K. J. Somaiya College of Engineering, Mumbai-77

Batch: B2	**Roll No.: 16010122221**
Experiment / assignment / tutorial No. 6	

Title: Implementation of Adversarial Search Algorithm

Objective: Implementation of Alpha-Beta Pruning algorithm

Expected Outcome of Experiment:

Course Outcome	After successful completion of the course students should be able to
CO 2	Analyse and solve problems for goal based agent architecture (searching and planning algorithms).

Books/ Journals/ Websites referred:

1. **"Artificial Intelligence: a Modern Approach" by Russel and Norving, Pearson education Publications**
2. **"Artificial Intelligence" By Rich and knight, Tata Mcgraw Hill Publications**
3. **www.cs.sfu.ca/CourseCentral/310/oschulte/mychapter5.pdf**
4. **http://cs.lmu.edu/~ray/notes/asearch/**
5. **www.cs.cornell.edu/courses/cs4700/2011fa/.../06_adversarialsearch.pdf**

Pre Lab/ Prior Concepts: Two/Multi player Games and rules, state-space tree, searching algorithms and their analysis properties

Historical Profile: - The game playing has been an integral part of human life. The multiplayer games are a competitive environment in which everyone tries to gain more points for himself and wishes the opponent to gain a minimum.

The game can be represented in the form of a state space tree and one can follow the path from root to some goal node, for either of the players.

New Concepts to be learned: Adversarial search, min-max algorithm, Alpha-Beta pruning,

Adversarial Search Concept:-

In game-playing scenarios where two players compete against each other, we use adversarial search. Unlike regular search problems (where an agent searches for a solution to a problem),

adversarial search involves two opposing agents, typically a Maximizing player (MAX) and a Minimizing player (MIN), each trying to maximize their own gains while minimizing the opponent's.

State-Space Representation in Adversarial Search

● The game is represented using a state-space tree.

● The root node represents the initial state of the game.

● Each branch represents a possible move by a player.

● Leaf nodes represent terminal states (i.e., game outcomes).

● The tree is explored using game tree search algorithms.

Minimax Algorithm

Minimax is a decision-making algorithm used in two-player games. It assumes that:

1. One player (MAX) tries to maximize the score.

2. The other player (MIN) tries to minimize the score.

3. The game alternates between these two roles.

Steps:

1. Start from the leaf nodes and assign them evaluation values.

2. Apply the Minimax principle:

○ At MIN nodes, choose the minimum value of child nodes.

○ At MAX nodes, choose the maximum value of child nodes.

3. Continue propagating the values up to the root.

Alpha-beta pruning algorithm:

The Alpha-Beta pruning technique improves Minimax by avoiding unnecessary branches in the search tree. It eliminates branches that won't affect the final decision, making the search more efficient.

Key Components:

● α (alpha): The best value that the MAX player can guarantee so far (initialized to -∞).

● β (beta): The best value that the MIN player can guarantee so far (initialized to +∞).

Working of Alpha-Beta Pruning

1. Traverse the tree in depth-first order.

2. Apply Minimax normally but:

○ Update alpha at MAX nodes.

○ Update beta at MIN nodes.

3. Pruning condition:

○ If at any point $\beta \leq \alpha$, the branch is pruned (ignored), as it won't influence the final decision

Chosen Problem:

Implement the Alpha-Beta Pruning algorithm on a predefined test tree and demonstrate its effect on pruning unnecessary branches.

Problem Statement:

● Given a predefined game tree, apply the Minimax algorithm with Alpha-Beta pruning.

● Accept 8 leaf node values as input.

● Compute and display the tree before and after pruning.

● Show the decision-making process at each step.

Enter 8 leaf node values:

Leaf 1: 6

Leaf 2: 4

Leaf 3: 7

Leaf 4: 8

Leaf 5: 3

Leaf 6: 9

Leaf 7: 5

Leaf 8: 7

code:-

```
import math
import sys


class Node:
    def __init__(self, value=None):
        self.value = value
        self.children = []
        self.alpha = float('-inf')
        self.beta = float('inf')
        self.pruned = False
        self.is_max_node = True
        self.best_child_index = None
        self.depth = 0


def build_tree_from_input():
    print("Alpha-Beta Pruning Algorithm")
    print("============================")

    try:
        depth = int(input("Enter the depth of the tree: "))
        if depth < 0:
            print("Depth must be non-negative. Using default depth
of 3.")
```

```
            depth = 3
    except ValueError:
        print("Invalid input. Using default depth of 3.")
        depth = 3

    root = Node()
    root.is_max_node = True
    root.depth = 0

    if depth == 0:
        root.value = int(input("Enter value for the root node: "))
        return root, depth

    nodes_by_level = {0: [root]}

    for level in range(1, depth):
        nodes_by_level[level] = []
        for parent in nodes_by_level[level - 1]:
            try:
                num_children = int(input(f"Enter number of children
for node at level {level-1}: "))
                if num_children <= 0:
                    print("Number of children must be positive.
Using 2 children.")
                    num_children = 2
            except ValueError:
```

```
                print("Invalid input. Using 2 children.")
                num_children = 2

            for _ in range(num_children):
                child = Node()
                child.is_max_node = not parent.is_max_node
                child.depth = level
                parent.children.append(child)
                nodes_by_level[level].append(child)

    leaf_level = depth
    nodes_by_level[leaf_level] = []

    for parent in nodes_by_level[leaf_level - 1]:
        try:
            num_children = int(input(f"Enter number of children for
node at level {leaf_level-1}: "))
            if num_children <= 0:
                print("Number of children must be positive. Using 2
children.")
                num_children = 2
        except ValueError:
            print("Invalid input. Using 2 children.")
            num_children = 2

        for i in range(num_children):
```

```
            leaf = Node()
            leaf.is_max_node = not parent.is_max_node
            leaf.depth = leaf_level
            try:
                value = int(input(f"Enter value for leaf node {i+1} of parent at level {leaf_level-1}: "))
                leaf.value = value
            except ValueError:
                print(f"Invalid input. Using default value of 0.")
                leaf.value = 0
            parent.children.append(leaf)
            nodes_by_level[leaf_level].append(leaf)

    return root, depth

def alpha_beta_pruning(node, alpha, beta, is_maximizing):
    node.alpha = alpha
    node.beta = beta

    if not node.children:
        return node.value

    if is_maximizing:
        value = float('-inf')
        for i, child in enumerate(node.children):
```

```
            child_value = alpha_beta_pruning(child, alpha, beta,
False)
            if child_value > value:
                value = child_value
                node.best_child_index = i
            alpha = max(alpha, value)
            if beta <= alpha:
                for j in range(i + 1, len(node.children)):
                    node.children[j].pruned = True
                break
        node.value = value
        return value
    else:
        value = float('inf')
        for i, child in enumerate(node.children):
            child_value = alpha_beta_pruning(child, alpha, beta,
True)
            if child_value < value:
                value = child_value
                node.best_child_index = i
            beta = min(beta, value)
            if beta <= alpha:
                for j in range(i + 1, len(node.children)):
                    node.children[j].pruned = True
                break
        node.value = value
```

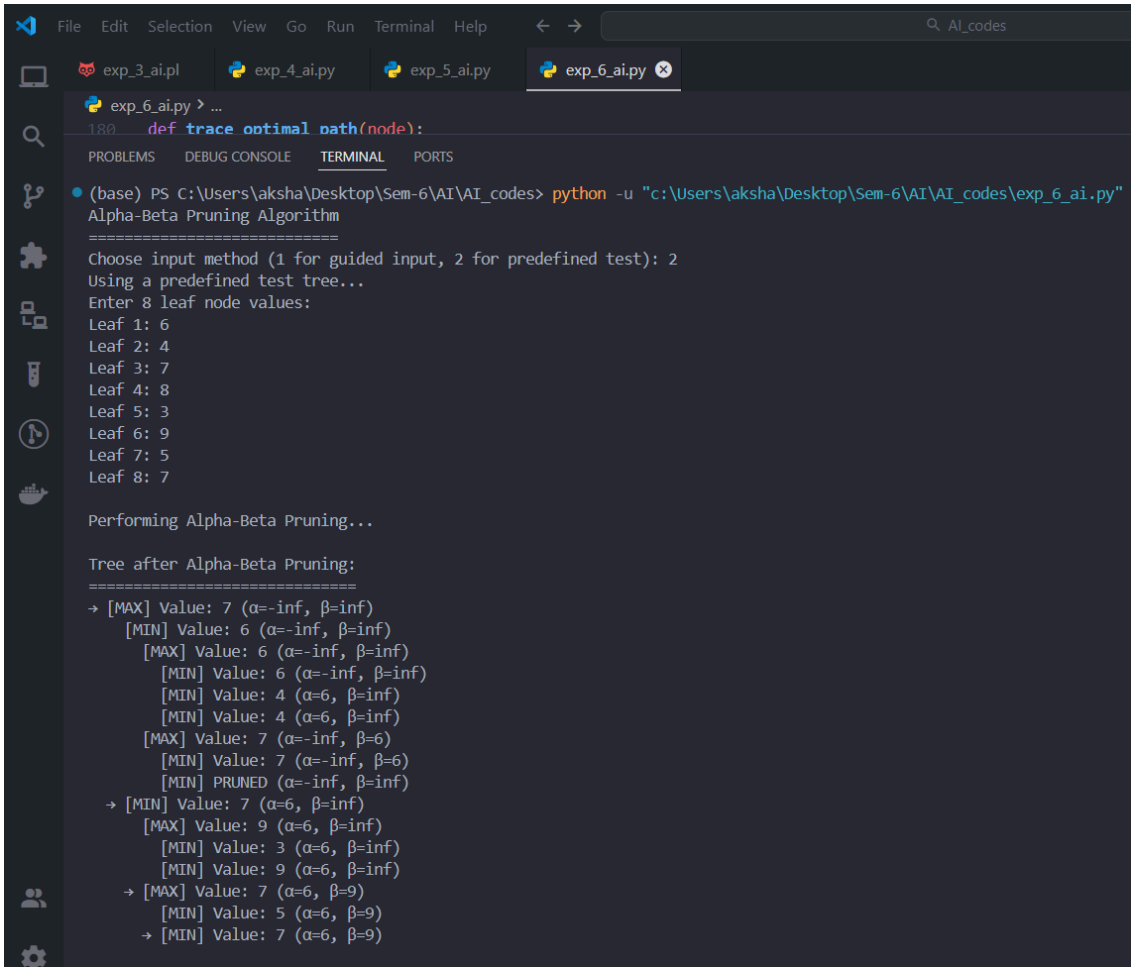
```
        return value


def manual_input_tree():
    print("Using a predefined test tree...")
    root = Node()
    root.is_max_node = True
    root.depth = 0

    for _ in range(2):
        child = Node()
        child.is_max_node = False
        child.depth = 1
        root.children.append(child)

    for parent in root.children:
        for _ in range(2):
            child = Node()
            child.is_max_node = True
            child.depth = 2
            parent.children.append(child)

    leaf_values = []
    print("Enter 8 leaf node values:")
    for i in range(8):
        try:
            value = int(input(f"Leaf {i+1}: "))
```

```
            leaf_values.append(value)
        except ValueError:
            print(f"Invalid input. Using default value of {i}.")
            leaf_values.append(i)

    value_index = 0
    for level1_node in root.children:
        for level2_node in level1_node.children:
            for _ in range(2):
                leaf = Node()
                leaf.is_max_node = False
                leaf.depth = 3
                if value_index < len(leaf_values):
                    leaf.value = leaf_values[value_index]
                    value_index += 1
                else:
                    leaf.value = 0
                level2_node.children.append(leaf)

    return root, 3

def print_tree(node, depth=0, path_str="", is_optimal=True):
    indent = "  " * depth
    path_marker = "→" if is_optimal else " "
    node_type = "MAX" if node.is_max_node else "MIN"
    if node.pruned:
```

```
        print(f"{indent}{path_marker} [{node_type}] PRUNED
(α={node.alpha}, β={node.beta})")

    else:

        value_str = str(node.value) if node.value is not None else
"None"

        print(f"{indent}{path_marker} [{node_type}] Value:
{value_str} (α={node.alpha}, β={node.beta})")

    for i, child in enumerate(node.children):

        new_path = f"{path_str},{i}" if path_str else f"{i}"

        is_best = is_optimal and (node.best_child_index == i)

        print_tree(child, depth + 1, new_path, is_best)


def trace_optimal_path(node):

    path = []

    current = node

    while current.children and current.best_child_index is not
None:

        path.append(current.best_child_index)

        current = current.children[current.best_child_index]

    return path, current.value


def main():

    print("Alpha-Beta Pruning Algorithm")

    print("============================")

    choice = input("Choose input method (1 for guided input, 2 for
predefined test): ")

    if choice == '2':
```

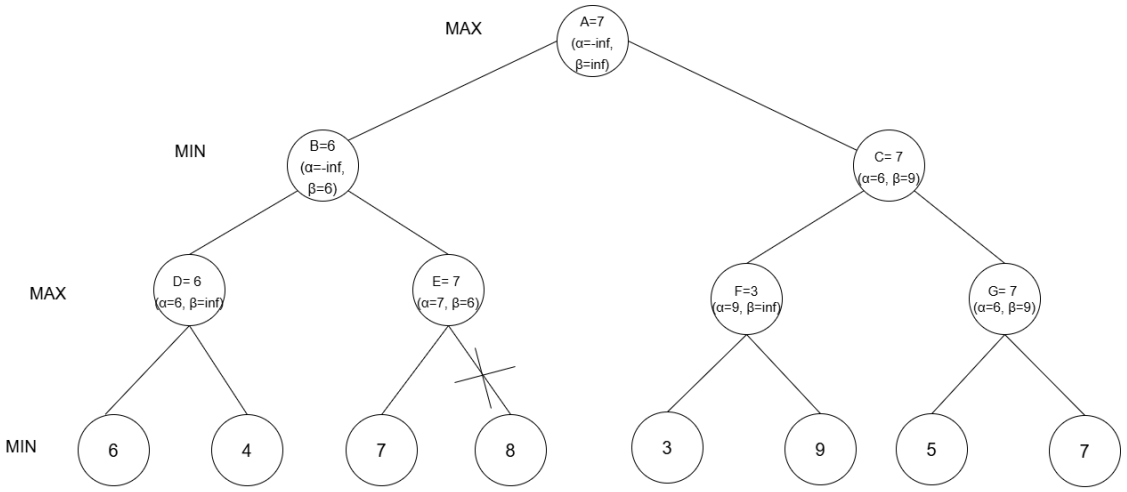
```
        root, depth = manual_input_tree()
    else:
        root, depth = build_tree_from_input()

    print("\nPerforming Alpha-Beta Pruning...")
    alpha_beta_pruning(root, float('-inf'), float('inf'),
root.is_max_node)
    print("\nTree after Alpha-Beta Pruning:")
    print("==============================")
    print_tree(root)

    path, value = trace_optimal_path(root)
    path_str = " → ".join([f"Child {p+1}" for p in path])
    print("\nOptimal Path:")
    print(f"Root → {path_str}")
    print(f"Final Value: {value}")

if __name__ == "__main__":
    main()
```

Output:-




```
Optimal Path:
Root → Child 2 → Child 2 → Child 2
Final Value: 7
(base) PS C:\Users\aksha\Desktop\Sem-6\AI\AI_codes>
```

Solution tree for chosen Problem:



Conclusion:

The implementation of the Alpha-Beta Pruning algorithm enhances the efficiency of the Minimax algorithm by reducing unnecessary evaluations in game trees. By leveraging boundary values, Alpha (α) and Beta (β), the algorithm prunes branches that cannot influence the final decision, significantly optimizing computational performance. The pruning condition, $\beta \leq \alpha$, minimizes exploration, resulting in a near O(b^(d/2)) time complexity compared to O(b^d) in standard Minimax. This efficiency allows faster decision-making in adversarial scenarios, particularly in two-player games where players alternate between maximizing and minimizing outcomes. The algorithm maintains the same decision accuracy as Minimax while cutting down on redundant computations. Overall, Alpha-Beta Pruning proves to be a vital optimization technique, enabling faster and more strategic problem-solving in complex game scenarios without compromising the quality of decisions. This experiment highlights its importance in improving adversarial search processes.

Post Lab objective Questions:

1. **Which search is equal to minmax search but eliminates the branches that can't influence the final decision?**
   a. Breadth-first search
   b. Depth first search
   c. Alpha-beta pruning
   d. None of the above

Answer:

2. **Which values are independent in minmax search alogirthm?**
   a. Pruned leaves x and y
   b. Every states are dependant
   c. Root is independent
   d. None of the above

Answer:

Post Lab Subjective Questions:

1. What is the main purpose of the Alpha-Beta pruning algorithm in game trees?

   Ans->The primary purpose of Alpha-Beta pruning is to optimize the Minimax algorithm by eliminating branches that won't affect the final decision. This helps in:
   1. Reducing Computation Time: By skipping unnecessary branches, the algorithm evaluates fewer nodes, making it faster.
   2. Improving Efficiency: Instead of exploring all possible moves, Alpha-Beta pruning allows the algorithm to focus only on relevant moves, improving performance.

2. How does Alpha-Beta pruning improve the efficiency of the Minimax algorithm?

   Ans->Alpha-Beta pruning improves efficiency by skipping unnecessary evaluations using two boundary values:
   ● Alpha (α): The best value that the Maximizing player can secure so far.
   ● Beta (β): The best value that the Minimizing player can secure so far.
   How does it work?
   ● As the algorithm explores the tree, it keeps track of alpha (α) and beta (β) values at each level.
   ● If, at any point, the condition $\beta \leq \alpha$ holds, further exploration of that branch is pruned (stopped).
   ● This means that part of the game tree is ignored because it won't affect the final decision. This reduces the number of nodes that need to be evaluated, making the search much faster than Minimax alone.

3. Explain the terms **alpha** and **beta** in the context of the Alpha-Beta pruning algorithm.

   Ans->Alpha (α):
   ● Represents the best (highest) value that the Maximizing player can secure at any point.
   ● It starts at $-\infty$ and gets updated as we explore the tree.
   ● The higher the alpha value, the better it is for the Maximizing player.
   Beta (β):
   ● Represents the best (lowest) value that the Minimizing player can secure at any point.
   ● It starts at $+\infty$ and gets updated as we explore the tree.
   ● The lower the beta value, the better it is for the Minimizing player.

4. What condition must be met for a node to be pruned during Alpha-Beta pruning?

Ans->A node is pruned when $\beta \leq \alpha$.
● This condition means that the Minimizing player has already found a better move elsewhere, making further exploration unnecessary.
● At this point, the subtree is cut off, preventing unnecessary computation.
For example:
● If Alpha = 7 (best Max value)
● If Beta = 5 (best Min value found so far) Since Beta (5) ≤ Alpha (7), the branch will be pruned because the Minimizing player would never allow the Maximizing player to get more than 5.

5. Compare and contrast Alpha-Beta pruning with the standard Minimax algorithm.

Feature	Minimax Algorithm	Alpha-Beta Pruning
Efficiency	Explores all nodes	Prunes unnecessary nodes
Time Complexity	O(b^d) (b = branching factor, d = depth)	O(b^(d/2)) (almost halves the time)
Speed	Slower (examines all possibilities)	Faster (skips irrelevant branches)
Decision Accuracy	Same as Alpha-Beta	Same as Minimax
Best-Case Performance	No optimizations	Works best with sorted branches