Report: Noughts and Crosses (Tic-Tac-Toe) with Alpha-Beta Pruning

1. Title:

Noughts and Crosses (Tic-Tac-Toe) with Alpha-Beta Pruning

2. Introduction:

Noughts and Crosses, also known as Tic-Tac-Toe, is a simple two-player game where one player (usually "X") attempts to form a line of three "X"s in a 3x3 grid, while the other player (usually "O") attempts to do the same. The game can be played manually or computationally, and the challenge often lies in finding the best possible moves.

Alpha-Beta Pruning is an optimization technique used to improve the efficiency of the Minimax algorithm. It "prunes" branches of the game tree that do not need to be explored because they cannot affect the final decision. This significantly reduces the number of nodes that need to be evaluated, speeding up the process.

In this implementation, **Alpha-Beta Pruning** is applied to the Tic-Tac-Toe game to help decide the best move for both players ('X' and 'O') by analyzing all possible future states and pruning branches that will not affect the outcome.

Methodology:

The core of the solution is built using **Alpha-Beta Pruning**, a search algorithm that explores possible moves in a game, evaluates them, and prunes unpromising branches to optimize the decision-making process. The game state is represented by a 3x3 grid, and the algorithm evaluates the board for every potential move. The key components of the methodology are as follows:

1. **State Representation**: The game board is a 3x3 grid where each position can either be empty (' ') or filled with 'X' or 'O'.
2. **Evaluation Function**: The evaluation function checks if there is a winner. It evaluates the board in terms of:
    - **1** for a win by player 'X' (maximizing player).
    - **-1** for a win by player 'O' (minimizing player).
    - **0** for a draw.
3. **Terminal State Check**: The algorithm checks for a terminal state where the game ends either by a win, loss, or draw. This is done by verifying all rows, columns, and diagonals for three matching marks. It also checks if the board is completely filled (indicating a draw).
4. **Alpha-Beta Pruning**: The Alpha-Beta Pruning technique is implemented within the Minimax framework:
    - **Maximizing Player ('X')** tries to maximize the score.
    - **Minimizing Player ('O')** tries to minimize the score. The pruning process reduces the number of nodes that need to be evaluated in the game tree, thus optimizing the search process.
5. **Move Selection**: The program generates all possible board states (children) for each move, evaluates them using Alpha-Beta

Pruning, and selects the best possible move based on the current player's strategy (maximize or minimize).

6. **Game Simulation**: The program runs a loop where each player alternates turns. It simulates the moves of both players until the game reaches a terminal state (win, loss, or draw).
7. **Game Over Condition**: The game ends when either player wins, the board is full, or a draw occurs.

CODE

```
import math

# Function to check if the game has ended (win, loss, or draw)
def is_terminal(board):
    # Check rows, columns, and diagonals for a win
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] and board[i][0] != ' ':
            return True
        if board[0][i] == board[1][i] == board[2][i] and board[0][i] != ' ':
            return True

    if board[0][0] == board[1][1] == board[2][2] and board[0][0] != ' ':
        return True
    if board[0][2] == board[1][1] == board[2][0] and board[0][2] != ' ':
        return True

    # Check if the board is full (draw)
    if all(board[i][j] != ' ' for i in range(3) for j in range(3)):
        return True

    return False

# Function to evaluate the board: 1 for 'X' win, -1 for 'O' win, 0 for draw
def evaluate(board):
    # Check rows, columns, and diagonals for a winner
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] and board[i][0] == 'X':
            return 1   # X wins
```

```
        if board[i][0] == board[i][1] == board[i][2] and board[i][0] == 'O':
            return -1   # O wins
        if board[0][i] == board[1][i] == board[2][i] and board[0][i] == 'X':
            return 1   # X wins
        if board[0][i] == board[1][i] == board[2][i] and board[0][i] == 'O':
            return -1   # O wins

    if board[0][0] == board[1][1] == board[2][2] and board[0][0] == 'X':
        return 1   # X wins
    if board[0][0] == board[1][1] == board[2][2] and board[0][0] == 'O':
        return -1   # O wins
    if board[0][2] == board[1][1] == board[2][0] and board[0][2] == 'X':
        return 1   # X wins
    if board[0][2] == board[1][1] == board[2][0] and board[0][2] == 'O':
        return -1   # O wins

    return 0   # Draw

# Function to get all possible moves (empty spaces) on the board
def get_children(board, player):
    children = []
    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':  # Find empty spaces
                new_board = [row[:] for row in board]  # Copy the current board state
                new_board[i][j] = player   # Place the current player's mark ('X' or 'O')
                children.append(new_board)  # Add the new board to the list of children
    return children

# Alpha-Beta Pruning implementation
def alpha_beta_pruning(board, depth, alpha, beta, maximizing_player):
```

```
    # Check if the game is over or we have reached the maximum depth
    if depth == 0 or is_terminal(board):
        return evaluate(board)    # Evaluate the board (win, loss, or draw)

    if maximizing_player:  # Maximizing player's turn (Player X)
        max_eval = -math.inf   # Start with the lowest possible value
        for child in get_children(board, 'X'):  # Get all possible moves for X
            eval = alpha_beta_pruning(child, depth - 1, alpha, beta, False)  # Minimize for the opponent (O)
            max_eval = max(max_eval, eval)  # Choose the maximum evaluation
            alpha = max(alpha, eval)  # Update alpha
            if beta <= alpha:  # Beta pruning: if beta is smaller than or equal to alpha, stop exploring this branch
                break
        return max_eval
    else:  # Minimizing player's turn (Player O)
        min_eval = math.inf  # Start with the highest possible value
        for child in get_children(board, 'O'):  # Get all possible moves for O
            eval = alpha_beta_pruning(child, depth - 1, alpha, beta, True)  # Maximize for the opponent (X)
            min_eval = min(min_eval, eval)  # Choose the minimum evaluation
            beta = min(beta, eval)  # Update beta
            if beta <= alpha:  # Alpha pruning: if alpha is larger than or equal to beta, stop exploring this branch
                break
        return min_eval

# Function to find the best move for the current player (X or O)
def find_best_move(board, depth, maximizing_player):
    best_move = None
    best_value = -math.inf if maximizing_player else math.inf
```

```
    # Try every possible move and apply Alpha-Beta pruning to find the best move
    for child in get_children(board, 'X' if maximizing_player else 'O'):
        move_value = alpha_beta_pruning(child, depth - 1, -math.inf, math.inf, not maximizing_player)
        if (maximizing_player and move_value > best_value) or (not maximizing_player and move_value < best_value):
            best_value = move_value  # Update the best value
            best_move = child  # Update the best move

    return best_move

# Function to display the board
def print_board(board):
    for row in board:
        print(" | ".join(row))  # Print each row
        print("-" * 5)  # Print a separator line

# Main game loop
def play_game():
    # Initial empty 3x3 Tic-Tac-Toe board
    board = [[' ' for _ in range(3)] for _ in range(3)]

    # Display the initial empty board
    print("Initial Board:")
    print_board(board)

    # Number of moves made
    moves = 0
    depth = 9  # Maximum number of moves is 9 (full board)

    # Game loop: X is the maximizing player and O is the minimizing player
    while not is_terminal(board) and moves < depth:
```

```
        print(f"\nMove {moves + 1}:")
        if moves % 2 == 0:
            print("Player X's turn (Maximizing)")
            best_move = find_best_move(board, depth, True)  # X is the maximizing player
        else:
            print("Player O's turn (Minimizing)")
            best_move = find_best_move(board, depth, False)  # O is the minimizing player

        # Display the board after the move
        print_board(best_move)
        board = best_move  # Update the board with the new move
        moves += 1

    print("\nGame Over!")
    print_board(board)

# Call the play_game function to simulate the game
play_game()
```

OUTPUT Noughts and Crosses with Alpha-Beta Pruning

```
Initial Board:
  |   |
-----
  |   |
-----
  |   |
-----

Move 1:
Player X's turn (Maximizing)
X |   |
-----
  |   |
-----
  |   |
-----

Move 2:
Player O's turn (Minimizing)
X |   |
-----
  | O |
-----
  |   |
-----

Move 3:
Player X's turn (Maximizing)
X | X |
-----
  | O |
-----
  |   |
-----
```

```
  |   |
-----

Move 4:
Player O's turn (Minimizing)
X | X | O
-----
  | O |
-----
  |   |
-----

Move 5:
Player X's turn (Maximizing)
X | X | O
-----
  | O |
-----
X |   |
-----

Move 6:
Player O's turn (Minimizing)
X | X | O
-----
O | O |
-----
X |   |
-----

Move 7:
Player X's turn (Maximizing)
X | X | O
-----
O | O | X
-----
X |   |
-----
```

```
Move 8:
Player O's turn (Minimizing)
X | X | O
-----
O | O | X
-----
X | O |
-----

Move 9:
Player X's turn (Maximizing)
X | X | O
-----
O | O | X
-----
X | O | X
-----

Game Over!
X | X | O
-----
O | O | X
-----
X | O | X
-----
```

5. Result:

- **The game alternates between the players ('X' and 'O'), with 'X' being the maximizing player and 'O' being the minimizing player.**

- **Alpha-Beta Pruning ensures that unnecessary branches are pruned, making the decision-making process faster and more efficient.**
- **The game ends when either player wins (forming a line of three marks) or when the board is full, leading to a draw.**

6. References:

- **Alpha-Beta Pruning: Wikipedia (https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning)**
- **Minimax Algorithm: Wikipedia(https://en.wikipedia.org/wiki/Minimax)**
- **Tic-Tac-Toe Game: https://en.wikipedia.org/wiki/Tic-tac-toe**