UNBEATABLE AI Tic-Tac-Toe

By Joydeep Dutta

25BCY10002

For VITyarthi

Course: Fundamentals in AI & ML

Faculty: Satyabrat Nath

Slot: B14+B23+D21

Introduction

Artificial Intelligence (AI) often deals with making optimal decisions in uncertain or competitive environments. This project explores the domain of Game Theory and Adversarial Search by implementing an intelligent agent capable of playing the zero-sum game of Tic-Tac-Toe. Unlike simple rule-based bots that play randomly, this agent utilizes the **Minimax Algorithm**—a recursive search strategy that evaluates all possible future game states to determine the mathematically perfect move. The goal is to create an agent that is impossible to defeat.

Problem Statement

Core Problem: In competitive environments, standard search algorithms (like BFS/DFS) fail because they do not account for an opponent who is actively trying to minimize the agent's score. **Specific Goal:** To design and implement a software agent that plays Tic-Tac-Toe against a human opponent. The agent must evaluate the "Game Tree" to ensure it never loses. It must either win or force a draw, regardless of the human player's strategy.

Functional Requirements

The system delivers the following core features:

1. **Game Board Interface:** A clear console-based display of the 3x3 grid that updates after every move.

2. **Move Validation:** Logic to check if a user’s move is within bounds (0-8) and on an empty space.
3. **Adversarial AI Agent:** An implementation of the Minimax algorithm that plays as the 'maximizing' player.
4. **Win/Loss Detection:** Automatic detection of winning combinations (rows, columns, diagonals) or draw conditions.

Non-functional Requirements

1. **Performance:** The AI must calculate and execute its move in under 2 seconds.
2. **Reliability:** The system must not crash upon receiving invalid input (e.g., letters or numbers > 8); it should prompt the user to try again.
3. **Usability:** The interface must be simple to understand, using standard numpad mapping (0-8).
4. **Logging:** The system must persistently store the result of every game (Win/Loss/Draw) in a text file (game_history.txt) for audit purposes.

System Architecture

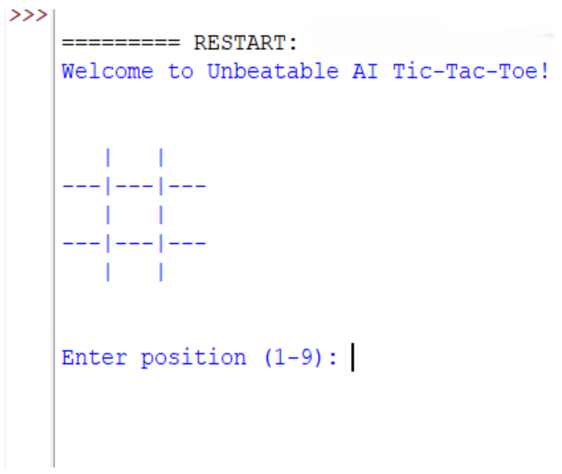
The system follows a modular architecture:

- **Controller (main.py):** Manages the flow of the game, switching turns between Human and AI.

- **UI Module (board.py):** Handles printing the grid to the console.
- **Logic Module (game_rules.py):** Contains the "physics" of the game (checking wins/draws).
- **Intelligence Module (ai_agent.py):** Contains the Minimax recursion logic.
- **Storage (logger.py):** Handles file I/O for game history.

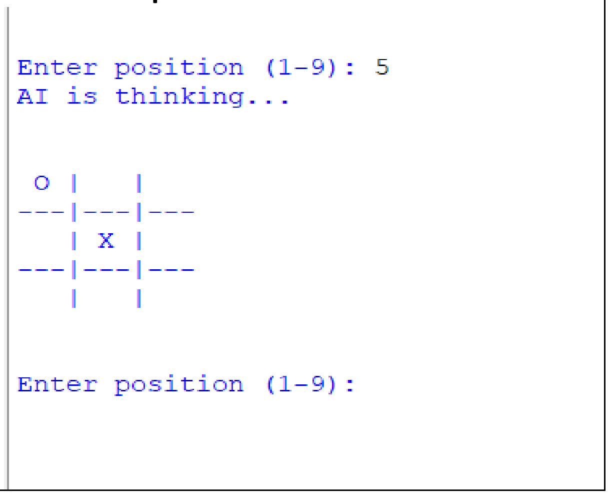
Screenshots / Results

1. Interface



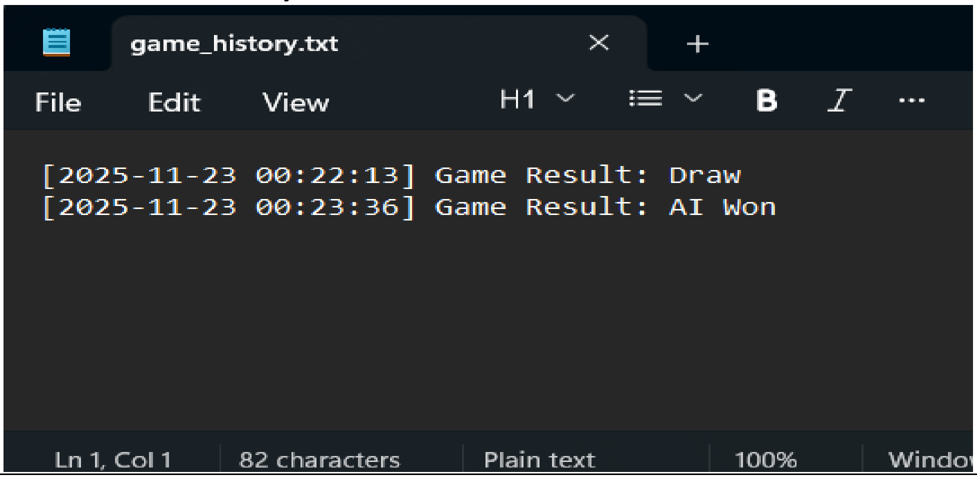
```
>>>
========= RESTART:
Welcome to Unbeatable AI Tic-Tac-Toe!

   |   |
---|---|---
   |   |
---|---|---
   |   |

Enter position (1-9):
```

2.Enter position to cross



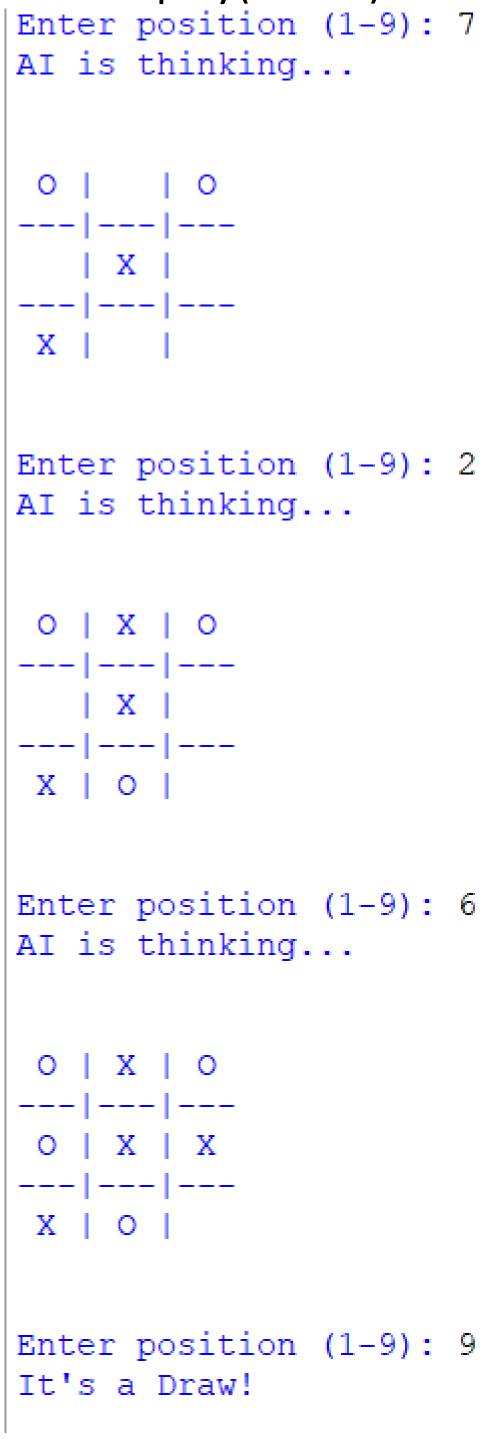
```
Enter position (1-9): 5
AI is thinking...

 O |   |
---|---|---
   | X |
---|---|---
   |   |

Enter position (1-9):
```

3.Game history



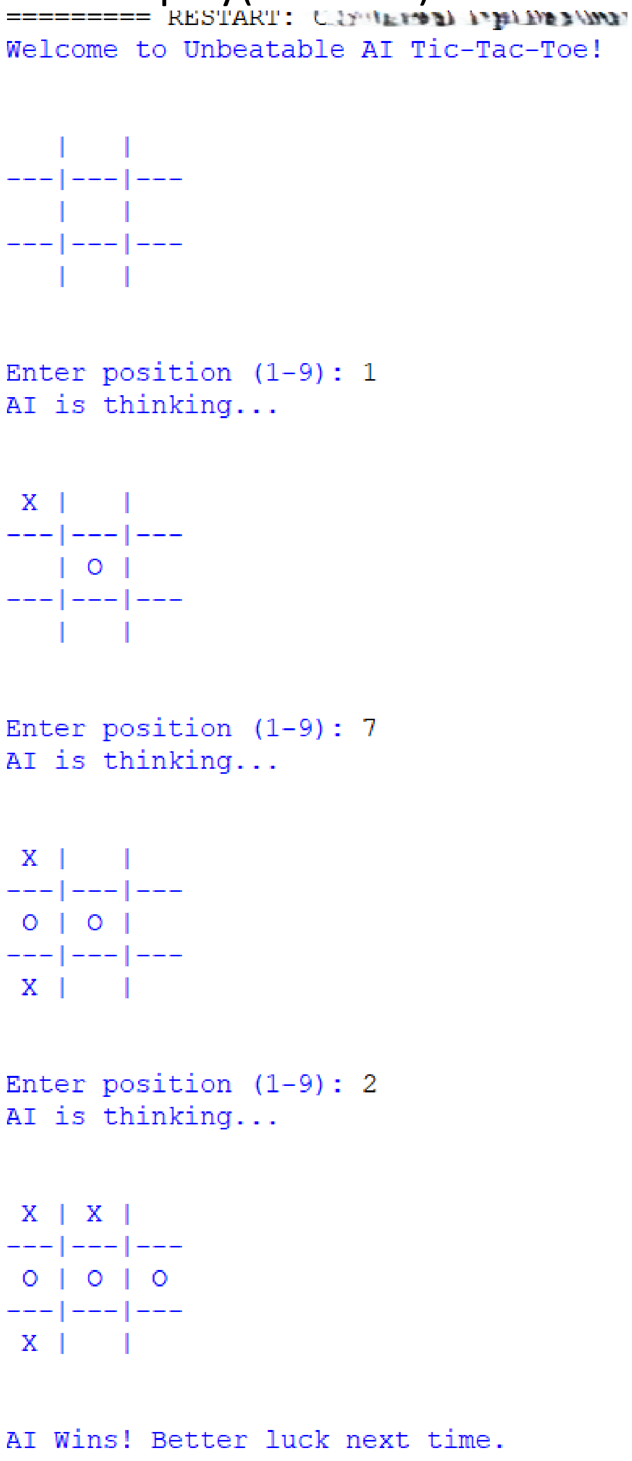
```
[2025-11-23 00:22:13] Game Result: Draw
[2025-11-23 00:23:36] Game Result: AI Won
```

4.Gameplay(Draw!)



```
Enter position (1-9): 7
AI is thinking...


 O |   | O
---|---|---
   | X |
---|---|---
 X |   |


Enter position (1-9): 2
AI is thinking...


 O | X | O
---|---|---
   | X |
---|---|---
 X | O |


Enter position (1-9): 6
AI is thinking...


 O | X | O
---|---|---
 O | X | X
---|---|---
 X | O |


Enter position (1-9): 9
It's a Draw!
```

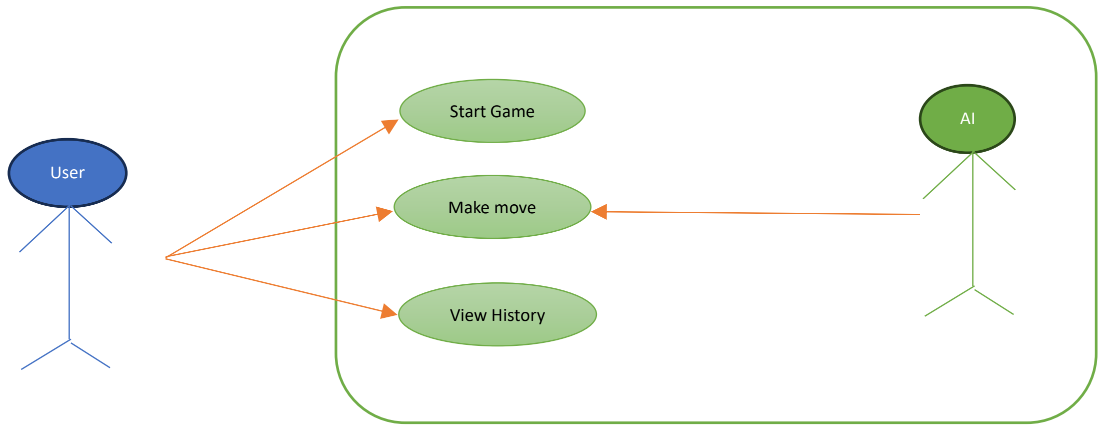
5.Gameplay(AI Wins!)



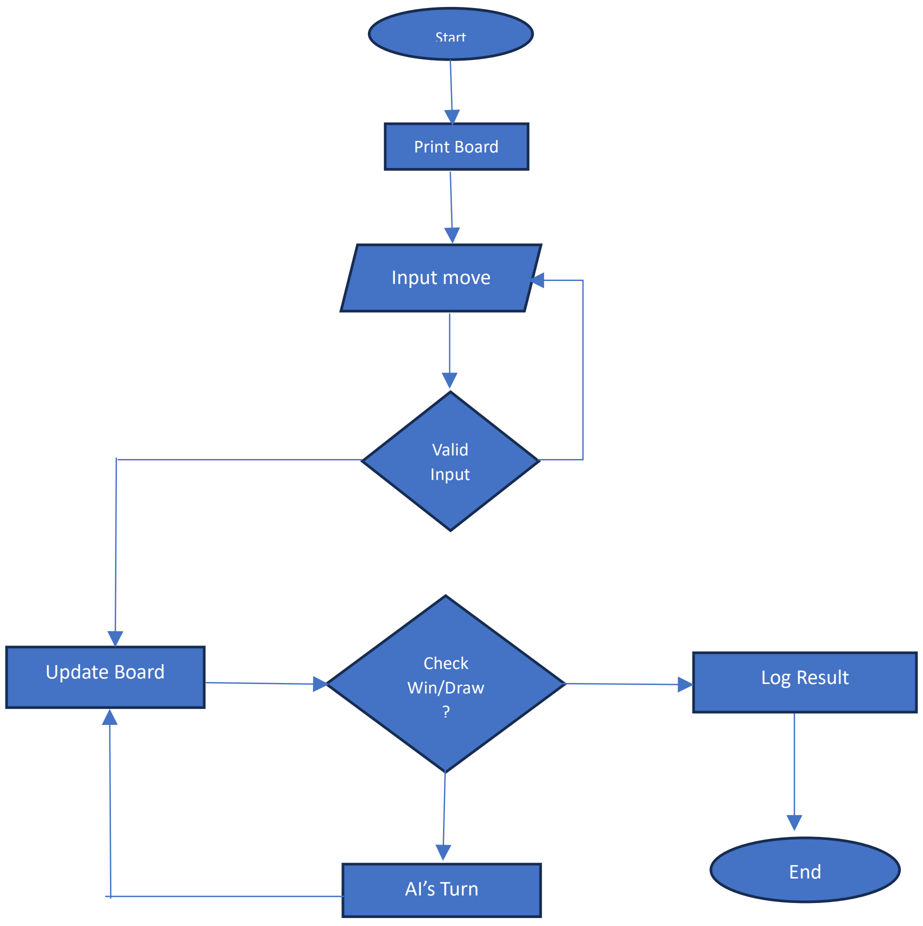
```
========== RESTART: [illegible]
Welcome to Unbeatable AI Tic-Tac-Toe!


   |   |
---|---|---
   |   |
---|---|---
   |   |


Enter position (1-9): 1
AI is thinking...


 X |   |
---|---|---
   | O |
---|---|---
   |   |


Enter position (1-9): 7
AI is thinking...


 X |   |
---|---|---
 O | O |
---|---|---
 X |   |


Enter position (1-9): 2
AI is thinking...


 X | X |
---|---|---
 O | O | O
---|---|---
 X |   |


AI Wins! Better luck next time.
```

Design Diagrams

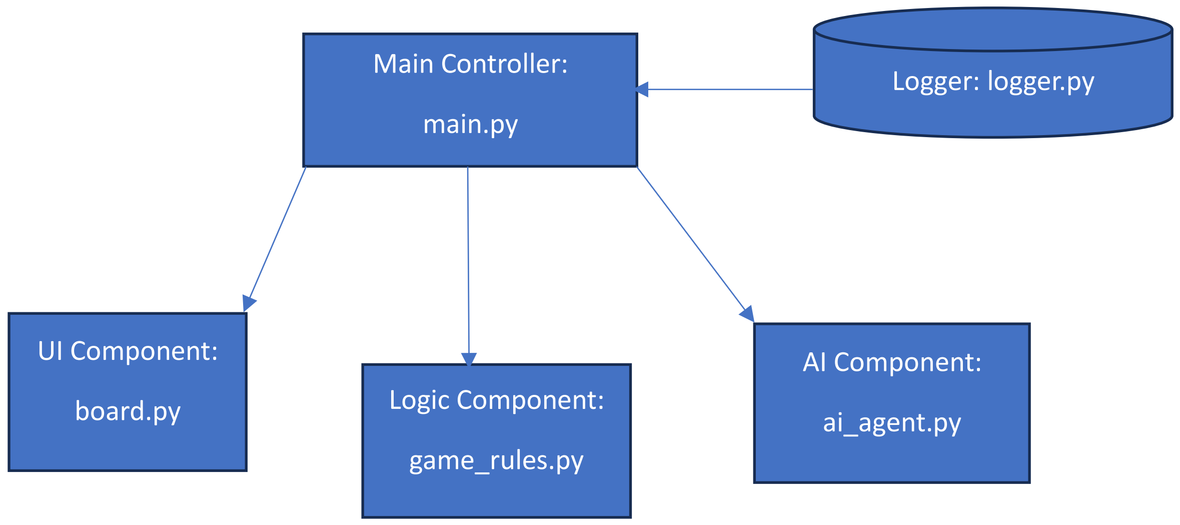
1. Use case Diagram:



2. Workflow Diagram:



3. Component Diagram:



Design Decisions & Rationale

- **Why Minimax?**
    - *Decision:* We chose Minimax over Reinforcement Learning or Random selection.
    - *Rationale:* Tic-Tac-Toe has a small "state space" (only 255,168 unique games). Minimax can traverse this entire tree instantly, guaranteeing a mathematically perfect result without the need for training data.
- **Why Python?**
    - *Decision:* The project is built in Python.
    - *Rationale:* Python's list handling makes grid manipulation easy, and it aligns with the lab requirements for AI implementation.

Implementation Details

The core logic resides in the minimax function inside ai_agent.py.

- **Recursion:** The function calls itself, alternating between maximizing the AI's score and minimizing the Human's score.
- **Scoring:** +1 for AI win, -1 for Human win, 0 for Draw.
- **Code Snippet:**

```
def minimax(board, depth, is_maximizing):
    if check_win(board, 'O'): return 1
    if check_win(board, 'X'): return -1
    if check_draw(board): return 0
    #recursive logic continues ...
```

Testing Approach

We used Black-box testing strategies:

- **Valid Inputs:** Tested entering numbers 0-8. Result: Success.
- **Boundary Testing:** Tested entering 0 and 8. Result: Success.
- **Invalid Inputs:** Tested entering "A", "9", and "-1". Result: System caught the error and requested new input (Pass).
- **Logic Testing:** Played 10 games trying to trick the AI using "fork" strategies. Result: AI blocked all attempts (10 Draws, 0 Losses).

Challenges Faced

1. **Recursion Depth:** Initially, it was difficult to visualize how the recursion stack worked. Debugging the "depth" variable helped understand the flow.
2. **Input Indexing:** Python lists are 0-indexed (0-8), but humans intuitively count 1-9. Translating this required careful subtraction in the input handler to avoid IndexError.

Learnings & Key Takeaways

- **Understanding State Space:** I learned that even simple games have thousands of possibilities, and computers "solve" them by looking at every single one.
- **Adversarial Search:** I gained practical experience with the Minimax algorithm, a foundational concept in AI for chess and go engines.
- **Modular Coding:** Breaking the code into 5 files made debugging significantly easier than writing one giant script.

Future Enhancements

1. **Alpha-Beta Pruning:** To optimize the algorithm for larger grids (like 4x4 or 5x5) where standard Minimax would be too slow.
2. **GUI:** Implementing a graphical interface using PyGame or Tkinter instead of the command line.
3. **Difficulty Levels:** Adding a "Easy" mode where the AI makes random errors occasionally.

References

1. Russell, S., & Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall.
2. CSA2001 Course Material, VIT Bhopal.
3. Python Documentation (docs.python.org).