# Assignment 02

**Name -** Sk Fardeen Hossain
**Enrollment No -** 2021CSB023
**Department -** Computer Science and Technology

## Question 01:-

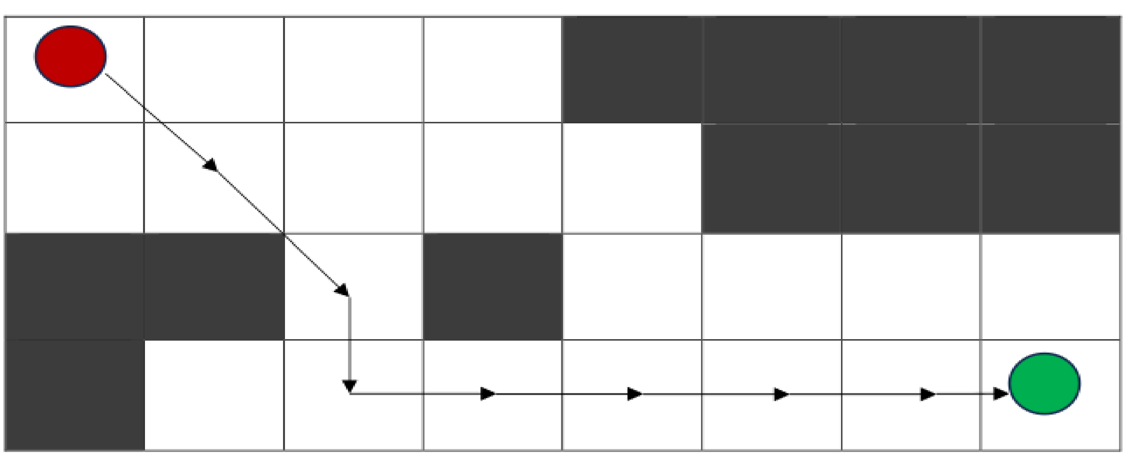
1. In a spatial context defined by a square grid featuring numerous obstacles, a task is presented wherein a starting cell, and a target cell are specified. The objective is to efficiently traverse from the starting cell to the target cell, optimizing for expeditious navigation. In this scenario, the A* Search algorithm proves instrumental.

   The A* Search algorithm operates by meticulously selecting nodes within the grid, employing a parameter denoted as 'f.' This parameter, critical to the decision-making process, is the summation of two distinct parameters – 'g' and 'h.' At each iterative step, the algorithm strategically identifies the node with the lowest 'f' value and progresses the exploration accordingly. The allowed actions are: ***left, right, top, bottom, and diagonal.***

   The parameters 'g' and 'h' are delineated as follows:
   - 'g': Represents the cumulative movement cost incurred in traversing the path from the designated starting point to the current square on the grid.

   - 'h': Constitutes the estimated movement cost anticipated for the traversal from the current square on the grid to the specified destination, by using either Manhattan or Euclidean distance. This element, often denoted as the heuristic, embodies an intelligent estimation.

   The A* Search algorithm, distinguished by its ability to efficiently find optimal or near-optimal paths amidst obstacles, holds significant applicability in diverse domains such as robotics, gaming, and route planning.



## Idea

Here we have to use the A* search algorithm, where we expand the node of a state that has the optimal f(n) value, where f(n)=g(n)+h(n),
g:- Actual cost to reach the state from the start state
h:- Predictive(Heuristic) cost to reach the goal state from the current state.

State in this problem can be described as a tuple containing 2 things:-

A. Current Cell's g(n) and h(n) values
B. Parent cell (comes previous in the sequence from start to goal state)

We stop the algorithm whenever we find the goal state. We maintain a priority queue to order the states based on their f(n) values. Optimality is guaranteed when the heuristic function (h(n)) suffices admissibility condition, that is it never overestimates or underestimates the actual cost.

Here we have used both manhattan and euclidean distance between the current cell and goal cell as a heuristic function.

## Code:-

```
/*
   A* search algorithm employs searching the search space using both
heuristic(h) and
   actual cost(g) combined into a objective function(f) that is used to
penetrate the search
   space by going from the start state to the final state making decisions
at the current
   state.
   The admissibility is ensured.
   The heuristic function h(n) is referred to as acceptable,
   and it is always less than or equal to the actual cost,
```

```cpp
   which is the true cost from the present point in the path.
   H(n) is never bigger than h*(n) in this context.

*/

#include <bits/stdc++.h>
using namespace std;

struct Node
{
   int x, y;                   // Current coordinates
   double g_cost = DBL_MAX; // Cost from start to current node
   double h_cost = 0;        // Estimated cost from current to goal
   double f_cost = DBL_MAX; // Total cost (g + h)
   Node *parent = nullptr;  // Parent node

   Node(int x, int y) : x(x), y(y) {}

   bool operator<(const Node &other) const
   {
       if (x != other.x)
           return x < other.x;
       return y < other.y;
   }
};

class Compare
{
public:
   bool operator()(const Node *below, const Node *above)
   {
       return below->f_cost > above->f_cost;
   };
};

class GridSolver
{
private:
   int width, height;
   vector<vector<int>> grid;
```

```
    vector<pair<int, int>> directions = {
        {-1, 0}, {1, 0}, {0, -1}, {0, 1}, {-1, -1}, {-1, 1}, {1, -1}, {1,
1}};

    double calculateHCostEuclidean(const Node &current, const Node &goal)
    {
        double dx = current.x - goal.x;
        double dy = current.y - goal.y;
        return sqrt(dx * dx + dy * dy);
    }

    double calculateHCostManhattan(const Node &current, const Node &goal)
    {
        double dx = abs(current.x - goal.x) * 1.0;
        double dy = abs(current.y - goal.y) * 1.0;
        return dx + dy;
    }

    bool isValid(int x, int y)
    {
        return x >= 0 && x < width && y >= 0 && y < height && grid[y][x] !=
0;
    }

public:
    GridSolver(int w, int h, const vector<vector<int>> &obstacles)
        : width(w), height(h), grid(obstacles) {}

    pair<vector<pair<int, int>>, double> findPath(
        pair<int, int> start,
        pair<int, int> goal, bool useManhattan)
    {
        // Check if start and end are valid
        if (!isValid(start.first, start.second) || !isValid(goal.first,
goal.second))
        {
            return {vector<pair<int, int>>(), DBL_MAX};
        }
```

```
    Node startNode(start.first, start.second);
    Node goalNode(goal.first, goal.second);

    priority_queue<Node *, vector<Node *>, Compare> pq;
    map<Node, Node *> nodes;

    startNode.g_cost = 0;
    startNode.h_cost = useManhattan ?
calculateHCostManhattan(startNode, goalNode) :
calculateHCostEuclidean(startNode, goalNode);
    startNode.f_cost = startNode.g_cost + startNode.h_cost;

    Node *current = new Node(startNode);
    nodes[startNode] = current;
    pq.push(current);

    while (!pq.empty())
    {
        current = pq.top();
        pq.pop();

        if (current->x == goalNode.x && current->y == goalNode.y)
        {
            vector<pair<int, int>> path;
            double finalCost = current->g_cost;

            while (current != nullptr)
            {
                path.push_back({current->x, current->y});
                current = current->parent;
            }
            reverse(path.begin(), path.end());

            for (auto &pair : nodes)
            {
                delete pair.second;
            }

            return {path, finalCost};
        }
```

```
            for (const auto &dir : directions)
            {
                int newX = current->x + dir.first;
                int newY = current->y + dir.second;

                if (!isValid(newX, newY))
                    continue;

                // double movementCost = (dir.first != 0 && dir.second !=
0) ? sqrt(2.0) : 1.0;
                double movementCost = 1.0; // Considering diagonal movement
is treated same as vertical or horizontal movement
                double tentative_g = current->g_cost + movementCost;

                Node neighbor(newX, newY);
                if (nodes.find(neighbor) == nodes.end())
                {
                    nodes[neighbor] = new Node(newX, newY);
                }

                Node *neighborNode = nodes[neighbor];
                if (tentative_g < neighborNode->g_cost)
                {
                    neighborNode->parent = current;
                    neighborNode->g_cost = tentative_g;
                    neighborNode->h_cost = useManhattan ?
calculateHCostManhattan(*neighborNode, goalNode) :
calculateHCostEuclidean(*neighborNode, goalNode);
                    neighborNode->f_cost = neighborNode->g_cost +
neighborNode->h_cost;
                    pq.push(neighborNode);
                }
            }
        }

        for (auto &pair : nodes)
        {
            delete pair.second;
        }
```

```cpp
        return {vector<pair<int, int>>(), DBL_MAX};
    }
};

int main()
{
    int m, n;
    cout << "Enter the number of rows of the grid:\n";
    cin >> m;
    cout << "Enter the number of columns in the grid:\n";
    cin >> n;

    vector<vector<int>> grid(m, vector<int>(n, 1));

    cout << "Enter the number of obstacles in the grid:\n";
    int q;
    cin >> q;
    while (q--)
    {
        cout << "Enter the obstacle coordinates:\n";
        int x, y;
        cin >> x >> y;
        if (x >= 0 && x < m && y >= 0 && y < n)
        {
            grid[x][y] = 0;
        }
    }

    GridSolver gs(n, m, grid);

    cout << "Enter the start state coordinates:\n";
    int sx, sy;
    cin >> sx >> sy;
    cout << "Enter the goal state coordinates:\n";
    int ex, ey;
    cin >> ex >> ey;

    // Manhattan heuristic
    auto result = gs.findPath({sy, sx}, {ey, ex}, true);
    if (result.second == DBL_MAX)
```

```
    {
        cout << "Could not find a path using Manhattan heuristic\n";
    }
    else
    {
        cout << "Path found using Manhattan heuristic with cost " <<
result.second << ":\n";
        for (int i = 0; i < result.first.size(); i++)
        {
            if (i < result.first.size() - 1)
                cout << "< " << result.first[i].second << " , " <<
result.first[i].first << " > ---- ";
            else
                cout << "< " << result.first[i].second << " , " <<
result.first[i].first << " >";
        }
        cout << "\n";
    }

    // Euclidean heuristic
    result = gs.findPath({sy, sx}, {ey, ex}, false);
    if (result.second == DBL_MAX)
    {
        cout << "Could not find a path using Euclidean heuristic\n";
    }
    else
    {
        cout << "Path found using Euclidean heuristic with cost " <<
result.second << ":\n";
        for (int i = 0; i < result.first.size(); i++)
        {
            if (i < result.first.size() - 1)
                cout << "< " << result.first[i].second << " , " <<
result.first[i].first << " > ---- ";
            else
                cout << "< " << result.first[i].second << " , " <<
result.first[i].first << " >";
        }
        cout << "\n";
    }
```

```
    return 0;
}
```

## Output :-

The grid given in the question was given as a input, in brief:-

1. (0,0) is the start state
2. (3,7) is the goal state
3. 11 obstacles are there namely (0,4),(0,5)....

*It is important to note here, that diagonal movements and linear movements are given the same weight in the actual cost computation, although geometrically diagonal movements cost roughly 1.414 times more than the linear movements.*

```
Enter the start state coordinates:
0 0
Enter the goal state coordinates:
3 7
Path found using Manhattan heuristic with cost 7:
< 0 , 0 > ---- < 1 , 1 > ---- < 2 , 2 > ---- < 3 , 3 > ---- < 3 , 4 > ---- < 3 , 5 > ---- < 3 , 6 > ---- < 3 , 7 >
Path found using Euclidean heuristic with cost 7:
< 0 , 0 > ---- < 1 , 1 > ---- < 2 , 2 > ---- < 3 , 3 > ---- < 3 , 4 > ---- < 3 , 5 > ---- < 3 , 6 > ---- < 3 , 7 >
```

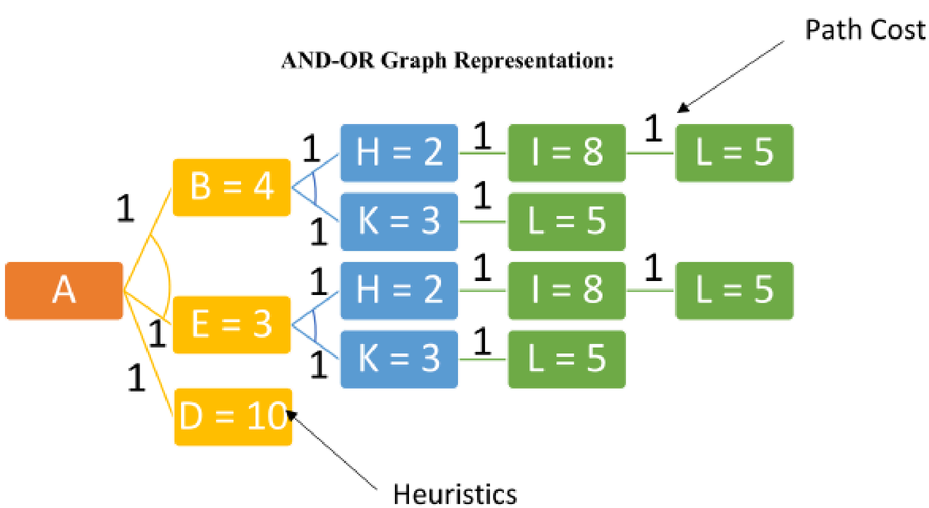
# Question 02:-

2. In a spatial context defined by a square matrix of order N * N, a rat is situated at the starting point (0,0), aiming to reach the destination at (N-1, N-1). The task at hand is to enumerate all feasible paths that the rat can undertake to traverse from the source to the destination. The permissible directions for the rat's movement are denoted as 'U' (up), 'D' (down), 'L' (left), and 'R' (right). Within this matrix, a cell assigned the value 0 signifies an obstruction, rendering it impassable for the rat, while a value of 1 indicates a traversable cell. The objective is to furnish a list of paths in lexicographically increasing order, with the constraint that no cell can be revisited along the path. Moreover, if the source cell is assigned a value of 0, the rat is precluded from moving to any other cell.

   To accomplish this, the AO* Search algorithm is employed to systematically explore the AND-OR graph and evaluate all conceivable paths from source to destination (with path cost = 1, and heuristic values given in the diagram). The algorithm dynamically adapts its heuristic function during the search, optimizing the exploration process. The resultant list of paths reflects a meticulous exploration of the matrix, ensuring lexicographical order and adherence to the specified constraints.

| | | |
|---|---|---|
| **Source (A)** | B = 4 | C |
| D = 10 | E = 3 | F |
| G | H = 2 | I = 8 |
| J | K = 3 | **Destination (L = 5)** |

**AND-OR Graph Representation:**



## Idea

The *AO algorithm** is a powerful best-first search method used to solve problems that can be represented as a directed acyclic graph (DAG). Unlike traditional algorithms like A*, which explore a single path, the AO* algorithm evaluates multiple paths simultaneously. This makes it more efficient for problems that involve AND-OR nodes. The AND nodes represent tasks where all child nodes must be satisfied, while OR nodes offer multiple alternative paths where only one child node needs to be satisfied to achieve the goal.

AO* search unlike A* does not guarantee optimality as it stops whenever a solution is found, although it takes less memory.

Pseudo Code:-

```
algorithm AOStar(Graph, StartNode):
    // INPUT
    //    Graph = the graph to search
    //    StartNode = the starting node
    // OUTPUT
    //    The minimum cost path from StartNode to GoalNode

    CurrentNode <- StartNode

    while there is a new path with lower cost from StartNode to GoalNode:
        calculate the cost of path from the current node to the goal node
          through each of its successor nodes

        if the successor node is connected to other successor nodes by AND-ARCS:
            sum up the cost of all paths in the AND-ARC
            return the total cost
        else:
            calculate the cost of the single path in the OR side
            return the single cost

        find the minimum cost path

        CurrentNode <- Successor Node Of Minimum Cost Path

        if CurrentNode has no successor node:
            do the backpropagation and correct the estimated costs
            CurrentNode <- StartNode
            return CurrentNode, New estimated costs
        else:
            return null

    return The minimum cost path
```

## Code:-

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Node
{
   string id;                                  // Label
   int value;                                  // Heuristic Value
   bool isGoal;                                // whether goal state or not
   bool isObstacle;                            // whether obstacle or not
   double totalCost;                           // totalCost -> actual +
heuristic
   vector<Node *> successors;                  // successor nodes in the path
   vector<pair<Node *, Node *>> andPairs; // Nodes joined via hyperedge

   Node(string id, int value, bool isGoal = false, bool isObstacle =
false)
       : id(id), value(value), isGoal(isGoal), isObstacle(isObstacle),
totalCost(numeric_limits<double>::infinity()) {}
};

class AOStarSearch
{
private:
   Node *startNode;
   unordered_map<string, Node *> graph; // Label to Node map

   void evaluateNode(Node *node, set<string> &visited)
   {
       if (visited.count(node->id))
           return;
       visited.insert(node->id);

       if (node->isGoal)
       {
           node->totalCost = node->value;
           return;
       }
```

```
        // First evaluate all successors
        for (auto &successor : node->successors)
        {
            if (!successor->isObstacle && !visited.count(successor->id))
            {
                evaluateNode(successor, visited);
            }
        }

        // OR nodes
        double minCost = numeric_limits<double>::infinity();
        for (auto &successor : node->successors)
        {
            if (!successor->isObstacle)
            {
                double cost = node->value + successor->totalCost + 1; // 1
for the actual edge cost
                minCost = min(minCost, cost);
            }
        }

        // AND pairs
        for (const auto &andPair : node->andPairs)
        {
            if (!andPair.first->isObstacle && !andPair.second->isObstacle)
            {
                double andCost = node->value + andPair.first->totalCost +
andPair.second->totalCost + 2; // taking twice edge cost
                minCost = min(minCost, andCost);
            }
        }

        node->totalCost = minCost;
    }

public:
    AOStarSearch(Node *start) : startNode(start)
    {
        graph[start->id] = start;
    }
```

```cpp
void addNode(Node *node)
{
    graph[node->id] = node;
}

void addOREdge(const string &u, const string &v)
{
    if (graph.count(u) && graph.count(v))
    {
        graph[u]->successors.push_back(graph[v]);
    }
}

void addANDPair(const string &u, const string &v1, const string &v2)
{
    if (graph.count(u) && graph.count(v1) && graph.count(v2))
    {
        graph[u]->andPairs.push_back({graph[v1], graph[v2]});
    }
}

vector<string> findPath()
{
    vector<string> path;
    set<string> visited;

    // Initial evaluation
    evaluateNode(startNode, visited);
    visited.clear();

    // Trace the path
    Node *current = startNode;
    while (current && !visited.count(current->id))
    {
        path.push_back(current->id);
        visited.insert(current->id);

        if (current->isGoal)
            break;
```

```
            // Find next node
            Node *next = NULL;
            double minCost = numeric_limits<double>::infinity();

            // Check regular successors
            for (auto &successor : current->successors)
            {
                if (!successor->isObstacle &&
!visited.count(successor->id))
                {
                    if (successor->totalCost + 1 < minCost)
                    {
                        minCost = successor->totalCost + 1;
                        next = successor;
                    }
                }
            }

            // Check AND pairs
            for (const auto &andPair : current->andPairs)
            {
                if (!andPair.first->isObstacle &&
!andPair.second->isObstacle)
                {
                    double pairCost = andPair.first->totalCost +
andPair.second->totalCost + 2;
                    if (pairCost < minCost)
                    {
                        minCost = pairCost;
                        // Choose the unvisited node with lower cost
                        if (!visited.count(andPair.first->id) &&
                            (!visited.count(andPair.second->id) &&
andPair.first->totalCost <= andPair.second->totalCost))
                        {
                            next = andPair.first;
                        }
                        else if (!visited.count(andPair.second->id))
                        {
                            next = andPair.second;
```

```cpp
                        }
                    }
                }
            }

            current = next;
        }

        return path;
    }

    void printCosts()
    {
        cout << "Final Node costs:\n";
        for (const auto &[id, node] : graph)
        {
            cout << id << ": " << node->totalCost << "\n";
        }
    }

    ~AOStarSearch()
    {
        for (auto &[id, node] : graph)
        {
            delete node;
        }
    }
};

int main()
{
    Node *nodeA = new Node("A", 0);
    Node *nodeB = new Node("B", 4);
    Node *nodeC = new Node("C", 0, false, true);
    Node *nodeD = new Node("D", 10);
    Node *nodeE = new Node("E", 3);
    Node *nodeF = new Node("F", 0, false, true);
    Node *nodeG = new Node("G", 0, false, true);
    Node *nodeH = new Node("H", 2);
    Node *nodeI = new Node("I", 8);
```

```cpp
Node *nodeJ = new Node("J", 0, false, true);
Node *nodeK = new Node("K", 3);
Node *nodeL = new Node("L", 5, true);

AOStarSearch search(nodeA);

vector<Node *> nodes = {nodeA, nodeB, nodeC, nodeD, nodeE, nodeF,
                        nodeG, nodeH, nodeI, nodeJ, nodeK, nodeL};
for (const auto &node : nodes)
{
    search.addNode(node);
}

// OR edges
search.addOREdge("A", "B");
search.addOREdge("A", "D");
search.addOREdge("A", "E");
search.addOREdge("B", "H");
search.addOREdge("B", "K");
search.addOREdge("E", "H");
search.addOREdge("E", "K");
search.addOREdge("H", "I");
search.addOREdge("K", "L");
search.addOREdge("I", "L");

// AND pairs    ---> HyperEdges
search.addANDPair("A", "B", "E");
search.addANDPair("B", "H", "K");
search.addANDPair("E", "H", "K");

vector<string> path = search.findPath();

cout << "Optimal path: ";
for (int i = 0; i < path.size(); ++i)
{
    cout << path[i];
    if (i < path.size() - 1)
    {
        cout << " -> ";
    }
```

```
    }
    cout << "\n";

    search.printCosts();

    return 0;
}
```

## Output :-

Input is not given, as it will become cumbersome if the ORedges and ANDpairs are given as user inputs, building the graph will take up more complexity, which is not the concern of our problem. So, I have embedded the initial configuration of the problem, namely the blocked cells, the OR edges and the AND pairs in the graph along with their corresponding heuristic values(which are changed dynamically) in the code itself.

```
Optimal path: A -> E -> K -> L
Final Node costs:
L: 5
J: inf
G: inf
F: inf
K: 9
E: 13
H: 17
B: 14
I: 14
D: inf
C: inf
A: 14
```