Experiment-1.1

Student Name: Tirukala Gowtham **UID:** 21BCS7240
Branch: BE-CSE **Section/Group:**21BCS_IOT-624A
Semester: 05 **Date of Performance:** 08_/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. Algorithm:

- Initialize the open list
- Initialize the closed list
  Put the starting node on the open list (You can leave its f at zero)
- While the open list is not empty
  - ➢ Find the node with the least f on the open list, call it "q"
  - ➢ Pop q off the open list
  - ➢ Generate q's 8 successors and set their parents to q
  - ➢ For each successor
    1. If successor is the goal, stop search
    2. Else compute both g and h for successor.
       G=q.g+distance between successor and q
       Successor h = distance between goal and successor
       Successor f=successor g + successor h
    3. If a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor.
    4. If a node with the same position as successor is in the CLOSED list which has a lower f than successor, skip the successor otherwise, add the node to the open list end (for loop).
  - ➢ Push q on the closed list end (while loop).

6. Code:

```
import heapq
graph = {
'A': [('B', 6), ('F', 3)],
'B': [('G', 8)],
'F': [('G', 1), ('H', 6)],
'G': [('I', 3)],
'H': [('E', 2), ('J', 3)],
'I': [('E', 5), ('H', 1), ('J', 3)],
'E': [],
'J': []
}
heuristic = {
'A': 10,
'B': 7,
'F': 3,
'G': 6,
'H': 4,
```

```
'I': 1,
'E': 5,
'J': 0,
}
def astar(start, goal):
    open_list = [(0, start)]
    came_from = {}
    g_score = {node: float('inf') for node in graph}
    g_score[start] = 0
    f_score = {node: float('inf') for node in graph}
    f_score[start] = heuristic[start]
    while open_list:
        current_f, current = heapq.heappop(open_list)
        if current == goal:
            path = []
            while current in came_from:
                    path.insert(0, current)
                    current = came_from[current]
            path.insert(0, start)
            return path
        for neighbor, cost in graph[current]:
            tentative_g = g_score[current] + cost
            if tentative_g < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score[neighbor] = g_score[neighbor] + heuristic[neighbor]
                heapq.heappush(open_list, (f_score[neighbor], neighbor) )
    return None # No path found
start_node = 'A'
goal_node = 'E'
shortest_path = astar(start_node, goal_node)
if shortest_path:
    print(f"Shortest path from {start_node} to {goal_node}: {'-
>'.join(shortest_path)}")
else:
    print(f"No path found from {start_node} to {goal_node}")
```

7. Result:

```
PS C:\Users\91739> & C:/Users/91739/AppData/Lo
Shortest path from A to E: A->F->G->I->H->E
```