www.kiet.edu
Delhi-NCR, Ghaziabad

KIET
GROUP OF INSTITUTIONS
Connecting Life with Learning

REPORT ON NOUGHTS AND CROSSES WITH ALPHA-BETA PRUNING

Submitted by: NITISH SABLOK
Course: INTRODUCTION TO AI
Date: 11/03/2025

1. Introduction

☑ Objective

This project's goal is to use the Alpha-Beta Pruning algorithm to build a game of Noughts and Crosses (Tic-Tac-Toe). The project shows how AI techniques may be used to build an opponent that is impossible to beat and plays at their best.

☑ Background

In the board game Noughts and Crosses, two players alternately mark spots in a 3x3 grid. The game is won by the person who is able to line up three of their markers in a row, either diagonally, upwards or horizontally.

AI that makes decisions and plays games frequently uses the Minimax algorithm. An optimization method for the minimax algorithm called alpha-beta pruning lowers the number of nodes the search tree evaluates, increasing the AI's efficiency.

☑ Scope

The following goals are the main emphasis of this project:

- Create a Noughts and Crosses game that works flawlessly.
- Employing Alpha-Beta Pruning, create an AI adversary.
- Make sure the AI performs at its best and is unbeatable.
- Offer an intuitive user interface so that the player can engage with the game..

---

3. Methodology

☑ 3.1 Game Rules

• A 3x3 grid is used to play the game.

• The AI is given the symbol 'O', whereas the player is given 'X'.

•The placement of the player's and AI's symbols on the grid alternates.

• The game is won by the first player to line up three symbols in a row, column, or diagonal.

• The game is deemed a tie if every cell is filled and there is no winner.

☑ 3.2 Algorithm Used

Minimax Algorithm:

• Minimax is a recursive algorithm that maximizes winning chances in two-player games.

• It considers every move that may be made and, depending on the result, chooses the best one.

• If AI prevails, the system gives it a score of o + 10.

• If the player wins, it is -10; if it is a draw, it is 0.

Alpha-Beta Pruning:

By disregarding branches that have no bearing on the ultimate choice, alpha-beta pruning, an optimization over minimax, lowers the number of nodes assessed in the game tree.

• Alpha: The maximum value that the maximizer can ensure.

• Beta: The best value that can be guaranteed by the minimizer.

☑ 3.3 Implementation Approach

1. Using a 2D list, make a 3x3 board.
2. Create a function to assess the board's condition.
3. Use alpha-beta pruning in conjunction with the minimax method.
4. Develop functions to manage AI movements and player input.
5. After every turn, display the board.
6. Look for winning circumstances and show the outcome.

CODE:-

```
import math

# Constants for player, AI, and empty spaces
PLAYER = 'X'
AI = 'O'
EMPTY = ' '

# Function to print the board in a readable format
def print_board(board):
    for row in board:
        print(' | '.join(row))  # Print row elements separated by '|'
        print('-' * 5)  # Print horizontal line after each row

# Function to check if there are any empty cells left on the board
def is_moves_left(board):
    for row in board:
        if EMPTY in row:
            return True
    return False

# Function to evaluate the current board state
def evaluate(board):
    # Check rows for a win
    for row in board:
        if row.count(PLAYER) == 3:
            return -10  # Player wins
        if row.count(AI) == 3:
            return 10  # AI wins

    # Check columns for a win
    for col in range(3):
        if all(board[row][col] == PLAYER for row in range(3)):
            return -10  # Player wins
        if all(board[row][col] == AI for row in range(3)):
            return 10  # AI wins

    # Check main diagonal for a win
    if all(board[i][i] == PLAYER for i in range(3)):
        return -10  # Player wins
    if all(board[i][i] == AI for i in range(3)):
        return 10  # AI wins

    # Check anti-diagonal for a win
    if all(board[i][2 - i] == PLAYER for i in range(3)):
        return -10  # Player wins
```

```
    if all(board[i][2 - i] == AI for i in range(3)):
        return 10  # AI wins

    # No winner yet — return 0
    return 0

# Alpha-Beta pruning function using the minimax algorithm
def alpha_beta(board, depth, alpha, beta, is_maximizing):
    score = evaluate(board)

    # If AI wins, return a positive score
    if score == 10:
        return score
    # If player wins, return a negative score
    if score == -10:
        return score
    # If no moves left, it's a draw — return 0
    if not is_moves_left(board):
        return 0

    if is_maximizing:
        best = -math.inf  # Start with lowest possible score
        for i in range(3):
            for j in range(3):
                if board[i][j] == EMPTY:
                    board[i][j] = AI  # Try AI's move
                    best = max(best, alpha_beta(board, depth + 1, alpha, beta, False))
                    board[i][j] = EMPTY  # Undo the move
                    alpha = max(alpha, best)  # Update alpha
                    if beta <= alpha:  # Prune branches if beta <= alpha
                        break
        return best
    else:
        best = math.inf  # Start with highest possible score
        for i in range(3):
            for j in range(3):
                if board[i][j] == EMPTY:
                    board[i][j] = PLAYER  # Try player's move
                    best = min(best, alpha_beta(board, depth + 1, alpha, beta, True))
                    board[i][j] = EMPTY  # Undo the move
                    beta = min(beta, best)  # Update beta
                    if beta <= alpha:  # Prune branches if beta <= alpha
                        break
        return best

# Function to find the best move for AI using alpha-beta pruning
def find_best_move(board):
    best_val = -math.inf
```

```
    best_move = (-1, -1)

    # Try every empty spot and calculate its score using alpha-beta pruning
    for i in range(3):
        for j in range(3):
            if board[i][j] == EMPTY:
                board[i][j] = AI
                move_val = alpha_beta(board, 0, -math.inf, math.inf, False)
                board[i][j] = EMPTY  # Undo the move

                # If this move is better than previous best, update best move
                if move_val > best_val:
                    best_val = move_val
                    best_move = (i, j)

    return best_move

# Function to handle player's input with validation
def player_move(board):
    while True:
        try:
            # Take input from user and split into row and column
            values = input("Enter row and column (0-2) separated by a space: ").split()
            if len(values) != 2:
                raise ValueError("Please enter two numbers separated by a space.")

            row, col = map(int, values)

            # Check if row and column are within valid range
            if row not in range(3) or col not in range(3):
                raise ValueError("Row and column must be between 0 and 2.")
            if board[row][col] != EMPTY:
                raise ValueError("Cell is already occupied. Try again.")
            return row, col
        except ValueError as e:
            print(f"Invalid input: {e}")

# Function to play the game
def play_game():
    # Initialize empty 3x3 board
    board = [[EMPTY for _ in range(3)] for _ in range(3)]
    print("\nWelcome to Noughts and Crosses!")
    print_board(board)

    while True:
        # Player's turn
        row, col = player_move(board)
        board[row][col] = PLAYER
```

```
        print_board(board)

        # Check if player wins
        if evaluate(board) == -10:
            print("🎉 You Win! 😎")
            break

        # Check for a draw
        if not is_moves_left(board):
            print("🤝 It's a Draw!")
            break

        # AI's turn
        print("🤖 AI is making a move...")
        row, col = find_best_move(board)
        board[row][col] = AI
        print_board(board)

        # Check if AI wins
        if evaluate(board) == 10:
            print("💀 AI Wins! 😈")
            break

        # Check for a draw
        if not is_moves_left(board):
            print("🤝 It's a Draw!")
            break

# Start the game when script is executed
if __name__ == "__main__":
    play_game()
```

OUTPUT 1 (DRAW CASE):-

```
Welcome to Noughts and Crosses!
  |   |
-----
  |   |
-----
  |   |
-----
Enter row and column (0-2) separated by a space: 2 2
  |   |
-----
  |   |
-----
  |   | X
-----
🤖 AI is making a move...
  |   |
-----
  | O |
-----
  |   | X
-----
Enter row and column (0-2) separated by a space: 1 2
  |   |
-----
  | O | X
-----
  |   | X
-----
🤖 AI is making a move...
  |   | O
-----
  | O | X
-----
```

```
  |   | X
-----
Enter row and column (0-2) separated by a space: 2 0
  |   | O
-----
  | O | X
-----
X |   | X
-----
🤖 AI is making a move...
  |   | O
-----
  | O | X
-----
X | O | X
-----
Enter row and column (0-2) separated by a space: 0 1
  | X | O
-----
  | O | X
-----
X | O | X
-----
🤖 AI is making a move...
O | X | O
-----
  | O | X
-----
X | O | X
-----
Enter row and column (0-2) separated by a space: 1 0
O | X | O
-----
X | O | X
-----
X | O | X
-----
🤝 It's a Draw!
```