Artificial Intelligence And Expert Systems
(CT-361)

Assignment 2

NAME	MUHAMMAD FURQAN PATEL
ROLL NO	CR-22032
BATCH	2022
YEAR	3RD YEAR
DEPARTMENT	CYBER SECURITY

Implementation of Minimax Algorithm and Alpha-Beta Pruning in Tic-Tac-Toe

Abstract

This report presents the design and implementation of the Minimax algorithm for the classic game of Tic-Tac-Toe, and its optimization using Alpha-Beta Pruning. The goal of the assignment was to create an AI that plays optimally by considering all possible moves and using Alpha-Beta Pruning to enhance the efficiency of the Minimax algorithm. The report discusses the implementation of the game logic, the AI's decision-making process, and compares the performance of both algorithms in terms of computation time.

1. Introduction

In this assignment, we implemented an AI for the game of Tic-Tac-Toe using two techniques: the **Minimax algorithm** and its optimization through **Alpha-Beta Pruning**. The aim was to create an AI that could play the game optimally and efficiently. The project also involved comparing the performance of both algorithms in terms of execution time.

2. Objective

The objectives of the assignment were:

1. **Implement the core logic of the Tic-Tac-Toe game.**
2. **Create an AI player using the Minimax algorithm that plays optimally.**
3. **Optimize the Minimax algorithm using Alpha-Beta Pruning.**
4. **Compare the performance of the standard Minimax algorithm and the Alpha-Beta Pruning optimized Minimax.**

3. Implementation

3.1 TicTacToe Class

The **TicTacToe** class is responsible for managing the game state and implementing game mechanics such as:

- **`__init__()`**: Initializes the board and the current winner.
- **`print_board()`**: Prints the current board to the console.
- **`print_board_nums()`**: Prints the board with numbers representing positions for user input.
- **`available_moves()`**: Returns a list of available moves.
- **`empty_squares()`**: Checks if there are any empty squares left.
- **`make_move()`**: Makes a move on the board and checks for a winner.
- **`winner()`**: Checks for a winner after a move (rows, columns, diagonals).

3.2 Minimax Algorithm

The **Minimax algorithm** works by recursively exploring all possible future game states and selecting the best move:

- **`minimax()`**: A recursive function that returns the best possible move for the current player, assuming both players play optimally.

- **`minimax_ab()`**: A version of Minimax that uses **Alpha-Beta Pruning** to eliminate unnecessary branches in the search tree.

3.3 Alpha-Beta Pruning

Alpha-Beta Pruning optimizes the Minimax algorithm by cutting off branches of the game tree that do not need to be explored. This results in a more efficient search:

- **Alpha** represents the best score that the maximizing player can guarantee so far.
- **Beta** represents the best score that the minimizing player can guarantee so far.
- If **Alpha** is greater than or equal to **Beta**, further exploration of the branch is unnecessary.

3.4 Best Move Calculation

The **`get_best_move()`** function calculates the best move for the AI by evaluating all possible moves using either the standard Minimax or the Alpha-Beta Pruning optimized Minimax.

4. Results

4.1 Game Simulation

A sample game was run where the user played as "O" and the AI played as "X". The game was played with Alpha-Beta Pruning enabled, and the result was a tie. Below is a summary of the moves and the final board state:

```
Welcome to Tic Tac Toe vs AI!
You are O, AI is X.
Use Alpha-Beta Pruning? (y/n): y

| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Your move (0-8): 4

|   |   |   |
|   | O |   |
|   |   |   |

AI is thinking...

| X |   |   |
|   | O |   |
|   |   |   |

Your move (0-8): 2

| X |   | O |
|   | O |   |
|   |   |   |

AI is thinking...

| X |   | O |
|   | O |   |
| X |   |   |
```

```
Your move (0-8): 3

| X |   | O |
| O | O |   |
| X |   |   |

AI is thinking...

| X |   | O |
| O | O | X |
| X |   |   |

Your move (0-8): 7

| X |   | O |
| O | O | X |
| X | O |   |

AI is thinking...

| X | X | O |
| O | O | X |
| X | O |   |

Your move (0-8): 8

| X | X | O |
| O | O | X |
| X | O | O |

It's a tie!

Run performance comparison? (y/n): y
```

4.2 Performance Comparison

The performance of both the standard **Minimax** algorithm and the **Alpha-Beta Pruning** algorithm was compared over three rounds. The following are the results of the performance comparison:

```
Run performance comparison? (y/n): y
Running test 1/3...
Running test 2/3...
Running test 3/3...

Performance Comparison:
Minimax Avg Time: 13.150418 seconds
Alpha-Beta Avg Time: 0.749547 seconds
```

From the comparison, it is evident that **Alpha-Beta Pruning** dramatically improved the computation time, reducing it by approximately **94%**.

5. Conclusion

This assignment successfully demonstrated how the **Minimax algorithm** can be used to create an optimal AI player for Tic-Tac-Toe. The **Alpha-Beta Pruning** optimization significantly improved the performance by reducing the number of nodes that needed to be explored in the decision tree, thereby speeding up the computation time.

- The **Minimax algorithm** ensures that the AI makes the best possible move at every step.
- **Alpha-Beta Pruning** reduces the time complexity of the Minimax algorithm by pruning branches of the game tree.
- The **performance comparison** clearly showed the time efficiency of Alpha-Beta Pruning, making it the preferred method for optimizing decision-making in games like Tic-Tac-Toe.