# Indian Institute of Information Technology Surat

# Lab Report on
# Artificial Intelligence (CS 701) Practical

**Submitted by**

**[RAHUL KUMAR SINGH] (UI21CS44)**

**Course Faculty**

**Dr. Ritesh Kumar**

**Mrs. Archana Balmik**

**Department of Computer Science and Engineering**

**Indian Institute of Information Technology Surat**

**Gujarat-394190, India**

**Aug-2024**

# Lab No: 7

## Aim:

Write a code to implement A* and AO* Algorithms.

## Description:

**A* Algorithm**

- **Node Class**: Represents positions, parent, and cost metrics (g, h, f).
- **Heuristic**: Uses Manhattan distance for cost estimation to the goal.
- **Pathfinding**: Explores nodes, reconstructing the optimal path from goal to start.

**AO* Algorithm**

- **Node Class**: Represents positions, parent, children, and costs.
- **Pathfinding**: Uses a queue to explore nodes, updating costs and parents.
- **Goal Check**: Determines success by matching the current node to the goal position.

## Code:

**A)A***

```
import heapq
class ANode:
    def __init__(self, position, parent=None):
        self.position = position
        self.parent = parent
        self.g = 0
        self.h = 0
        self.f = 0
    def __eq__(self, other):
        return self.position == other.position
    def __lt__(self, other):
        return self.f < other.f

def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def get_neighbors(position, grid):
    neighbors = []
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    for d in directions:
        new_pos = (position[0] + d[0], position[1] + d[1])
        if 0 <= new_pos[0] < len(grid) and 0 <= new_pos[1] < len(grid[0]) and
grid[new_pos[0]][new_pos[1]] == 0:
            neighbors.append(new_pos)
    return neighbors

def astar(grid, start, goal):
    open_list = []
    closed_list = []
```

```python
    start_node = ANode(start)
    goal_node = ANode(goal)
    heapq.heappush(open_list, (start_node.f, start_node))
    while open_list:
        current_node = heapq.heappop(open_list)[1]
        closed_list.append(current_node)
        if current_node == goal_node:
            path = []
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]
        neighbors = get_neighbors(current_node.position, grid)
        for neighbor_pos in neighbors:
            node_position = neighbor_pos
            neighbor_node = ANode(node_position, current_node)
            if neighbor_node in closed_list:
                continue
            neighbor_node.g = current_node.g + 1
            neighbor_node.h = heuristic(neighbor_node.position, goal_node.position)
            neighbor_node.f = neighbor_node.g + neighbor_node.h
            if add_to_open(open_list, neighbor_node):
                heapq.heappush(open_list, (neighbor_node.f, neighbor_node))
    return None

def add_to_open(open_list, neighbor):
    for ANode in open_list:
        if neighbor == ANode[1] and neighbor.g > ANode[1].g:
            return False
    return True


grid = [[0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0],
        [0, 0, 0, 0, 0],
        [0, 1, 1, 0, 0],
        [0, 0, 0, 0, 0]]

start = (0, 0)
goal = (4, 4)
path = astar(grid, start, goal)
print("A* Path:", path)
```

**B)AO***

```python
class AONode:
    def __init__(self, position, parent=None):
        self.position = position
        self.parent = parent
        self.children = []
        self.cost = float('inf')
        self.is_goal = False
```

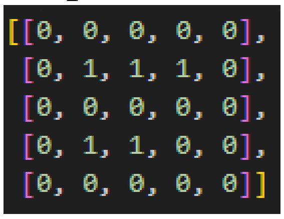
```
def get_neighbors(position, grid):
    neighbors = []
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    for d in directions:
        new_pos = (position[0] + d[0], position[1] + d[1])
        if 0 <= new_pos[0] < len(grid) and 0 <= new_pos[1] < len(grid[0]) and
grid[new_pos[0]][new_pos[1]] == 0:
            neighbors.append(new_pos)
    return neighbors

def ao_star(start_node, goal_position, grid):
    start_node.cost = 0
    agenda = [start_node]
    while agenda:
        current_node = agenda.pop(0)
        if current_node.position == goal_position:
            return current_node
        neighbors = get_neighbors(current_node.position, grid)
        for neighbor_pos in neighbors:
            child_node = AONode(neighbor_pos)
            child_cost = current_node.cost + 1
            if child_cost < child_node.cost:
                child_node.cost = child_cost
                child_node.parent = current_node
                if child_node not in agenda:
                    agenda.append(child_node)
    return None

def get_path(node):
    path = []
    while node:
        path.append(node.position)
        node = node.parent
    return path[::-1]

grid = [[0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0],
        [0, 0, 0, 0, 0],
        [0, 1, 1, 0, 0],
        [0, 0, 0, 0, 0]]
start_position = (0, 0)
goal_position = (4, 4)
start_node = AONode(start_position)
goal_node = ao_star(start_node, goal_position, grid)
if goal_node:
    path = get_path(goal_node)
    print("AO* Path:", path)
else:
    print("No path found")
```

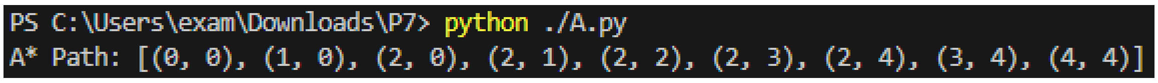
## Input:



```
[[0, 0, 0, 0, 0],
 [0, 1, 1, 1, 0],
 [0, 0, 0, 0, 0],
 [0, 1, 1, 0, 0],
 [0, 0, 0, 0, 0]]
```

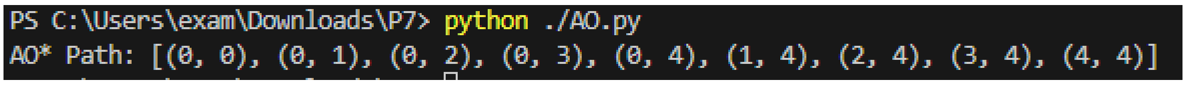
## Output:

**A)A***



```
PS C:\Users\exam\Downloads\P7> python ./A.py
A* Path: [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (2, 3), (2, 4), (3, 4), (4, 4)]
```

**B)AO***



```
PS C:\Users\exam\Downloads\P7> python ./AO.py
AO* Path: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (4, 4)]
```

## Conclusion:

- Increasing demand in robotics and autonomous vehicles.
- A* Benefits: Optimal and efficient for grid-based pathfinding.
- AO* Benefits: Ideal for multi-agent and goal-oriented problems.
- Dijkstra’s, RRT, and neural networks are recent alternatives that are far more efficient methods
- Combining algorithms yields better results but it depends on environment complexity