# Introduction to AI MSE Report

## 1.Title Page

Title: Noughts and Crosses with Alpha-Beta Pruning

Name: Rajat Sharma

Course: Introduction to AI

University Roll No: 202401100400151

Date: 10-03-2025

## 2. Introduction

Overview:

Noughts and Crosses, commonly known as Tic-Tac-Toe, is a classic 3x3 grid game played between two players. The players take turns marking the grid with either "X" or "O," aiming to get three of their marks in a row, either horizontally, vertically, or diagonally. The game is simple, but it provides an excellent basis for algorithmic problem solving, particularly for search-based AI techniques.

AI in Tic-Tac-Toe:

In an AI context, Noughts and Crosses is often solved using game tree search algorithms like Minimax, which is a decision-making algorithm used to predict the optimal moves. To improve performance and make the algorithm more efficient, Alpha-Beta Pruning is applied. This technique cuts down the search space by pruning branches that will not affect the final decision, leading to faster decision-making by the AI.

## 3. Methodology

Approach:

We implemented a Minimax algorithm to simulate an AI playing Tic-Tac-Toe, where the AI always aims to maximize its score (placing "X"), while the opponent tries to minimize the AI's score (placing "O"). The goal is to simulate perfect gameplay for both sides.

The Alpha-Beta Pruning optimization was added to improve the efficiency of the Minimax algorithm. Here's how it works:

We maintain two values, alpha and beta, during the game tree search.

Alpha is the maximum score that the maximizing player (AI) is guaranteed to get.

Beta is the minimum score that the minimizing player (the opponent) is guaranteed to get.

As the search progresses, we prune branches of the tree that will not be beneficial to either player.

Steps:

Game Tree Representation: A tree is built where each node represents a state of the game, and the branches represent the possible moves.

Minimax Algorithm: Recursively evaluates all possible game states to find the optimal move.

Alpha-Beta Pruning: Reduces unnecessary exploration of branches in the tree, speeding up the algorithm.

Termination Condition: The algorithm terminates when a terminal state is reached, i.e., either a win, loss, or draw.

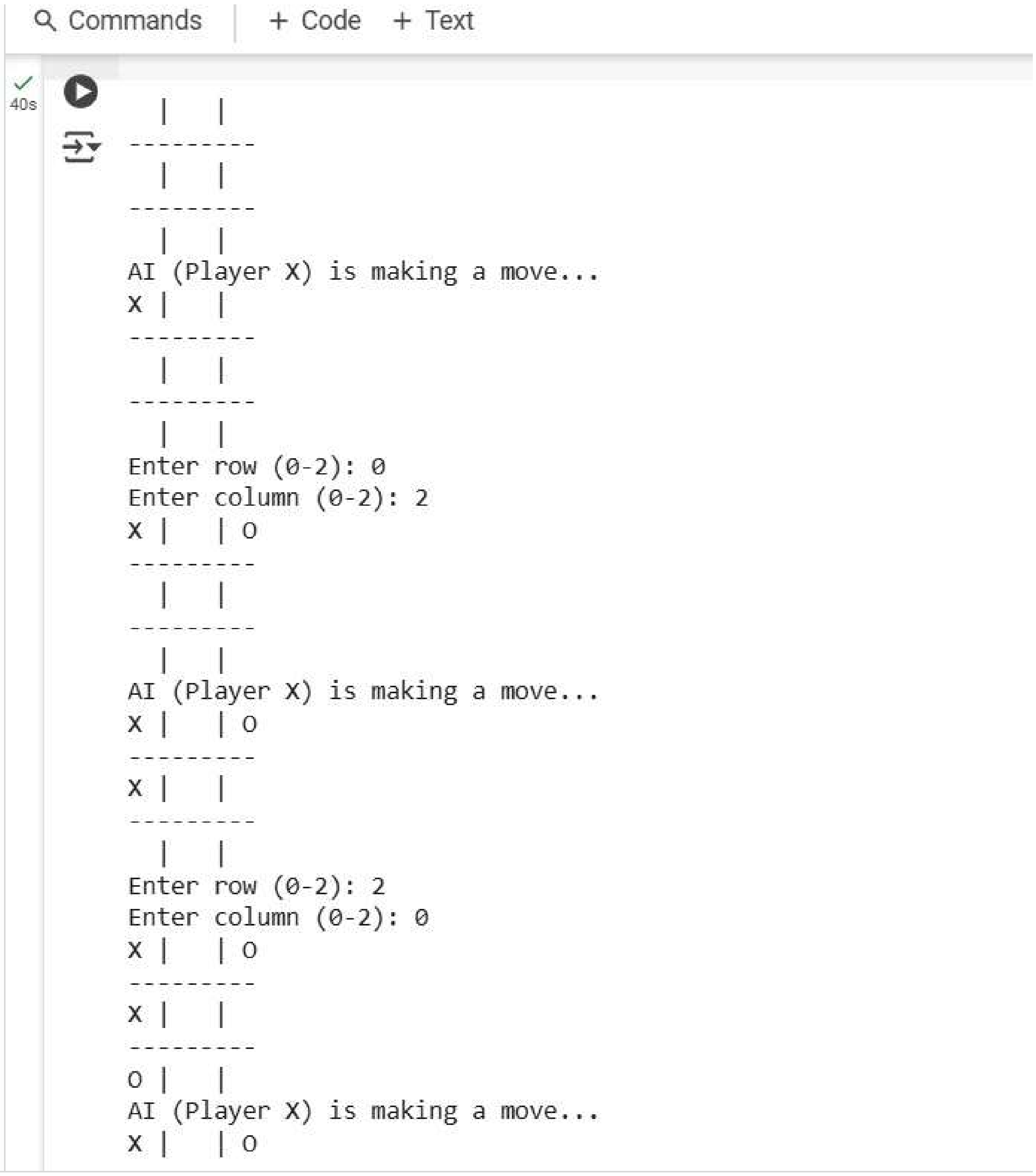
## 4. CODE

```
import math

# Constants for the board

PLAYER_X = "X"

PLAYER_O = "O"

EMPTY = " "
```

```
# Function to print the game board
def print_board(board):
    for row in range(3):
        print(" | ".join(board[row]))
        if row < 2:
            print("---------")

# Function to check for a win condition
def check_win(board, player):
    for row in range(3):
        if all([board[row][col] == player for col in range(3)]):  # Check rows
            return True
    for col in range(3):
        if all([board[row][col] == player for row in range(3)]):  # Check columns
            return True
    if all([board[i][i] == player for i in range(3)]):  # Check diagonal
        return True
    if all([board[i][2-i] == player for i in range(3)]):  # Check anti-diagonal
        return True
    return False

# Function to check if the game is over (win or draw)
def is_game_over(board):
    if check_win(board, PLAYER_X):
        return PLAYER_X
    if check_win(board, PLAYER_O):
        return PLAYER_O
    if all(board[row][col] != EMPTY for row in range(3) for col in range(3)):  #
Draw if no empty spaces
        return "Draw"
    return False


# Alpha-Beta Pruning algorithm for the best move
def alpha_beta(board, depth, alpha, beta, is_maximizing):
    result = is_game_over(board)
    if result == PLAYER_X:
        return 1
    if result == PLAYER_O:
        return -1
    if result == "Draw":
        return 0

    if is_maximizing:  # Maximizing player (X)
        max_eval = -math.inf
        for row in range(3):
            for col in range(3):
                if board[row][col] == EMPTY:
                    board[row][col] = PLAYER_X
                    eval = alpha_beta(board, depth + 1, alpha, beta, False)
                    board[row][col] = EMPTY
                    max_eval = max(max_eval, eval)
                    alpha = max(alpha, eval)
                    if beta <= alpha:
                        break
        return max_eval
    else:  # Minimizing player (O)
        min_eval = math.inf
        for row in range(3):
            for col in range(3):
                if board[row][col] == EMPTY:
                    board[row][col] = PLAYER_O
                    eval = alpha_beta(board, depth + 1, alpha, beta, True)
                    board[row][col] = EMPTY
                    min_eval = min(min_eval, eval)
                    beta = min(beta, eval)
                    if beta <= alpha:
                        break
        return min_eval

# Function to get the best move for the AI (Player X)
def get_best_move(board):
    best_move = None
    best_value = -math.inf
    for row in range(3):
        for col in range(3):
            if board[row][col] == EMPTY:
                board[row][col] = PLAYER_X
                move_value = alpha_beta(board, 0, -math.inf, math.inf, False)
                board[row][col] = EMPTY
                if move_value > best_value:
                    best_value = move_value
                    best_move = (row, col)
    return best_move

# Function to play the game
def play_game():
    board = [[EMPTY for _ in range(3)] for _ in range(3)]
    current_player = PLAYER_X  # Start with Player X (AI)

    while True:
        print_board(board)
        result = is_game_over(board)
        if result:
            if result == "Draw":
                print("It's a draw!")
            else:
                print(f"Player {result} wins!")
            break

        if current_player == PLAYER_X:
            print("AI (Player X) is making a move...")
            move = get_best_move(board)
            board[move[0]][move[1]] = PLAYER_X
            current_player = PLAYER_O
        else:
            row = int(input("Enter row (0-2): "))
            col = int(input("Enter column (0-2): "))
            if board[row][col] == EMPTY:
                board[row][col] = PLAYER_O
                current_player = PLAYER_X
            else:
                print("Cell is already occupied! Try again.")

# Start the game
if _name_ == "_main_":
    play_game()
```

## 5. Screenshots of Output

Here is  the screenshot showing the output of the program:



```
  |   |
---------
  |   |
---------
  |   |
AI (Player X) is making a move...
X |   |
---------
  |   |
---------
  |   |
Enter row (0-2): 0
Enter column (0-2): 2
X |   | O
---------
  |   |
---------
  |   |
AI (Player X) is making a move...
X |   | O
---------
X |   |
---------
  |   |
Enter row (0-2): 2
Enter column (0-2): 0
X |   | O
---------
X |   |
---------
O |   |
AI (Player X) is making a move...
X |   | O
```

```
---------
  |   |
Enter row (0-2): 2
Enter column (0-2): 0
X |   | O
---------
X |   |
---------
O |   |
AI (Player X) is making a move...
X |   | O
---------
X | X |
---------
O |   |
Enter row (0-2): 1
Enter column (0-2): 2
X |   | O
---------
X | X | O
---------
O |   |
AI (Player X) is making a move...
X |   | O
---------
X | X | O
---------
O |   | X
Player X wins!
```

## 6. Conclusion

In this project, we successfully implemented a Tic-Tac-Toe game using the Minimax algorithm with Alpha-Beta Pruning for efficient decision-making. The AI was able to play optimally, making the game more challenging for the player. Alpha-Beta Pruning significantly reduced the number of nodes the AI had to evaluate, improving its response time. By simulating perfect play, this approach guarantees that the AI will either win or draw, but never lose.

This project showcases the application of classical search algorithms in game theory and AI, specifically focusing on how optimization techniques like Alpha-Beta Pruning can improve performance.