Optimizing Game Tree Search using Minimax Algorithm and Alpha-Beta Pruning Heuristics in Tic-Tac-Toe AI

Al Hasib Reshad
Department of Electrical and Computer Engineering
North South University
al.reshad@northsouth.edu

Fariha Akter Resha
Department of Electrical and Computer Engineering
North South University
fariha.resha@northsouth.edu

Sakila Anjum
Department of Electrical and Computer Engineering
North South University
sakila.anjum@northsouth.edu

Sanjida Akhter Sadia
Department of Electrical and Computer Engineering
North South University
sanjida.sadia@northsouth.edu

Md Hasibur Rahman
Department of Electrical and Computer Engineering North South University
hasibur.rahman08@northsouth.edu

Abstract—**This project showcases an intelligent Tic-Tac-Toe AI player developed in Python, utilizing the Minimax algorithm with Alpha-Beta pruning as the main decision-making method. The AI is intended to come out as the winner or at least tie always if a full-depth search of the game tree is made. Pruning the worst-case scenario with Alpha-Beta is among the major ways that the algorithm has been made more efficient. This is achieved by searching through what branches of the tree cannot possibly affect the final decision hence, less time is consumed without losing precision. When it is not practical to explore the whole game tree—this could be due to limited computing resources or the user imposing depth constraints—the system resorts to a heuristic evaluation function. This heuristic gives a rough estimate of how strong the intermediate positions on the board are, thus allowing the AI to make decisions based on good strategy even it cannot foresee every possible outcome. Accuracy and efficiency are weighed against one another, the AI thus becomes quick in response and yet, still, shows the quality of smart play. The complete system is developed on a modular software architecture which divides the game mechanics, AI logic, and user interaction into separate components. Such a design makes it easier to understand, maintain, and expand the codebase. It also allows for improvements in the future such as enlarging the game to accommodate larger boards or adding a graphical user interface for better convenience. In a nutshell, the present paper introduces the AI's primary algorithms, discusses the implementation strategy, performance analysis through experimental results, and outlines the prospective areas for research and development.**

I. Introduction

Tic-Tac-Toe, even if it looks simple, is still one of the best teaching methods for studying the algorithms for searching for opponents in games. The game is very small, completely visible, and has a deterministic state space that makes it perfect for demonstrating the classical AI techniques such as Minimax algorithm, Alpha-Beta pruning, heuristic evaluation, and the like. A game that is so small and simple grants the teaching and research of the game's strategies without having to deal with the difficulties of computation found in more complex games. The main goal of this project is to build an AI agent that is always capable to select the best possible move at every moment of the game. This agent will always be able to win in case it is mathematically possible and assure the draw in every other situation where both players cannot gain an advantage due to perfect play. To act that way, the AI must consider the different configurations of the board in the future and the opponent's possible reactions with great accuracy.

The Minimax algorithm is the one that lays down the theoretical foundation for this reasoning. By performing recursively all the move sequences for both players, Minimax then ranks the resulting game states according to their desirability and selects the corresponding strategic choice as optimal. Nevertheless, the classic Minimax algorithm can be very time-consuming in terms of computations, even in a comparatively small domain like Tic-Tac-Toe. During the early game phases, when the number of free cells on the board is still high, the branching factor rises and the algorithm might be ending up looking through thousands of states in order to pinpoint just one optimal move.

To cope with this limitation, Alpha–Beta pruning is introduced in the search procedure. The application of this optimization method prunes out the portions of the decision tree that do not affect the outcome and thus improves the overall efficiency of the computation without changing the algorithm's outcome. As a consequence of this pruning mechanism, the AI can determine the best moves much quicker.

Additionally, the project is equipped with a heuristic evaluation function for scenarios with depth-limited search. Heuristics give an estimate of the value of the intermediate board states when a full traversal of the game tree is not needed or is deliberately restricted for performance reasons. This allows the AI to behave strategically, detect threats and take advantage of them, even when it is unable to look through every possible future outcome.

II. Methodology

Tic-Tac-Toe AI creation process was directed by the essential mainstays of artificial intelligence for game playing, namely adversarial search that included the decision tree exploration and efficient computation. Here the representation of the game in code and the managing of the turns between player and AI are explained. Moreover, the Minimax, Alpha-Beta Pruning, and Heuristic Evaluation key algorithms work together to make strategic decisions during the 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] is the main decision-making tool of the AI. It surveys the game tree by changing every legal move simulation at the current position, always under the assumption that the two players are perfect. The AI's turn is represented by the maximizing layer of nodes (the algorithm is looking for the largest score achievable), while the opponent's turn is represented by the minimizing layer (the algorithm assumes that the opponent will make the move that gives the AI the least score). The recursion comes to an end at terminal states, which are scored as +1 for a win, -1 for a defeat, and 0 for a draw, as demonstrated in Fig. 1 below. These terminal values are then sent upwards: 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 accepted. In this manner, Minimax traces back from the leaves to the root, providing an optimal value for the present position as well as a principled selection of the best move. In reality, it is the case that Minimax assures the AI not to make a wrong move during an entire deterministic, finite and perfectly informed game (just like Tic-Tac-Toe).

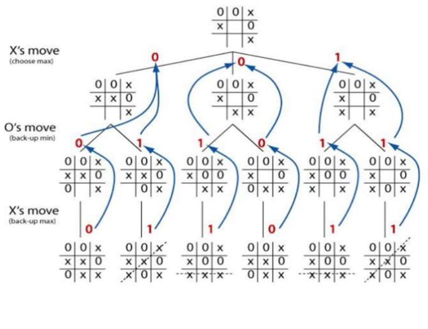


Fig. 1. A breakdown of the minimax algorithm in AI

If the whole search to terminal positions is possible (as it is in this case due to the shallow depth and small branching factor), then the exact game-theoretic value of the position is computed by the algorithm (for Tic-Tac-Toe this means that optimal play results in a draw). In the case where deeper trees or time limits are involved, the same framework can use a depth cutoff alongside a lightweight evaluation function to approximate terminal scores and move tie-breaking rules can be incorporated without affecting the underlying optimality guarantees when a full search is completed. Combined with Alpha–Beta 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.

C. Alpha-Beta Pruning

Alpha-Beta pruning is used in the Minimax search to improve the performance [4]. It is a different way of looking at the same decision rule as the plain Minimax but it does not explore the moves that cannot in any way affect the final choice. The procedure operates by taking two bounds along the search: alpha (α), the maximum score that the maximizing player (AI) has already obtained on the path so far, and beta (β), the minimum score that the minimizing opponent can force so far. At the root, we set α = −∞ and β = +∞. When we check the children of a max node, we adjust α ← max(α, childValue); if at any time α ≥ β, we cease—no possible remaining sibling could give Max a better outcome because the opponent already has a reply that is at least as good as β. In the same way, at a min node we adjust β ← min(β, childValue)** and cut when α ≥ β, because Max is already holding an option that is at least as good as α and Min cannot accept anything worse than β. This "cutoff" reasoning (shown in Fig. 2) is correct: the branches eliminated will not affect the final value of Minimax, so the precision of the decision is maintained. However, in many cases, the pruning can reduce the number of states evaluated significantly in small games like Tic-Tac-Toe (the branching factor is up to 9, depth is ≤ 9), and very soon good cutoffs come and mostly the speed-ups are around one order of magnitude compared to the 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.

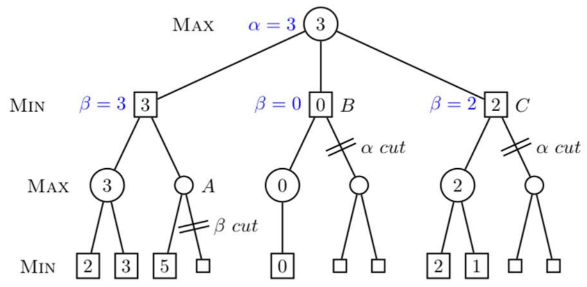


Fig. 2. A visual of Alpha-Beta Pruning

The order in which the moves are considered does matter: if we first pick the strongest candidates (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), then α and β will be narrowed down more quickly leading to more cutouts. 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.

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. When the search depth is
 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:

a. *The current state of the board is shown to the player.*

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

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

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

e. *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.

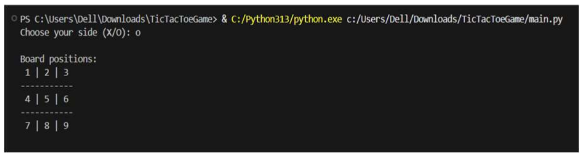
1) Code Structure Overview

Table.1. Modular Structure Of Our Project

Module	Description
tictactoe_game.py	Contains functions for rendering the board, handling player moves and calling the AI.
tictactoe_ai.py	Implements the Minimax algorithm, alpha-beta pruning, evaluation, heuristic winner checking logic.
simulate.py	For generating the performance results.
main.py	Entry point of the application; manages game loop and user interaction.

2) Board Representation

The board is implemented as a one-dimensional Python list of length 9: board = [" "] * 9



```
PS C:\Users\Dell\Downloads\TicTacToeGame> & C:/Python313/python.exe c:/Users/Dell/Downloads/TicTacToeGame/main.py
Choose your side (X/O): o

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

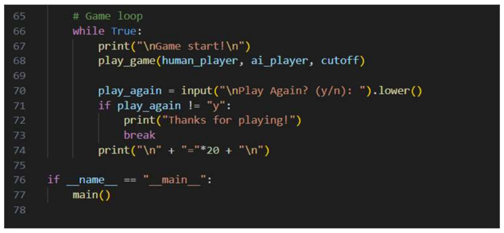
Fig.3. Board Representation Output

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]
```

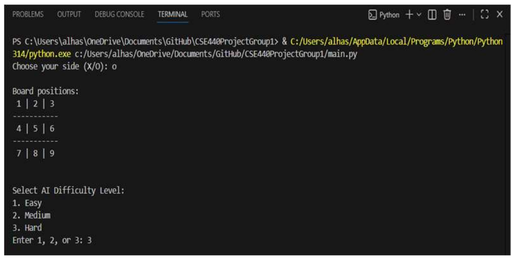
3) Main Game Loop

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.



```python
    # Game loop
    while True:
        print("\nGame start!\n")
        play_game(human_player, ai_player, cutoff)

        play_again = input("\nPlay Again? (y/n): ").lower()
        if play_again != "y":
            print("Thanks for playing!")
            break
        print("\n" + "="*20 + "\n")

if __name__ == "__main__":
    main()
```

Fig 4: Coding Game Loop



```
PS C:\Users\alhas\OneDrive\Documents\GitHub\CSE440ProjectGroup1> & C:/Users/alhas/AppData/Local/Programs/Python/Python
314/python.exe c:/Users/alhas/OneDrive/Documents/GitHub/CSE440ProjectGroup1/main.py
Choose your side (X/O): o

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


Select AI Difficulty Level:
1. Easy
2. Medium
3. Hard
Enter 1, 2, or 3: 3
```

Fig 5: Permission taken before implementing heuristics

Once the game begins, it runs in a loop that continues until accurately block immediate threats (e.g., when the opponent was about to complete a row with two markers). It would take winning moves when available and could also force a draw in 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.

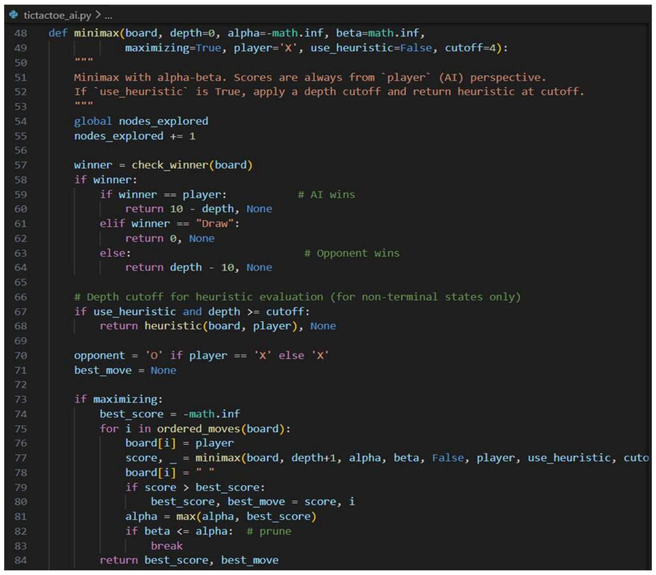
c) ai_move(board, ai_player, use_heuristic)

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.



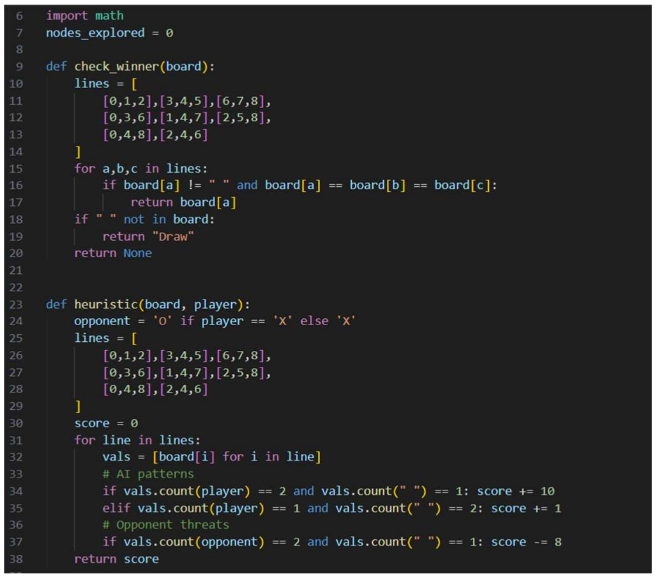
```python
def minimax(board, depth=0, alpha=-math.inf, beta=math.inf,
            maximizing=True, player='X', use_heuristic=False, cutoff=4):
    """
    Minimax with alpha-beta. Scores are always from `player` (AI) perspective.
    If `use_heuristic` is True, apply a depth cutoff and return heuristic at cutoff.
    """
    global nodes_explored
    nodes_explored += 1

    winner = check_winner(board)
    if winner:
        if winner == player:            # AI wins
            return 10 - depth, None
        elif winner == "Draw":
            return 0, None
        else:                            # Opponent wins
            return depth - 10, None

    # Depth cutoff for heuristic evaluation (for non-terminal states only)
    if use_heuristic and depth >= cutoff:
        return heuristic(board, player), None

    opponent = 'O' if player == 'X' else 'X'
    best_move = None

    if maximizing:
        best_score = -math.inf
        for i in ordered_moves(board):
            board[i] = player
            score, _ = minimax(board, depth+1, alpha, beta, False, player, use_heuristic, cuto
            board[i] = " "
            if score > best_score:
                best_score, best_move = score, i
            alpha = max(alpha, best_score)
            if beta <= alpha:  # prune
                break
        return best_score, best_move
```

Fig 6: 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. Here's how the function is structured: minimax(board, depth, alpha, beta, maximizing, player, use_heuristic.

5) Heuristics Function

The heuristic logic is handled by the heuristic(board, player) function as we see on fig. 7 below.



```python
import math
nodes_explored = 0

def check_winner(board):
    lines = [
        [0,1,2],[3,4,5],[6,7,8],
        [0,3,6],[1,4,7],[2,5,8],
        [0,4,8],[2,4,6]
    ]
    for a,b,c in lines:
        if board[a] != " " and board[a] == board[b] == board[c]:
            return board[a]
    if " " not in board:
        return "Draw"
    return None


def heuristic(board, player):
    opponent = 'O' if player == 'X' else 'X'
    lines = [
        [0,1,2],[3,4,5],[6,7,8],
        [0,3,6],[1,4,7],[2,5,8],
        [0,4,8],[2,4,6]
    ]
    score = 0
    for line in lines:
        vals = [board[i] for i in line]
        # AI patterns
        if vals.count(player) == 2 and vals.count(" ") == 1: score += 10
        elif vals.count(player) == 1 and vals.count(" ") == 2: score += 1
        # Opponent threats
        if vals.count(opponent) == 2 and vals.count(" ") == 1: score -= 8
    return score
```

Fig 7: 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.

III. Results and Analysis

The evaluation of the Tic-Tac-Toe AI's performance was done through various game simulations and interactive play sessions. The primary motive was to check if the AI could always take the right approach, not lose, and change its strategy according to the situation whether the full Minimax algorithm was being used or the optimized versions with alpha-beta pruning and heuristic support [8]. This part of the document discusses the tests conducted and reveals the AI's ability to play accurately, its efficiency, and behavior in the overall workflow.

A. Functional Accuracy

The AI was put to the test through several complete game scenarios where:

1) the human player made either random or strategic moves and

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

It was noted that the AI never lost when given the opportunity of computing the full depth Minimax.

B. Performance with and without Alpha Beta Pruning

Alpha-beta pruning was a major factor in the strong reduction of the number of board states that were evaluated in the AI's decision-making process. The number of possible states, which is around 255,000 for Tic-Tac-Toe, is quite limited but the difference was noticeable. How to visualize the performance effect with or without the use of pruning can be seen in the table below:

Table 2. Overall Analysis with or without Pruning

Configuration	Avg Nodes Evaluated Per Moves	Avg Time per Move (ms)
Minimax without Pruinmg	~6314	~62.35ms
Minimax with Pruning	~540	~4.67 ms

Alpha-beta pruning was the major factor contributing to performance gain as it could cut off the decision tree by 75% which consequently lowered the time until AI response and made it perceived as quicker and more responsive during the game. This is a proof of how effective pruning is.

C. Heuristic

AI utilized heuristic as its main function in intermediate game states when not more than certain number of moves (e.g., 2=Medium or 3=Hard) was set. Heuristic Play Behavior: First it blocked most of the immediate threats, then it identified the opportunities to build towards potential wins. Long-term strategies were sometimes missed due to limited depth.

Table. 3. Observed Behavior using Pruning

Depth Limit	Win Rate vs Random Player	Observed Behavior
Hard	100%	Always optimal
Medium Heuristic	~90%	Generally strong, sometimes block-first
Easy Heuristic	~75%	Defensive bias, missed offensive setups

This indicated that even at limited depth, the heuristics supplied the AI with a strategic advantage over random play, yet it sometimes gave up optimality for speed. It also kept full accuracy.

D. Limitations

The AI is good on the 3×3 board, but it has a lot of limitations in its current form: It does not change its strategy depending on the results or the opponent's actions; it treats the positions in the same way for every game. Therefore, it cannot take advantage of human errors that happen often, will not be able to see patterns that are only for that opponent, and will not be able to improve its play over time. To put it in simple terms, a novice who keeps making bad moves will be met with the same unchanging replies of the AI instead of an AI that learns and continues to push those weaknesses and a pro player will find the machine totally predictable once its reasoning is clear. The current state of affairs is confined to a 3×3 board and a 'three-in-a-row' victory condition that makes it impossible to generalize about the core routines (state representation, move generation, win checks, and evaluation). Thus, if the scenario is stretched to 4×4 or 5×5 the initial motion would be very human-like by simulating suboptimal moves at lower difficulties. The space is enormously inflated, making the simple Minimax that is not changed architecturally, prohibitively costly without stronger pruning or using methods like MCTS as alternatives. Due to these limitations, the present codebase is not suitable for testing connect-k variants or for conducting research that compares the efficiency of algorithms by board sizes. Having purely terminal experience is sufficient for debugging and quick demonstrations but it limits usability, accessibility, and attraction for non-technical users. The absence of GUI means that there are no visual aids like mouse-clickable boxes, previews of moves, line of victory being lit, and animations that assist in comprehending strategy. It also hinders the distribution of the game to a larger audience (for example, as a school exercise or a casual web app) and collecting organized user feedback. The engine always operates perfectly at full capacity, which, though awe-inspiring, eliminates uncertainty and might even be annoying or dull to the human competitors. There isn't a built-in method to reduce the CPU's power (for example, deeper search, stochastic tie-breaking, or style profiles), thus the entire game ends up being played on the same optimal lines. Without the possibility of depth adjustment, introversion, or different heuristics, the AI cannot impersonate various "characters," making training modes or gradual challenges hard to devise.

IV. Future Work

Even though the present Tic Tac-Toe AI version is a superb performer and makes wise moves through Minimax, alpha beta pruning, and heuristics, there is still a lot to be done. Future enhancements might make the game more engaging, challenging, and customizable while at the same time, giving a chance for more sophisticated AI techniques and bigger game areas to be explored. The future improvements are listed below:

A. Graphical User Interface (GUI)

At present, the game operates in a text-based console, which restricts the player's interaction with the game. The use of a graphical user interface through the application of different frameworks like Tkinter, Pygame, or Kivy would: Make the game more lively and user-friendly. Let the players choose cells by mouse clicks instead of by pressing for the numbers. Show the AI moves, the game developments, and the win/draw states using visual cues.

B. Support for Larger Board Sizes

The present version is designed specifically for a 3x3 grid. Moving up to large boards, like 4x4 or 5x5, would: open up a gigantic game tree. The difficulty of the decision-making would go up. The Minimax algorithm would need to be fine-tuned, possibly by incorporating iterative deepening or Monte Carlo Tree Search (MCTS) for speed.

C. Adaptive Difficulty and AI Behavior

The game is to become more challenging and more often resumed: Create difficulty levels by changing the max search depth and switching heuristic evaluation on/off. Incorporate learning-based components like Q-learning or reinforcement learning so that the AI can get better with time. Give the AI personality more human-like by simulating sub optimal moves at lower difficulties.

D. Multiplayer and Online Play

A multiplayer mode either locally or over a network can be implemented that would allow two humans to play against the AI with the option of having AI act as an onlooker or referee. Thus, AI will learn from games played by real users. It will be a trial and error kind of game for two players. This can also be done in a very resource-efficient manner.

G. AI Enhancements

First of all, one has to understand that the classic search technique is no longer sufficient for the development of engaging and powerful engines. The layering of classic search upgrades together with learning and explainability will definitely result in getting both a more powerful and an engaging engine. The first thing that comes to mind is to improve the move ordering such that Alpha-Beta pruning would cut off more branches. One way to achieve this is through the application of domain knowledge (center before corners, corners before edges) and dynamic techniques like killer-move and history heuristics that prioritize moves which previously caused big cutoffs. The result would be a responsive UI and a more natural “thinking” animation. This move would be the pairing of iterative deepening under a fixed time budget so that AI would search depth 1, then 2, etc., all the way retaining the best move.

Next, an addition of a transposition table that possesses Zobrist hashing for caching evaluated positions would speed up the process a great deal since the engine would be able to avoid recomputation—thus, it can search considerably deeper at higher difficulty levels. Besides pure search, there is a lot of room for introducing learning-augmented play, and so one can train the Q-learning or the TD(λ) agent via self-play to learn value estimates and then to benchmark them against the hand-crafted heuristics, or one can plug a small neural network in as the Minimax evaluation to study the trade-off between learned evaluation and brute-force search. For a different decision paradigm, one could implement Monte Carlo Tree Search (UCT) and compare its strength and speed to Minimax across larger board sizes.

Moreover, to make the opponent less mechanical, one can expose “style profiles” that customize heuristics and tie-breaking to produce personalities such as aggressive, defensive, or playful randomness, especially at lower difficulties. Finally, explainability will be kept at a light-weight level, so to speak: after each move, a one-line rationale such as “blocks an immediate fork” or “creates two winning threats” will be shown, which would make the AI’s choices transparent, improve pedagogy, and, most importantly, build player trust.

H. Promising Research Directions

The authors mention three primary research paths that are likely to yield fruitful results, which include: curriculum learning, explainability, and algorithmic trade-offs [10]. In the case of curriculum learning, the agents can initially train themselves on smaller boards (for example, 3×3 with $k=3$) and then step by step move on to larger and tougher tasks (4×4 with $k=3$, 5×5 with $k=4$, and so on) where the transfer will be reflected by the number of episodes needed to reach the target win/draw rate after each upshift, while how fast policies are stabilized and how much prior knowledge (features or value nets) is transferred will also be monitored; and comparisons will be made between “from-scratch” baselines, frozen-feature transfer, and fine-tuning to measure the sample-efficiency improvements through ablations. As for the area of explication, the research can be carried out as controlled, between-subjects experiments where the same engine can be presented to the human players with and without post-move justifications; t-tests or mixed-effects models can be used to analyze the data coming from the outcomes such as error rate, duration of puzzle-solving, trust/competence perceived (Likert scales), and retention on the follow-up tasks with the setting being nontrivial boards ($\geq4\times4$) to remove the trivial draw of optimal

3×3 being the most preferable. The last point is algorithmic trade-offs, which should be done systematically, measuring the performance of the traditional Minimax against Alpha-Beta (with the addition of move ordering and transposition tables) and Monte Carlo Tree Search (UCT variants) and the distribution of learning agents over different board sizes and compute budgets, considering in terms of 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).

V. Conclusions

The first phase of the project has led to the successful establishment of the Tic-Tac-Toe AI that made use of the Minimax algorithm for the best possible decision-making and then later on took up the Alpha-Beta pruning technique for even better performance. The AI was always a winner or at least tied with the widest range of opponents, which demonstrated its skill, It was resolved to carry out the full optimization of the Alpha-Beta pruning technique and to introduce heuristic evaluations to support limited-depth searches, and, possibly, larger board settings will be taken into account for the project expansion. Besides, a GUI is to be created which will enable better interaction and user experience with the system. In short, this work is a strong basis for the building of intelligent agents for classical turn-based games using classical search methods. The project aimed to the extent of creating a smart and dependable Tic-Tac-Toe AI and that was the attained goal. Through the Minimax algorithm, the AI is able to predict all the possible outcomes and subsequently make the best move out of turn. At the same time, it applies the α-β pruning technique which makes it function in a more efficient manner, that is, it excludes the consideration of moves that are no good. Still, in the case of depth full search not being the most appropriate, a simple but powerful heuristic steps in to help the AI keep a tactical play without missing out on critical opportunities. The AI has been doing great, making fast and accurate decisions, and victory or a tie is guaranteed if it is utilized correctly. The project's modular coding structure makes it easy for the project to be extended which implies that it can be the addition of user interface, larger boards support, new AI techniques being tried out, and so on. Taking everything into consideration, this project turned out to be a great way to transform the theoretical AI concepts into the practical world in a playful and interactive way. By the above-mentioned classes of the game, it built a solid foundation for comprehending choice making, search optimization, and game strategy.

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.