Taken**: The time spent by players during gameplay sessions.
3. **User Interface (UI/UX) Design**:
   * Develop a visually appealing and child-friendly interface, focusing on:
     + Simplicity in layout for easy navigation.
     + Bright and engaging colors to maintain user interest.
     + Clear icons and game elements suitable for younger audiences.

## 3.2 Design:

**The design phase covered three key aspects: gameplay AI, leaderboard system, and UI/UX.**

1. **UI/UX Design:**
   * **Created prototypes to visualize the game interface with an emphasis on accessibility and engagement for children.**
   * **Selected a bright and vibrant color palette to make the interface visually appealing.**
2. **AI Logic:**
   * **Represented game states as a tree structure:**
     + **Nodes: Represented board configurations after each move.**
     + **Branches: Represented potential transitions between states.**
   * **Used the Minimax algorithm to evaluate possible game states, ensuring optimal moves by the AI in "Hard Mode."**



**Figure 1: Decision tree visualization representing the Minimax algorithm, illustrating possible game states and transitions for optimal AI decision-making**

1. **Leaderboard Schema:**
   * **Designed the Leaderboard table to store and manage player performance metrics:**

|  |
| --- |
| **CREATE TABLE Leaderboard (**  **games\_played INT NOT NULL,**  **player\_name VARCHAR(255) NOT NULL,**  **score INT NOT NULL,**  **time\_taken FLOAT NOT NULL**  **);** |

* + **games\_played: Tracks the total number of games played by a player.**
  + **player\_name: Stores the player's name as a unique identifier.**
  + **score: Maintains the cumulative score of the player.**
  + **time\_taken: Captures the total time spent by the player during gameplay.**

## 3.3 Development:

1. **Game Logic and AI Development**:

* Implemented the fundamental game mechanics, including:
  + A grid structure for the game board.
  + Turn-based gameplay for human vs. computer mode.
* Developed AI functionality:
  + **Easy Mode**: Used random move generation:

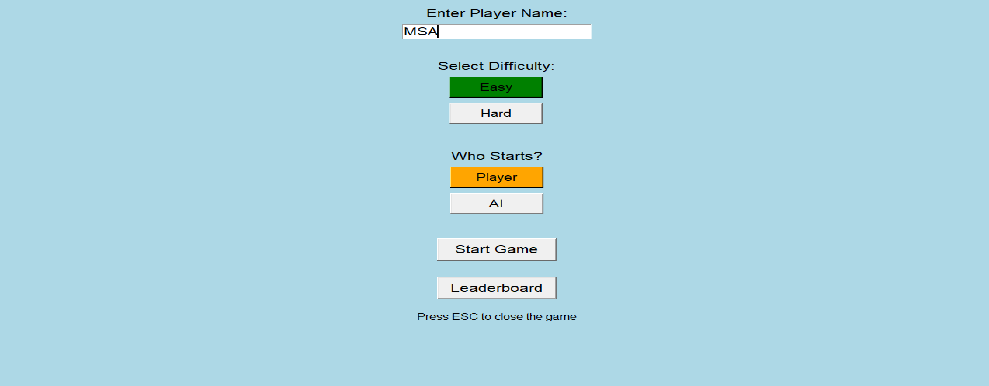
|  |
| --- |
| import random  def easy\_mode(board):  available\_moves = [i for i in range(9) if board[i] == ' ']  return random.choice(available\_moves) |

* + **Hard Mode:** Implemented the Minimax algorithm to evaluate all possible game states and select optimal moves:

|  |
| --- |
| def minimax(board, is\_maximizing):  winner = check\_winner(board)  if winner == "X":  return 1  elif winner == "O":  return -1  elif is\_draw(board):  return 0  if is\_maximizing:  best\_score = -float('inf')  for move in get\_available\_moves(board):  board[move] = "X"  score = minimax(board, False)  board[move] = None  best\_score = max(score, best\_score)  return best\_score  else:  best\_score = float('inf')  for move in get\_available\_moves(board):  board[move] = "O"  score = minimax(board, True)  board[move] = None  best\_score = min(score, best\_score)  return best\_score |

1. **Interactive Elements**:

* Incorporated interactive components, such as clickable cells for player moves.



**Figure 2: Dynamic color-changing buttons demonstrating user interaction for selecting game difficulty or player turn.**

* Added dynamic win/lose messages to provide feedback after each game session.

1. **Leaderboard Integration**:

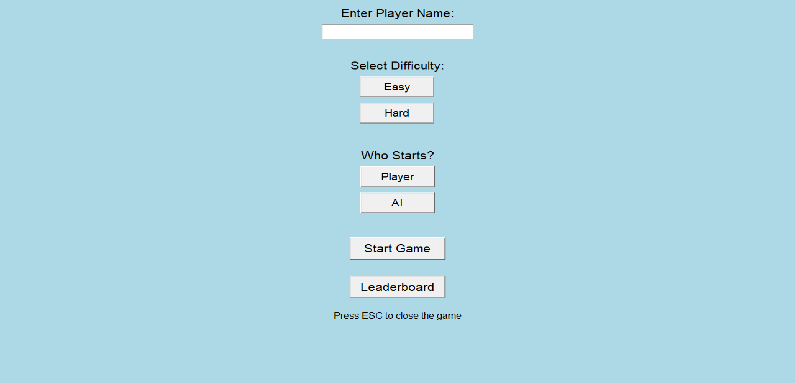
* The leaderboard automatically updated player statistics after each game. Below is the code snippet for updating the leaderboard:

|  |
| --- |
| import mysql.connector  def update\_leaderboard(player\_name, games\_played, score, time\_taken):  connection = mysql.connector.connect(host="localhost", user="root", password="", database="tictactoe")  cursor = connection.cursor()  query = """  INSERT INTO Leaderboard (player\_name, games\_played, score, time\_taken)  VALUES (%s, %s, %s, %s)  ON DUPLICATE KEY UPDATE  games\_played = games\_played + VALUES(games\_played),  score = score + VALUES(score),  time\_taken = time\_taken + VALUES(time\_taken);  """  cursor.execute(query, (player\_name, games\_played, score, time\_taken))  connection.commit()  connection.close() |

1. **UI Implementation**:

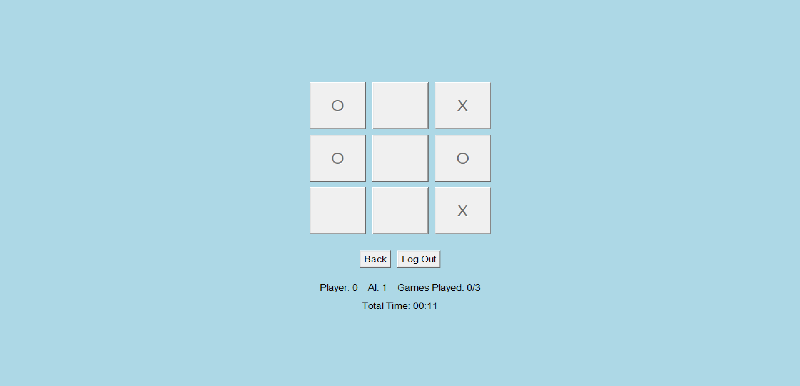
The graphical user interface (GUI) of the Tic-Tac-Toe game is designed using Python's **Tkinter** library. It provides an engaging and intuitive experience, particularly for younger players. The design emphasizes interactivity, simplicity, and visual appeal while supporting multiple game features.

* + - 1. **Interactive Frames for Navigation**:
  + The UI consists of multiple frames for seamless navigation between screens:
    - **Start Frame**: Allows players to input their name, choose difficulty, and decide the starting player.



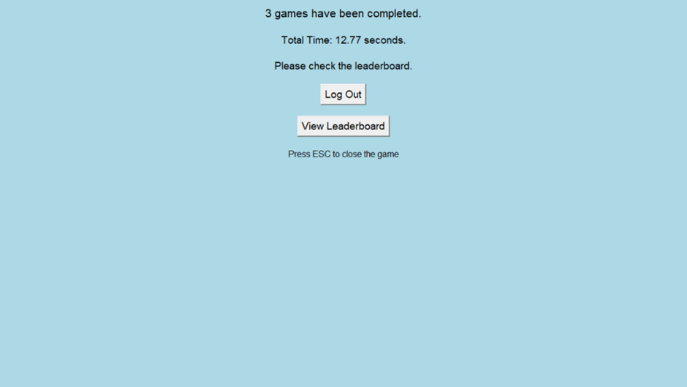
**Figure 3: Start screen interface where players can enter their name and select game options, including difficulty level and who plays first.and game options.**

* + - **Game Frame**: Displays the Tic-Tac-Toe board, score tracker, timer, and control buttons.



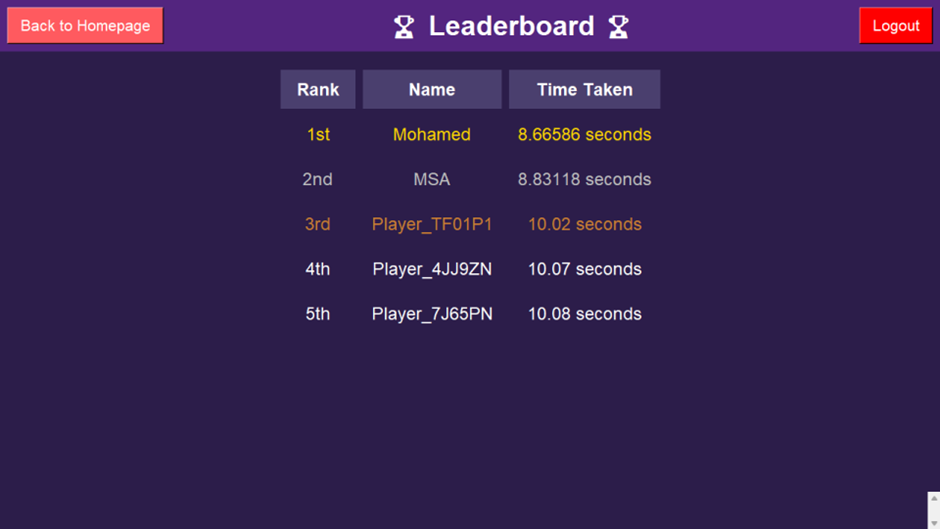
**Figure 4: n-game view showing the Tic-Tac-Toe grid during gameplay, highlighting player and AI moves.**

* + - **Results Frame**: Shows the results after completing three games, including total time taken.



**Figure 5: Winning screen displaying the congratulatory message after a successful game, indicating the player's victory.**

* + - **Leaderboard Frame**: Displays the top players based on their performance, retrieved from a MySQL database.



**Figure 6: Leaderboard interface presenting player rankings, scores, games played, and time taken, showcasing the database integration.**

|  |
| --- |
| def hide\_all\_frames():  """Hides all frames for smooth navigation between screens."""  if start\_frame: start\_frame.pack\_forget()  if game\_frame: game\_frame.pack\_forget()  if results\_frame: results\_frame.pack\_forget()  if leaderboard\_frame: leaderboard\_frame.pack\_forget() |

* + - 1. **Buttons for "Play," "Quit," and "Restart"**:
* Buttons allow players to:
  + **Play**: Start a new game with selected settings.
  + **Quit**: Exit the application.
  + **Restart**: Reset the current game session.

Code example for control buttons:

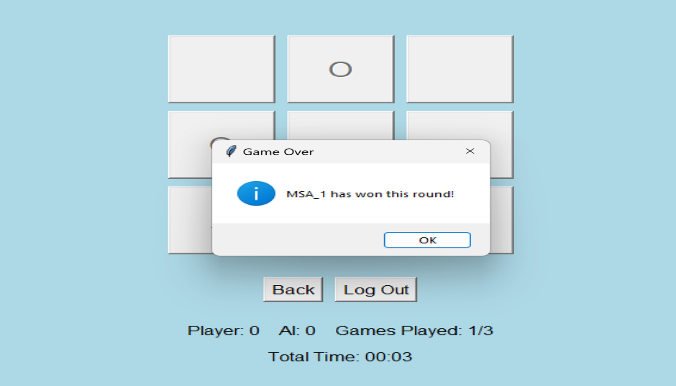
|  |
| --- |
| reset\_button = tk.Button(control\_frame, text="Back", font=('Arial', 12), command=show\_start\_screen)  reset\_button.grid(row=0, column=0, padx=5, pady=5)  log\_out\_button = tk.Button(control\_frame, text="Log Out", font=('Arial', 12), command=log\_out)  log\_out\_button.grid(row=0, column=1, padx=5, pady=5) |

* + - 1. **Dynamic Color Changes**:
* Buttons dynamically update colors to indicate actions, such as selecting a difficulty or who starts the game:
  + **Green** for selected options.
  + **System default** for unselected options.
* Game board buttons change to reflect player moves and indicate the current state.

Example snippet:

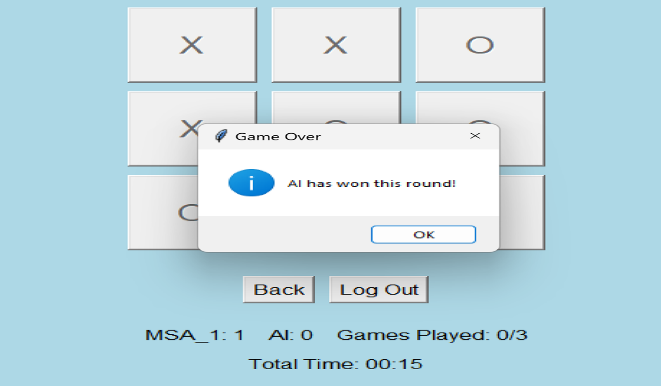
|  |
| --- |
| def set\_difficulty(selected\_difficulty):  if selected\_difficulty == 'easy':  easy\_button.config(bg='green')  Hard\_button.config(bg='SystemButtonFace')  elif selected\_difficulty == 'Hard':  Hard\_button.config(bg='red')  easy\_button.config(bg='SystemButtonFace') |

* + - 1. **Popup Messages**:
* The game uses popup messages to notify players of results:
  + "Player X Wins!"



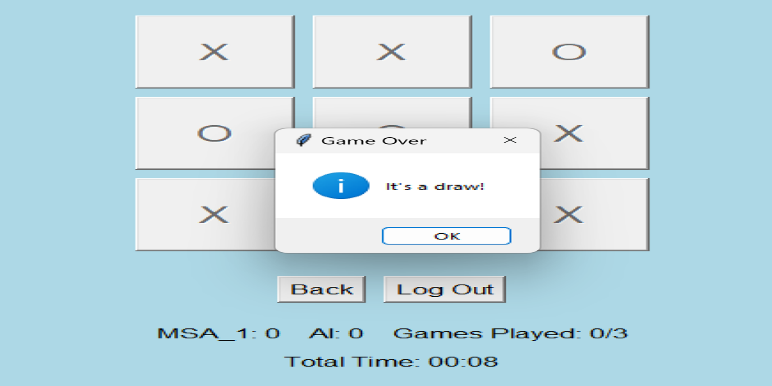
**Figure 7: Popup message indicating a player's win, offering instant feedback and enhancing user engagement.**

* + "AI Wins!"



**Figure 8: Popup message displayed when the AI wins, providing clear feedback about the outcome and emphasizing the challenge of defeating the AI.**

* + "It's a Draw!"



**Figure 9: Popup message displayed when the game ends in a draw, providing immediate feedback to the players.**

Example snippet:

|  |
| --- |
| def update\_score(winner):  if winner == "player":  messagebox.showinfo("Game Over", f"{player\_name} has won this round!")  elif winner == "ai":  messagebox.showinfo("Game Over", "AI has won this round!")  elif winner == "draw":  messagebox.showinfo("Game Over", "It's a draw!") |

1. **Leaderboard Integration**:

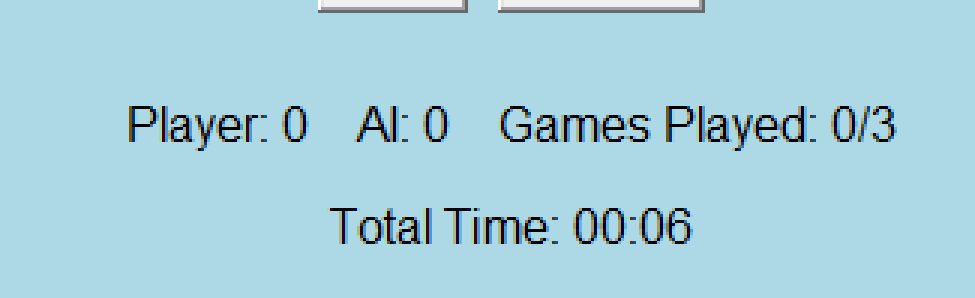
* Displays the top players and their statistics, fetched from a MySQL database. The leaderboard includes features like ranking and time taken for games.
* A scrollable leaderboard frame ensures user-friendly navigation.

Example snippet for leaderboard display:

|  |
| --- |
| cursor.execute("""  SELECT player\_name, time\_taken,  (SELECT COUNT(\*) + 1 FROM leaderboard AS t2 WHERE t2.time\_taken < t1.time\_taken) AS player\_rank  FROM leaderboard AS t1  ORDER BY time\_taken ASC LIMIT 5;  """) |

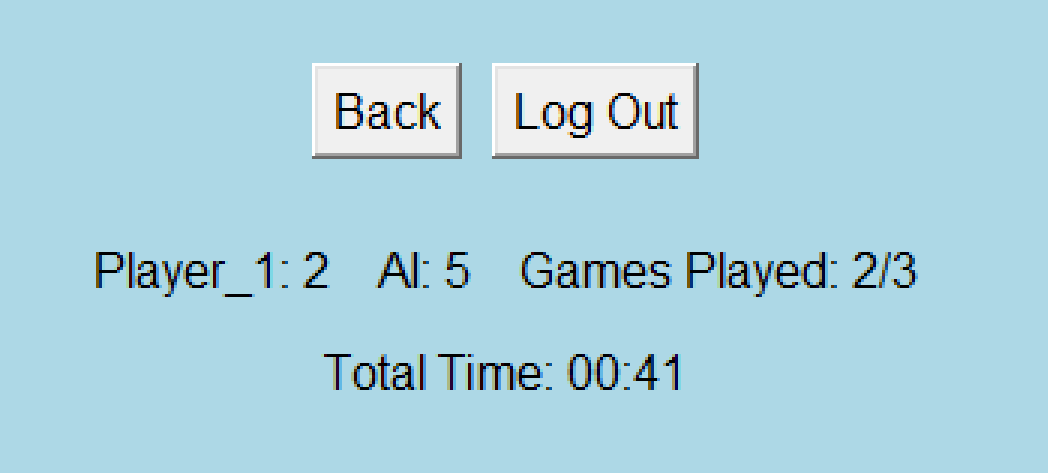
1. **Timer and Score Tracking**:

* A real-time timer tracks the duration of the game.



**Figure 10: Real-time timer interface, showing the duration of the game in seconds alongside the initial scores and game count before any rounds are played**

* Labels dynamically update scores for the player and AI after each game.



**Figure 11: Score tracking interface, dynamically updating the player and AI scores, total games played, and overall time after a game round.**

Example snippet:

|  |
| --- |
| def update\_timer():  if timer\_running and game\_start\_time is not None:  elapsed\_time = datetime.datetime.now() - game\_start\_time  minutes, seconds = divmod(elapsed\_time.total\_seconds(), 60)  timer\_label.config(text=f"Total Time: {int(minutes):02}:{int(seconds):02}")  window.after(1000, update\_timer) |

1. **Game Board**:

* The 3x3 Tic-Tac-Toe grid uses buttons for interactive gameplay.
* Each button updates dynamically based on player or AI moves.

Example snippet:

|  |
| --- |
| for i in range(9):  button = tk.Button(board\_frame, text='', font=('Arial', 20), width=5, height=2, command=lambda i=i: player\_move(i))  button.grid(row=i // 3, column=i % 3, padx=5, pady=5)  buttons.append(button) |

# 4. Outcomes:

The development of the interactive Tic-Tac-Toe game using Python and Tkinter resulted in a fully functional, user-friendly application that met the project's objectives. Below are the key outcomes achieved:

## 4.1 Design and Implementation of a Tree Structure:

The game’s decision-making system was based on a tree structure, representing all possible game states.

* **START**: The initial state of the game board before any moves are made.
* **Nodes**: Represent subsequent board configurations generated after each move.
* **Lines**: Depict possible moves or transitions between states.

The tree structure was crucial for implementing the **Minimax algorithm**, which systematically evaluates all possible moves. Each node in the tree corresponds to a unique board state, while branches represent the choices available to the player or the AI. This design allowed the AI to explore the search space effectively and make optimal decisions.

## 4.2 Tree Search Techniques:

The project demonstrated the practical application of tree search techniques through the implementation of the **Minimax algorithm**, a cornerstone of game theory and artificial intelligence.

* **Recursive Nature**: The algorithm traverses the tree to evaluate potential outcomes of moves.
* **Terminal Nodes**: Represent game end states—win, loss, or draw—with assigned evaluation scores:
  + Positive for wins.
  + Negative for losses.
  + Neutral for draws.
* **Backtracking**: Evaluated scores propagate back through the tree, guiding the AI to select the most favorable move.

This approach provided a systematic framework for decision-making, enabling the AI to anticipate opponent strategies and maximize its chances of winning. The project strengthened the understanding of recursive thinking, predictive modeling, and computational efficiency.

## 4.3 predicting winning strategy:

The AI’s strategy, particularly in the **‘Hard’ mode**, relied heavily on the Minimax algorithm’s predictive capabilities:

* **Planning Ahead**: The AI evaluated not just the immediate consequences of its moves but also potential counter-moves by the opponent.
* **Complete Game Space Analysis**: All possible game outcomes were represented as nodes in the decision tree, enabling the AI to plan several steps ahead.
* **Optimal Decisions**: By assigning values to terminal states and backtracking through the tree, the AI identified the most advantageous path. This made the AI unbeatable, as it consistently selected moves that maximized its chances of winning.

This strategy highlighted the effectiveness of advanced search techniques in real-time decision-making and strategic gameplay.

## 4.4 Advantages and Limitation:

The implementation of the Minimax algorithm provided valuable insights into the strengths and challenges of using search techniques in games:

**Advantages**:

1. **Optimal Decision-Making**: The Minimax algorithm ensured the AI consistently selected the best possible moves, enhancing the gameplay experience.
2. **Simplicity and Effectiveness**: The algorithm was well-suited to Tic-Tac-Toe due to its finite and straightforward game state space.
3. **Educational Value**: The project provided hands-on experience in systematic problem-solving and computational strategies.

**Limitations**:

1. **Scalability Issues**: For games with larger state spaces, the computational demand of the Minimax algorithm increases exponentially, making it less practical.
2. **Predictability**: The fixed strategy of the algorithm made the AI predictable after repeated gameplay, as it could not learn or adapt dynamically.
3. **Resource Intensive**: Even in Tic-Tac-Toe, exhaustive search through all possible states can be computationally demanding without optimizations like pruning.

## 4.5 Key Takeaways

The project successfully implemented and demonstrated the power of the Minimax algorithm in creating an unbeatable AI for a well-defined problem. While the algorithm was effective in optimizing gameplay decisions, its limitations in scalability and adaptability highlight the need for more advanced AI techniques, such as machine learning, in more complex scenarios. This experience serves as a foundation for future work in AI and game design.

# 5. Conclusion:

The development of the Tic-Tac-Toe game using Python and Tkinter, combined with AI strategies and a leaderboard system, successfully achieved its objectives of creating an engaging and interactive gaming experience. This project served as a practical demonstration of combining artificial intelligence, user interface design, and database integration to deliver a robust application

# 6. CODE:

|  |
| --- |
| import tkinter as tk  from tkinter import messagebox  import mysql.connector  import datetime  import random  # Create the main window  window = tk.Tk()  window.title("Tic Tac Toe")  window.configure(bg='lightblue')  # Set background color for better UI contrast  # Frames (defined globally so they can be referenced anywhere)  # These frames represent different screens or sections of the game  start\_frame = None  # The starting screen frame where players input their name and choose settings  game\_frame = None  # The frame used during the game, displaying the Tic Tac Toe board and controls  results\_frame = None  # The frame that displays the results after the game ends  leaderboard\_frame = None  # The frame showing the leaderboard with the top player scores  # Player name entry field (defined globally for easy access across functions)  player\_name\_entry = None  # Input field where the player enters their name  # Difficulty selection buttons (defined globally for easy access and modification)  easy\_button = None  # Button for selecting 'Easy' difficulty level  Hard\_button = None  # Button for selecting 'Hard' difficulty level  # Starting player selection buttons (defined globally for easy access and modification)  player\_button = None  # Button for allowing the player to start the game  ai\_button = None  # Button for allowing the AI to start the game  def hide\_all\_frames():      """      Hide all frames in the main window to ensure that only the desired frame is visible.        This function is used to switch between different screens in the application. For example,      when navigating from the start screen to the game screen, all other frames need to be hidden      to provide a clean and user-friendly interface.        By hiding all frames initially, the application can then display the correct frame depending      on the current state of the game.        The function also uses conditionals to verify if the frames have been initialized before attempting to hide them.      This prevents potential errors due to uninitialized frames, ensuring smoother navigation between screens.      """      # Check if start\_frame exists and hide it if it does      if start\_frame:          start\_frame.pack\_forget()  # Hides the start frame to make room for other frames      # Check if game\_frame exists and hide it if it does      if game\_frame:          game\_frame.pack\_forget()  # Hides the game frame, ensuring other frames can be displayed      # Check if results\_frame exists and hide it if it does      if results\_frame:          results\_frame.pack\_forget()  # Hides the results frame to prepare for another frame      # Check if leaderboard\_frame exists and hide it if it does      if leaderboard\_frame:          leaderboard\_frame.pack\_forget()  # Hides the leaderboard frame, allowing another frame to take focus    # Global variable to keep track of the current player  current\_player\_name = None  # Stores the name of the player currently using the game  # Flag to indicate whether the user is logged out or logged in  logged\_out = True  # Default to logged out state initially. True means no player is logged in.  def show\_leaderboard():      """  This function displays the leaderboard screen in the application. It hides all other frames first,  then creates and packs the leaderboard UI components, including a header with navigation buttons,  a list of top players, and optionally the rank of the currently logged-in player.  It interacts with the database to fetch leaderboard data, and ensures that the information is presented  in a user-friendly way, with features such as scrollability for easy navigation.  """      global leaderboard\_frame, current\_player\_name, logged\_out      # Hide all other frames before showing the leaderboard      hide\_all\_frames()      # Create a new frame for the leaderboard      leaderboard\_frame = tk.Frame(window, bg='#2C1D4A')      leaderboard\_frame.pack(expand=True, fill='both')      # Header setup for navigation buttons      header\_frame = tk.Frame(leaderboard\_frame, bg='#53257F', pady=10)      header\_frame.pack(fill='x')      # Back button to return to the homepage      back\_button = tk.Button(header\_frame, text="Back to Homepage", font=('Arial', 16),                              command=show\_start\_screen, bg='#FF5A5F', fg='white', padx=10, pady=5)      back\_button.pack(side='left', padx=10)      # Logout button to log the user out      logout\_button = tk.Button(header\_frame, text="Logout", font=('Arial', 16),                                command=log\_out, bg='red', fg='white', padx=10, pady=5)      logout\_button.pack(side='right', padx=10)      # Title label for the leaderboard      title\_label = tk.Label(header\_frame, text="🏆 Leaderboard 🏆", font=('Arial', 28, 'bold'), bg='#53257F', fg='white')      title\_label.pack(side='left', expand=True)      # Create a canvas for the leaderboard to enable scrolling if needed      leaderboard\_canvas = tk.Canvas(leaderboard\_frame, bg='#2C1D4A', highlightthickness=0)      leaderboard\_canvas.pack(expand=True, fill="both")      # Create a frame centered in the canvas for content      centered\_frame = tk.Frame(leaderboard\_canvas, bg='#2C1D4A')      leaderboard\_canvas.create\_window((window.winfo\_screenwidth() // 2, 0), window=centered\_frame, anchor="n")      # Container for the leaderboard list      leaderboard\_list = tk.Frame(centered\_frame, bg='#2C1D4A')      leaderboard\_list.pack(pady=20, padx=50)  # Padding to center horizontally      # Scrollbar for the leaderboard      scrollbar = tk.Scrollbar(leaderboard\_frame, orient="vertical", command=leaderboard\_canvas.yview)      leaderboard\_canvas.configure(yscrollcommand=scrollbar.set)      scrollbar.pack(side="right", fill="y")      # Add headers to the leaderboard table      headers = ["Rank", "Name", "Time Taken"]      for col, header in enumerate(headers):          tk.Label(leaderboard\_list, text=header, font=('Arial', 18, 'bold'),                   bg='#4A3F6E', fg='white', padx=20, pady=10).grid(row=0, column=col, sticky='nsew', padx=5, pady=5)      try:          # Establish a connection to the database          conn = mysql.connector.connect(              host="localhost",              user="root",              password="Mohamed@123",              database="tictactoe"          )          cursor = conn.cursor()          # Fetch the top 5 players based on the lowest time taken          cursor.execute("""              SELECT                  player\_name,                  time\_taken,                  (SELECT COUNT(\*) + 1                   FROM leaderboard AS t2                   WHERE t2.time\_taken < t1.time\_taken) AS player\_rank              FROM leaderboard AS t1              ORDER BY time\_taken ASC              LIMIT 5;          """)          top\_5\_rows = cursor.fetchall()          # Define colors for the top 3 ranks (gold, silver, bronze) and white for others          colors = ['#FFD700', '#C0C0C0', '#CD7F32'] + ['white'] \* (len(top\_5\_rows) - 3)          # Display each player in the leaderboard          for i, (name, time\_taken, rank) in enumerate(top\_5\_rows):              # Format rank with appropriate suffix (st, nd, rd, th)              rank\_text = f"{rank}{'st' if rank == 1 else 'nd' if rank == 2 else 'rd' if rank == 3 else 'th'}"              font\_color = colors[i]              # Add rank, player name, and time taken to the table              tk.Label(leaderboard\_list, text=rank\_text, font=('Arial', 18),                       bg='#2C1D4A', fg=font\_color, padx=10, pady=10).grid(row=i + 1, column=0, sticky='nsew', padx=5, pady=5)              tk.Label(leaderboard\_list, text=name, font=('Arial', 18),                       bg='#2C1D4A', fg=font\_color, padx=10, pady=10).grid(row=i + 1, column=1, sticky='nsew', padx=5, pady=5)              tk.Label(leaderboard\_list, text=f"{time\_taken} seconds", font=('Arial', 18),                       bg='#2C1D4A', fg=font\_color, padx=10, pady=10).grid(row=i + 1, column=2, sticky='nsew', padx=5, pady=5)          # Show "Your Rank" section if the user is logged in and has recent data          if current\_player\_name and not logged\_out:              cursor.execute("""                  SELECT                      player\_name,                      time\_taken,                      (SELECT COUNT(\*) + 1                       FROM leaderboard AS t2                       WHERE t2.time\_taken < t1.time\_taken) AS player\_rank                  FROM leaderboard AS t1                  WHERE player\_name = %s              """, (current\_player\_name,))              recent\_player = cursor.fetchone()                if recent\_player:                  name, time\_taken, rank = recent\_player                  # Display a label indicating the user's rank                  tk.Label(leaderboard\_list, text="Your Rank", font=('Arial', 18, 'bold'), bg='#FF5A5F', fg='white').grid(                      row=len(top\_5\_rows) + 2, column=0, columnspan=3, pady=10)                    # Display user's rank, name, and time taken                  rank\_text = f"{rank}{'st' if rank == 1 else 'nd' if rank == 2 else 'rd' if rank == 3 else 'th'}"                  tk.Label(leaderboard\_list, text=rank\_text, font=('Arial', 18),                           bg='#2C1D4A', fg='green', padx=10, pady=10).grid(row=len(top\_5\_rows) + 3, column=0, sticky='nsew', padx=5, pady=5)                  tk.Label(leaderboard\_list, text=name, font=('Arial', 18),                           bg='#2C1D4A', fg='green', padx=10, pady=10).grid(row=len(top\_5\_rows) + 3, column=1, sticky='nsew', padx=5, pady=5)                  tk.Label(leaderboard\_list, text=f"{time\_taken} seconds", font=('Arial', 18),                           bg='#2C1D4A', fg='green', padx=10, pady=10).grid(row=len(top\_5\_rows) + 3, column=2, sticky='nsew', padx=5, pady=5)      except mysql.connector.Error as err:          # Show an error message if there's an issue with the database          messagebox.showerror("Database Error", f"Error fetching leaderboard data: {err}")      finally:          # Close the cursor and connection to the database          cursor.close()          conn.close()          # Update the scrollable region of the leaderboard canvas          leaderboard\_list.update\_idletasks()          leaderboard\_canvas.config(scrollregion=leaderboard\_canvas.bbox("all"))  # Function to handle logout and hide the recent rank from the leaderboard  def logout\_and\_hide\_rank():      """      This function is responsible for handling user logout actions. It sets the logged\_out flag to True,      indicating that no player is currently logged in. This is used to update the UI accordingly, such as      hiding the player's recent rank from the leaderboard.      After logging out, it refreshes the leaderboard to reflect that the user is no longer logged in,      thereby hiding personalized ranking information.      """      global logged\_out      logged\_out = True  # Set logged out status to true      show\_leaderboard()  # Refresh leaderboard to hide the recent player's rank  # Constants for players  PLAYER = 'X'  # Constant to represent the human player in the game (usually 'X')  AI = 'O'      # Constant to represent the AI opponent in the game (usually 'O')  # Variables to track the start time of the game, number of games played, and game timer label  game\_start\_time = None  # Variable to store the time when the game starts for tracking game duration  games\_played = 0        # Counter to track the total number of games played  total\_time = 0          # Tracks total time spent across all games to help calculate average playtime  # Initialize the board as a list of empty strings  board = ['' for \_ in range(9)]  # Initialize a 3x3 board with empty strings to represent an empty game board  # Variables to store player choices and game difficulty  difficulty = None      # Variable to hold the current difficulty setting chosen by the player (e.g., easy, medium, hard)  player\_starts = None   # Boolean or flag to indicate if the player starts first in the game  player\_name = "Player" # Default player name to "Player" in case no specific name is provided  # Variables to track scores  player\_score = 0  # Score counter for the human player  ai\_score = 0      # Score counter for the AI opponent  # Start screen frame function  def reset\_homepage\_fields():      """      Resets selections and the name entry on the homepage.      This function is used to reset all fields on the homepage of the game. This includes:      - Clearing the player name entry field.      - Resetting the difficulty level selection.      - Resetting who starts the game (either the player or the AI).        It ensures that when the player navigates back to the start screen or restarts, all previous      selections are cleared, providing a fresh and consistent user experience.      """      global difficulty, player\_starts      # Reset the player name entry field      if player\_name\_entry:          player\_name\_entry.delete(0, tk.END)  # Clear any text in the player name entry box      # Reset difficulty selection buttons if they exist      if easy\_button:          easy\_button.config(bg='SystemButtonFace')  # Reset the background color to default (unselected)      if Hard\_button:          Hard\_button.config(bg='SystemButtonFace')  # Reset the background color to default (unselected)      # Reset starting player selection buttons if they exist      if player\_button:          player\_button.config(bg='SystemButtonFace')  # Reset the background color to default (unselected)      if ai\_button:          ai\_button.config(bg='SystemButtonFace')  # Reset the background color to default (unselected)      # Reset difficulty and player start choice variables      difficulty = None  # Clear the difficulty selection      player\_starts = None  # Clear the starting player selection  def show\_start\_screen():      """      Displays the start screen of the game.      This function is responsible for creating and showing the start screen of the game. It initializes      various components such as the player name entry field, difficulty selection buttons, and buttons      for selecting who starts the game. It also provides options to start the game or view the leaderboard.      The function ensures that the homepage is reset every time it is displayed to give the player a fresh      experience, and only creates the start frame if it does not already exist, to avoid redundant UI creation.      """      global start\_frame, easy\_button, Hard\_button, player\_button, ai\_button, player\_name\_entry      # Hide all other frames to ensure only the start screen is visible      hide\_all\_frames()      # Reset all fields (e.g., player name, difficulty, and starter options) to default state      reset\_homepage\_fields()      # Create and pack the start screen frame if not already created      if not start\_frame:          # Main start screen frame setup          start\_frame = tk.Frame(window, bg='lightblue')            # Player Name Entry Section          # Frame for the player name input          name\_frame = tk.Frame(start\_frame, bg='lightblue')          name\_frame.pack(pady=10)            # Label prompting the player to enter their name          tk.Label(name\_frame, text="Enter Player Name:", font=('Arial', 16), bg='lightblue').pack()            # Entry field for player name input          player\_name\_entry = tk.Entry(name\_frame, font=('Arial', 16))          player\_name\_entry.pack(pady=5)          # Difficulty Selection Section          # Frame for difficulty selection buttons          difficulty\_frame = tk.Frame(start\_frame, bg='lightblue')          difficulty\_frame.pack(pady=20)            # Label prompting player to choose the game difficulty level          tk.Label(difficulty\_frame, text="Select Difficulty:", font=('Arial', 16), bg='lightblue').pack()            # Button to select "Easy" difficulty level          easy\_button = tk.Button(difficulty\_frame, text="Easy", font=('Arial', 14), width=10, command=lambda: set\_difficulty('easy'))          easy\_button.pack(pady=5)            # Button to select "Hard" difficulty level          Hard\_button = tk.Button(difficulty\_frame, text="Hard", font=('Arial', 14), width=10, command=lambda: set\_difficulty('Hard'))          Hard\_button.pack(pady=5)          # Starting Player Section          # Frame for choosing who starts the game (Player or AI)          start\_choice\_frame = tk.Frame(start\_frame, bg='lightblue')          start\_choice\_frame.pack(pady=20)            # Label prompting player to select who starts first          tk.Label(start\_choice\_frame, text="Who Starts?", font=('Arial', 16), bg='lightblue').pack()            # Button to select the player as the starter          player\_button = tk.Button(start\_choice\_frame, text="Player", font=('Arial', 14), width=10, command=lambda: set\_start\_options(True))          player\_button.pack(pady=5)            # Button to select the AI as the starter          ai\_button = tk.Button(start\_choice\_frame, text="AI", font=('Arial', 14), width=10, command=lambda: set\_start\_options(False))          ai\_button.pack(pady=5)          # Start Game Button          # Button to start the game after all selections are made          start\_game\_button = tk.Button(start\_frame, text="Start Game", font=('Arial', 16), width=12, command=save\_player\_name)          start\_game\_button.pack(pady=20)          # Leaderboard Button          # Button to navigate to the leaderboard screen          leaderboard\_button = tk.Button(start\_frame, text="Leaderboard", font=('Arial', 16), width=12, command=show\_leaderboard)          leaderboard\_button.pack(pady=10)          # Close Game Instruction          # Label displaying instructions on how to exit the game          close\_instruction = tk.Label(start\_frame, text="Press ESC to close the game", font=('Arial', 12), bg='lightblue')          close\_instruction.pack(pady=10)      # Pack the start\_frame to make it visible      start\_frame.pack(expand=True, fill='both')  def set\_difficulty(selected\_difficulty):      """      Sets the difficulty for the game and visually highlights the selected button.      Args:          selected\_difficulty (str): The difficulty level chosen by the player, either 'easy' or 'Hard'.        This function updates the global difficulty variable with the player's selection and provides      a visual indication of the selected difficulty by changing the button color. The selected      button's background color is changed to green to show it is active, while the unselected button      reverts to its default state.      """      global difficulty, easy\_button, Hard\_button      # Set the global difficulty variable to the selected value      difficulty = selected\_difficulty      # Highlight the selected button by changing its background color to green      if selected\_difficulty == 'easy':          easy\_button.config(bg='green')  # Mark 'Easy' button as selected          Hard\_button.config(bg='SystemButtonFace')  # Reset 'Hard' button to default color      elif selected\_difficulty == 'Hard':          Hard\_button.config(bg='red')  # Mark 'Hard' button as selected          easy\_button.config(bg='SystemButtonFace')  # Reset 'Easy' button to default color  def set\_start\_options(player\_starts\_choice):      """      Sets who starts the game (player or AI) and visually highlights the selected button.      Args:          player\_starts\_choice (bool): Boolean value indicating who starts the game.                                       True if the player starts, False if the AI starts.      This function updates the global player\_starts variable based on the player's selection and      highlights the selected button. The button representing the starting player is highlighted in green,      while the other button is reset to its default color.      """      global player\_starts, player\_button, ai\_button      # Set the global player\_starts variable to indicate whether the player starts or not      player\_starts = player\_starts\_choice      # Highlight the selected button by changing its background color to green      if player\_starts\_choice:          player\_button.config(bg='Orange')  # Mark 'Player' button as selected          ai\_button.config(bg='SystemButtonFace')  # Reset 'AI' button to default color      else:          ai\_button.config(bg='yellow')  # Mark 'AI' button as selected          player\_button.config(bg='SystemButtonFace')  # Reset 'Player' button to default color  def save\_player\_name():      """      Saves the player's name, ensuring it is unique, and sets it for the current game session.      This function handles saving the player's name for the game, ensuring that the name is unique by      checking against existing names in the database. If the entered name is not unique, a suffix is appended      to create a unique identifier. It then stores this unique name and initiates the game.      It interacts with the database to check for name uniqueness and updates global variables to track the current      player throughout the game session.      """      global player\_name, player\_name\_entry, current\_player\_name, logged\_out      # Retrieve the player's name from the entry field or default to "Player" if none is entered      entered\_name = player\_name\_entry.get().strip() or "Player"  # Default to "Player" if no name is provided      # Connect to the database to ensure uniqueness of the player's name      try:          conn = mysql.connector.connect(              host="localhost",              user="root",              password="Mohamed@123",              database="tictactoe"          )          cursor = conn.cursor()          # Check if the entered name already exists in the leaderboard and create a unique name if necessary          unique\_name = entered\_name          count = 1          while True:              cursor.execute("SELECT player\_name FROM leaderboard WHERE player\_name = %s", (unique\_name,))              results = cursor.fetchall()              if results:  # If the name already exists, append a number to make it unique                  unique\_name = f"{entered\_name}\_{count}"                  count += 1              else:                  break          # Set the unique player name for the session          player\_name = unique\_name  # Assign the final unique name to the global player\_name variable          current\_player\_name = player\_name  # Update the global current\_player\_name for tracking purposes          logged\_out = False  # Set logged\_out to False to indicate that the player is now logged in      except mysql.connector.Error as err:          # Show an error message if there is an issue with connecting to the database or checking the name          messagebox.showerror("Database Error", f"Error checking player name uniqueness: {err}")      finally:          # Always close the cursor and connection to prevent database locks or resource leaks          cursor.close()          conn.close()      # Start the game with the unique player name      start\_game()  def log\_out():      """      Logs the user out, automatically saves progress only if a game was played, waits 10 milliseconds, and then returns to the homepage.      This function checks if any game has been played before logging out. If so, it automatically saves the player's progress      to the leaderboard. Then it resets all player-related data, updates the UI labels to reflect these changes, and navigates      back to the homepage.      """      global player\_name, player\_score, ai\_score, games\_played, total\_time, logged\_out, current\_player\_name      # Check if the player name is set and at least one game has been played      if player\_name != "Player" and games\_played > 0:          # Automatically save the player's progress to the leaderboard          time\_taken = calculate\_game\_time()  # Calculate the time taken for the current session          save\_to\_leaderboard(player\_name, player\_score, time\_taken, games\_played)  # Save progress to the leaderboard      # Reset player-related data      player\_name = "Player"  # Set the player name to the default value ("Player")      player\_score = 0  # Reset the player's score to zero      ai\_score = 0  # Reset the AI's score to zero      games\_played = 0  # Reset the number of games played to zero      total\_time = 0  # Reset the total time spent on games      current\_player\_name = None  # Clear the current player name for tracking purposes      logged\_out = True  # Mark the player as logged out      # Update displayed labels to reflect the reset values      player\_score\_label.config(text=f"{player\_name}: {player\_score}")  # Update player score label      ai\_score\_label.config(text="AI: 0")  # Update AI score label to show zero      games\_played\_label.config(text="Games Played: 0/3")  # Update games played label to reset count      timer\_label.config(text="Total Time: 00:00")  # Reset the timer label to show zero time spent      # Redirect to the leaderboard screen first      show\_leaderboard()      # After 10 milliseconds, go back to the start screen      window.after(10, show\_start\_screen)    # Bind ESC key to close the game  # - Allows the user to close the game by pressing the ESC key.  # - Enhances convenience for exiting the application quickly.  window.bind('<Escape>', lambda event: window.destroy())  # Set the window to fullscreen  # - Provides an immersive experience by setting the window to fullscreen mode.  window.attributes('-fullscreen', True)  # Button list to access buttons by index  # - Initializes an empty list to store the buttons representing the game board.  # - This makes it easier to access buttons by index.  buttons = []  # Main game frame  # - Creates the main frame to hold all sub-frames of the game interface.  # - Background color is set to 'lightblue' for a cohesive look.  game\_frame = tk.Frame(window, bg='lightblue')  # Separate frames for different sections of the game interface  # - Frame for the game board (9 buttons representing the grid)  board\_frame = tk.Frame(game\_frame, bg='lightblue')  # - Frame for control buttons like "Back" and "Log Out"  control\_frame = tk.Frame(game\_frame, bg='lightblue')  # - Frame for score and game-related labels  score\_frame = tk.Frame(game\_frame, bg='lightblue')  # Arrange frames within the main game\_frame  # - Frames are positioned within the game\_frame using a grid layout.  # - Ensures the board, controls, and scores are visually distinct and accessible.  board\_frame.grid(row=0, column=0, columnspan=3, padx=10, pady=10)  control\_frame.grid(row=1, column=0, columnspan=3, pady=10)  score\_frame.grid(row=2, column=0, columnspan=3, pady=10)  # Create the grid of buttons for the game board within board\_frame  # - Creates 9 buttons to represent the 3x3 Tic-Tac-Toe board.  # - Each button is added to the board\_frame in a 3x3 layout using grid positioning.  # - Buttons are stored in the `buttons` list for easy access by index.  # - `command=lambda i=i: player\_move(i)` allows tracking of the specific button clicked.  for i in range(9):      button = tk.Button(board\_frame, text='', font=('Arial', 20), width=5, height=2,                         command=lambda i=i: player\_move(i))      button.grid(row=i // 3, column=i % 3, padx=5, pady=5)      buttons.append(button)  # Control buttons  # - "Back" button: Allows the player to return to the start screen.  reset\_button = tk.Button(control\_frame, text="Back", font=('Arial', 12), command=show\_start\_screen)  reset\_button.grid(row=0, column=0, padx=5, pady=5)  # - "Log Out" button: Logs the player out, saves progress if necessary, and returns to the start screen.  log\_out\_button = tk.Button(control\_frame, text="Log Out", font=('Arial', 12), command=log\_out)  log\_out\_button.grid(row=0, column=1, padx=5, pady=5)  # Score, timer, and games played labels in the score\_frame  # - Displays the player's score, showing the player name and current score.  # - Dynamically updated during the game.  player\_score\_label = tk.Label(score\_frame, text=f"{player\_name}: {player\_score}", font=('Arial', 12), bg='lightblue')  player\_score\_label.grid(row=0, column=0, padx=5)  # - Displays the AI's score to track the player's performance against the AI.  ai\_score\_label = tk.Label(score\_frame, text=f"AI: {ai\_score}", font=('Arial', 12), bg='lightblue')  ai\_score\_label.grid(row=0, column=1, padx=5)  # - Displays the number of games played in the current session.  # - Provides a sense of progress, with a maximum of 3 games.  games\_played\_label = tk.Label(score\_frame, text=f"Games Played: {games\_played}/3", font=('Arial', 12), bg='lightblue')  games\_played\_label.grid(row=0, column=2, padx=5)  # Function to update scores and track games played  def update\_score(winner):      """      Updates the score based on the winner of the game, keeps track of games played,      and performs actions accordingly like displaying messages, updating labels, and starting a new session.      """      global player\_score, ai\_score, games\_played, timer\_running, total\_time      # Update scores based on the winner of the current round      if winner == "player":          player\_score += 1  # Increment player's score          games\_played += 1  # Increment number of games played by the player          games\_played\_label.config(text=f"Games Played: {games\_played}/3")  # Update games played label          messagebox.showinfo("Game Over", f"{player\_name} has won this round!")  # Display a message indicating the player won      elif winner == "ai":          ai\_score += 1  # Increment AI's score          # Decrement the games\_played if AI wins, but ensure it does not drop below zero          if games\_played > 0:              games\_played -= 1          games\_played\_label.config(text=f"Games Played: {games\_played}/3")  # Update games played label          messagebox.showinfo("Game Over", "AI has won this round!")  # Display a message indicating the AI won      elif winner == "draw":          # Handle a draw situation          messagebox.showinfo("Game Over", "It's a draw!")  # Display a message indicating the game is a draw      # Update score labels after each game      # - Reflect the latest score changes for both the player and AI      player\_score\_label.config(text=f"{player\_name}: {player\_score}")      ai\_score\_label.config(text=f"AI: {ai\_score}")      # If the player has won 3 rounds, handle the end of the session      if games\_played == 3:          # Stop the timer and calculate the total time taken for the session          timer\_running = False  # Stop the timer          total\_time = calculate\_game\_time()  # Calculate and store the total time for the session          # Show results in a new frame instead of a popup for a better user experience          show\_results\_screen()          # Save the current session results to the leaderboard for tracking player progress          save\_to\_leaderboard(player\_name, player\_score, total\_time)          # Reset all necessary variables for a new session after leaderboard update          reset\_for\_new\_session()      else:          # Reset the board for the next game round to continue playing          reset\_board()            # If AI starts, initiate AI's move          if not player\_starts:              ai\_turn()  # Function to save the player's performance to the leaderboard in the database  def save\_to\_leaderboard(player\_name, score, time\_taken, games\_played=3):      """      Save the player's name, score, time taken, and games played to the leaderboard in the database.      Ensures that the player name is unique before saving.      """      try:          # Establish a connection to the MySQL database          conn = mysql.connector.connect(              host="localhost",              user="root",              password="Mohamed@123",              database="tictactoe"          )          cursor = conn.cursor()          # Ensure that the player name is unique          # - If the player's name already exists in the leaderboard, append a suffix to make it unique.          unique\_name = player\_name          count = 1          while True:              cursor.execute("SELECT player\_name FROM leaderboard WHERE player\_name = %s", (unique\_name,))              result = cursor.fetchone()              if result:                  # If the name exists, create a new unique name by appending a suffix (e.g., Player\_1, Player\_2, etc.)                  unique\_name = f"{player\_name}\_{count}"                  count += 1              else:                  # Found a unique name that doesn't exist in the leaderboard                  break          # Insert the player's information into the leaderboard          cursor.execute("""              INSERT INTO leaderboard (player\_name, score, time\_taken, games\_played)              VALUES (%s, %s, %s, %s)          """, (unique\_name, score, time\_taken, games\_played))            # Commit the transaction to save the changes to the database          conn.commit()      except mysql.connector.Error as err:          # Handle database errors          # - Suppress duplicate entry errors (error code 1062), as unique names are being generated.          # - Show other errors to the user to notify of potential issues with the database.          if err.errno != 1062:              messagebox.showerror("Database Error", f"Error saving to leaderboard: {err}")      finally:          # Ensure that the cursor and database connection are closed properly          if cursor:              cursor.close()          if conn:              conn.close()  # Function to show results after 3 games are completed  def show\_results\_screen():      """      Replaces the main game window with a results screen showing game statistics.      Displays the total time taken, allows viewing the leaderboard, and provides a logout option.      """      global results\_frame      # Hide the current game frame to transition to the results screen      # - Ensures the main game interface is not visible while displaying the results.      game\_frame.pack\_forget()      # Initialize the results\_frame if it does not already exist      # - The frame is used to display the results after the player completes 3 games.      if results\_frame is None:          results\_frame = tk.Frame(window, bg='lightblue')      # Clear any previous content in results\_frame to ensure it is clean before displaying new information      for widget in results\_frame.winfo\_children():          widget.destroy()      # Pack and display the results frame      # - Set the frame to expand fully within the window for better visual coverage.      results\_frame.pack(expand=True, fill='both')      # Display results header to indicate that 3 games have been completed      tk.Label(results\_frame, text="3 games have been completed.", font=('Arial', 16), bg='lightblue').pack(pady=10)        # Display total time taken for the session      # - Provides the player with the statistics of their gameplay.      tk.Label(results\_frame, text=f"Total Time: {total\_time:.2f} seconds.", font=('Arial', 14), bg='lightblue').pack(pady=10)        # Prompt to check the leaderboard after completing 3 games      tk.Label(results\_frame, text="Please check the leaderboard.", font=('Arial', 14), bg='lightblue').pack(pady=10)      # Button to log out      # - Allows the player to log out from the results screen.      logout\_button = tk.Button(results\_frame, text="Log Out", font=('Arial', 14), command=log\_out)      logout\_button.pack(pady=10)      # Button to view the leaderboard      # - Clicking this button hides the results frame and shows the leaderboard screen.      leaderboard\_button = tk.Button(results\_frame, text="View Leaderboard", font=('Arial', 14),                                     command=lambda: [results\_frame.pack\_forget(), show\_leaderboard()])      leaderboard\_button.pack(pady=10)      # Instructions to close the game using the ESC key      # - Provides the player with a quick way to exit the game.      tk.Label(results\_frame, text="Press ESC to close the game", font=('Arial', 12), bg='lightblue').pack(pady=10)  # Function to reset the board and prepare for a new game  def reset\_board():      """      Resets the board to its initial empty state, preparing for a new game.      Clears all board cells and enables all buttons for the next game round.      """      global board      # Reset the game board to an empty state      # - Initializes the board list with empty strings to represent an empty 3x3 Tic-Tac-Toe grid.      board = ['' for \_ in range(9)]      # Update the button states to reflect the empty board      # - Clears the text in all buttons and sets them to a normal (enabled) state.      for button in buttons:          button.config(text='', state=tk.NORMAL)  # Function to reset all variables and prepare for a new session  def reset\_for\_new\_session():      """      Reset variables for a new session after 3 games are completed.      Resets scores, games played count, timer, and game board to start a fresh new session.      """      global player\_score, ai\_score, games\_played, total\_time, game\_start\_time      # Reset scores, games played count, and the timer      # - Start the session anew by setting all counters and time-related variables back to zero.      games\_played = 0      player\_score = 0      ai\_score = 0      total\_time = 0      game\_start\_time = None  # Clear the start time to indicate a fresh session      # Update the displayed labels to reflect the reset state      # - Ensures all score-related UI elements are updated to show zero scores and progress.      games\_played\_label.config(text=f"Games Played: {games\_played}/3")      player\_score\_label.config(text=f"{player\_name}: {player\_score}")      ai\_score\_label.config(text=f"AI: {ai\_score}")      timer\_label.config(text="Total Time: 00:00")      # Reset the game board to an initial state for the new session      reset\_board()  # Function to check if a player has won the game  def check\_winner(player):      """      Checks if the given player has won the game.      Args:          player (str): The symbol representing the player ('X' or 'O').      Returns:          bool: True if the player has a winning combination, False otherwise.      """      # Define all possible winning combinations for a 3x3 Tic-Tac-Toe board      # - Includes rows, columns, and diagonals.      winning\_combinations = [          [0, 1, 2], [3, 4, 5], [6, 7, 8],  # Rows          [0, 3, 6], [1, 4, 7], [2, 5, 8],  # Columns          [0, 4, 8], [2, 4, 6]              # Diagonals      ]      # Check each winning combination to determine if the player has won      for combo in winning\_combinations:          if board[combo[0]] == board[combo[1]] == board[combo[2]] == player:              return True  # Return True if the player has three symbols in a row      # Return False if no winning combination is found      return False  # Function to calculate the time taken for the game  def calculate\_game\_time():      """Calculates the time taken for the game in seconds and updates the start time for continuous tracking."""      global game\_start\_time      # Check if the game start time is set      if game\_start\_time is None:          return 0  # Return 0 if the game start time is not available      # Record the current time as the end time      end\_time = datetime.datetime.now()      # Calculate the difference between the start and end time in seconds      time\_taken = (end\_time - game\_start\_time).total\_seconds()      # Update the start time for ongoing time tracking      game\_start\_time = end\_time      # Return the total time taken      return time\_taken  # Function to handle the player's move  def player\_move(index):      """Handles the player's move by updating the board, checking for a winner or a draw."""        # Ensure the cell is empty and there is no winner before proceeding      if board[index] == '' and not check\_winner(PLAYER) and not check\_winner(AI):          # Update the board with the player's move          board[index] = PLAYER          # Update the button text to reflect the player's move and disable the button          buttons[index].config(text=PLAYER, state=tk.DISABLED)          # Check if the player won after their move          if check\_winner(PLAYER):              # Update the score to reflect the player's win              update\_score("player")              return          # Check for a draw if no empty cells remain          if '' not in board:              # Update the score to reflect a draw              update\_score("draw")              return          # Let AI make a move if the game is still ongoing          ai\_turn()  # Function for AI's move based on selected difficulty  def ai\_turn():      """Handles the AI's move based on the selected difficulty level."""      # Choose AI's move based on the difficulty selected by the player      if difficulty == 'easy':          move = ai\_move\_easy()  # AI selects a random empty cell for easy difficulty      else:          move = find\_best\_move()  # AI uses minimax to find the best move for higher difficulty      # If a valid move is found, update the board and disable the button      if move != -1:          board[move] = AI          buttons[move].config(text=AI, state=tk.DISABLED)          # Check if AI won after the move          if check\_winner(AI):              # Update the score to reflect AI's win              update\_score("ai")              return          # If no empty cells are left, declare a draw          if '' not in board:              update\_score("draw")  # Function for easy AI move (random choice)  def ai\_move\_easy():      """AI move for easy difficulty - randomly selects an empty cell."""        # Create a list of all empty cells      empty\_spots = [i for i in range(9) if board[i] == '']      # Return a random choice from empty cells if available      if empty\_spots:          return random.choice(empty\_spots)      return -1  # Return -1 if no moves are possible  # Helper function to evaluate the board for minimax (AI) purposes  def evaluate():      """Evaluates the board and returns a score based on the game result."""      # Check if AI has won and return +10 if true      if check\_winner(AI):          return 10      # Check if the player has won and return -10 if true      elif check\_winner(PLAYER):          return -10      # Return 0 for a draw or if the game is ongoing      return 0  # Minimax algorithm for AI to find the best move  def minimax(is\_maximizing):      """Minimax algorithm to find the best move for the AI."""      # Evaluate the board to determine if the game has ended      score = evaluate()      # Return score if a winning or losing condition has been reached      if score == 10 or score == -10:          return score      # If no cells are empty, return a draw score of 0      if '' not in board:          return 0      if is\_maximizing:          # Maximize AI's score          best\_score = -1000          for i in range(9):              if board[i] == '':                  board[i] = AI  # Make a hypothetical move                  best\_score = max(best\_score, minimax(False))  # Recursively find the best score                  board[i] = ''  # Undo the move          return best\_score      else:          # Minimize player's score          best\_score = 1000          for i in range(9):              if board[i] == '':                  board[i] = PLAYER  # Make a hypothetical move                  best\_score = min(best\_score, minimax(True))  # Recursively find the best score                  board[i] = ''  # Undo the move          return best\_score  # Function to find the best move for AI using minimax  def find\_best\_move():      """Finds the best move for the AI using the Minimax algorithm."""      best\_val = -1000  # Initialize the best score for AI      best\_move = -1  # Initialize the best move as invalid (-1)      # Iterate through all cells to evaluate the best possible move      for i in range(9):          if board[i] == '':  # Check if the cell is empty              board[i] = AI  # Make a hypothetical move              move\_val = minimax(False)  # Evaluate the move using minimax              board[i] = ''  # Undo the move              # Update the best move if the current move value is higher              if move\_val > best\_val:                  best\_move = i                  best\_val = move\_val      # Return the index of the best move      return best\_move  # Function to start the game  def start\_game():      """Initializes the game by recording start time, starting timer, and hiding the start screen."""      global game\_start\_time, timer\_running      # Ensure the player name is set before starting      if not player\_name:          messagebox.showerror("Error", "Player name is not set. Please enter a valid name.")          return      # Set the game start time if not already set      if game\_start\_time is None:          game\_start\_time = datetime.datetime.now()  # Record the start time      # Start the game timer      timer\_running = True      update\_timer()  # Begin the timer update loop      # Hide the start screen and show the game frame      start\_frame.pack\_forget()      game\_frame.pack(expand=True)      # Reset the board for a new game      reset\_board()      # If AI starts, let AI make the first move      if not player\_starts:          ai\_turn()  # Function to update the timer every second  def update\_timer():      """Updates the timer label with the time elapsed since the game started."""      if timer\_running and game\_start\_time is not None:          # Calculate the elapsed time          elapsed\_time = datetime.datetime.now() - game\_start\_time          seconds = elapsed\_time.total\_seconds()          # Format the time in MM:SS format          minutes = int(seconds // 60)          seconds = int(seconds % 60)          time\_string = f"Total Time: {minutes:02}:{seconds:02}"          # Update the timer label          timer\_label.config(text=time\_string)          # Re-call this function after 1 second for continuous update          window.after(1000, update\_timer)  # Adding timer label to the game frame  # - Displays the total time elapsed in the current session  timer\_label = tk.Label(game\_frame, text="Total Time: 00:00", font=('Arial', 12), bg='lightblue')  timer\_label.grid(row=3, column=1, columnspan=1)  # Position the timer in the game frame  # Show the start screen initially  # - Display the initial start screen when the application runs  show\_start\_screen()  # Run the Tkinter event loop  # - Starts the application's main event loop for user interaction  window.mainloop() |