Experiment No. 7

Aim

To write a program to implement the **AO*** (And-Or Star) algorithm for solving problems using heuristic search in AND-OR graphs.

Theory

The **AO*** algorithm is a **graph-based search algorithm** used in **Artificial Intelligence** to find optimal solutions in **AND-OR graphs**.
Unlike the A* algorithm, which is used for simple pathfinding problems, **AO*** is designed for problems that involve **decomposable sub-problems**, where a node may require solving **multiple sub-goals (AND nodes)** or **alternative goals (OR nodes)**.

Key Concepts

1. **AND-OR Graphs:**
   - A **graph structure** where nodes may represent **AND** or **OR** relationships.
   - **OR nodes**: Choosing any one of the successor nodes satisfies the condition.
   - **AND nodes**: All the child nodes must be solved to satisfy the condition.
2. **Heuristic Search:**
   AO *uses heuristic values to estimate the cost of reaching a goal, similar to A*, but adapted for AND-OR relationships.
3. **Cost Function:**
   Each node maintains a cost function:
   [
   f(n) = g(n) + h(n)
   - **g(n):** Actual cost to reach node *n*.
   - **h(n):** Heuristic estimate to reach the goal from node *n*.
4. **Optimality:**
   AO* guarantees an optimal solution if the heuristic is **admissible** (does not overestimate the true cost).
5. **Applications:**
   Used in areas like **problem reduction**, **expert systems**, and **planning systems**, where complex problems can be divided into simpler sub-problems.

Algorithm

Algorithm: AO* Search Algorithm

1. **Start** with the initial node as the current node.
2. **Expand** the current node — generate all possible child nodes (successors).
3. **Compute** the cost estimates for each node using the heuristic function.
4. **Mark** each node as either:
    - **AND-node:** all children must be solved.
    - **OR-node:** any one child can lead to a solution.
5. **Select** the node with the minimum cost path estimate.
6. **Backtrack:** Update parent nodes with new cost values based on children.
7. **Repeat** until the start node is marked as solved (goal state found) or no further expansion is possible.
8. **Terminate** with the optimal solution path.

Code

Java Program to Implement AO* Algorithm

```java
import java.util.*;

class Node {
    String name;
    boolean solved;
    boolean isAND;
    double heuristic;
    double cost;
    List<List<Node>> children; // Each child list represents an AND/OR branch
    Node parent;

    Node(String name, double heuristic, boolean isAND) {
        this.name = name;
        this.heuristic = heuristic;
        this.isAND = isAND;
        this.solved = false;
        this.cost = heuristic;
        this.children = new ArrayList<>();
    }

    void addChildren(List<Node> childGroup) {
        children.add(childGroup);
    }
}
```

```java
public class AOStarAlgorithm {

    // Function to calculate minimum cost and choose the best child path
    static double computeCost(Node node) {
        double minCost = Double.MAX_VALUE;

        for (List<Node> group : node.children) {
            double sum = 0;
            for (Node child : group) sum += child.cost;
            if (sum < minCost) minCost = sum;
        }
        node.cost = node.heuristic + minCost;
        return node.cost;
    }

    // Recursive AO* function
    static void AOStar(Node node) {
        if (node.solved || node.children.isEmpty()) return;

        computeCost(node);
        for (List<Node> group : node.children) {
            for (Node child : group) {
                child.parent = node;
                AOStar(child); // Recursive call
            }
        }

        // Mark node as solved if all AND children are solved
        if (node.isAND) {
            boolean allSolved = true;
            for (List<Node> group : node.children)
                for (Node child : group)
                    if (!child.solved) allSolved = false;

            if (allSolved) node.solved = true;
        } else {
            // OR node is solved if any one child is solved
            for (List<Node> group : node.children)
                for (Node child : group)
                    if (child.solved) node.solved = true;
        }
    }

    public static void main(String[] args) {
        // Create nodes with heuristic values
        Node A = new Node("A", 5, false);
        Node B = new Node("B", 3, false);
        Node C = new Node("C", 2, true);
        Node D = new Node("D", 4, false);
        Node E = new Node("E", 1, false);
```

```
        // Construct the AND-OR graph
        A.addChildren(Arrays.asList(B));            // OR branch
        A.addChildren(Arrays.asList(C));            // OR branch
        B.addChildren(Arrays.asList(D, E));         // AND branch

        // Execute AO* Algorithm
        AOStar(A);

        // Output result
        System.out.println("Node\tCost\tSolved");
        System.out.println(A.name + "\t" + A.cost + "\t" + A.solved);
        System.out.println(B.name + "\t" + B.cost + "\t" + B.solved);
        System.out.println(C.name + "\t" + C.cost + "\t" + C.solved);
        System.out.println(D.name + "\t" + D.cost + "\t" + D.solved);
        System.out.println(E.name + "\t" + E.cost + "\t" + E.solved);

        System.out.println("\nOptimal Solution Found Using AO* Algorithm!");
    }
}
```

Input/Output Example

Input:
A simple AND-OR graph with nodes and heuristic values:

- A → (B OR C)
- B → (D AND E)
- C, D, E are terminal nodes with heuristic values.

Output:

```
Node    Cost    Solved
A   8.0 true
B   8.0 true
C   2.0 true
D   4.0 true
E   1.0 true

Optimal Solution Found Using AO* Algorithm!
```

Analysis of Code and Algorithm

- **Logic Flow:**
  The algorithm recursively expands nodes, updates costs, and backtracks to mark solved nodes until the start node is solved.
- **Efficiency:**
  AO* avoids unnecessary computation by maintaining the best cost estimates during traversal.
- **Time Complexity:**
  Depends on the graph structure — approximately (O(b^d)), where *b* is the branching factor and *d* is the graph depth.
- **Space Complexity:**
  (O(n)), where *n* is the number of nodes stored in memory during search.
- **Advantages:**
  - Handles **decomposable problems** efficiently.
  - Guarantees **optimal solution** if heuristic is admissible.

Real-Life Applications

1. **Expert Systems:** Used in medical diagnosis or troubleshooting systems to combine evidence (AND) or select alternatives (OR).
2. **Automated Planning:** In AI planning where sub-goals must be achieved simultaneously or selectively.
3. **Game Playing & Decision Making:** Helps in analyzing multi-step strategies where multiple sub-decisions are required.

Conclusion

In this experiment, the **AO*** algorithm was implemented to solve a problem using **AND-OR graph-based heuristic search**.
The experiment demonstrated how AO* can efficiently handle problems that involve both **conjunctive (AND)** and **disjunctive (OR)** decisions.
This experiment enhanced understanding of **heuristic problem-solving techniques** and **graph-based search algorithms** in Artificial Intelligence.
