## 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 = []

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\***

**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)**

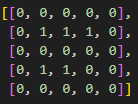
**print("AO\* Path:", path)**

**else:**

**print("No path found")**

## 

## Input:



## Output:

**A)A\***



**B)AO\***

****

## 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