Experiment-1.1

Student Name: Aayushi Kumari | **UID:** 21BCS3403
Branch: BE-CSE | **Section/Group:** 21BCS_IOT-620A
Semester: 05 | **Date of Performance:** 14/08/23
Subject Name: AI&ML with Lab | **Subject Code:** 21CSH-316

1. **Aim:** Evaluate the performance and effectiveness of the A* algorithm implementation in Python.

2. **Objective:** The objective is to assess how well the A* algorithm in solving a specific problem or scenario, and to analyze its effectiveness in comparison to other algorithms or approaches.

3. **Input/Apparatus Used:** PC, Python Programming Language, A* Implementation, Problem scenario for testing the algorithm.

4. **Theory:** The A* (Pronounced 'A Star') algorithm is a popular and graph traversal algorithm commonly used in various real-world applications that involve navigating through spaces or networks. It's particularly useful when you need the cost of moving between different nodes or locations.

Here are some real-world applications of the A* algorithm:

Robotics and Autonomous Navigation,
Video games, Maps and GPS Navigation,
Network Routing, Path Planning for unnamed Aerial Vehicles (UAV's),
Puzzle Solving, Medical Language,
Natural Language Processing.

Strengths:

Completeness
Efficiency (in many Cases),
Adaptability
Optimization Potential.

Weakness:

Heuristic Sensitivity, Memory Usage, Time Complexity in Worst Cases, Graphs with High Branching Factor, Noisy or Changing Environment.

5. Code:

```
import heapq
class Node:
    def___init_(self, position, parent=None):
        self.position = position
        self.parent = parent
        self.g = 0  # Cost from start node to current node
        self.h = 0  # Heuristic (estimated cost) from current node to goal node
        self.f = 0  # Total cost (g + h)

    def___lt_(self, other):
        return self.f < other.f
def heuristic(node, goal):
    # Manhattan distance heuristic (can be changed to Euclidean distance or others)
    return abs(node.position[0] - goal[0]) + abs(node.position[1] - goal[1])
def astar(grid, start, goal):
    open_list =  []
    closed_set = set()
    start_node = Node(start)
    goal_node = Node(goal)
    heapq.heappush(open_list, start_node)
    while open_list:
        current_node = heapq.heappop(open_list)
        if current_node.position == goal_node.position:
            path = []
            while current_node is not None:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]
        closed_set.add(current_node.position)
        for next_position in [(0, -1), (0, 1), (-1, 0), (1, 0)]:  # Possible adjacent positions
            node_position = (current_node.position[0] + next_position[0], current_node.position[1]
      + next_position[1])
            if node_position[0] < 0 or node_position[0] >= len(grid) or node_position[1] < 0 or
      node_position[1] >= len(grid[0]):
                continue
            if grid[node_position[0]][node_position[1]] == 1:
                continue
```

```
            if node_position in closed_set:
                continue
            new_node = Node(node_position, current_node)
            new_node.g = current_node.g + 1
            new_node.h = heuristic(new_node, goal_node.position)
            new_node.f = new_node.g + new_node.h

            for node in open_list:
                if new_node.position == node.position and new_node.f >= node.f:
                    break
            else:
                heapq.heappush(open_list, new_node)
    return None  # No path found

# Example usage:
grid = [
    [0, 0, 0, 0],
    [0, 1, 1, 0],
    [0, 0, 0, 0],
    [0, 0, 1, 0]
]

start_point = (0, 0)
goal_point = (3, 3)
path = astar(grid, start_point, goal_point)
      print(path)  # Output: [(0, 0), (1, 0), (2, 0), (3, 0), (3, 1), (3, 2), (3, 3)]
```

6. Result: