Assignment 10

Name : Mayuresh Pathak - SE-B-B3-46

Aim : Implementation of Alpha-Beta Pruning Algorithm for Game Tree Search

---

Objectives

1. To understand the concept of game playing algorithms in Artificial Intelligence.
2. To implement alpha-beta pruning as an optimization to the Minimax algorithm.
3. To analyze how pruning reduces the number of nodes evaluated in a search tree.

Theory

Game Playing in AI

Game playing problems in AI, such as Chess or Tic-Tac-Toe, are typically modeled as adversarial search problems. These games involve two or more players with conflicting goals. In a two-player game, the players are commonly referred to as **MAX** (our agent, who tries to maximize the score) and **MIN** (the opponent, who tries to minimize the score). The game can be represented as a **game tree**, where nodes are game states and edges are moves.

Minimax Algorithm

The Minimax algorithm is a recursive, depth-first search algorithm used to determine the optimal move for a player. It explores the entire game tree to find the best possible move, assuming that the opponent also plays optimally.

- **MAX's turn:** Choose the move that leads to the maximum score.
- **MIN's turn:** Choose the move that leads to the minimum score.

Problem with Minimax

The primary drawback of the Minimax algorithm is its computational complexity. It must explore every single node in the game tree to determine the optimal value. For games with a large branching factor (many possible moves from each state), such as Chess or Go, this exhaustive search becomes computationally infeasible very quickly.

Alpha-Beta Pruning

Alpha-Beta pruning is an optimization technique for the Minimax algorithm. It reduces the number of nodes that need to be evaluated by "pruning" or cutting off branches of the game tree that cannot possibly influence the final decision. It does this by keeping track of the best scores found so far for both MAX and MIN.

Key Terms

The algorithm maintains two values, alpha and beta, which represent the best possible scores for MAX and MIN along the current path in the tree.

- **α (alpha):** The best value (highest score) that the **MAX** player can guarantee so far at that node or any of its ancestors. It starts at .
- **β (beta):** The best value (lowest score) that the **MIN** player can guarantee so far at that node or any of its ancestors. It starts at .

The core principle is to prune a branch when the current value being evaluated is already worse than a choice we've already found. The pruning condition is **if β ≤ α**.

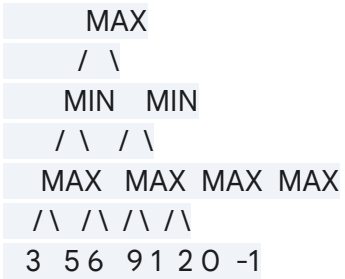
Algorithm (Pseudocode)

```
function alpha_beta(node, depth, α, β, maximizingPlayer):
    if depth = 0 or node is a terminal node:
        return heuristic_value(of node)

    if maximizingPlayer:
        value = -∞
        for each child of node:
            value = max(value, alpha_beta(child, depth-1, α, β, False))
            α = max(α, value)
            if β ≤ α:
                break  // β cut-off
        return value
    else:  // minimizingPlayer
        value = +∞
        for each child of node:
            value = min(value, alpha_beta(child, depth-1, α, β, True))
            β = min(β, value)
            if β ≤ α:
                break  // α cut-off
        return value
```

Python Implementation

The following game tree is used, where the numbers are the terminal node (leaf) values:



Python

```
# Global counters to track the number of terminal nodes evaluated
nodes_evaluated_minimax = 0
nodes_evaluated_ab = 0

# The game tree is represented as a nested list
# Leaf nodes are integers, internal nodes are lists of children
game_tree = [
    [[3, 5], [6, 9]],
    [[1, 2], [0, -1]]
]

# --- Minimax Algorithm Implementation ---
def minimax(node, depth, maximizingPlayer):
    global nodes_evaluated_minimax
    # If it's a terminal node (leaf), return its value
    if not isinstance(node, list):
        nodes_evaluated_minimax += 1
        return node

    if maximizingPlayer:
        bestValue = -float('inf')
        for child in node:
            value = minimax(child, depth - 1, False)
            bestValue = max(bestValue, value)
        return bestValue
    else:  # Minimizing player
        bestValue = float('inf')
        for child in node:
            value = minimax(child, depth - 1, True)
```

```
            bestValue = min(bestValue, value)
        return bestValue

# --- Alpha-Beta Pruning Algorithm Implementation ---
def alpha_beta(node, depth, alpha, beta, maximizingPlayer):
    global nodes_evaluated_ab
    # If it's a terminal node (leaf), return its value
    if not isinstance(node, list):
        nodes_evaluated_ab += 1
        return node

    if maximizingPlayer:
        value = -float('inf')
        for child in node:
            value = max(value, alpha_beta(child, depth - 1, alpha, beta, False))
            alpha = max(alpha, value)
            if beta <= alpha:
                break  # Beta cut-off
        return value
    else:  # Minimizing player
        value = float('inf')
        for child in node:
            value = min(value, alpha_beta(child, depth - 1, alpha, beta, True))
            beta = min(beta, value)
            if beta <= alpha:
                break  # Alpha cut-off
        return value

# --- Running the algorithms ---
minimax_result = minimax(game_tree, 3, True)
alpha_beta_result = alpha_beta(game_tree, 3, -float('inf'), float('inf'), True)

# --- Displaying the results ---
print("--- Minimax Algorithm ---")
print(f"Optimal value: {minimax_result}")
print(f"Terminal nodes evaluated: {nodes_evaluated_minimax}\n")

print("--- Alpha-Beta Pruning ---")
print(f"Optimal value (with Alpha-Beta Pruning): {alpha_beta_result}")
print(f"Terminal nodes evaluated: {nodes_evaluated_ab}")
```

Sample Output

The output from the Python script above is:

```
--- Minimax Algorithm ---
Optimal value: 5
Terminal nodes evaluated: 8

--- Alpha-Beta Pruning ---
Optimal value (with Alpha-Beta Pruning): 5
Terminal nodes evaluated: 5
```

Observation Table

This table compares the performance of the two algorithms on the given game tree.

Parameter	Minimax	Alpha-Beta
Nodes evaluated	8	5
Optimal value	5	5

Conclusion

The experiment successfully demonstrates the effectiveness of the Alpha-Beta Pruning algorithm. By intelligently cutting off branches that won't affect the final outcome, it **significantly reduces the number of nodes evaluated** (from 8 to 5 in this case) compared to the standard Minimax algorithm. Crucially, this optimization is achieved **without sacrificing accuracy**, as both algorithms arrive at the same optimal decision. For complex games with deep and wide search trees, Alpha-Beta Pruning is an essential technique for making AI game-playing agents computationally feasible.