Noughts and Crushes using Alpha-Beta Pruning

A PROJECT REPORT
for
Artificial intelligence (AI101B) Session
(2024-25)

Submitted by

Aman Kumar (2024101161000020)
Alok Kumar (2024101161000018)
Ambikeshwar Dutt Dwivedi (2024101161000022)
Anand Patel (2024101161000023)

Submitted in partial fulfilment of the
Requirements for the Degree of

MASTER OF COMPUTER APPLICATION

Under the Supervision of
Mr. Apoorv Jain (Assistant Professor)

Submitted to

DEPARTMENT OF COMPUTER APPLICATIONS
KIET Group of Institutions, Ghaziabad
Uttar Pradesh-201206

CERTIFICATE

Certified that **Aman Kumar (202410116100020)**, **Alok Kumar (202410116100018), Ambikeshwar Dutt Dwivedi, (202410116100022)**, and **Anand Patel (202410116100023) have successfully carried out the project work titled "Noughts and Crushes using Alpha-Beta Pruning" (Artificial intelligence, AI101B) as part of the curriculum for the Master of Computer Application (MCA) program at Dr. A.P.J. Abdul Kalam Technical University (AKTU) (formerly UPTU), Lucknow, under my supervision.**

The project report embodies original work and research undertaken by the students themselves. The contents of the project report do not form the basis for the award of any other degree or diploma to the candidates or any other individual from this or any other university/institution.

Mr. Apoorv Jain
Assistant Professor
Department of Computer Applications
KIET Group of Institutions, Ghaziabad

Dr. Akash Rajak
Dean & Professor
Department of Computer Applications
KIET Group of Institutions, Ghaziabad

ABSTRACT

This project presents an AI-driven Tic Tac Toe game where a computer opponent challenges a human player using the powerful Minimax algorithm enhanced by Alpha-Beta Pruning. While Tic Tac Toe appears simple, its underlying game theory serves as an excellent demonstration of adversarial search techniques.

By leveraging exhaustive evaluation within a manageable state space, the AI ensures optimal gameplay—either securing a win or forcing a draw when both players make optimal moves. Beyond just a game, this project provides a practical application of fundamental artificial intelligence techniques in decision-making, offering educational insights into recursion, heuristics, and performance optimization.

This report explores the theoretical foundations, system architecture, and in-depth analysis of the implemented algorithms, making it a valuable resource for students, educators, and AI enthusiasts.

ACKNOWLEDGEMENT

Success in life is never attained single-handedly. My deepest gratitude goes to my project supervisor, **Mr. Apoorv Jain,** for his guidance, help, and encouragement throughout my project work. Their enlightening ideas, comments, and suggestions.

Words are not enough to express my gratitude to Dr. Akash Rajak, Dean and Professor, Department of Computer Applications, for his insightful comments and administrative help on various occasions.

Fortunately, I have many understanding friends, who have helped me a lot on many critical conditions.

Finally, my sincere thanks go to my family members and all those who have directly and indirectly provided me with moral support and other kinds of help. Without their support, completion of this work would not have been possible in time. They keep my life filled with enjoyment and happiness.

Aman Kumar
Alok Kumar
Ambikeshwar Dutt Dwivedi
Anand Patel

Detailed Project Report

Tic Tac Toe AI Using Minimax and Alpha-Beta Pruning

Table of Contents

---

1. Introduction

This project presents an AI-driven **Tic Tac Toe** game where a computer agent challenges a human opponent using the powerful **minimax algorithm** enhanced by **alpha-beta pruning**. Although Tic Tac Toe is a simple game on the surface, its underlying game theory offers an ideal demonstration of adversarial search methods.

The AI leverages exhaustive evaluation within a manageable state space to guarantee optimal play—either securing a win or forcing a draw when both parties perform optimally. This project is more than just a game; it's a concrete application of key artificial intelligence techniques in decision-making, offering educational insight into recursion, heuristics, and performance optimization.

The report delves into the theoretical foundations, system architecture, and detailed analysis of the algorithms used, making it a valuable resource for students, educators, and practitioners interested in AI-powered game development.

---

2. Background and Theoretical Foundations

2.1 Tic Tac Toe as an AI Problem

Tic Tac Toe's simplicity—as a 3×3 grid with only nine cells—allows for complete analysis of the game space, making it a well-known zero-sum problem in game theory. Every game can be represented as a game tree where each node signifies a board state, and the exhaustive exploration of this tree teaches fundamental AI concepts such as state-space search and strategic planning.

2.2 The Minimax Algorithm

The minimax algorithm is a recursive decision-making process. For every possible move, it simulates game outcomes, assigning scores to terminal states:

- A win for the AI ('X') scores **+1**.
- A win for the human opponent ('O') scores **–1**.
- A draw scores **0**.

This systematic evaluation ensures that the AI selects the move that maximizes its chances of winning while minimizing potential losses.

2.3 Alpha-Beta Pruning Optimization

Alpha-beta pruning streamlines the minimax algorithm by eliminating branches in the game tree that won't influence the final decision. By maintaining two thresholds—**alpha** (the best score for the maximizing player) and **beta** (the best score for the minimizing player)—the algorithm skips further exploration on irrelevant branches. This technique dramatically reduces computation time while retaining the optimal decision found by exhaustive search.

---

3. System Architecture and Methodology

3.1 Overview of the Project Structure

The project is organized into several functional modules:

- **Board Representation:** The game board is a list of nine strings representing cells.
- **Display Functions:** A dedicated function renders the board into a human-friendly 3×3 grid.
- **Evaluation Functions:** These functions check for win conditions, draw states, and provide board evaluation scores.
- **Decision Making:** The core of the project uses the minimax algorithm enhanced with alpha-beta pruning to evaluate moves.
- **Game Loop:** A main loop alternates turns between the AI and the human player, updating and displaying the board continuously until termination.

3.2 Development Environment

Developed in Python, the project takes advantage of Python's readability and dynamic features. Python's support for recursion and list manipulation makes it an excellent choice for implementing the game logic and AI search algorithm. The code is designed to be modular, facilitating future enhancements like graphical user interfaces or integration with machine learning models.

---

4. In-Depth Code Walkthrough

Code-

```
import random

def print_board(board):
    print("-------------")
    for i in range(3):
        print("|", board[i*3], "|", board[i*3 + 1], "|", board[i*3 + 2], "|")
        print("-------------")

def check_win(board, player):
    win_combinations = [(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 combo in win_combinations:
        if all(board[i] == player for i in combo):
            return True
    return False

def check_draw(board):
    return all(cell != ' ' for cell in board)

def evaluate(board):
    if check_win(board, 'X'):
        return 1
```

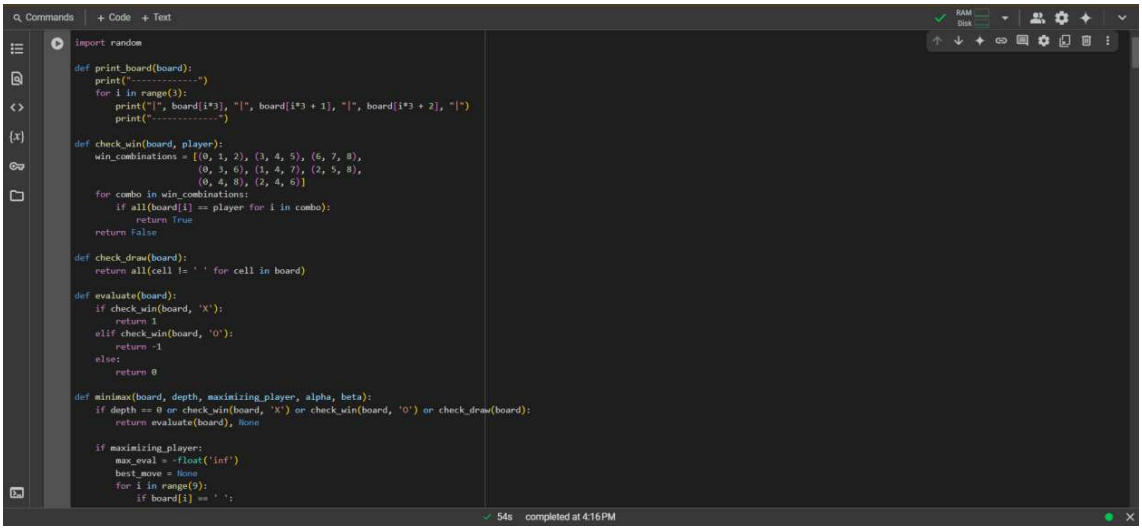
```
    elif check_win(board, 'O'):
        return -1
    else:
        return 0

def minimax(board, depth, maximizing_player, alpha, beta):
    if depth == 0 or check_win(board, 'X') or check_win(board, 'O') or check_draw(board):
        return evaluate(board), None

    if maximizing_player:
        max_eval = -float('inf')
        best_move = None
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'X'
                eval, _ = minimax(board, depth - 1, False, alpha, beta)
                board[i] = ' '
                if eval > max_eval:
                    max_eval = eval
                    best_move = i
                alpha = max(alpha, eval)
                if beta <= alpha:
                    break
        return max_eval, best_move
    else:
        min_eval = float('inf')
```

```
        best_move = None
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'O'
                eval, _ = minimax(board, depth - 1, True, alpha, beta)
                board[i] = ' '
                if eval < min_eval:
                    min_eval = eval
                    best_move = i
                beta = min(beta, eval)
                if beta <= alpha:
                    break
        return min_eval, best_move

def ai_move(board):
  _, move = minimax(board, 9, True, -float('inf'), float('inf'))
  return move

def play_game():
    board = [' '] * 9
    current_player = 'X'

    while True:
        print_board(board)
        if current_player == 'X':
            move = ai_move(board)
```

```
            print(f"AI plays at position {move + 1}")
            board[move] = current_player
        else:
            while True:
                try:
                    move = int(input("Enter your move (1-9): ")) - 1
                    if 0 <= move <= 8 and board[move] == ' ':
                        board[move] = current_player
                        break
                    else:
                        print("Invalid move. Try again.")
                except ValueError:
                    print("Invalid input. Please enter a number.")

        if check_win(board, current_player):
            print_board(board)
            print(f"{current_player} wins!")
            break
        elif check_draw(board):
            print_board(board)
            print("It's a draw!")
            break

        current_player = 'O' if current_player == 'X' else 'X'

if __name__ == "__main__":
```

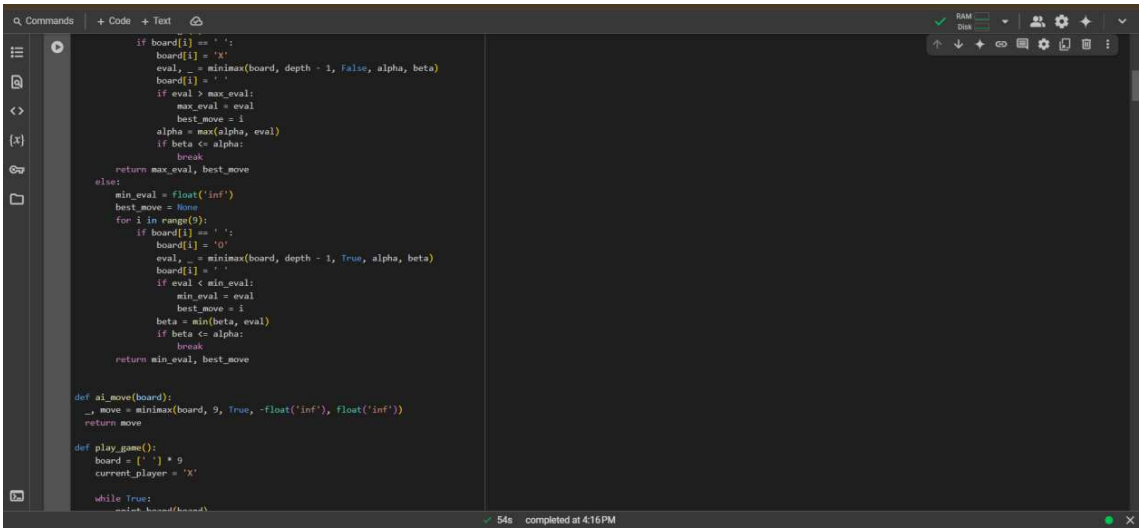
play_game()

Screenshots-



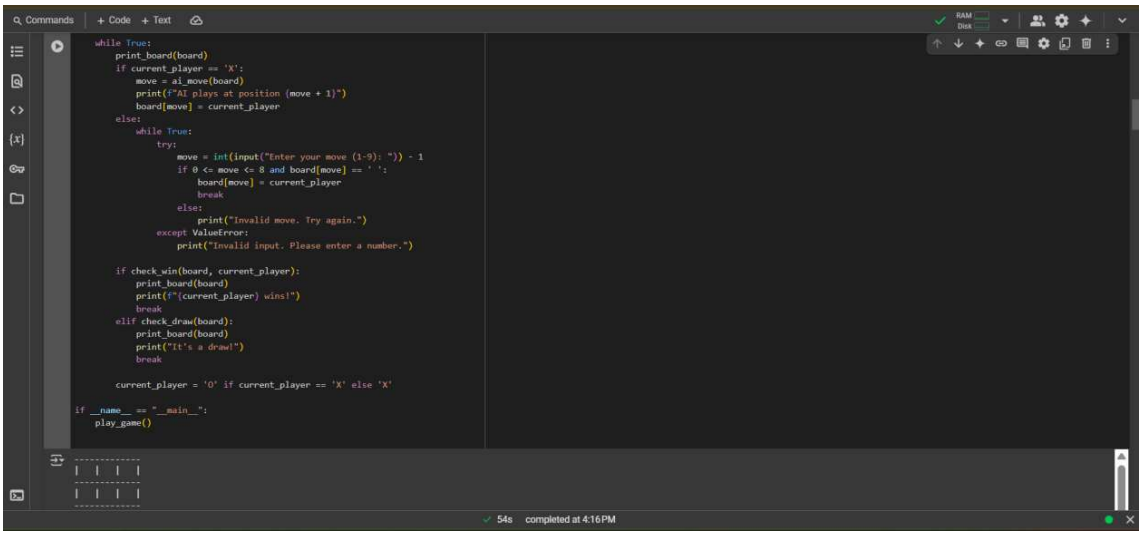
```
import random

def print_board(board):
    print("-------------")
    for i in range(3):
        print("|", board[i*3], "|", board[i*3 + 1], "|", board[i*3 + 2], "|")
        print("-------------")

def check_win(board, player):
    win_combinations = [(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 combo in win_combinations:
        if all(board[i] == player for i in combo):
            return True
    return False

def check_draw(board):
    return all(cell != ' ' for cell in board)

def evaluate(board):
    if check_win(board, 'X'):
        return 1
    elif check_win(board, 'O'):
        return -1
    else:
        return 0

def minimax(board, depth, maximizing_player, alpha, beta):
    if depth == 0 or check_win(board, 'X') or check_win(board, 'O') or check_draw(board):
        return evaluate(board), None

    if maximizing_player:
        max_eval = -float('inf')
        best_move = None
        for i in range(9):
            if board[i] == ' ':
```

54s completed at 4:16PM



```
            if board[i] == ' ':
                board[i] = 'X'
                eval, _ = minimax(board, depth - 1, False, alpha, beta)
                board[i] = ' '
                if eval > max_eval:
                    max_eval = eval
                    best_move = i
                alpha = max(alpha, eval)
                if beta <= alpha:
                    break
        return max_eval, best_move
    else:
        min_eval = float('inf')
        best_move = None
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'O'
                eval, _ = minimax(board, depth - 1, True, alpha, beta)
                board[i] = ' '
                if eval < min_eval:
                    min_eval = eval
                    best_move = i
                beta = min(beta, eval)
                if beta <= alpha:
                    break
        return min_eval, best_move


def ai_move(board):
    _, move = minimax(board, 9, True, -float('inf'), float('inf'))
    return move

def play_game():
    board = [' '] * 9
    current_player = 'X'

    while True:
```

54s completed at 4:16PM



```
    while True:
        print_board(board)
        if current_player == 'X':
            move = ai_move(board)
            print(f"AI plays at position {move + 1}")
            board[move] = current_player
        else:
            while True:
                try:
                    move = int(input("Enter your move (1-9): ")) - 1
                    if 0 <= move <= 8 and board[move] == ' ':
                        board[move] = current_player
                        break
                    else:
                        print("Invalid move. Try again.")
                except ValueError:
                    print("Invalid input. Please enter a number.")

        if check_win(board, current_player):
            print_board(board)
            print(f"{current_player} wins!")
            break
        elif check_draw(board):
            print_board(board)
            print("It's a draw!")
            break

        current_player = 'O' if current_player == 'X' else 'X'

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

```
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
```

54s completed at 4:16PM

```
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
AI plays at position 1
-------------
| X |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
Enter your move (1-9): 1
Invalid move. Try again.
Enter your move (1-9): 2
-------------
| X | O |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
AI plays at position 4
-------------
| X | O |   |
-------------
| X |   |   |
-------------
|   |   |   |
-------------
Enter your move (1-9): 4
Invalid move. Try again.
Enter your move (1-9): 8
-------------
| X | O |   |
-------------
| X |   |   |
```

54s completed at 4:16PM

```
Invalid move. Try again.
Enter your move (1-9): 8
-------------
| X | O |   |
-------------
| X |   |   |
-------------
|   | O |   |
-------------
AI plays at position 5
-------------
| X | O |   |
-------------
| X | X |   |
-------------
|   | O |   |
-------------
Enter your move (1-9): 2
Invalid move. Try again.
Enter your move (1-9): 6
-------------
| X | O |   |
-------------
| X | X | O |
-------------
|   | O |   |
-------------
AI plays at position 3
-------------
| X | O | X |
-------------
| X | X | O |
-------------
|   | O |   |
-------------
Enter your move (1-9): 7
-------------
| X | O | X |
-------------
| X | X | O |
```

54s completed at 4:16PM

4.1 Key Functions Overview

- **print_board(board):**
  Displays the Tic Tac Toe board on the console in a structured 3×3 format.
- **check_win(board, player):**
  Checks whether the specified player has achieved any of the winning conditions (rows, columns, diagonals).
- **check_draw(board):**
  Determines if the board is completely filled with no winners, thereby declaring a draw.
- **evaluate(board):**
  Returns an evaluation score based on the state: **+1** for an AI win, **–1** for a human win, **0** for no terminal state.
- **minimax(board, depth, maximizing_player, alpha, beta):**
  Recursively explores the game tree to select the optimal move, implementing alpha-beta pruning to skip unnecessary evaluations.
- **ai_move(board) and play_game():**
  These functions integrate decision-making into the game loop and manage user interaction.

4.2 Condensed Code Summary

Below is a shortened version of the code that highlights the core functionality without listing every detail. For a complete code listing, please refer to the full project repository.

```
def print_board(board):
    # Render the board as a 3x3 grid
    for i in range(3):
        print("|", board[i*3], board[i*3+1], board[i*3+2], "|")

def check_win(board, player):
    # Check all winning combinations (rows, columns, diagonals)
    combos = [(0,1,2), (3,4,5), (6,7,8), (0,3,6), (1,4,7), (2,5,8), (0,4,8), (2,4,6)]
    return any(all(board[i] == player for i in combo) for combo in combos)

def evaluate(board):
    # Returns +1, -1, or 0 based on the board state
    if check_win(board, 'X'):
        return 1
    elif check_win(board, 'O'):
        return -1
    return 0

def minimax(board, depth, is_max, alpha, beta):
    # Base condition: terminal state reached or depth limit attained.
    if depth == 0 or check_win(board, 'X') or check_win(board, 'O') or ' ' not in
board:
        return evaluate(board), None
```

```
# Recursive exploration with alpha-beta pruning
best_move = None
if is_max:
    max_eval = -float('inf')
    for i, cell in enumerate(board):
        if cell == ' ':
            board[i] = 'X'
            val, _ = minimax(board, depth - 1, False, alpha, beta)
            board[i] = ' '
            if val > max_eval:
                max_eval, best_move = val, i
            alpha = max(alpha, val)
            if beta <= alpha:
                break
    return max_eval, best_move
else:
    min_eval = float('inf')
    for i, cell in enumerate(board):
        if cell == ' ':
            board[i] = 'O'
            val, _ = minimax(board, depth - 1, True, alpha, beta)
            board[i] = ' '
```

```
            if val < min_eval:
                min_eval, best_move = val, i
            beta = min(beta, val)
            if beta <= alpha:
                break
        return min_eval, best_move


def play_game():
    board = [' '] * 9
    player = 'X'
    while True:
        print_board(board)
        if player == 'X':
            _, move = minimax(board, 9, True, -float('inf'), float('inf'))
            board[move] = 'X'
        else:
            move = int(input("Enter move (1-9): ")) - 1
            if board[move] == ' ':
                board[move] = 'O'
        if check_win(board, player):
            print_board(board)
            print(player + " wins!")
```

```
            break
        elif ' ' not in board:
            print_board(board)
            print("It's a draw!")
            break
        player = 'O' if player == 'X' else 'X'
```

Note: The above snippet illustrates the core logic. The full code includes additional error checking and interface improvements.

---

5. Algorithmic Analysis and Complexity

5.1 Recursive Decision Making

The minimax algorithm explores every possible future move, simulating the entire game tree for Tic Tac Toe. Given the small game space (at most 9 moves), recursion remains efficient even with a complete search.

5.2 Performance Improvements with Alpha-Beta Pruning

Alpha-beta pruning dramatically reduces the number of nodes evaluated by the recursive search, meaning that many branches in the game tree are eliminated once it is clear they cannot yield a better outcome than what has already been found. This optimization ensures a near-optimal response time even in a traditionally exhaustive search.

5.3 Complexity Analysis

- **Time Complexity:** The worst-case performance without pruning is $O(b^d)$, where b is the branching factor (≤ 9) and d is the maximum depth (≤ 9). With alpha-beta pruning, the effective time complexity is significantly reduced.
- **Space Complexity:** Primarily determined by the recursion depth, the space complexity remains O(d).

---

6. Case Studies and Scenario-Based Evaluation

6.1 Optimal AI Play

The AI, employing minimax with alpha-beta pruning, always forces either a win or a draw when playing optimally. In practice, when a human opponent makes a suboptimal move, the AI exploits the mistake to pursue a win.

6.2 Dynamics Between Human and AI

- **Aggressive vs. Defensive Strategies:** The AI adapts based on the opponent's moves, balancing offense and defense.
- **Learning Insights:** Although the AI is not self-learning, the observed interactions provide insight into game dynamics and possible areas for further development or heuristic adjustments.

---

7. Potential Enhancements and Future Work

Future directions for this project could include:

- **Graphical User Interface (GUI):** Enhancing user experience with a visual interface using Python frameworks like Tkinter.
- **Adaptive Difficulty Levels:** Adjusting minimax search depth dynamically to cater to varying player skill levels.
- **Machine Learning Integration:** Experimenting with reinforcement learning to observe how adaptive algorithms can play the game.
- **Expansion to More Complex Games:** Adapting the current architecture to larger games like Connect Four or Chess for a deeper challenge.
- **Performance Monitoring:** Integrating logging and analytics for further understanding and optimization of the decision-making process.

---

8. Conclusion

This project has demonstrated the practical application of the minimax algorithm enhanced by alpha-beta pruning in creating an AI that plays Tic Tac Toe. Through a modular code design, we have covered board representation, game state evaluation, and recursive decision-making, ensuring that our AI plays optimally.

The theoretical discussions on game theory, search optimization, and algorithmic complexity reinforce the underlying principles and provide a strong foundation for those looking to extend these concepts to more complex scenarios. This document not only acts as a project report but also as a guide to understanding and applying AI techniques in strategic game development.

---

9. Appendices

Appendix A: Code Summary

A condensed version of the project code (see Section 4.2) encapsulates the essential functionality of our Tic Tac Toe game AI. For a detailed exploration and additional improvements (such as robust error handling and interface polish), readers are encouraged to refer to the complete code repository attached as supplementary material.

Appendix B: References & Further Reading

- Research papers and articles on minimax, alpha-beta pruning, and AI game strategy.
- Python documentation on recursion and functional programming.
- Tutorials on developing interactive Python applications with both CLI and GUI.

This comprehensive report outlines the theoretical and practical aspects of building an AI for Tic Tac Toe. It serves both as an academic study and a practical guide, paving the way for future explorations in artificial intelligence and algorithmic strategies.

Would you be interested in discussing ideas for extending this project—for instance, integrating a GUI or adapting the algorithm for more complex games?