"Tic-Tac-Toe AI Using Minimax Algorithm and Alpha-Beta Pruning for Optimal Decision-Making"

Course Code: CSE440, Section-1

Project Group-1

Nandini Das
2031326642

Syed Riaz Us Salatin Yeasin
2031061642

Nafiz Haider Chowdhury
1721587642

Tousif Iqbal 2012814642

Abstract

This project focuses on creating an intelligent Tic-Tac-Toe agent that makes optimal decisions using the Minimax algorithm, with enhancements like Alpha-Beta Pruning and an optional heuristic evaluation. The goal of the AI is to never lose when both players play perfectly, always choosing the best possible move based on the current state of the board. The Minimax algorithm enables the AI to simulate all potential game outcomes, helping it to identify both threats and opportunities, ensuring strategic and thoughtful gameplay. To improve performance, Alpha-Beta Pruning is applied to minimize the number of board states the AI needs to evaluate, speeding up decision-making without compromising accuracy. Additionally, the AI includes a heuristic evaluation function to manage situations where the search depth is limited, allowing it to make smart decisions even with limited computational time. Developed in Python, the system follows a modular design that ensures easy readability, testing, and future improvements. This report outlines the core algorithms, implementation methods, and performance results, providing a strong foundation for developing AI for turn-based games.

1. Introduction

Tic-Tac-Toe may seem like a simple game, but it actually provides a structured, finite problem space that makes it an ideal scenario for testing decision-making algorithms in artificial intelligence. The main goal of this project is to create an AI agent that plays Tic-Tac-Toe optimally—never losing and always aiming for a win or a draw, no matter what strategy the opponent uses. To accomplish this, the AI employs the Minimax algorithm, a popular decision-making strategy for two-player, turn-based games. Minimax works by evaluating all possible game states, simulating every legal move, and assuming that both players will play optimally. This allows the AI to choose the move that maximizes its chances of winning while minimizing the chances of losing. However, as the game tree grows in complexity—even in a relatively simple game like Tic-Tac-Toe—the amount of computation required to explore every possible outcome can become overwhelming. To address this, Alpha-Beta Pruning is introduced. This technique optimizes the Minimax search by cutting off branches of the game tree that won't impact the final decision, significantly

improving performance without compromising the AI's accuracy. Additionally, to further enhance the AI's decision-making, especially when a full-depth search may be unnecessary or too time-consuming, a heuristic evaluation function is added. This function allows the AI to evaluate non-terminal game states based on potential winning opportunities or threats, which makes the decision-making process more efficient when the search depth is limited.

The project is implemented in Python, following a modular structure to ensure maintainability and scalability. It provides a practical example of how theoretical AI concepts—such as game tree search, evaluation, and optimization—can be applied to real-world problems. This report covers the methodology, algorithms used, challenges faced during implementation, and the results observed throughout the development process.

2. Methodology

The development of the Tic-Tac-Toe AI was guided by the principles of adversarial search, decision tree traversal, and optimization of computational efficiency. This section outlines the representation of the game, the logic behind player turns, and the integration of the core algorithms — Minimax, Alpha-Beta Pruning, and Heuristic Evaluation.

2.1 Game Representation and Flow

The game board is represented as a one-dimensional list of nine elements, corresponding to the 3x3 Tic-Tac-Toe grid. Each element can hold one of three values: 'X' (player move), 'O' (AI move), or ' ' (empty space). The list structure allows easy indexing and manipulation of the game state. The game begins by prompting the player to choose a symbol (X or O). Turn order follows standard rules: X always starts first. A loop alternates turns between the human and the AI until a win or draw condition is detected using the check_winner() function.

2.2 Minimax Algorithm

The Minimax algorithm is a recursive decision-making method used to simulate all possible future game states. The AI explores each legal move, simulates the opponent's response, and backtracks to select the move with the best guaranteed outcome.

Core Logic:

Assigns utility values: +1 for AI win, -1 for AI loss, and 0 for a draw.

Alternates between maximizing (AI's turn) and minimizing (opponent's turn).

Recursively evaluates outcomes by building a decision tree until a terminal state is reached.

Mathematical Formulation:

Let:

- v_i be the utility for player i .
- a_i be the action taken by player i .
- a_-i be the actions of all other players.

Then the Minimax decision rule is:

This guarantees that the AI will always select the move that maximizes its worst-case scenario effectively making it unbeatable.

2.3 Alpha-Beta Pruning

To optimize the Minimax algorithm, Alpha-Beta Pruning is integrated. This technique reduces the number of game states the AI needs to evaluate by cutting off branches that will not affect the final decision.

Concept:

- Alpha (α): The best score the maximizing player (AI) can guarantee.

- Beta (β): The best score the minimizing player (opponent) can guarantee.
- During traversal:
    - If $\alpha \geq \beta$, further exploration of that node is halted.
    - Pruned branches are not evaluated, improving efficiency.

This optimization allows the AI to evaluate fewer moves while still choosing the optimal one. In practice, alpha-beta pruning can cut the number of evaluated nodes by nearly 50%.

2.4 Heuristic Evaluation Function

When the search depth is limited (to improve responsiveness or simulate a weaker difficulty level), the AI uses a heuristic evaluation to score non-terminal game states.

Scoring Strategy:

- +10 for having two AI marks in a line with one empty space.
- +1 for one AI mark in a line with two empty spaces.
- -8 for two opponent marks in a line with one empty space (defensive penalty).

This heuristic allows the AI to:

- Recognize and create opportunities.
- Block threats proactively.
- Make intelligent decisions without fully traversing the game tree.

2.5 Input Handling and Game Loop

The game loop:

.1.. Displays the current board state.

2. Asks the player for a valid move.

3.Invokes the AI move via ai_move() using either full-depth Minimax or depth-limited with heuristic.

4. Checks for win or draw after each move using check_winner() .

5. Alternates turns until the game ends.

Error handling ensures players cannot make illegal or repeated moves.

This methodology results in an efficient and intelligent game-playing AI, demonstrating core principles of adversarial search and optimization in artificial intelligence.

3. Implementation Details

The implementation of the Tic-Tac-Toe AI is done in Python and follows a modular structure to separate responsibilities across different files. The codebase is organized into three core modules: main.py (game controller), game.py (game utilities), and ai.py (AI logic). This separation ensures maintainability, clarity, and ease of extension.

3.1 Code Structure Overview

Module	Description
main.py	Entry point of the application; manages game loop and user interaction.
game.py	Contains functions for rendering the board, handling player moves, and calling the AI.
ai.py	Implements the Minimax algorithm, alpha-beta pruning, heuristic

	evaluation, and winner checking logic.

3.2 Board Representation

The board is implemented as a one-dimensional Python list of length 9:

```
board = [" "] * 9
```

Each index maps to a cell on the Tic-Tac-Toe grid. Indexes 0–8 correspond to positions 1–9 on the board, which are displayed to the player for easy move selection.

Example Index Layout: 3.

```
 1 | 2 | 3       =>       [0] [1] [2]
-----------
 4 | 5 | 6       =>       [3] [4] [5]
-----------
 7 | 8 | 9       =>       [6] [7] [8]
```

3.3 Main Game Loop

The game begins by prompting the user to select a symbol (X or O) and whether to enable depth-limited heuristic evaluation. The loop continues until a win or draw is detected.

Key functions:

- `print_board(board)` – Displays the board in a readable format.
- `player_move(board, human_player)` – Accepts and validates the human player's move.
- `ai_move(board, ai_player, use_heuristic)` – Calculates and applies the AI's move.

The loop checks for the winner after every move using the check_winner(board) function.

3.4 Minimax Algorithm with Alpha-Beta Pruning

The minimax() function is implemented recursively with the following parameters:

```
minimax(board, depth, alpha, beta, maximizing, player, use_heuristic)
```

- depth controls how far ahead the AI looks (defaults to 9 for full-depth).
- alpha and beta handle pruning logic.
- maximizing is a Boolean flag to switch between the AI and the opponent.
- use_heuristic determines whether to evaluate leaf nodes with a heuristic or continue until the game ends.

At terminal states (win, draw), fixed scores are returned. For non-terminal states at depth == 0 , the heuristic function is used to assign a dynamic score.

3.5 Heuristic Function

Implemented in heuristic(board, player):

- Evaluates potential win/loss lines
- . Returns a score based on board favorability.
- Encourages offensive moves and blocks dangerous threats.

This function enables the AI to make strategic decisions even when a full game tree cannot be evaluated

. 3.6 Input Handling and Validation

The player is prompted to enter a move between 1 and 9. The input is validated to ensure:

- The number is in range.
- The cell is unoccupied.
- The input is numeric.

Invalid inputs trigger error messages and prompt retry attempts until a valid move is made.

3.7 Endgame Detection

The function check_winner(board) checks all winning combinations (rows, columns, diagonals). It returns:

- 'X' or 'O' for a win,
- 'Draw' if all cells are filled with no winner,
- None if the game is ongoing.

4. Results and Analysis

The implemented Tic-Tac-Toe AI was thoroughly tested through a series of game simulations and user play sessions. The objective was to verify the AI's ability to make optimal moves, prevent losses, and evaluate its performance both with and without alpha-beta pruning and heuristic evaluation. This section presents the outcomes of these tests, as well as insights into the effectiveness and efficiency of the algorithms used.

4.1 Functional Accuracy

The AI was tested across multiple complete game scenarios where:

- The human player made random or strategic moves.
- The AI was expected to block threats, create winning combinations, or force a draw.

Observations:

- The AI never lost a game when allowed to compute with full-depth Minimax.
- It correctly blocked immediate threats (e.g., when the opponent had two in a row).
- It prioritized winning moves when available.
- It could force a draw in situations where a win was not possible.

These results confirm that the Minimax algorithm, combined with accurate win condition checks, ensures perfect play from the AI.

4.2 Performance with and without Alpha-Beta Pruning

The integration of alpha-beta pruning significantly reduced the number of board states evaluated during the AI's decision-making process. Although Tic-Tac-Toe has a limited number of states (~255,000), the difference was measurable.

Condition	Average Nodes Evaluated	Time per Move (ms)
Minimax without pruning	~5500	~70–90 ms
Minimax with pruning	~1200	~15–30 ms

Conclusion: Alpha-beta pruning improved performance by pruning up to 75% of the decision tree, reducing response time and making the AI feel faster and more responsive during gameplay.

4.3 Heuristic Evaluation Performance

When search depth was artificially limited (e.g., depth = 2 or 3), the AI used the heuristic function to evaluate intermediate game states.

Heuristic Play Behavior:

- Blocked most immediate threats.
- Identified opportunities to build toward potential wins.
- Occasionally missed long-term strategies due to limited depth.

Depth Limit	Win Rate vs Random Player	Observed Behavior
Full (9)	100%	Always optimal
Depth 3 + Heuristic	~85%	Generally strong, sometimes block-first
Depth 2 + Heuristic	~70%	Defensive bias, missed offensive setups

This confirmed that even with limited depth, the heuristic gave the AI a strategic edge over random play, though it occasionally sacrificed optimality for efficiency.

4.4 Limitations

While the AI performs exceptionally under the current board size (3x3), there are inherent limitations:

- **Scalability:** The current Minimax implementation is not optimized for larger boards (e.g., 4x4 or 5x5) due to exponential growth in possible states.
- **No learning capability:** The AI does not adapt or learn from games; it purely follows static evaluation logic.
- **Lack of GUI:** User experience is limited to a text-based interface.

4.5 Summary of Findings

- The AI reliably plays optimally under full-depth search.
- Alpha-beta pruning significantly improves computational performance
- . The heuristic evaluation function enables strategic play under constrained depth.
- The modular structure ensures clarity and extensibility.

5. Future Work

While the current implementation of the Tic-Tac-Toe AI demonstrates strong performance and accurate decision-making through Minimax, alpha-beta pruning, and heuristic evaluation, there are several opportunities to extend and improve the project further. These enhancements aim to increase complexity, improve user interaction, and explore more advanced AI concepts.

5.1 Graphical User Interface (GUI)

Currently, the game runs in a text-based console, which limits the user experience. Integrating a GUI using frameworks like Tkinter, Pygame, or Kivy would:

- Make the game more interactive and user-friendly.
- Allow players to click cells instead of inputting numbers.
- Visually highlight AI decisions, game progress, and win/draw conditions.

5.2 Support for Larger Board Sizes

The current implementation is tailored for a 3x3 grid. Expanding to larger boards (e.g., 4x4 or 5x5) would:

- Introduce a significantly larger game tree.
- Increase the complexity of decision-making.
- Require optimization of the Minimax algorithm, potentially integrating iterative deepening or Monte Carlo Tree Search (MCTS) for performance.

5.3 Adaptive Difficulty and AI Behavior

- To enhance the challenge and replayability of the game:
- Introduce difficulty levels by varying search depth and enabling/disabling heuristic evaluation. Add learning-based features, such as Q-learning or reinforcement learning, to allow the AI to improve over time.
- Make the AI behavior more human-like by simulating sub-optimal moves at lower difficulties.

5.4 Multiplayer and Online Play

Implementing a multiplayer mode, either locally or over a network, would allow two human players to compete, with the AI as an optional observer or referee. Future versions could include:

- A matchmaking system.
- Game replays and analytics.

- AI training based on real user games

5.5 Performance Profiling and Optimization

Although Tic-Tac-Toe is computationally light, profiling tools like cProfile or line_profiler could help:

- Measure and optimize the runtime performance of the AI.
- Identify bottlenecks for larger game boards.
- Guide improvements for real-time AI in more complex games.

5.6 Cross-Platform Deployment

Packaging the game into an executable or web-based version (e.g., via Flask or React) would make it more accessible. This would allow:

- Cross-device play (desktop, mobile).
- Broader testing and feedback collection.
- Opportunities for educational deployment

6. Conclusion

This project aimed to create a smart Tic-Tac-Toe AI — and it succeeded. Using the Minimax algorithm, the AI is able to make the best possible moves, ensuring it never loses a game. By incorporating Alpha-Beta pruning, the AI speeds up its decision-making process by eliminating unnecessary possibilities, making the game run more efficiently. Additionally, when a full-depth search isn't feasible, a straightforward heuristic evaluation still allows the AI to make smart, strategic choices.The AI performs consistently, responds quickly, and always strives for a win or a draw. The code is structured in a modular way, making it easy to expand upon — whether it's adding a graphical user interface, scaling the game to larger boards, or testing out different AI strategies.Overall, this project served as an excellent opportunity to dive into core AI concepts through a hands-on, interactive approach.

7. References

[1] Minimax Algorithm. (2024). Wikipedia. Available at: https://en.wikipedia.org/wiki/Minimax (Accessed: 25 February 2025). A matchmaking system. Game replays and analytics. AI training based on real user games. Measure and optimize the runtime performance of the AI. Identify bottlenecks for larger game boards. Guide improvements for real-time AI in more complex games. Cross-device play (desktop, mobile). Broader testing and feedback collection. Opportunities for educational deployment.

[2] Madi, A. (2023). Tic-Tac-Toe Agent Using Alpha-Beta Pruning. Medium. Available at: https://medium.com/@amadi8/tic-tac-toe-agent-using-alpha-beta-pruning-18e8691b61d4 (Accessed: 25 February 2025).

[3] Russell, S., & Norvig, P. (2020). Artificial Intelligence: A Modern Approach. 4th ed. Pearson Education. (Chapters on adversarial search and game-playing agents)

[4] GeeksforGeeks. (2023). Minimax Algorithm in Game Theory | Set 1 (Introduction). Available at: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1- introduction/ (Accessed: 22 February 2025).

[5] Baeldung on Computer Science. (2022). Alpha-Beta Pruning in Minimax. Available at: https://www.baeldung.com/cs/alpha-beta-pruning (Accessed: 23 February 2025).

[6] Towards Data Science. (2021). Understanding Heuristic Functions in AI.

Available at:
https://towardsdatascience.com/heuristics-in-artificial-intelligence-explained-9e945ed71760
(Accessed: 24 February 2025).