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

Nandini Das
2031326642
Department of Computer Science and Engineering,
North South University
Dhaka, Bangladesh

Syed Riaz Us Salatin Yeasin
2031061642
Department of Computer Science and Engineering,
North South University
Dhaka, Bangladesh

Tousif Iqbal
2012814642
Department of Computer Science and Engineering,
North South University
Dhaka, Bangladesh

Nafiz Haider Chowdhury
1721587642
Department of Computer Science and Engineering,
North South University
Dhaka, Bangladesh

Abstract

This project focuses on building a smart Tic-Tac-Toe game where the computer opponent can play with near-perfect strategy. The AI uses the Minimax algorithm to evaluate all possible moves and pick the one that leads to the best outcome. To make the decision-making process faster and more efficient, alpha-beta pruning is applied, which helps avoid checking unnecessary moves that don't affect the final result. When the AI isn't allowed to search all the way to the end of the game (due to performance constraints), a simple heuristic is used to estimate which board positions are stronger or weaker. The goal is to make the AI capable of playing intelligently even with limited resources. The project is fully developed in Python and organized into clean, reusable modules, making it easy to maintain and expand in the future. This report covers the design choices, core algorithms, and results observed during testing, and offers a practical example of how AI techniques can be applied in classic turn-based games.

Keywords — Tic-Tac-Toe AI, Minimax Algorithm, Alpha-Beta Pruning, Search Space Optimization, Computational Complexity, Game Tree Search

1. Introduction

Tic-Tac-Toe may be a simple game on the surface, but it presents a clear and limited problem space that makes it ideal for

experimenting with decision-making strategies in artificial intelligence. The goal of this project is to build an AI that can play the game perfectly—always going for a win or, at the very least, securing a draw—no matter how the human opponent plays.

To accomplish this, the AI relies on the Minimax algorithm, a classic approach used in two-player, turn-based games. Minimax works by simulating every possible legal move in the game and predicting how the opponent might respond. It assigns scores to each outcome and chooses the path that gives the best possible result assuming both players make the best choices available to them.

Even though Tic-Tac-Toe has a relatively small number of possible board states, exhaustively evaluating all of them can still be inefficient, especially as the game progresses. To speed things up, the project uses Alpha-Beta Pruning, which helps cut down the number of moves the AI needs to consider. This optimization skips over branches of the game tree that won't affect the final decision, improving performance without compromising the AI's accuracy.

To make the AI even more efficient—especially in cases where a full-depth search isn't practical—a heuristic evaluation function is used. Instead of exploring to the very end of the game, the heuristic gives an estimate of how good a board position is by considering opportunities for winning and the need to block threats.

The project is developed in Python and designed with modularity in mind, making it easier to test, maintain, and build upon. It offers a hands-on demonstration of how key AI concepts like game tree traversal, optimization, and evaluation come together in a working system. This report outlines the steps taken to build the AI, the algorithms behind it, and the outcomes of testing its performance.

2. Methodology

The Tic-Tac-Toe AI was developed with a focus on core principles of game-playing artificial intelligence, including adversarial search, decision tree exploration, and efficient computation. This section explains how the game is represented in code, how turns are managed between the player and the AI, and how key algorithms—Minimax, Alpha-Beta Pruning, and Heuristic Evaluation—work together to make strategic decisions during gameplay.

2.1 Game Representation and Flow

The game board is implemented as a simple one-dimensional list with nine elements, each representing a cell in the 3×3 grid. Every position in the list can hold either `'X'` for the player, `'O'` for the AI, or a blank space (`' '`) if the cell is empty. This format makes it easy to access and update specific positions on the board using basic indexing.

At the start of the game, the player is asked to choose a symbol—either `X` or `O`. Following the standard rules of Tic-Tac-Toe, `X` always goes first. The game then enters a loop where the player and AI take turns making their moves. After each turn, the program checks for a winner or a draw using the `check_winner()` function, and the loop continues until the game reaches a conclusion.

2.2 Minimax Algorithm

The Minimax algorithm is a recursive strategy used in two-player games to help the AI make optimal decisions by simulating future game states. In this project, the AI explores each valid move, assumes the opponent will respond with their best possible counter-move, and continues this process recursively to build a complete decision tree.

During this evaluation, the AI alternates between maximizing its score and minimizing the opponent's. Final game states are scored numerically—typically +1 for a win, -1 for a loss, and 0 for a draw. Once the full tree is evaluated, the algorithm backtracks and selects the move that offers the best outcome for the AI, assuming both players play optimally.

By following this process, the AI ensures it always chooses the move with the most favorable result. As a result, it plays perfectly—guaranteeing a win or at least a draw.

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:

$$v_i = max_{a_i} min_{a\_i}\left(v_i(a_i, a\_i)\right)$$

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 improve the efficiency of the Minimax algorithm, Alpha-Beta Pruning is applied. This technique cuts off branches of the game tree that cannot possibly affect the final decision, allowing the AI to avoid evaluating moves that are already worse than previously explored options. By keeping track of the best scores for both the AI and the opponent during the search, it significantly reduces the number of board states that need to be checked, making the algorithm faster without changing the outcome.

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 α ≥ β, 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—either to keep things fast or to make the gameplay feel less difficult—it uses a heuristic evaluation function to help decide its next move. Instead of calculating every possible outcome, the heuristic gives the AI a quick estimate of how good the current board looks, based on things like potential winning lines or immediate threats. This allows the AI to still make smart choices, even when it's working with limited information.

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 runs in a simple, turn-based loop that continues until someone wins or the game ends in a draw. Each round of the loop follows a clear set of steps:

1. The current state of the board is shown to the player.

2. The player is asked to make a move by choosing a position.

3. The AI takes its turn, using either full-depth Minimax or a faster, heuristic-based approach—depending on the selected settings.

4. After each move, the program checks whether there's a winner or if the board is full, indicating a draw.

5. The turn then switches to the other player, and the cycle repeats.

To make sure the game runs smoothly, the code includes error handling to catch invalid inputs—like choosing a cell that's already occupied or entering an out-of-range number. This approach keeps the game user-friendly while allowing the AI to demonstrate intelligent decision-making in a dynamic, interactive setting.

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 element in the list represents a cell on the 3×3 Tic-Tac-Toe grid. The list uses indexes from 0 to 8, which correspond to positions 1 through 9 as seen by the player. This layout makes it easy to update and check specific positions on the board during gameplay.

Example Index Layout:

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

3.3 Main Game Loop

At the start of the game, the player is asked to choose a symbol—either `X` or `O`—and decide whether to enable depth-limited search, which lets the AI use a faster but slightly less thorough decision-making strategy.

Once the game begins, it runs in a loop that continues until someone wins or the game ends in a draw. On each turn, either the player or the AI makes a move, and the board is updated and displayed.

Here are the key functions that keep the loop running smoothly:

- `print_board(board)` – Shows the current state of the board in a clean, readable format.
- `player_move(board, human_player)` – Gets the player's input and ensures the move is valid.
- `ai_move(board, ai_player, use_heuristic)` – Determines the AI's move using either the full-depth Minimax algorithm or a depth-limited version with heuristic support.

After each move, the game checks if there's a winner or if the board is full using the `check_winner(board)` function. The loop keeps alternating between the player and the AI until the game is over.

3.4 Minimax Algorithm with Alpha-Beta Pruning

The core of the AI's decision-making is the `minimax()` function, which is implemented recursively to explore possible future moves and determine the best one. It works by simulating every possible move for both the AI and the opponent, and evaluating the outcomes to choose the most favorable path.

Here's how the function is structured:

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

- **`depth`**: Controls how many moves ahead the AI will look. A value of 9 means it will search the entire game tree.
- **`alpha and beta`**: Used for pruning—these values help skip over moves that don't need to be evaluated because they won't affect the final decision.
- **`maximizing`**: A flag that tells the function whether it's the AI's turn (maximize the score) or the opponent's turn (minimize the score).
- **`use_heuristic`**: If set to `True`, the AI will use a heuristic to evaluate the board when it hits the depth limit, rather than searching to the end of the game.

When the algorithm reaches a terminal state (win, loss, or draw), it returns a fixed score: typically +1 for a win, -1 for a loss, and 0 for a draw. If it reaches the depth limit before the game ends, it uses the heuristic function to assign a score based on the current board situation. This lets the AI make smart decisions even when it can't simulate the entire game.

3.5 Heuristic Function

The heuristic logic is handled by the `heuristic(board, player)` function. It's designed to evaluate the current state of the board and give it a score based on how favorable it looks for the AI. Here's what it focuses on:

- Identifying lines where the AI is close to winning and rewarding those positions.
- Detecting threats from the opponent and applying penalties to encourage blocking moves.
- Scoring based on how many potential win paths are still open.

This function becomes especially useful when the AI isn't searching all the way to the end of the game. Instead of trying every possible move, it makes a well-informed guess about which moves are strongest based on the current situation. This keeps the AI playing strategically, even with limited information.

3.6 Input Handling and Validation

During the player's turn, they're asked to enter a number between 1 and 9 to choose a position on the board. The game performs several checks to make sure the input is valid:

- The number must be within the valid range (1–9).
- The selected cell must be empty. - The input must be a number, not a letter or symbol.

If the input doesn't meet these conditions, the game shows an appropriate error message and asks the player to try again. This ensures a smooth and user-friendly experience, preventing accidental or invalid moves from disrupting the game.

3.7 Endgame Detection

The `check_winner(board)` function is responsible for determining if the game has ended. It checks all possible winning combinations—rows, columns, and diagonals—to see if either player has achieved three in a row.

The function returns: - `'X'` or `'O'` if one of the players wins, - `'Draw'` if the board is full and there's no winner, - `None` if the game is still in progress.

This logic runs after every move to ensure the game ends immediately once a win or draw condition is met.

4. Results and Analysis

To evaluate how well the Tic-Tac-Toe AI performs, it was tested through multiple game simulations and hands-on play sessions. The main goal was to see if the AI could consistently make smart decisions, avoid losing, and adapt well whether using the full Minimax algorithm or the optimized versions with alpha-beta pruning and heuristic support. This section highlights what those tests revealed about the AI's accuracy, efficiency, and overall behavior during gameplay.

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

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

Although the AI performs well on the 3×3 board, there are several areas where the current implementation has limitations:

- **No Learning Capability**: The AI doesn't learn from past games or adjust its strategy. It follows the same logic every time, without any ability to adapt or improve based on experience.
- **Fixed Board Size**: The system is built specifically for a 3×3 grid. It doesn't support larger variations like 4×4 or 5×5, which would require major changes to the logic and more efficient algorithms to handle the increased complexity.
- **No Graphical Interface**: The game runs entirely in the terminal, which works fine for testing but lacks the interactivity and visual appeal of a graphical user interface.
- **Static Behavior**: The AI plays flawlessly in every game, which makes it highly predictable. Without adjusting heuristic depth or logic, there's no built-in way to vary difficulty levels.

5. Future Work

Although the current version of the Tic-Tac-Toe AI performs well and makes smart decisions using Minimax, alpha-beta pruning, and heuristics, there's still plenty of room to take the project further. Future improvements could make the game more interactive, challenging, and adaptable, while also opening the door to exploring more advanced AI techniques and larger game spaces.

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.

6. Conclusion

This project set out to create a smart and reliable Tic-Tac-Toe AI—and it successfully achieved that goal. By leveraging the Minimax algorithm, the AI is able to evaluate all possible outcomes and consistently make the best possible move. With alpha-beta pruning, it does so more efficiently, cutting out moves that don't need to be considered. And when full-depth search isn't ideal, a simple but effective heuristic allows it to keep playing strategically without missing key opportunities.

The AI consistently performs well, making fast and accurate decisions that guarantee a win or draw when used correctly. Thanks to its modular code structure, the project is easy to extend—whether it's adding a user interface, supporting bigger boards, or experimenting with new AI techniques.

Overall, this project was a great way to bring theoretical AI concepts to life in a fun and interactive setting. It provided a solid foundation for understanding decision-making, search optimization, and game strategy—all within a classic game that never gets old.

7. References

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Hoboken, NJ, USA: Pearson, 2020.

[2] D. Fudenberg and J. Tirole, *Game Theory*, 1st ed. Cambridge, MA, USA: MIT Press, 1991. [3] E. Rich, K. Knight, and S. B. Nair, *Artificial Intelligence*, 3rd ed. New York, NY, USA: McGraw-Hill, 2009.

[4] S. J. Isenberg, "Minimax and Alpha-Beta Pruning in Games," *Journal of Game Theory and Decision Making*, vol. 12, no. 2, pp. 65–72, Apr. 2018.

[5] R. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial Intelligence*, vol. 27, no. 1, pp. 97–109, Sep. 1985.

[6] L. V. Allis, "Searching for Solutions in Games and Artificial Intelligence," Ph.D. dissertation, Dept. Comput. Sci., Vrije Universiteit, Amsterdam, The Netherlands, 1994.

[7] M. Campbell, A. J. Hoane Jr., and F. Hsu, "Deep Blue," *Artificial Intelligence*, vol. 134, no. 1–2, pp. 57–83, Jan. 2002.

[8] C. Browne et al., "A Survey of Monte Carlo Tree Search Methods," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012.