Comparative Analysis of Alpha-Beta Pruning and Min-Max Algorithms in Tic-Tac-Toe AI

Shanmukha Sudha Kiran .T -211FA04003; RaviKiran .K-211FA04043;
Deepthi .U -211FA04046; Sahil Raj -211FA04677
BATCH-17,3 Rd B. TECH, CSE BRANCH, VFSTR DEEMED TO BE UNIVERSITY

ABSTRACT:

This paper presents a comprehensive analysis of the application of the Alpha-Beta pruning algorithm in the context of a Tic-Tac-Toe game-playing AI. The primary objective is to evaluate and compare the performance of Alpha-Beta pruning against the conventional Min-Max algorithm in terms of computational efficiency and the ability to identify optimal game moves.

INTRODUCTION:

Alpha-Beta pruning: It is a method that optimizes the Minimax algorithm. The number of states to be visited by the minimax algorithm are exponential, which shoots up the time complexity. Some of the branches of the decision tree are useless, and the same result can be achieved if they were never visited.

Min-Max algorithm*:* Mini-max is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-tac-toe, Backgammon, Mancala, Chess, etc.

ALGORITHM:

1) Start

2) Import the math module to use mathematical functions.

3) Define constants for representing empty cells, player X, and player O.

4) Create the game board as a list of 9 empty cells.

5) Define a function print_board(board) to print the current state of the game board.

6) Define a function check_winner(board) to check if there is a winner or if the game is a tie.

7) Define a function evaluate(board) to evaluate the current state of the board and return a score for the minimax algorithm.

8) Define the minimax(board, depth, alpha, beta, maximizing_player) function that implements the minimax algorithm with alpha-beta pruning. This function recursively explores possible game states.

9) Define a function find_best_move(board) that uses the minimax algorithm to find the best move for the AI player (X).

10) Implement the main game loop, which continues until there's a winner or a tie.

11) Inside the main game loop, print the current game board.

12) Check if there's a winner or a tie. If so, print the result and exit the loop.

13) If the game is still ongoing, determine whose turn it is (X or O) based on the number of moves made.

14) If it's O's turn, prompt the human player for input and update the board. If it's X's

turn, use the find_best_move function to make the AI move and update the board.

15) Print the result.

16)stop

SOURCE CODE:

```
import math


# Constants for representing the players and empty cells
EMPTY = "-"
PLAYER_X = "X"
PLAYER_O = "O"
# The game board
board = [EMPTY, EMPTY, EMPTY,
         EMPTY, EMPTY, EMPTY,
         EMPTY, EMPTY, EMPTY]
# Function to print the game board
def print_board(board):
    print(" --------")
    for i in range(3):
        print("|", board[i*3], "|", board[i*3 + 1], "|", board[i*3 + 2], "|")
        print(" --------")
# Function to check if a player has won
def check_winner(board):
    winning_combinations = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # rows
        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # columns
        [0, 4, 8], [2, 4, 6]  # diagonals
    ]
    for combination in winning_combinations:
        if board[combination[0]] == board[combination[1]] == board[combination[2]] != EMPTY:
            return board[combination[0]]
    if EMPTY not in board:
        return "tie"
    return None
# Function to evaluate the game board
def evaluate(board):
    winner = check_winner(board)
    if winner == PLAYER_X:
        return 1
    elif winner == PLAYER_O:
        return -1
    else:
        return 0
# Minimax function with alpha-beta pruning
def minimax(board, depth, alpha, beta, maximizing_player):
    if check_winner(board) is not None or depth == 0:
        return evaluate(board)
    if maximizing_player:
        max_eval = -math.inf
        for i in range(9):
            if board[i] == EMPTY:
                board[i] = PLAYER_X
                eval_score = minimax(board, depth - 1, alpha, beta, False)
                board[i] = EMPTY
```

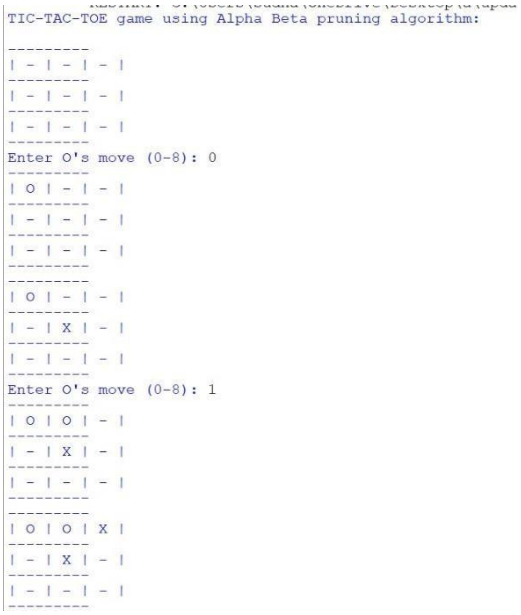
```
            max_eval = max(max_eval, eval_score)
            alpha = max(alpha, eval_score)
            if beta <= alpha:
                break
    return max_eval
    else:
        min_eval = math.inf
        for i in range(9):
            if board[i] == EMPTY:
                board[i] = PLAYER_O
                eval_score = minimax(board, depth - 1, alpha, beta, True)
                board[i] = EMPTY
                min_eval = min(min_eval, eval_score)
                beta = min(beta, eval_score)
                if beta <= alpha:
                    break
        return min_eval
# Function to find the best move using minimax with alpha-beta pruning
def find_best_move(board):
    best_score = -math.inf
    best_move = None
    for i in range(9):
        if board[i] == EMPTY:
            board[i] = PLAYER_X
            move_score = minimax(board, 9, -math.inf, math.inf, False)
            board[i] = EMPTY
            if move_score > best_score:
                best_score = move_score
                best_move = i
    return best_move
# Main game loop
while True:
    print_board(board)
    winner = check_winner(board)
    if winner is not None:
        if winner == "tie":
            print("It's a tie!")
        else:
            print("Player", winner, "wins!")
        break
    if len([cell for cell in board if cell != EMPTY]) % 2 == 0:
        # Player O's turn
        while True:
            move = int(input("Enter O's move (0-8): "))
            if board[move] == EMPTY:
                board[move] = PLAYER_O
                break
            else:
                print("Invalid move! Try again.")
    else:
        # Player X's turn
        move = find_best_move(board)
        board[move] = PLAYER_X
```

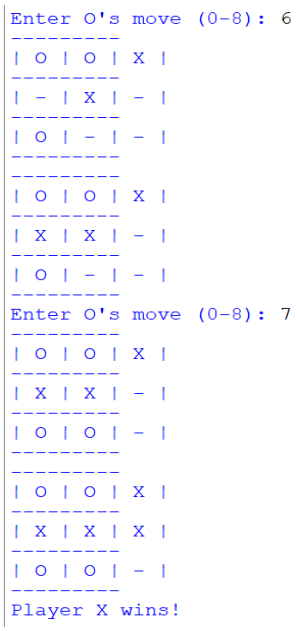
OUTPUT:

Test case-1: AI vs Human

Player O: Human

Player X:AI





RESULTS:

Our experimental results show that Alpha-Beta pruning significantly reduces the number of nodes explored during the search, leading to faster decision-making compared to the Min-Max algorithm. This performance improvement becomes more pronounced as the depth of the game tree increases.

CONCLUSION:

In the context of playing Tic-Tac-Toe, the Alpha-Beta pruning algorithm demonstrates its effectiveness in reducing computational complexity while maintaining competitive gameplay. It outperforms the traditional Min-Max algorithm in terms of computational efficiency.

This research contributes to the broader field of game AI and showcases the practical advantages of Alpha-Beta pruning, especially when applied to games with complex state spaces.

REFERENCE:

1) Tic Tac Toe with AI (MinMax and alpha-beta pruning) – Virtualanup
2) Artificial Intelligence | Alpha-Beta Pruning - Javatpoint