TITLE PAGE

Problem Statement: In this problem, the game is played between a human player and an AI. The AI uses a minimax algorithm with alpha-beta pruning to make optimal moves, ensuring that the AI always tries to win or force a draw if it cannot win.

Name- Akshat Shandilya
Roll No- 202401100400025
Branch- CSEAIML
Section- A

INTRODUCTION

EXPLAINATION OF PROBLEM: Noughts and Crosses (commonly known as Tic-Tac-Toe) is a simple 2-player game played on a 3x3 grid. The objective of the game is to place three of your marks (either "X" or "O") in a row, column, or diagonal to win the game. The game ends when one player wins or when all cells in the grid are filled, resulting in a draw.

METHODOLOGY

APPROACH:

1. BOARD SETUP**:** Represent the board as a 3x3 grid.

2. GAME FLOW**:** Alternate turns between the human player and the AI, updating the board after each move.

3. MINIMAX ALGORITHM**:** The AI uses the minimax algorithm with alpha-beta pruning to evaluate all possible moves and choose the optimal one.

4. EVALUATING GAME STATES: Use the evaluate() function to determine the value of a board state based on whether the AI or player has won.

5. WINNER CHECK**:** After each move, check if there's a winner or if the game is a draw.

6. GAME CONTINUATION: Continue the game until there's a winner or the board is full.

CODE

```
#importing math library
import math

# Define constants
PLAYER_X = 'X'
PLAYER_O = 'O'
EMPTY = ' '

# The game board is a 3x3 grid
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 5)

# Check if the board is full
def is_full(board):
    for row in board:
        if EMPTY in row:
            return False
    return True

# Check if a player has won
def check_winner(board, player):

    # Check rows, columns, and diagonals
    for row in board:
        if all(s == player for s in row):
            return True
    for col in range(3):
        if all(board[row][col] == player for row in range(3)):
            return True
```

```
    if all(board[i][i] == player for i in range(3)):
        return True
    if all(board[i][2 - i] == player for i in range(3)):
        return True
    return False

# Evaluate the board state
def evaluate(board):
    if check_winner(board, PLAYER_X):
        return 10
    elif check_winner(board, PLAYER_O):
        return -10
    else:
        return 0

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

    # If the maximizer wins, return the score
    if score == 10:
        return score

    # If the minimizer wins, return the score
    if score == -10:
        return score

    # If the board is full, it's a tie
    if is_full(board):
        return 0

    # If it is the maximizer's (AI's) turn
    if is_maximizing_player:
        max_eval = -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == EMPTY:
```

```
                    board[i][j] = PLAYER_X
                    eval = minimax(board, depth + 1, alpha, beta, False)
                    max_eval = max(max_eval, eval)
                    alpha = max(alpha, eval)
                    board[i][j] = EMPTY
                    if beta <= alpha:
                        break
        return max_eval

    # If it is the minimizer's (player's) turn
    else:
        min_eval = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == EMPTY:
                    board[i][j] = PLAYER_O
                    eval = minimax(board, depth + 1, alpha, beta, True)
                    min_eval = min(min_eval, eval)
                    beta = min(beta, eval)
                    board[i][j] = EMPTY
                    if beta <= alpha:
                        break
        return min_eval

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

    # Try all possible moves for the AI
    for i in range(3):
        for j in range(3):
            if board[i][j] == EMPTY:
                board[i][j] = PLAYER_X
                move_val = minimax(board, 0, -math.inf, math.inf, False)
                board[i][j] = EMPTY
                if move_val > best_val:
```

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

    return best_move

# Play the game
def play_game():
    board = [[EMPTY for _ in range(3)] for _ in range(3)]
    current_player = PLAYER_O  # Player O starts

    while True:
        print_board(board)

        if current_player == PLAYER_X:
            print("AI's turn (X):")
            move = find_best_move(board)
            board[move[0]][move[1]] = PLAYER_X
        else:
            print("Player O's turn:")
            move = None
            while move is None:
                try:
                    row, col = map(int, input("Enter row and column (0-2)
separated by space: ").split())
                    if board[row][col] == EMPTY:
                        move = (row, col)
                        board[row][col] = PLAYER_O
                    else:
                        print("This spot is already taken, try again.")
                except (ValueError, IndexError):
                    print("Invalid input, please enter row and column (0-2).")

        # Check if the game has ended
        if check_winner(board, PLAYER_X):
            print_board(board)
            print("AI wins!")
            break
```

```
        elif check_winner(board, PLAYER_O):
            print_board(board)
            print("Player O wins!")
            break
        elif is_full(board):
            print_board(board)
            print("It's a draw!")
            break

        # Switch turns
        current_player = PLAYER_X if current_player == PLAYER_O else
PLAYER_O

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

OUTPUT OF THE CODE

Noughts and Crosses with Alpha-Beta Pruning

```
-----
Player O's turn:
Enter row and column (0-2) separated by space: 2 0
O |   |
-----
  | X |
-----
O |   |
-----
AI's turn (X):
O |   |
-----
X | X |
-----
O |   |
-----
Player O's turn:
Enter row and column (0-2) separated by space: 2 1
O |   |
-----
X | X |
-----
O | O |
-----
AI's turn (X):
O |   |
-----
X | X | X
-----
O | O |
-----
AI wins!
```

CREDITS

1. **Concept:** The game is based on the classic Tic-Tac-Toe (Noughts and Crosses) board game.

2. **AI Algorithm:** The AI opponent utilizes the Minimax Algorithm with Alpha-Beta Pruning to make optimal decisions during the game, ensuring a challenging opponent for the player.

DATASET USED:

Passenger	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	Braund, M	male	22	1	0	A/5 21171	7.25		S
2	1	1	Cumings, M	female	38	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen,	female	26	0	0	STON/O2.	7.925		S
4	1	1	Futrelle, M	female	35	1	0	113803	53.1	C123	S
5	0	3	Allen, Mr. '	male	35	0	0	373450	8.05		S
6	0	3	Moran, Mr	male		0	0	330877	8.4583		Q
7	0	1	McCarthy,	male	54	0	0	17463	51.8625	E46	S
8	0	3	Palsson, M	male	2	3	1	349909	21.075		S
9	1	3	Johnson, M	female	27	0	2	347742	11.1333		S
10	1	2	Nasser, Mr	female	14	1	0	237736	30.0708		C
11	1	3	Sandstrom	female	4	1	1	PP 9549	16.7	G6	S
12	1	1	Bonnell, M	female	58	0	0	113783	26.55	C103	S
13	0	3	Saunderco	male	20	0	0	A/5. 2151	8.05		S
14	0	3	Andersson	male	39	1	5	347082	31.275		S
15	0	3	Vestrom, M	female	14	0	0	350406	7.8542		S
16	1	2	Hewlett, M	female	55	0	0	248706	16		S
17	0	3	Rice, Mast	male	2	4	1	382652	29.125		Q
18	1	2	Williams, M	male		0	0	244373	13		S
19	0	3	Vander Pla	female	31	1	0	345763	18		S
20	1	3	Masselmai	female		0	0	2649	7.225		C
21	0	2	Fynney, Mr	male	35	0	0	239865	26		S
22	1	2	Beesley, M	male	34	0	0	248698	13	D56	S
23	1	3	McGowan	female	15	0	0	330923	8.0292		Q
24	1	1	Sloper, Mr	male	28	0	0	113788	35.5	A6	S
25	0	3	Palsson, M	female	8	3	1	349909	21.075		S
26	1	3	Asplund, M	female	38	1	5	347077	31.3875		S