CS2007-Artificial Intelligence

CIA 2

Rohan Joshua 22011102079

IoT -B

AO* and Minimax Algorithm:

The **Minimax algorithm** is a classic decision-making strategy primarily used in game theory and artificial intelligence, particularly for two-player, turn-based games such as chess, tic-tac-toe, or checkers. It systematically evaluates all possible moves by simulating the game to its conclusion for each option, assuming optimal play from both sides. Minimax aims to minimize the maximum potential loss of a move, thus “minimizing the opponent's maximum possible gain” and allowing the player to make the most advantageous decision at each step. The algorithm builds a decision tree, examining each node's outcome to determine the best possible move for the player by assuming that both players will choose their optimal moves until the game ends.

Alpha-Beta pruning often accompanies Minimax to improve efficiency. This pruning technique discards paths that don't affect the final decision, thus skipping unnecessary evaluations and speeding up the search process. However, Alpha-Beta pruning still requires examining relevant paths up to a defined depth, even when some may be less optimal, which can lead to redundant evaluations.

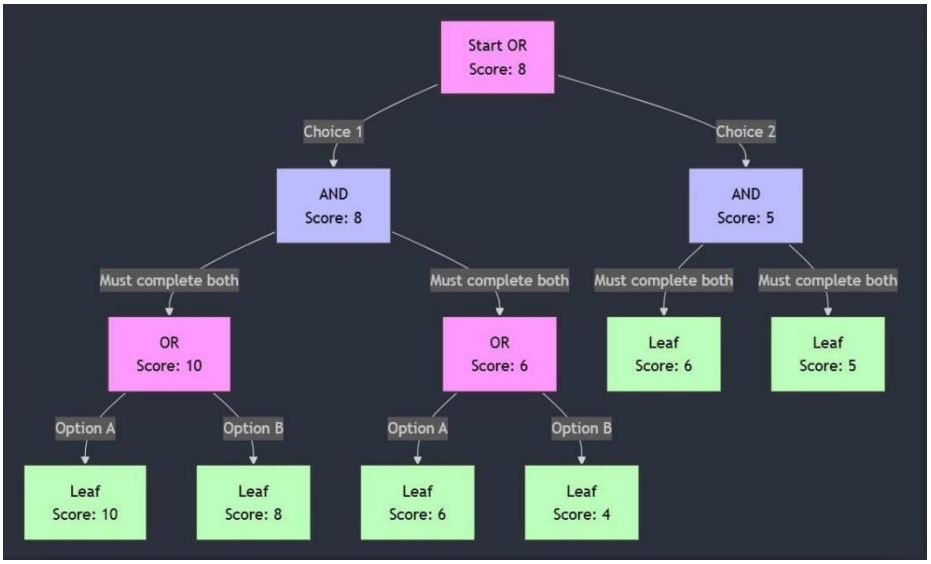
On the other hand, the *AO (AND-OR) algorithm** tackles more complex problems beyond strictly adversarial games, often in scenarios that involve multiple objectives or dependencies. AO* relies on **AND-OR graphs** to model problems that can be broken down into smaller subproblems, where some subproblems (represented by AND nodes) require multiple conditions to be met, while others (OR nodes) represent individual tasks that can achieve a goal independently.

In an **AND-OR graph**:

- **AND nodes** represent interconnected sub-tasks, meaning all child nodes must be solved together to achieve the parent task. This structure suits situations where multiple factors contribute to a solution.
- **OR nodes** allow any one of the child nodes to solve the parent task, providing flexibility when there are multiple ways to reach a solution.

AO* incorporates **heuristics** to guide the search toward the most promising paths first, making it efficient in scenarios where only certain subproblems are essential to achieving the goal. It dynamically adjusts to eliminate irrelevant nodes, which reduces the search time significantly compared to exhaustive approaches. This adaptiveness makes AO* particularly effective for decision trees with interconnected tasks (AND-OR structures), where paths may be pruned based on cumulative progress toward an optimal solution, unlike the static approach of Minimax with Alpha-Beta pruning.

Game Tree:



For this tree, nodes represent game states, and they are divided into:

- OR Nodes (NodeType.OR): Represents decisions where the player can choose one option from many. The goal is to maximize the score by selecting the best option.
- AND Nodes (NodeType.AND): Represents states where multiple actions or events must be completed to proceed, minimizing risk by selecting the lowest score.
- LEAF Nodes (NodeType.LEAF): Terminal states where scores are already defined.

A- Root node OR(score 8)

B- And node(score 8)

C- And node(score 5)

D- B's left child OR node(score 10)

E- B's right child OR node(score 6)

F- D's left child LEAF(score 10)

G- D's right child LEAF(score 8)

H- E's left child LEAF(score 6)

I- E's right child LEAF(score 4)

J- C's left child LEAF(Score 6)

K- C's right child LEAF(Score 5)

Algorithm:

1. OR nodes maximize the score of the player, so they select the child with the highest score.
2. AND Nodes minimize risk or cost, selecting the child with the lowest score. The g_cost or

the actual cost of the node is calculated based on its type.

3. If the node is a LEAF, it simply returns its score.
4. If the node is an OR Node, it takes the maximum score among its children (applying Minimax's max function).
5. If the node is an AND Node, it takes the minimum score (applying Minimax's min function).
6. For path reconstruction, best_child is recorded in each node to trace the path leading to the best possible outcome. Starting from the root, ao_star_search traces the path by following best_child pointers until a LEAF is reached.

Program Code:

```
from enum import Enum
import math
from typing import List, Optional

class NodeType(Enum):
    OR = "OR"
    AND = "AND"
    LEAF = "LEAF"

class Node:
```

```
    def __init__(self, name: str, node_type: NodeType, score: float = 0): self.name = name
        self.node_type = node_type self.score =
        score self.children: List[Node] = []
        self.parent: Optional[Node] = None self.is_solved =
        False self.best_child: Optional[Node] = None
        self.h_cost = float('inf')  # heuristic cost self.g_cost =
        float('inf')  # actual cost

    def add_child(self, child: 'Node'):
        self.children.append(child) child.parent =
        self

class AOStarMinimax: def __init__
    (self):
        self.root = self._create_game_tree()

    def _create_game_tree(self):
        root = Node("A", NodeType.OR, 8)
        b = Node("B", NodeType.AND, 8)
        c = Node("C", NodeType.AND, 5)
        d = Node("D", NodeType.OR, 10)
        e = Node("E", NodeType.OR, 6)
        f = Node("F", NodeType.LEAF, 10)
        g = Node("G", NodeType.LEAF, 8)
        h = Node("H", NodeType.LEAF, 6)
        i = Node("I", NodeType.LEAF, 4)
        j = Node("J", NodeType.LEAF, 6)
        k = Node("K", NodeType.LEAF, 5)

        root.add_child(b)
        root.add_child(c) b.add_child(d)
        b.add_child(e) d.add_child(f)
        d.add_child(g) e.add_child(h)
        e.add_child(i) c.add_child(j)
        c.add_child(k)
```

```
        return root

    def evaluate_node(self, node: Node):
        if node.node_type == NodeType.LEAF: return
            node.score
        if node.node_type == NodeType.OR:
            return max(self.evaluate_node(child) for child in node.children) if node.node_type ==
        NodeType.AND:
            return min(self.evaluate_node(child) for child in node.children) return float('-inf')

    def ao_star_search(self) -> (float, List[str]): def
        update_node_cost(node: Node) -> float:
            if node.node_type == NodeType.LEAF: node.g_cost =
                node.score
                return node.g_cost
            children_costs = [update_node_cost(child) for child in node.children] if node.node_type ==
            NodeType.OR:
                node.g_cost = max(children_costs)
                node.best_child =
node.children[children_costs.index(node.g_cost)] else:
                node.g_cost = min(children_costs) node.best_child
                =
node.children[children_costs.index(node.g_cost)] return node.g_cost

        # Perform the search
        final_cost = update_node_cost(self.root) optimal_path = []
        current = self.root while
        current:
            optimal_path.append(current.name) current =
            current.best_child
        return final_cost, optimal_path

    def print_tree(self, node: Node, level: int = 0): """Print the tree
        structure with scores."""  indent = " " * level
        print(f"{indent}{node.name} ({node.node_type.value}) - Score:
{node.score}")
        for child in node.children: self.print_tree(child, level + 1)
```

```
solver = AOStarMinimax()
print("Initial Game Tree Structure:")
solver.print_tree(solver.root)
optimal_score, optimal_path = solver.ao_star_search()
print("\nResults:")
print(f"Optimal Score: {optimal_score}")
print(f"Optimal Path: {' -> '.join(optimal_path)}")
print("\nBranch Evaluations:")
for child in solver.root.children:
    score = solver.evaluate_node(child)
    print(f"Branch starting at {child.name}: {score}")
```

Sample I/O:

```
Initial Game Tree Structure:
A (OR) - Score: 8
  B (AND) - Score: 8
    D (OR) - Score: 10
      F (LEAF) - Score: 10
      G (LEAF) - Score: 8
    E (OR) - Score: 6
      H (LEAF) - Score: 6
      I (LEAF) - Score: 4
  C (AND) - Score: 5
    J (LEAF) - Score: 6
    K (LEAF) - Score: 5

Results:
Optimal Score: 6
Optimal Path: A -> B -> E -> H

Branch Evaluations:
Branch starting at B: 6
Branch starting at C: 5
```

When we use Minimax algorithm for the same tree, we get the optimal score. In the recursive evaluation, if it's the maximizing player's turn (max_player=True), the function initializes max_eval as negative infinity. For each child of the current node, it recursively calls min_max with max_player=False. It updates max_eval to the maximum evaluation of the child nodes.

The same is done for minimum values when max_player=False

Alpha-beta pruning also gives the same solution. We add alpha and beta parameters to the function to represent the best values for the maximizing and minimizing players respectively. For a maximizing node, if beta <= alpha, it indicates that further children don't need to be evaluated, as the opponent has a better option, so the loop breaks early.

Similarly, for a minimizing node, if beta <= alpha, this branch is pruned. This prevents us from visiting any unnecessary nodes and paths.

```
Minimax
Depth 3 - max evaluates 8
Depth 2 - min evaluates 8
Depth 1 - max evaluates 10
Depth 0 - min evaluates 10
Depth 0 - min evaluates 8
Depth 1 - max evaluates 6
Depth 0 - min evaluates 6
Depth 0 - min evaluates 4
Depth 2 - min evaluates 5
Depth 1 - max evaluates 6
Depth 1 - max evaluates 5
Minimax Final Value: 6
```

```
Alpha-Beta Pruning
Depth 3 - max evaluates 8
Depth 2 - min evaluates 8
Depth 1 - max evaluates 10
Depth 0 - min evaluates 10
Depth 0 - min evaluates 8
Depth 1 - max evaluates 6
Depth 0 - min evaluates 6
Depth 0 - min evaluates 4
Depth 2 - min evaluates 5
Depth 1 - max evaluates 6
Alpha-Beta Pruning Final Value: 6
```

Result:

AO* Algorithm uses heuristics to prioritize promising paths, enabling faster convergence in AND-OR decision trees by focusing only on relevant nodes, which reduces search time. Unlike Alpha-Beta pruning, which statically examines nodes up to a set depth, AO* dynamically adjusts paths, making it more adaptive and efficient in finding optimal solutions in decision trees with interconnected subproblems.