REPORT

TITLE PAGE :

Development of an AI-based Noughts and Crosses (Tic-Tac-Toe) Game using Minimax Algorithm with Alpha-Beta Pruning

Problem Statement:
Tic-Tac-Toe (also known as Noughts and Crosses) is a two-player game where players alternately place their symbols ("X" and "O") on a 3×3 grid. The goal is to align three of their symbols in a row, column, or diagonal. While the game may seem simple, developing an AI that can play optimally involves complex decision-making. This project aims to create an AI-based solution using the Minimax algorithm with Alpha-Beta Pruning to make efficient decisions and improve performance by reducing the number of explored game states. The AI should maximize its chances of winning while minimizing the opponent's chances.

Name: Shivansh Gupta

Roll No.:202401100300236

Course: Computer Science and Engineering (AI)-(B-Tech)

Institute: KIET GROUP OF INSTITUTION

Introduction:

Tic-Tac-Toe is a classic and well-known two-player game that involves strategy and logical thinking. The game is played on a 3×3 grid, where two players take turns marking the spaces with "X" and "O." The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal line wins the game. If all nine spaces are filled without a winner, the game ends in a draw.

Despite its simplicity, the game involves complex decision-making due to the high number of possible game states (over 255,168 unique board configurations). Developing an AI player for Tic-Tac-Toe requires the ability to analyze all possible moves and select the optimal one.

Why Minimax Algorithm?

The Minimax algorithm is a backtracking algorithm used for decision-making in two-player games. It works by simulating all possible future moves and assigning scores based on the outcome:

- **+10** → If the AI wins.
- **-10** → If the opponent wins.
- **0** → If the game is a draw.

The AI selects the move that maximizes its chances of winning while minimizing the opponent's chances. The algorithm explores the entire game tree to make the best decision. However, this process becomes computationally expensive as the search space increases.

Why Alpha-Beta Pruning?

Alpha-Beta Pruning is an optimization technique that improves the efficiency of the Minimax algorithm by eliminating branches that cannot influence the final decision.

- **Alpha** → Best score that the maximizer (AI) can guarantee.
- **Beta** → Best score that the minimizer (opponent) can guarantee.
- If at any point **alpha ≥ beta**, further exploration of the branch is stopped because it cannot affect the final decision.

By pruning irrelevant branches, Alpha-Beta Pruning significantly reduces the number of explored nodes, making the algorithm faster and more efficient without affecting the outcome.

Project Motivation

The motivation behind this project is to explore the fundamentals of AI-based decision-making using classic algorithms. While Tic-Tac-Toe is a simple game, the underlying logic used in Minimax and Alpha-Beta Pruning forms the foundation of more complex AI systems used in games like Chess and Go. The project serves as a practical demonstration of AI techniques such as:

- ☑ State-space exploration
- ☑ Game tree traversal
- ☑ Optimal decision-making
- ☑ Search space reduction using pruning

This project will help deepen the understanding of game theory, algorithm design, and AI decision-making strategies. Furthermore, it provides a strong foundation for developing more complex AI models in real-world scenarios.

Challenges and Complexity

- The number of possible board states is relatively low for Tic-Tac-Toe (255,168).
- However, the challenge lies in ensuring that the AI makes the optimal decision without increasing computational overhead.
- Alpha-Beta Pruning addresses this by cutting off unnecessary branches, allowing the AI to make faster decisions

Code:

```python
import math

# Constants for player symbols
PLAYER = 'X'  # Maximizing player (AI)
OPPONENT = 'O'  # Minimizing player (Human)

# Function to print the current state of the board
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 5)

# Function to check for a winner
def check_winner(board):
    # Check rows for winner
    for row in board:
        if row.count(row[0]) == 3 and row[0] != ' ':
            return row[0]

    # Check columns for winner
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != ' ':
            return board[0][col]

    # Check diagonals for winner
    if board[0][0] == board[1][1] == board[2][2] != ' ':
        return board[0][0]

    if board[0][2] == board[1][1] == board[2][0] != ' ':
```

```python
            return board[0][2]

    # No winner yet
    return None

# Function to check if the game is a draw (board is full)
def is_draw(board):
    for row in board:
        if ' ' in row:
            return False
    return True

# Function to evaluate the board score
def evaluate(board):
    winner = check_winner(board)
    if winner == PLAYER:
        return 10  # AI wins
    elif winner == OPPONENT:
        return -10  # Human wins
    return 0  # Draw or game still ongoing

# Minimax algorithm with Alpha-Beta Pruning
def minimax(board, depth, is_maximizing, alpha, beta):
    score = evaluate(board)

    # If the game is won or drawn, return the score
    if score == 10 or score == -10:
        return score
```

```python
    if is_draw(board):
        return 0

    if is_maximizing:
        # AI's turn (maximize score)
        best = -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    # Try the move
                    board[i][j] = PLAYER
                    # Recur to check the outcome of this move
                    best = max(best, minimax(board, depth + 1, False, alpha, beta))
                    # Undo the move
                    board[i][j] = ' '
                    # Update alpha
                    alpha = max(alpha, best)
                    # Prune if beta <= alpha (no need to check other branches)
                    if beta <= alpha:
                        break
        return best
    else:
        # Opponent's turn (minimize score)
        best = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    # Try the move
                    board[i][j] = OPPONENT
```

```python
                    # Recur to check the outcome of this move
                    best = min(best, minimax(board, depth + 1, True, alpha, beta))
                    # Undo the move
                    board[i][j] = ' '
                    # Update beta
                    beta = min(beta, best)
                    # Prune if beta <= alpha
                    if beta <= alpha:
                        break
        return best

# Function to find the best move for the AI
def find_best_move(board):
    best_val = -math.inf
    best_move = (-1, -1)

    # Try every possible move and evaluate using minimax
    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                # Try the move
                board[i][j] = PLAYER
                # Get the score of this move using minimax
                move_val = minimax(board, 0, False, -math.inf, math.inf)
                # Undo the move
                board[i][j] = ' '

                # If this move is better than previous best, update best move
                if move_val > best_val:
```

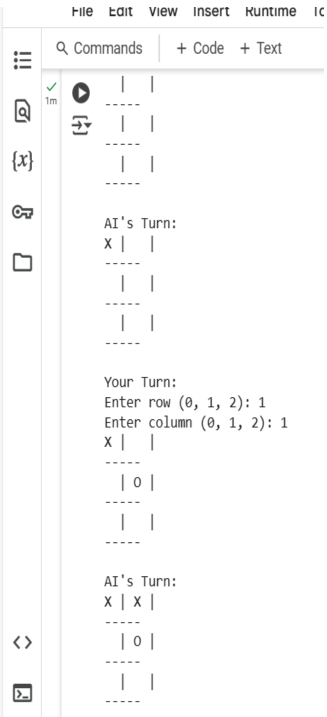
```python
                    best_val = move_val
                    best_move = (i, j)

    return best_move

# Main game loop
def play_game():
    # Initialize empty board
    board = [[' ' for _ in range(3)] for _ in range(3)]
    print_board(board)

    for turn in range(9):
        if turn % 2 == 0:
            # AI's turn (Maximizing player)
            print("\nAI's Turn:")
            i, j = find_best_move(board)
            board[i][j] = PLAYER
        else:
            # Human's turn (Minimizing player)
            print("\nYour Turn:")
            while True:
                # Get user input for row and column
                row = int(input("Enter row (0, 1, 2): "))
                col = int(input("Enter column (0, 1, 2): "))
                if 0 <= row < 3 and 0 <= col < 3 and board[row][col] == ' ':
                    board[row][col] = OPPONENT
                    break
                else:
                    print("Invalid move! Try again.")
```

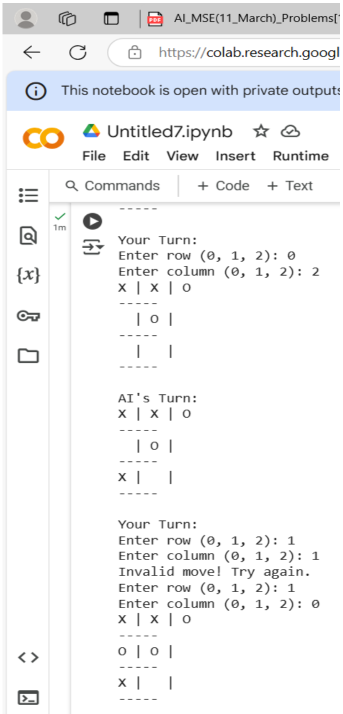
```python
        # Display the board after each turn
        print_board(board)

        # Check for a winner after each move
        winner = check_winner(board)
        if winner:
            if winner == PLAYER:
                print("\nAI wins! 😎")
            else:
                print("\nYou win! 🎉")
            return

        # Check for a draw
        if is_draw(board):
            print("\nIt's a draw! 😐")
            return

# Start the game
if __name__ == "__main__":
    play_game()
```

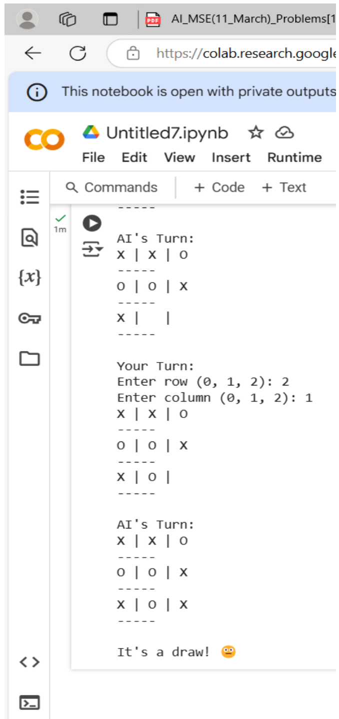
```
  |   |
-----
  |   |
-----
  |   |
-----
```

Output/Result:

Example Output:







References/Credits:

ChatGPT – ChatGPT played a key role in various aspects of the project:

- Code Generation: Provided the initial structure for the Minimax algorithm with Alpha-Beta Pruning, helping to set up the recursive logic and game state evaluation.
- Optimization: Suggested improvements to the Alpha-Beta Pruning logic, reducing the search space and enhancing the AI's decision-making efficiency.
- Debugging: Helped identify and resolve coding issues, such as handling edge cases and improving the recursive depth management.
- Documentation: Assisted in writing well-organized and professional documentation, including the report structure and detailed explanations of the algorithm.
- Performance Tuning: Recommended changes to optimize the algorithm’s speed and minimize computational overhead without affecting accuracy.