**Date: 03rd January 2025**

**Experiment 3**

**AIM:** Implement the AO\* algorithm to find the optimal solution path.

**Introduction:**

The AO\* (And-Or Star) algorithm is a search algorithm used in problem-solving and AI. It is designed for problems where solutions can be found through a combination of AND and OR conditions, unlike traditional A\* which only uses a single path. It is commonly used in graph-based searches and hierarchical problem solving, such as expert systems and decision-making applications.

**Core Concept**

Imagine you are trying to solve a puzzle where some steps need to be done together (AND), while others give you multiple choices (OR). The AO\* algorithm searches for the best way to solve the puzzle by considering both types of steps.

* OR nodes: You have different paths to choose from, and you only need one.
* AND nodes: You must complete multiple steps together.

**How AO\* works:**

* It looks at both options and finds the cheapest or best way to complete the goal.
* If buying the toy is faster and cheaper, it chooses that (OR).
* If making it is better, it ensures all steps are done together (AND).
* Unlike normal searches (like A\*), AO\* understands that sometimes multiple steps must happen together and searches accordingly.

**Code:**

|  |
| --- |
| ***AO\**** |
| **Code:**  #include <stdio.h>  #include <stdlib.h>  #include <limits.h>  #define INF INT\_MAX  **// Structure for a node in the AO\* graph**  typedef struct GraphNode {      char name;                    **// Node label (e.g., 'A', 'B')**      int heuristic;                **// Estimated cost to goal**      int cost;                      **// Actual cost from AO\* search**      struct GraphNode\*\* children; **// Child nodes**      int\* edgeCosts;               **// Costs to each child node**      int numChildren;             **// Number of children**      int isAndNode;               **// 1 for AND node, 0 for OR node**  } GraphNode;  **// Function to create a new node**  GraphNode\* createNode(char name, int heuristic, int isAndNode) {      GraphNode\* newNode = (GraphNode\*)malloc(sizeof(GraphNode));      newNode->name = name;      newNode->heuristic = heuristic;      newNode->cost = INF;      newNode->children = NULL;      newNode->edgeCosts = NULL;      newNode->numChildren = 0;      newNode->isAndNode = isAndNode;      return newNode;  }  **// Function to connect a node with its children**  void addEdges(GraphNode\* parent, GraphNode\*\* children, int\* costs, int numChildren) {      parent->children = children;      parent->edgeCosts = costs;      parent->numChildren = numChildren;  }  **// AO\* Algorithm to compute the optimal cost**  void aoStar(GraphNode\* node) {      if (node->numChildren == 0) {          node->cost = node->heuristic; **// Leaf node cost is its heuristic**          return;      }      printf("Processing Node %c...\n", node->name);      int minCost = INF;    **// AND node: Sum up all child costs**      if (node->isAndNode) {          int totalCost = 0;          for (int i = 0; i < node->numChildren; i++) {              aoStar(node->children[i]);              totalCost += node->children[i]->cost + node->edgeCosts[i];          }          minCost = totalCost;      }  **// OR node: Choose the minimum cost path**  else {          for (int i = 0; i < node->numChildren; i++) {              aoStar(node->children[i]);              int pathCost = node->children[i]->cost + node->edgeCosts[i];              printf("\nChecking path %c -> %c | Cost: %d\n", node->name, node->children[i]->name, pathCost);              if (pathCost < minCost) {                  minCost = pathCost;              }          }      }      node->cost = minCost;      printf("\nFinal cost at Node %c: %d\n", node->name, node->cost);  }  **// Function to print the optimal path**  void printOptimalPath(GraphNode\* node) {      if (node == NULL) return;      printf("%c (Cost: %d) ", node->name, node->cost);        if (node->numChildren > 0) {          if (node->isAndNode) {              for (int i = 0; i < node->numChildren; i++) {                  printOptimalPath(node->children[i]);              }          } else {              int minCost = INF;              GraphNode\* bestChild = NULL;              for (int i = 0; i < node->numChildren; i++) {                  int pathCost = node->children[i]->cost + node->edgeCosts[i];                  if (pathCost < minCost) {                      minCost = pathCost;                      bestChild = node->children[i];                  }              }              printOptimalPath(bestChild);          }      }  }  int main() {  **// Creating nodes**      GraphNode\* S = createNode('S', 10, 0);      GraphNode\* A = createNode('A', 5, 1);      GraphNode\* B = createNode('B', 6, 1);      GraphNode\* C = createNode('C', 2, 0);      GraphNode\* D = createNode('D', 1, 0);      GraphNode\* E = createNode('E', 3, 0);      GraphNode\* P = createNode('P', 0, 0);      GraphNode\* Q = createNode('Q', 0, 0);      GraphNode\* R = createNode('R', 0, 0);  **// Connecting edges**      GraphNode\* childrenA[] = {C, D};      int costsA[] = {3, 4};      addEdges(A, childrenA, costsA, 2);      GraphNode\* childrenB[] = {E, P};      int costsB[] = {4, 5};      addEdges(B, childrenB, costsB, 2);      GraphNode\* childrenC[] = {Q};      int costsC[] = {2};      addEdges(C, childrenC, costsC, 1);      GraphNode\* childrenD[] = {R};      int costsD[] = {1};      addEdges(D, childrenD, costsD, 1);      GraphNode\* childrenE[] = {P};      int costsE[] = {3};      addEdges(E, childrenE, costsE, 1);      GraphNode\* childrenS[] = {A, B};      int costsS[] = {2, 3};      addEdges(S, childrenS, costsS, 2);  **// Running AO\***      printf("Running AO\* Algorithm...\n\n");      aoStar(S);    **// Printing results**      printf("\nOptimal cost from start node: %d\n", S->cost);      printf("Optimal Path: ");      printOptimalPath(S);      printf("\n");        return 0;  } |
| **Output:** |

**Conclusion:**

* In an AND/OR graph, the optimal path represents the best sequence of decisions leading to a solution while considering both AND and OR relationships between nodes.
* The AO\* algorithm, which integrates elements of A\* search with decision trees and structured problem spaces, helps efficiently determine this path.
* It evaluates path costs, explores subgoals while managing constraints, and provides suboptimal solutions when necessary to ensure the best possible outcome.