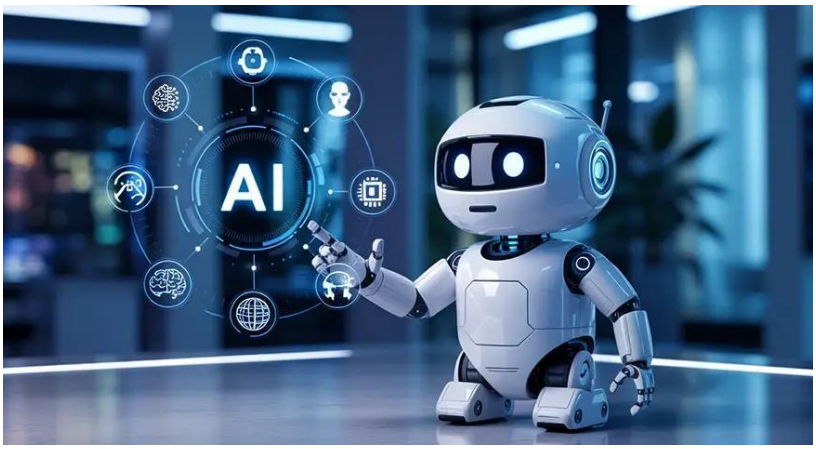
Tic_Tac_Toe
AI Agent Game Report



Name : L.M.R. Perera

Table of Contents

1.List of Figures 3
2.Abstract 4
3.Introduction 5
4.Background and Literature Review 6
4.1. Game Playing Agents 6
4.2. Minimax Algorithm 6
4.3. Alpha Beta Pruning 7
5.PEAS analysis 9
6.Methodology 10
6.1. Algorithm justification 10
7.Implementation 11
7.1. Board Representation 11
7.2. AI Decision Making 11
7.3. Game Statistics Tracking 12
7.3. Help Feature 13
8.Evaluation and results 14
8.1. Observations 15
9.Critical Analysis 16
10.Discussion 17
10.1. Strengths 17
10.2. Limitation 17
10.3. Future Improvements 17
11.Conclusion 18
12. References 19

1.List of Figures

Fiuger_1: Main menu Interface

Fiuger_2: Game Board Representation

Figure_3: Minimax function implementation with Alpha-Beta

Figure_4: Board class and winner detection

Figure_5: Minimax function with Alpha Beta pruning.

Figure_ 6: Tracking move times and statics.

Figure_7: Help feature logic

Fiuger_8: Level 1 Summery and Performance

Fiuger_9: Level 2 Summery and Performance

Fiuger_10: Level 3 Summery and Performance

Figure_11: Performance Summary

Table_01: -summery of PEAS

Table_02: Result Table

2.Abstract

A multi-level Tic-Tac-Toe game playing agent was designed and developed using Python and Pygame. The agent uses the Minimax algorithm together with Alpha beta pruning to make intelligent and efficient decisions. Classical Al search techniques are applied to simulate rational gameplay and to evaluate performance within a controlled environment. The system offers three difficulty levels,

- 3*3 Board- Level 1
- 4*4 Board- level 2
- 5*5 Board- level 3

Each with its own search depth, allowing the AI to adjust to increasing complexity. A help function lets the player press “H” key up to five times to receive AI generated suggestions during gameplay. Even move whether made by humans or the AI is recorded and analyzed for time taken and accuracy providing clear performance metrics.

Results show that the AI makes fast and accurate decisions across all levels, largely because Alpha Beta pruning removes unnecessary calculations. The help option improves the player’s performance and interaction showing a cooperative side of an AI behavior.

Overall, the system represents a well-balanced intelligence agent that demonstrates both competitive and supportive capabilities offering useful insights into reasoning efficiency and scalability in artificial intelligence.

3.Introduction

Artificial intelligence aims to build systems that can reason, learn and solve problems in ways like human thinking. Game playing agents are an effective way to study these behaviors because they work in structured environments with clear goals and measurable outcomes. Tic-tac-Toe, although a simple game, is ideal for testing decision making algorithms because it has a small state space and easy to understand rules.

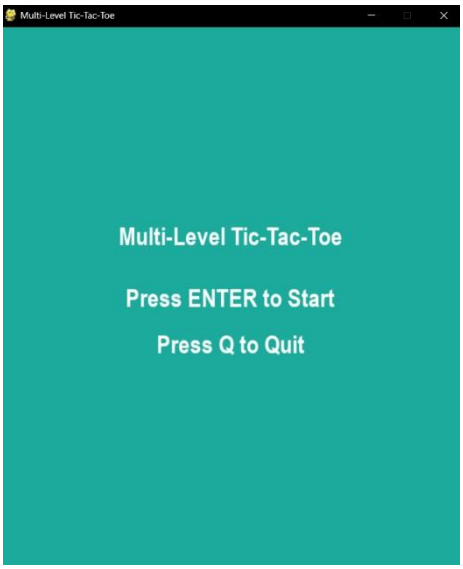
The developed system is a Tic-Tac-Toe AI agent that applies Minimax algorithms combined with Alpha-Beta pruning to play intelligently. The agent completes against a human player through three levels of difficulty,

1. Level 1- 3*3 Board
2. Level 2- 4*4 Board
3. Level 3- 5*5 Board

With each level adding more complexity and requiring deeper reasoning.

A key feature is the built in help option which lets the player press a key to request AI suggestions during the game. This feature changes the agent from being only an opponent to also acting as a supportive guide helping user make better decisions. During play the system tracks move times, total moves and accuracy to fairly evaluate both human and AI performance.

The following section explains the theoretical background, design choices and evaluation of the system, along with a reflection on its performance, challenges and future improvements.



Fiuger_1: Main menu Interface

4.Background and Literature Review

4.1. Game Playing Agents

Game playing agents are AI programs designed to make optimal decisions within a defined environment by predicting future outcomes of possible moves. They typically operate in deterministic, fully observable and turn based settings. In such environments, both players have complete information about the current game state, and no randomness affects the outcome.

Tic-Tac-Toe is a good example of this kind of environment.

The game has a simple rule, a small playing area and both players can always see the full board. This makes it ideal for testing how AI agents think and plan their next move.

In this project the agent analyses the board current configuration and calculates all possible future states to select the move that maximizes its winning potential. This type of reasoning forms the foundation for many early AI systems including chess playing programs like IBM Deep Blue. The Tic-tac-Toe agent therefore represents a scaled down but conceptually complete example of intelligent decision-making system.

Fiuger_2: Game Board Representation

4.2. Minimax Algorithm

The Minimax algorithm is a decision-making process used for two player zero- sum games where one player's is equal to the other's loss. It assumes that both players act rationally and always

make the best possible move. The algorithm generates a search three of all potential moves until research terminal states such as a win, lose or draw.

Each state is evaluated and given a score of +1 for a win, -1 for a loss and 0 for a draw. These values are propagated upward to determine the best possible action at the current state.

In this project the human player acts as the maximizer (aiming for the highest score) and the AI acts as the minimizer (trying to reduce the score). This alternating logic is implemented recursively in the minimax () function within the AI class of the program.

```
    return best
def minimax(self, board, win_len, depth, maximizing, alpha, beta):
    winner = board.check_winner(win_len)
    if winner == 1: return 1
    if winner == 2: return -1
    if board.full(): return 0
    if self.depth_limit and depth >= self.depth_limit: return 0
    if maximizing:
        value = -math.inf
        for (r, c) in board.empty_squares():
            nb = Board(board.size)
            nb.squares = board.squares.copy()
            nb.marked = board.marked
            nb.mark(r, c, 1)
            value = max(value, self.minimax(nb, win_len, depth+1, False, alpha, beta))
            alpha = max(alpha, value)
            if alpha >= beta: break
        return value
    else:
        value = math.inf
        for (r, c) in board.empty_squares():
            nb = Board(board.size)
            nb.squares = board.squares.copy()
            nb.marked = board.marked
            nb.mark(r, c, self.player)
            value = min(value, self.minimax(nb, win_len, depth+1, True, alpha, beta))
            beta = min(beta, value)
            if alpha >= beta: break
        return value
```

Figure_3: Minimax function implementation with Alpha-Beta

4.3. Alpha Beta Pruning

Alpha-Beta pruning is an optimization technique that enhances Minimax performance without changing its outcome. It introduces two threshold variables,

1. Alpha: - The best value the maximiser can achieve so far
2. Beta: - The best value the minimizer can achieve

If during the search the current branch cannot possibly produce a better result than previously explored branches the algorithm stops exploring, it. This process is known as pruning.

In this project Alpha beta pruning is critical for making higher levels playable in real time. It reduces redundant calculations, allowing the AI to maintain competitive response times. This efficiency makes the algorithm suitable even for slightly larger board configurations while preserving optimal decisions.

5.PEAS analysis

The PEAS model is used to describe how an intelligent agent interacts with its surroundings. It helps to clearly define what the agent is trying to achieve, where it operates and how it senses and acts within that environment.

In Tic-Tac-Toe game the agent's main goal is to win the game or avoid losing by choosing the best possible moves. The environment is a simple fully observable game board where both the AI and the human player can see all available moves. The agent sensors read the current board state and detect human moves while its actuators mark the AI chosen positions and update the visual display. Performance is measured through metrics such as win rate, move accuracy and decision time.

This analysis shows that the system behaves a goal based rational agent since it makes decisions that maximize performance based on the current environment.

Table_01: -summery of PEAS

Component	**Description for Tic-Tac-Toe Agent**
Performance Measure	Number of wins, Draws, Move accuracy and Average move time
Environment	A grid-based board environment (3*3, 4*4, 5*5)
Actuators	Place X or O in an empty square, generating help suggestions and updating the game display
Sensors	Read board state, Checking win/ draw conditions

6.Methodology

The project follows an object-oriented approach using Python with three main components.

- The board class
- The AI class
- Main game loop

The board handles the grid and validates moves. The AI performs decision making using Minimax and Alpha-Beta pruning. Finally, the main game loop manages gameplay flow user inputs and display outputs.

To ensure progressive complexity three levels of difficulty were introduced:

1. Level 1 uses full depth Minimax search, ensuring perfect play
2. level 2 introduces a depth limit of 4 to balanced accuracy and performance
3. level 3 uses a shallower search depth to maintain real time processing.

Additionally, a help system was implemented. Triggered by pressing “H” key on keyboard. When activated it generates a visual suggestion on the board showing where the AI should place their next move.

6.1. Algorithm justification

The Minimax algorithm was selected for this project because it provides guaranteed optimal play for deterministic games like Tic-Tac-Toe. If requires no training data and can handle every possible board configuration through exhaustive search.

Finally, Minimax is a good choice because it is easy to understand and explain. It clearly shows how an AI can “think ahead” by simulating all possible outcomes of a game before making move This not only helps the AI play perfectly but also makes it an excellent example for demonstrating key AI concepts such as decision making, adversarial search and recursion.

7.Implementation

The system was implemented in Python using Pygame library to manage graphics and user interaction and NumPy to handle the game board as a matrix.

7.1. Board Representation

Show how I defend the game board and check for winner.

```python
class Board:
    def __init__(self, size):
        self.size = size
        self.squares = np.zeros((size, size), dtype=int)
        self.marked = 0

    def mark(self, r, c, player):
        if self.squares[r, c] == 0:
            self.squares[r, c] = player
            self.marked += 1
            return True
        return False

    def empty_sqr(self, r, c):
        return self.squares[r, c] == 0

    def empty_squares(self):
        return [(r, c) for r in range(self.size) for c in range(self.size) if self.empty_sqr(r, c)]

    def full(self):
        return self.marked == self.size * self.size

    def check_winner(self, win_len):
        S, n, w = self.squares, self.size, win_len
        for r in range(n):
            for c in range(n - w + 1):
                seg = S[r, c:c + w]
                if seg[0] != 0 and np.all(seg == seg[0]):
                    return int(seg[0])
        for c in range(n):
            for r in range(n - w + 1):
                seg = S[r:r + w, c]
                if seg[0] != 0 and np.all(seg == seg[0]):
                    return int(seg[0])
        for r in range(n - w + 1):
            for c in range(n - w + 1):
                d1 = [S[r + i, c + i] for i in range(w)]
                d2 = [S[r + w - 1 - i, c + i] for i in range(w)]
                if d1[0] != 0 and all(x == d1[0] for x in d1):
                    return int(d1[0])
                if d2[0] != 0 and all(x == d2[0] for x in d2):
                    return int(d2[0])
        return 0
```

Figure_4: Board class and winner detection

7.2. AI Decision Making

Implements the Minimax algorithm with Alpha- Beta pruning logic. Which simulates all possible moves and outcomes to choose the optimal one.

```python
def minimax(self, board, win_len, depth, maximizing, alpha, beta):
    winner = board.check_winner(win_len)
    if winner == 1:
        return 1
    if winner == 2:
        return -1
    if board.full():
        return 0
    if self.depth_limit and depth >= self.depth_limit:
        return 0

    if maximizing:
        value = -math.inf
        for (r, c) in board.empty_squares():
            nb = Board(board.size)
            nb.squares = board.squares.copy()
            nb.marked = board.marked
            nb.mark(r, c, 1)
            value = max(value, self.minimax(nb, win_len, depth + 1, False, alpha, beta))
            alpha = max(alpha, value)
            if alpha >= beta:
                break
        return value
    else:
        value = math.inf
        for (r, c) in board.empty_squares():
            nb = Board(board.size)
            nb.squares = board.squares.copy()
            nb.marked = board.marked
            nb.mark(r, c, self.player)
            value = min(value, self.minimax(nb, win_len, depth + 1, True, alpha, beta))
            beta = min(beta, value)
            if alpha >= beta:
                break
        return value
```

Figure_5: Minimax function with Alpha Beta pruning.

7.3. Game Statistics Tracking

Show how I measured move times and store data for evaluation.

```python
if running and not human_turn:
    pygame.time.delay(200)
    start_ai = time.time()
    mv = ai.best_move(board, lvl["win"])
    end_ai = time.time()
    if mv:
        board.mark(mv[0], mv[1], 2)
        ai_moves += 1
    ai_times.append((end_ai - start_ai) * 1000)
```

Figure_ 6: Tracking move times and statics.

7.3. Help Feature

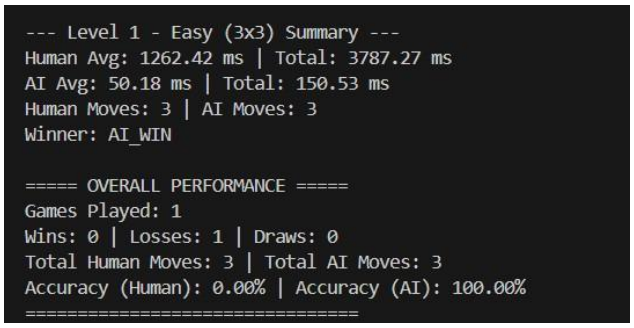
```
            pygame.quit(); sys.exit()

        # Help feature
        if e.type == pygame.KEYDOWN and e.key == pygame.K_h and human_turn:
            if help_remaining > 0:
                mv = ai.best_move(board, lvl["win"])
                help_position = mv
                help_remaining -= 1
```

Figure_7: Help feature logic

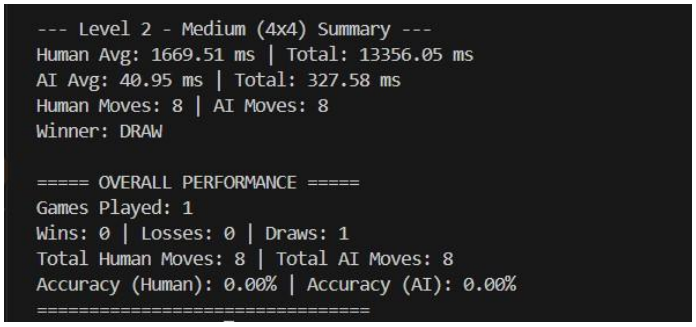
8.Evaluation and results

The following table presents the system's performance across all three levels. *Table_02: Result Table*

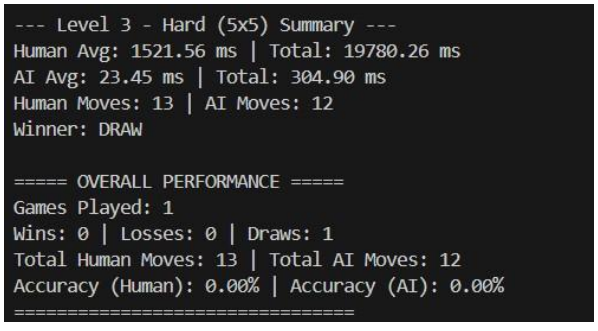
Level	Board Size	Human Avg(ms)	AI Avg(ms)	Human moves	AI moves	Winner	Help uses
1	3*3	1262.42	50.18	3	3	AI	0
2	4*4	1669.51	40.95	8	8	Draw	3
3	5*5	1521.56	23.45	13	12	Draw	2



Fiuger_8: Level 1 Summery and Performance



Fiuger_9: Level 2 Summery and Performance



Fiuger_10: Level 3 Summery and Performance

The AI made faster decisions than the human player. As board size increased both human and AI move time due to larger state space. However, Alpha-Beta pruning successfully maintained real-time playability. The human player's performance improved when the help system was used demonstrating that cooperative AI can enhance user learning.

8.1. Observations

The AI never lost. Human wins were rare and only possible if the AI was intentionally weak. Most games ended in draws reflecting Minmax's optimal strategy.

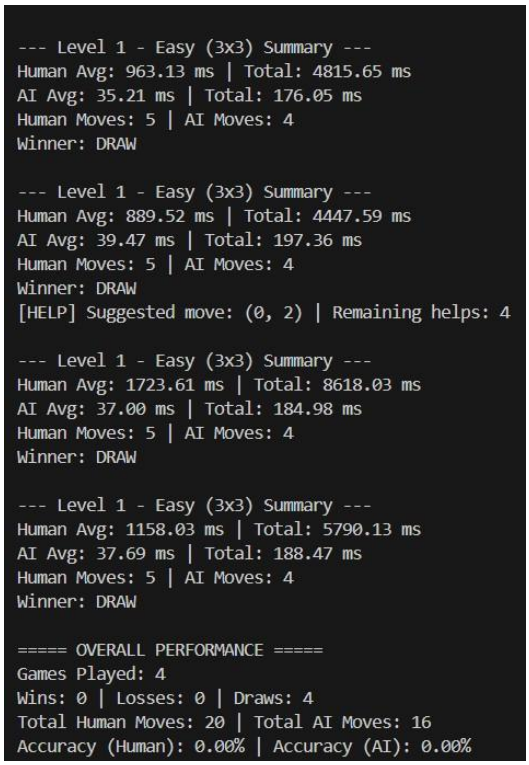


Figure_11: Performance Summary

9.Critical Analysis

The Tic-Tac-toe Ai works well and shows how an intelligent system can make smart moves by thinking ahead. The Minimax algorithm helps the AI plan its next move by checking all possible outcomes and choosing the best one. Using Alpha-Beta pruning makes this process faster because it skips moves that do not affect the result.

This helps the AI respond quickly and plan smoothly. Even though the system works well it has some limits. When the board size gets bigger the AI becomes slower because it has to check too many moves. This makes it hard for the Minimax algorithm to handle larger games. Adding heuristic function could make it faster by letting the AI estimate good moves instead of checking every option.

The help feature makes the game more fun and friendly. It helps players when they get suck and shows that AI can also guide not just complete. The graphics made with the Pygame are simple but clear and easy to understand. The print result like move time and accuracy show how both human and AI perform.

Overall, the system proves that classical AI methods can still be useful and understandable today.

10.Discussion

The project demonstrates that even in simple environments, adversarial search algorithms can guarantee unbeatable performance. Minimax performs perfectly in Tic-Tac-Toe because the state space is small. The inclusion of the help feature made gameplay more interactive and supported users in making strategic decisions.

10.1. Strengths

1. The AI users clear and easy to understand logic.
2. Minimax with Alpha-Beta pruning works well and gives quick decisions.
3. The game is interactive and fun to play
4. The system shows how classic AI methods can still be useful today.
5. The Ai always plays in a logical and fair way.

10.2. Limitation

1. The Minimax algorithm becomes slow on bigger boards like 6*6.
2. The AI cannot learn from past games or change its strategy.
3. The graphics and interface are quite simple.
4. The game only works for one human and one AI player
5. Players must restart or go back manually after finishing a level

10.3. Future Improvements

There are several ways to make this Tic-Tac-Toe system better in the future. One improvement would be to add a learning feature so the AI can study how the player moves and change its strategy over time. Using reinforce3ment learning could make the AI more adaptive and realistic. The graphics and interface could also be improved with animations, sound effects and better visual design to make the game more enjoyable. A scoreboard could be added to keep track performance across multiple sessions.

Finally adding multiplayer or online play would make the system more interactive allowing two human players or multiple AI agents to compete.

These improvements would make the system smarter, faster and more engaging for users.

11.Conclusion

This project successfully developed a functional multi-level Tic-Tac-Toe AI agent using the Minimax algorithm with Alpha-Beta pruning. The system demonstrated intelligent decision making, efficiency through pruning and meaningful human AI interaction through us help feature. The evaluation results confirmed that the AI made faster and more accurate decisions than the human player while maintaining fairness and engagement.

12. References

[1] *Pygame Documentation*. [Online]. Available: https://www.pygame.org/docs/. [Accessed: Sep. 20, 2025].

[2] M. Berman, "Deep Blue Beats Kasparov in Chess," EBSCO Research Starters. [Online]. Available: https://www.ebsco.com/research-starters/sports-and-leisure/deep-blue-beatskasparov-chess. [Accessed: Sep. 20, 2025].

[3] "Minimax Algorithm in Game Theory | Set 4 (Alpha-Beta Pruning)," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/dsa/minimax-algorithm-in-gametheory-set-4-alpha-beta-pruning/. [Accessed: Sep. 20, 2025].

[4] AlejoG10/python-tictactoe-ai-yt: Tic-Tac-Toe played by an unbeatable Artificial Intelligence using Python and Pygame. GitHub. [Online]. Available: https://github.com/AlejoG10/python-tictactoe-ai-yt. [Accessed: Sep. 20, 2025].