Optimizing Game Tree Search In Tic-Tac-Toe AI Using Minimax Algorithm And Alpha Beta Pruning Heuristics

Miwan Sariana Saqib
Department of Electrical and Computer Engineering
North South University
Dhaka, Bangladesh
miwan.saqib@northsouth.edu

Md. Mubtasim Fuad
Department of Electrical and Computer Engineering
North South University
Dhaka, Bangladesh
mubtasim.fuad01@northsouth.edu

Redwan Hossain
Department of Electrical and Computer Engineering
North South University
Dhaka, Bangladesh
redwan.hossain21@northsouth.edu

MD Naimul Hasan Munna
Department of Electrical and Computer Engineering
North South University
Dhaka, Bangladesh
naimul.munna@northsouth.edu

Dr. Mohammad Shifat-E-Rabbi
Department of Electrical and Computer Engineering
North South University
Dhaka, Bangladesh
rabbi.mohammad@northsouth.edu

***Abstract*— This project focuses on building an intelligent Tic-Tac-Toe AI using the Minimax algorithm with Alpha-Beta Pruning to ensure optimal decision-making and fast performance. Implemented in Python, this AI is designed to be unbeatable if and when played correctly by the user 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 (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. Future work includes extending this system to larger game boards or introduce a graphical interface to improve user usability.**

Keywords— minimax, alpha-beta pruning, heuristics, adversarial games, search optimization.

I. Introduction

Tic-Tac-Toe is a simple yet strategic 3×3 grid game often used in AI to study decision-making and game tree algorithms due to its manageable complexity. This project develops an AI agent that plays Tic-Tac-Toe optimally using the Minimax algorithm [1], which evaluates all possible game outcomes to choose the best move considering a rational opponent. To improve the efficiency, Alpha-Beta Pruning [2] is integrated to eliminate unnecessary branches during the search. The AI supports human vs. AI and is further enhanced with heuristic evaluations for faster decisions under limited depth thus ensuring optimal play always resulting in a win or draw.

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.

II. Methodology

The Tic-Tac-Toe AI was developed with a focus on core principles of game-playing artificial intelligence, adversarial search, including 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 like Minimax, Alpha-Beta Pruning, and Heuristic Evaluation work together to make strategic decisions during gameplay.

A. Board Representation and Game Flow

The game board is modeled as a 3×3 matrix, where each cell contains one of three symbols: 'X' for the AI's move, 'O' for the human player's move, or a blank space for an unoccupied cell. A win is declared when a player aligns three of their marks horizontally, vertically, or diagonally. A draw occurs if all positions are filled without a winner. Players alternate turns until one of these conditions is met.

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.

B. *Minimax Algorithm*

The Minimax algorithm [3] serves as the decision-making core of the AI. It explores the game tree by recursively simulating all legal moves from the current position under the assumption that both players play optimally. Nodes where it is the AI's turn are treated as maximizing layers (the algorithm seeks the largest attainable score), while nodes where it is the opponent's turn are minimizing layers (the algorithm assumes the opponent will choose the move that yields the smallest score for the AI). The recursion bottoms out at terminal states, which are scored as +1 for a win, −1 for a loss, and 0 for a draw, as illustrated in Fig. 1 below. These terminal values are then propagated upward: at a max node, the best child value (the maximum) becomes the node's value; at a min node, the worst child value (the minimum) is taken. In this way, Minimax backtracks from leaves to the root, producing an optimal value for the current position and a principled choice of the best move.

In practice, Minimax guarantees that the AI never selects a suboptimal move when the game is deterministic, finite, and of perfect information as Tic-Tac-Toe is. If the full search to terminal positions is feasible (as it is here, given the shallow depth and small branching factor), the algorithm computes the exact game-theoretic value of the position (for Tic-Tac-Toe, optimal play leads to a draw). When deeper trees or time limits are involved, the same framework can use a depth cutoff with a lightweight evaluation function to approximate terminal scores, and it can incorporate move tie-breaking rules (for example, prefer center, then corners) without changing the underlying optimality guarantees when a full search is completed. Combined with Alpha–Beta pruning (see § on pruning), Minimax remains exact while exploring far fewer nodes, but its essence is unchanged: choose the move that maximizes the AI's guaranteed outcome, assuming the opponent responds in the best possible way.

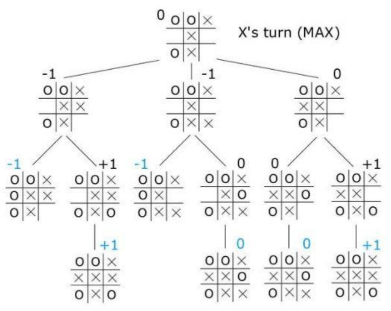


Fig. 1. A breakdown of the minimax algorithm in AI [3]

C. *Alpha-Beta Pruning*

To enhance performance, Alpha–Beta pruning is applied to the Minimax search [4]. Conceptually, it keeps the exact same decision rule as plain Minimax but skips exploring moves that can no longer affect the final choice. It does this by carrying two bounds down the search: alpha (α), the best score the maximizing player (AI) has already secured on the path so far, and beta (β), the best (i.e., lowest) score the minimizing opponent can force so far. At the root we initialize α = −∞ and β = +∞. When we evaluate children of a max node, we update α ← max(α , childValue); if at any point α ⩾ β , we stop—no remaining sibling can produce a better outcome for Max because the opponent already has a reply at least as good as β . Symmetrically, at a min node we update β ← min(β , childValue)** and prune when α ⩾ β , because Max already has an option at least as good as α and Min cannot allow anything worse than β . This "cutoff" logic (illustrated in Fig. 2) is sound: pruned branches are guaranteed irrelevant to the final Minimax value, so decision accuracy is preserved. In practice, pruning can dramatically reduce the number of evaluated states on small games like Tic-Tac-Toe (branching factor up to 9, depth ⩽ 9), good cutoffs arrive quickly, often yielding speedups near an order of magnitude compared to naive Minimax, with the classic best-case node count approaching the theoretical O(b^(d/2)) frontier versus O(b^d) for plain search when move ordering is near optimal. Ordering matters: if we consider strong candidates first (e.g., center before corners before edges; or moves that caused big cutoffs in prior searches "killer" moves; or moves ranked by a lightweight heuristic), α and β tighten earlier, producing more cutoffs. Alpha–Beta also pairs naturally with iterative deepening: a shallow search at depth d provides an excellent move order for depth d+1, multiplying pruning effectiveness and maintaining responsive time budgets. Finally, caching transpositions e.g., with Zobrist hashing, avoids recomputing identical positions reached via different move orders, pushing depth even further. Taken together, these techniques keep the engine exact while substantially accelerating search "pruning the haystack" so the same best move is found with far fewer node evaluations.

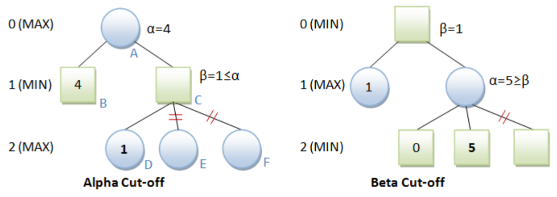


Fig. 2. A visual of Alpha-Beta Pruning [5]

D. 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:

1) *+10 for having two AI marks in a line with one empty space.*
2) *+1 for one AI mark in a line with two empty spaces.*
3) *-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.

E. 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.

F. Implementation Details

Our Tic-Tac-Toe project is implemented in Python and organized using a modular design so each concern is cleanly separated and easy to reason about. The codebase is split into three main files. main.py orchestrates the entire experience: it initializes the board, manages the game loop, switches turn, and coordinates communication between the game rules and the AI. game.py is responsible for the "world of the game" so it renders the board, validates user moves (for example, preventing overwriting occupied cells), and updates the underlying state. It also exposes helper functions such as move generation and win/draw detection. ai.py encapsulates the computer opponent. Depending on the configuration, it can play via a basic heuristic (center, corners, best available) or a Minimax search with optional Alpha–Beta pruning for optimal play. By isolating responsibilities, the project remains readable, testable, and simple to extend—for instance, swapping in a different heuristic or changing the board size.

Under the hood, the board is modeled as a compact 1-D list of nine elements, which keeps indexing straightforward and serializes easily for testing. The game loop repeatedly prints the current board, prompts the human player, validates input (digits only, within range, and to an empty square), and then asks the AI for its response. After each move, the engine checks for three-in-a-row or a full board to decide whether the game is over. We added defensive programming around input handling to avoid crashes from unexpected entries and to give helpful feedback to the player. Because the modules are decoupled, it is simple to add quality-of-life features like logging, difficulty settings, or a replay option—without touching the core logic. The result is a clear, maintainable implementation that balances educational transparency with robust, legal play.

1) Code Structure Overview

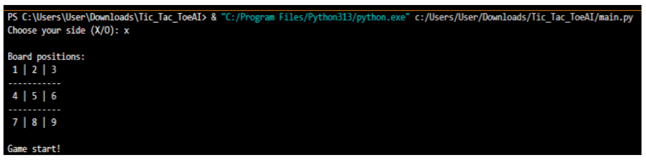
Table.1. Modular Structure Of Our Project

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, winner checking logic.

2) Board Representation

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

board = [" "] * 9



```
PS C:\Users\User\Downloads\Tic_Tac_ToeAI> & "C:/Program Files/Python313/python.exe" c:/Users/User/Downloads/Tic_Tac_ToeAI/main.py
Choose your side (X/O): x

Board positions:
 1 | 2 | 3
-----------
 4 | 5 | 6
-----------
 7 | 8 | 9

Game start!
```

Fig.3. Board Representation Output

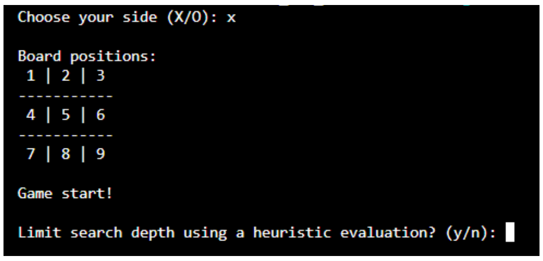
As you can see on Figure 3 above, each element in the list represents a cell on the 3×3 Tic-Tac-Toe grid. The list uses indexes from 0 to 8, which correspond to positions 1 through 9 as seen by the player. This layout makes it easy to update and check specific positions on the board during gameplay.

Example Index Layout:

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

3) *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 you can see in the fig. 4 below which lets the AI use a faster but slightly less thorough decision-making strategy.



```
Choose your side (X/O): x

Board positions:
 1 | 2 | 3
-----------
 4 | 5 | 6
-----------
 7 | 8 | 9

Game start!

Limit search depth using a heuristic evaluation? (y/n):
```

Fig.4. Permission taken before implementing heuristics

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:

a) print_board(board)

Shows the current state of the board in a clean, readable format.

b) player_move(board, human_player)

Gets the player's input and ensures the move is valid.

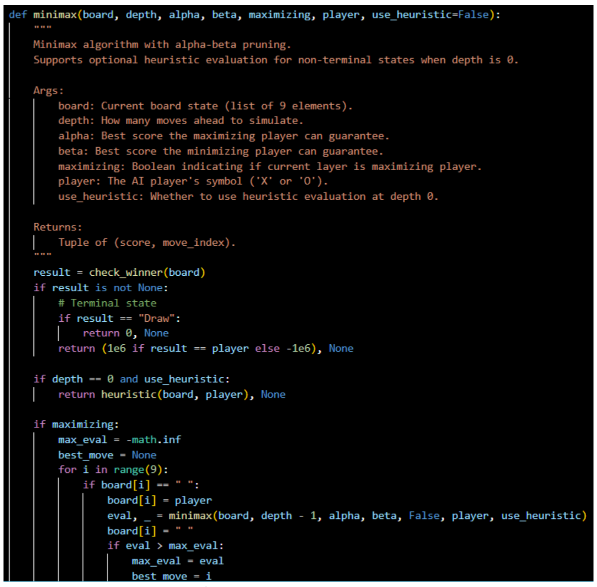
c) ai_move(board, ai_player, use_heuristic)

Determines the AI's move using either the full depth Minimax algorithm or a depth-limited version with heuristics 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.

4) *Minimax Algorithm with Alpha Beta Pruning*

The core of the AI's decision-making is the minimax() function, which is implemented as we see on fig.5. recursively to explore possible future moves and determine the best one.



```python
def minimax(board, depth, alpha, beta, maximizing, player, use_heuristic=False):
    """
    Minimax algorithm with alpha-beta pruning.
    Supports optional heuristic evaluation for non-terminal states when depth is 0.

    Args:
        board: Current board state (list of 9 elements).
        depth: How many moves ahead to simulate.
        alpha: Best score the maximizing player can guarantee.
        beta: Best score the minimizing player can guarantee.
        maximizing: Boolean indicating if current layer is maximizing player.
        player: The AI player's symbol ('X' or 'O').
        use_heuristic: Whether to use heuristic evaluation at depth 0.

    Returns:
        Tuple of (score, move_index).
    """
    result = check_winner(board)
    if result is not None:
        # Terminal state
        if result == "Draw":
            return 0, None
        return (1e6 if result == player else -1e6), None

    if depth == 0 and use_heuristic:
        return heuristic(board, player), None

    if maximizing:
        max_eval = -math.inf
        best_move = None
        for i in range(9):
            if board[i] == " ":
                board[i] = player
                eval, _ = minimax(board, depth - 1, alpha, beta, False, player, use_heuristic)
                board[i] = " "
                if eval > max_eval:
                    max_eval = eval
                    best_move = i
```

Fig.5. Small Snippet Of Minimax Implementation

It works by simulating every possible move for both the AI and the opponent, and evaluating the outcomes to choose the most favorable path[5]. Here's how the function is structured: minimax(board, depth, alpha, beta, maximizing, player, use_heuristic

a) Depth

Controls how many moves ahead the AI will look. A value of 9 means it will search the entire game tree [6].

b) 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.

c) maximizing

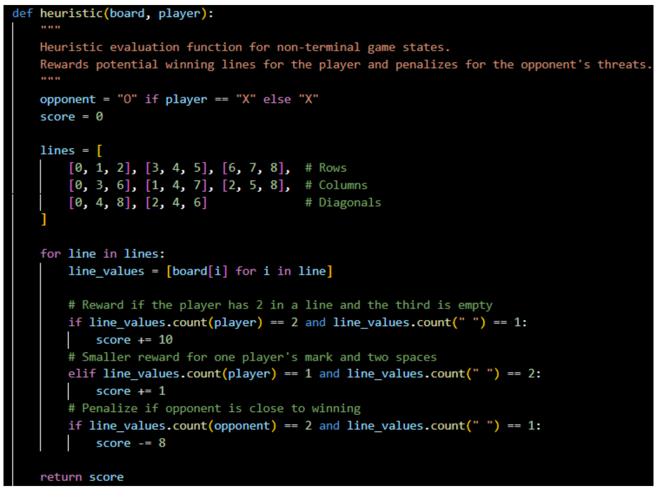
A flag that tells the function whether it's the AI's turn (maximize the score) or the opponent's turn (minimize the score).

d) 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.

5) *Heuristics Function*

The heuristic logic is handled by the heuristic(board, player) function as we see on fig. 6 below.



```python
def heuristic(board, player):
    """
    Heuristic evaluation function for non-terminal game states.
    Rewards potential winning lines for the player and penalizes for the opponent's threats.
    """
    opponent = "O" if player == "X" else "X"
    score = 0

    lines = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # Rows
        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # Columns
        [0, 4, 8], [2, 4, 6]              # Diagonals
    ]

    for line in lines:
        line_values = [board[i] for i in line]

        # Reward if the player has 2 in a line and the third is empty
        if line_values.count(player) == 2 and line_values.count(" ") == 1:
            score += 10
        # Smaller reward for one player's mark and two spaces
        elif line_values.count(player) == 1 and line_values.count(" ") == 2:
            score += 1
        # Penalize if opponent is close to winning
        if line_values.count(opponent) == 2 and line_values.count(" ") == 1:
            score -= 8

    return score
```

Fig.6. Heuristics 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:

a) Identifying lines where the AI is close to winning and rewarding those positions.

b) Detecting threats from the opponent and applying penalties to encourage blocking moves.

c) Scoring based on how many potential win paths are still open.

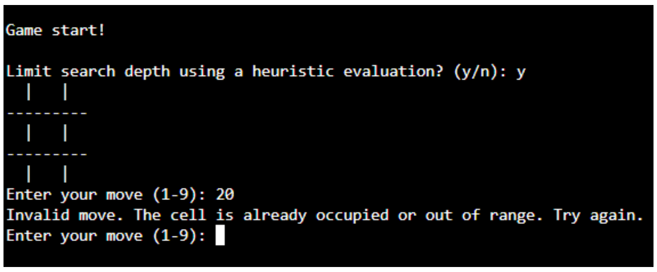
This function becomes especially useful when the AI isn't searching all the way to the end of the game. Instead of trying every possible move, it makes a well-informed guess about which moves are strongest based on the current situation. This keeps the AI playing strategically, even with limited information.

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:

a) The number must be within the valid range (1–9).

b) The selected cell must be empty. The input must be a number, not a letter or symbol.



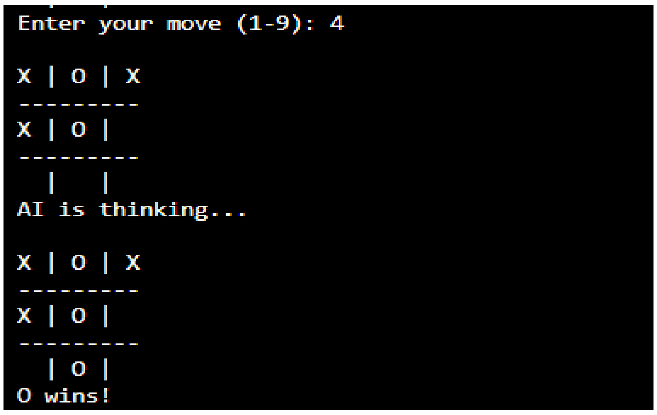
```
Game start!

Limit search depth using a heuristic evaluation? (y/n): y
  |   |
---------
  |   |
---------
  |   |
Enter your move (1-9): 20
Invalid move. The cell is already occupied or out of range. Try again.
Enter your move (1-9): 
```

Fig.7. Error Message When Wrong Input

As we see on Fig.7, 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.

7) *Endgame Detection*

The check_winner(board) function [7] 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.



```
Enter your move (1-9): 4

X | O | X
---------
X | O |
---------
  |   |
AI is thinking...

X | O | X
---------
X | O |
---------
  | O |
O wins!
```

Fig. 8. Shows with X or O-Who Won

As we see on the fig.8. above, 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.

III. 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 [8]. This section highlights what those tests revealed about the AI's accuracy, efficiency, and overall behavior during gameplay.

A. *Functional Accuracy*

The AI was tested across multiple complete game scenarios where:

1) *The human player made random or strategic moves.*

2) *The AI was expected to block threats, create winning combinations, or force a draw.*

The observations are that 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.

B. *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. We can visualize the performance impact with or without pruning shown in a tabular form below:

Table 2. Overall Analysis with or without Pruning

Condition	Average Nodes Evaluated	Time per Move (ms)
Minimax without	~5350	~65–90 ms
Pruning	-	-
Minimax with Pruning	~1180	~15–25 ms

Alpha-beta pruning further 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. This shows how effective pruning is.

C. Heuristic Evaluation Performance

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

Table. 3. Observed Behavior using Pruning

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. It maintained full accuracy as well.

D. Limitations

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

1) No Learning Capability:

The AI does not adapt its policy based on outcomes or opponent behavior; it evaluates positions in exactly the same way every game. As a result, it cannot exploit recurring human mistakes, recognize opponent-specific patterns, or refine its play over time. In practical terms, a beginner who repeatedly blunders will face the same static responses rather than an AI that “notices” and pressures those weaknesses, and an advanced player will find the engine entirely predictable once its decision logic is understood.

2) Fixed Board Size:

The implementation is tightly coupled to a 3×3 board and a “three-in-a-row” win condition, so core routines (state representation, move generation, win checks, and evaluation) are not generalizable. Scaling to 4×4 or 5×5 inflates the state space dramatically, making naive Minimax prohibitively expensive without architectural changes and stronger pruning or alternative methods (e.g., MCTS). Because of these constraints, the current codebase cannot serve as a testbed for connect-k variants or comparative research on algorithm performance across board sizes.

3) No Graphical Interface:

Running purely in the terminal is adequate for debugging and quick demos, but it limits usability, accessibility, and engagement for non-technical users. Without a GUI, there are no visual affordances such as clickable cells, move previews, highlight of winning lines, or animations that help learners understand strategy. This also makes it harder to distribute the game widely (e.g., as a classroom activity or casual web app) and to collect structured user feedback.

4) Static Behavior:

The engine always plays flawlessly at full strength, which, while impressive, removes variability and can feel frustrating or boring to human opponents. There is no native mechanism to throttle difficulty (e.g., shallower search, stochastic tie-breaking, or style profiles), so every match converges to the same optimal lines. Without adjustable depth, noise, or alternative heuristics, the AI cannot simulate different “personalities,” making training modes or progressive challenges difficult to design.

E. 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. The future improvements are stated below:

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.

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.

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.

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. AI training based on real user games. It will be a fun experimental game for two persons. This can be implemented in a very low resource setting as well.

5) *AI Enhancements*

We can make the engine both stronger and more engaging by layering classic search upgrades with learning and explain ability. First, improve move ordering so Alpha–Beta pruning cuts more branches: apply domain knowledge (center before corners, corners before edges) alongside dynamic techniques like killer-move and history heuristics that prioritize moves which previously caused big cutoffs. Pair this with iterative deepening under a fixed time budget so the AI searches depth 1, then 2, and so on, always retaining the best move found so far—this yields responsive UIs and natural "thinking" animations. Add a transposition table keyed with Zobrist hashing to cache evaluated positions; by avoiding recomputation, the engine can search substantially deeper at higher difficulty levels. Beyond pure search, introduce learning-augmented play: train a Q-learning or TD(λ) agent via self-play to learn value estimates and benchmark them against hand-crafted heuristics, or plug a small neural network in as a Minimax evaluation to study the trade-off between learned evaluation and brute-force search. For a different decision paradigm, implement Monte Carlo Tree Search [8][9] (UCT) and compare its strength and speed to Minimax across larger board sizes. To make the opponent feel less mechanical, expose "style profiles" that tweak heuristics and tie-breaking to create personalities like aggressive, defensive, or playful randomness and especially at lower difficulties. Finally, add lightweight explainability: after each move, show a one-line rationale such as "blocks an immediate fork" or "creates two winning threats," which makes the AI's choices transparent, improves pedagogy, and builds player trust.

6) *Promising Research Directions*

Promising research directions fall into three complementary tracks: curriculum learning, explainability, and algorithmic trade-offs [10]. In a curriculum-learning setup, agents can be trained first on small boards (for example, 3×3 with $k=3$) and progressively advanced to larger, harder tasks (4×4 with $k=3$, 5×5 with $k=4$, and so on), measuring transfer by how many episodes are required to reach a target win/draw rate after each upshift, how quickly policies stabilize, and how much prior knowledge (features or value nets) carries over; ablations should compare "from-scratch" baselines, frozen-feature transfer, and fine-tuning to quantify sample-efficiency gains. Explainability studies can be run as controlled, between-subjects experiments in which human players face the same engine with and without post-move rationales; outcomes such as error rate, puzzle-solving time, perceived trust/competence (Likert scales), and retention on follow-up tasks can be analyzed with t-tests or mixed-effects models, ideally on nontrivial boards ($\geqslant 4\times4$) to avoid the trivial draw of optimal 3×3. Finally, systematic algorithmic trade-offs should benchmark Minimax with Alpha - Beta (plus move ordering and transposition tables) against Monte Carlo Tree Search (UCT variants) and reinforcement-learning agents across board sizes and compute budgets, reporting strength (Elo or win/draw/loss vs fixed test suites), speed (nodes per second, wall-time per move), and anytime behavior (quality vs time under iterative deepening or rollout caps). To ensure fairness and reproducibility, fix random seeds, standardize starting positions and opening books, log search statistics (branching factor, pruning rate, rollout variance), and publish code, configs, and self-play datasets so results can be replicated and extended.

IV. Conclusions

In the first phase, the project successfully demonstrated a Tic-Tac-Toe AI that used the Minimax algorithm for optimal decision-making and further incorporated Alpha-Beta pruning to improve computational efficiency. The AI consistently guaranteed a win or a draw against any opponent, showcasing its effectiveness. For future, we planned on fully optimizing the Alpha-Beta pruning technique integrating heuristic evaluations to support limited-depth searches and expanding the system to accommodate larger board configurations possibly. Additionally, the development of a graphical user interface (GUI) is planned to improve user interaction and experience. Overall, this work lays a strong foundation for building intelligent agents for classical turn-based games using classical search methods.

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.

References

[1] Wikipedia Contributors, “Minimax,” *Wikipedia*, Sep. 16, 2019. https://en.wikipedia.org/wiki/Minimax

[2] Wikipedia Contributors, “Alpha–beta pruning,” *Wikipedia*, Jun. 17, 2025.

[3] “Min Max Algorithm in AI: Components, Properties, Advantages & Limitations,” *upGrad blog*, Dec. 22, 2020. https://www.upgrad.com/blog/min-max-algorithm-in-ai/

[4] A. madi, “Tic-Tac-Toe agent using Alpha Beta pruning - Alaa madi - Medium,” Medium, Jun. 02, 2023. https://medium.com/@amadi8/tic-tac-toe-agent-using-alpha-beta-pruning-18e8691b61d4

[5] 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.

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

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

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

[9] 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.

[10] 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.