جامعة مصر للمعلوماتية
EGYPT UNIVERSITY OF INFORMATICS

Egypt University of Informatics
Computer and Information Systems
Introduction to Artificial Intelligence Course

Tic Tac Toe Game

Submitted by:

- Basmala Yasser
- Renad Sameh
- Abeer Sherif
- Shahd Abo Elfotouh

Supervised by:

- Dr. Mohamed Taher Alrefaie
- Dr.Farid Zaki
- Eng. Tukka Mohammad

Issued Date: 5/1/2025

Table of Contents

1. Introduction:

The 3-by-3 Tic Tac Toe grid requires players to place three identical markers (X or O) in a row, column, or diagonal to win. In this game, a user-friendly tkinter interface is paired with two distinct AI approaches: Normal Mode, which uses heuristics (immediate win/block checks) and some randomness, and Unbeatable Mode, powered by a Minimax algorithm with memoization so the AI never loses. Players place “X,” the AI responds with “O,” and any winning line is highlighted. After each match, a post-game analysis provides brief feedback on every move, showcasing basic GUI design alongside classic game AI concepts in Python.

2. Searching algorithms used

2.1. Heuristic (Greedy) Search — Used in Normal Mode:

In **Normal Mode**, the AI uses a heuristic (greedy) strategy that relies on immediate checks for winning and blocking moves—**without** exploring multiple moves ahead, making it a **heuristic or greedy** strategy. It also mixes in **randomness** to differentiate Easy, Medium, and Hard difficulties:

1. **Easy**: 70% random, 30% “smart” (immediate checks).
2. **Medium**: 50% random, 50% “smart.”
3. **Hard**: 20% random, 80% “smart.”

2.1.1. Logic Flow (Heuristic)

- **Immediate Win**: The AI checks if it can **instantly** win on this move by placing “O” somewhere that creates three-in-a-row.
- **Immediate Block**: If no winning move is found, the AI checks if the player (“X”) can win on the next turn and **blocks** it.
- **Strategic Move or Random**: If neither an immediate win nor block is needed, the AI picks a strategic or random move (like center, corners, or edges).

- **Difficulty Variation**: Depending on the difficulty level, there's a certain percentage chance the AI will just do a **random** move rather than the "smart" one.

2.1.2. Code Snippet:

```
2. def find_heuristic_move(self):
        difficulty_probability = {
            "Easy": 0.3,
            "Medium": 0.5,
            "Hard": 0.8
        }
        prob = difficulty_probability[self.ai_difficulty]
        if random.random() > prob:
            return self.random_choice_move()

        # If AI can win in the next move, take it
        for i in range(3):
            for j in range(3):
                if self.board[i][j] is None:
                    self.board[i][j] = "O"
                    if self.winner(self.board) == "O":
                        self.board[i][j] = None
                        return (i, j)
                    self.board[i][j] = None

        #If player can win in the next move, block it
        for i in range(3):
            for j in range(3):
                if self.board[i][j] is None:
                    self.board[i][j] = "X"
                    if self.winner(self.board) == "X":
                        self.board[i][j] = None
                        return (i, j)
                    self.board[i][j] = None

        # Take the center if available
        if self.board[1][1] is None:
            return (1, 1)

        #Take one of the corners if available
        corners = [(0, 0), (0, 2), (2, 0), (2, 2)]
        available_corners = []
        for corner in corners:
            row = corner[0]
            column = corner[1]
            if self.board[row][column] is None:
                available_corners.append(corner)

        #Take one of the edges
        edges = [(0, 1), (1, 0), (1, 2), (2, 1)]
        available_edges = []
        for edge in edges:
            row = edge[0]
            column = edge[1]
            if self.board[row][column] is None:
                available_edges.append(edge)
        if available_edges:
            return random.choice(available_edges)
        return None
```

2.2. Minimax Search (with Memoization) — Used in Unbeatable Mode:

In **Unbeatable Mode**, the AI performs a **full-depth search** of all possible board states using **Minimax**, enhanced by **memoization** to skip repeated configurations, ensuring efficient and optimal gameplay. This ensures the AI **never loses** if it can force a win or tie

2.2.1. Logic Flow (Minimax)

- **Full Depth Search**: Tries every possible way to place an “O” (if maximizing) or “X” (if minimizing) until a terminal state (win, lose, or tie).
- **Maximizing / Minimizing**:
    - On the AI’s turn, the algorithm maximizes the outcome for “O” (trying to reach +1 for a sure win).
    - On the opponent’s turn (the player, placing “X”), the algorithm minimizes the AI’s outcome (trying to push it toward –1 pushing it toward a loss).
- **Scoring**:
    - +1 if the AI (“O”) eventually forces a win.
    - -1 if the player (“X”) forces a win.
    - 0 if it ends in a tie.
- **Memoization**

Each board layout (converted to a tuple) is stored in `self.memo`. If the AI sees the same layout again, it uses the saved result instead of calculating from scratch.

2.2.2. Snippet for Unbeatable Mode:

```
def unbeatable_ai_move(self):
    best_score = float('-inf')
    best_move = None

    for i in range(3):
        for j in range(3):
            if self.board[i][j] is None:
                self.board[i][j] = "O"
                sc = self.minimax(self.serialize_board(self.board), False)
                self.board[i][j] = None
                if sc > best_score:
                    best_score = sc
                    best_move = (i, j)
    if best_move:
        i, j = best_move
        self.board[i][j] = "O"
        self.buttons[i][j].config(text="O", state=tk.DISABLED,
bg="lightcoral")
        self.move_history.append(("AI", i, j))

def minimax(self, board, maximizing):
    if board in self.memo:
        return self.memo[board]
    w, _ = self.get_winner_and_line(board)
    if w == "O":
        return 1
    if w == "X":
        return -1
    if self.is_board_full(board):
        return 0
    best = float('-inf') if maximizing else float('inf')
    for i in range(3):
        for j in range(3):
            if board[i][j] is None:
                nb = [list(r) for r in board]
                nb[i][j] = "O" if maximizing else "X"
                sc = self.minimax(self.serialize_board(nb), not maximizing)
                best = max(best, sc) if maximizing else min(best, sc)

    self.memo[board] = best
    return best
```

2.2.3. Why It's Unbeatable

- By **examining every possible move** sequence, the AI identifies a forced **win** (score +1) or at least a **tie** (score 0) whenever available.
- If there is **any** route to avoid losing, Minimax will find and follow it, ensuring the AI can't be beaten under optimal play.

3. Objectives

- **Primary Goals:**
    - Develop an interactive Tic Tac Toe game with a graphical user interface.
    - Implement two AI modes: Normal (with varying difficulties) and Unbeatable (using Minimax).
    - Provide post-game analysis and feedback to enhance user experience.
- **Secondary Goals:**
    1. Learn and apply Python's tkinter library for GUI development.
    2. Understand and implement AI search algorithms in game development

4. Function Map

4.1. GUI Methods (Handling Screens, Layout, and User Interaction)

These functions are responsible for creating and managing the graphical user interface, handling user inputs, and updating the display accordingly.

- **`create_intro_screen(self)`**
  Sets up the home screen with instructions and mode-selection buttons (Normal / Unbeatable).
- **`create_normal_mode_selection(self)`**
  Allows the user to choose the Easy, Medium, or Hard difficulty level in Normal Mode.
- **`create_unbeatable_mode_start_selection(self)`**
  Lets the user decide who starts first in Unbeatable Mode (You or AI).
- **`set_normal_mode_difficulty(self,difficulty)`**
  Updates the difficulty level for Normal Mode and navigates to the next selection screen.
- **`create_normal_mode_start_selection(self)`**
  Prompts the user to choose who starts first in Normal Mode.
- **`set_mode_and_start(self,mode,ai_starts)`**
  Configures the game mode (Normal or Unbeatable), sets who starts first, resets game states, and initiates the game board.
- **`create_game_board(self)`**
  Constructs the **3×3** game board with buttons, initializes the status label, and adds a "Return" button to go back to the intro screen.
- **`player_move(self,i, j)`**
  Handles the player's move when they click on a cell, updates the board, and triggers the AI move if the game continues.
- **`end_game(self)`**
  Finalizes the game by displaying the result, highlighting the winning line if applicable, disabling all buttons, and providing a "Play Again" option.
- **`reset_game(self)`**
  Resets the game by clearing the current board and returning to the intro screen.

- **`create_algorithms_screen(self)`**

  Explains the AI algorithms based on the selected mode (Normal or Unbeatable).

- **`update_possible_moves()`**

  Updates the list of possible moves in the GUI.

4.2. Game Logic / AI Methods

These functions manage the game state, implement the AI strategies for both Normal and Unbeatable modes, and provide post-game analysis.

4.2.1. Normal Mode AI (Heuristic/Greedy Approach)

Handles the AI behavior based on different difficulty levels using a heuristic approach combined with randomness.

- **`normal_ai_move(self)`**

  Chooses and executes a move based on the selected difficulty (Easy, Medium, Hard).

- **`find_heuristic_move(self)`**

  Implements the heuristic logic for move selection:

  1. Winning move.
  2. Block opponent's win.
  3. Take the center.
  4. Occupy a corner.
  5. Fallback to edges.

- **`random_ai_move(self)`**

  Executes a random move if no heuristic-based move is found.

- **`random_choice_move(self)`**

  Selects a random move from available cells.

- **`ai_move(self)`**

  Executes the AI’s turn, calling either the Normal or Unbeatable AI method.

4.2.2. Unbeatable Mode AI (Minimax Algorithm with Memoization)

Implements the Minimax algorithm enhanced with memoization to ensure the AI never loses by exhaustively evaluating all possible game states.

- **`unbeatable_ai_move(self)`**

  Iterates through all empty cells, temporarily places an “O”, evaluates the move using `minimax()`, and selects the move with the highest score to ensure the best possible outcome.
- **`minimax(self,board,maximizing)`**

  The core recursive Minimax function:
  1. **Memoization Check**: Returns the cached score if the board state has been evaluated before.
  2. **Terminal State Check**: Determines if the current board state is a win, loss, or tie.
  3. **Recursion**:
     - **Maximizing Player (AI)**: Tries all possible moves for “O” and selects the move with the highest score.
     - **Minimizing Player (Human)**: Simulates the opponent’s moves by placing “X” and selects the move with the lowest score.
  4. **Caching**: Stores the evaluated score for the current board state to optimize future computations.

4.2.3. Post-Game Analysis

Provides feedback on each move after the game concludes.

- **`post_game_analysis(self)`**

  Generates and displays a summary of all moves made during the game along with feedback for each move.
- **`analyze_move(self,idx)`**

  Produces a feedback message for each move based on its sequence in the game (e.g., early moves, mid-game moves, critical moments).

4.3. Winner / Terminal Checks

These functions determine the game's outcome by checking for winners or a full board.

- **`get_winner_and_line(self,board)`**
  Checks all possible winning combinations (rows, columns, diagonals) to identify if there is a winner. Returns the winner ("X" or "O") and the winning line's coordinates.
- **`winner(self,b)`**
  A helper function that returns only the winner ("X" or "O") by calling **`get_winner_and_line(self,board).`**
- **`is_board_full(self,b)`**
  Checks if the board is completely filled with no empty cells remaining.
- **`terminal(self,b)`**
  Determines if the game has ended either by a win or a tie by checking **`winner(b)`** and **`is_board_full(self,b).`**
- **`serialize_board(self,b)`**
  Converts the 2D board list into a tuple of tuples, making it hashable for use as keys in the **`self.memo`** dictionary for memoization.

5. Challenges and Solutions

- **Development Challenges**:
  - Implementing the Minimax algorithm efficiently.
  - Managing the GUI state transitions smoothly.
  - Ensuring the AI’s Unbeatable Mode performs optimally without delays.
- **Solutions Implemented**:
  - Used memoization to optimize Minimax.
  - Structured the code with clear function categorization for better manageability.
  - Tested thoroughly to balance AI performance and responsiveness.

6. Conclusion

- **Project Summary:** Successfully developed a functional Tic Tac Toe game featuring both heuristic-based Normal Mode and optimal Unbeatable Mode AI strategies.
- **Learning Outcomes:** Gained valuable experience in GUI development using tkinter, implemented AI algorithms including heuristic methods and the Minimax algorithm with memoization, and learned to seamlessly integrate the interface with game logic.
- **Overall Experience:** The project was a rewarding success, overcoming challenges related to AI optimization and interface design, leading to personal growth in programming and problem-solving skills.

7. Any potential issues

- **Enhancements**:
    - Implementing additional features like undoing moves or replaying games.
    - Expanding the game to larger grids (e.g., 4×4) with more complex AI.
- **Optimization**:
    - Improving the GUI with better graphics or animations