Experiment 1.1

Student Name: Utkarsh Joshi
UID:21BCS9158
Branch: CSE
Section/Group-802/A
Semester: 5th
Date of Performance:10/08/23
Subject Name-AIML
Subject Code:21CSH-316

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

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

3. **Algorithm/Flowchart:**

Step 1: Add the beginning node to the open list
Step 2: Repeat the following step

In the open list, find the square with the lowest F cost, which denotes the current square. Now we move to the closed square.

Consider 8 squares adjacent to the current square and Ignore it if it is on the closed list or if it is not workable. Do the following if it is workable.

Check if it is on the open list; if not, add it. You need to make the current square as this square's a parent. You will now record the different costs of the square, like the F, G, and H costs.

If it is on the open list, use G cost to measure the better path. The lower the G cost, the better the path. If this path is better, make the current square as the parent square. Now you need to recalculate the other scores – the G and F scores of this square.

– You'll stop:

If you find the path, you need to check the closed list and add the target square to it.

There is no path if the open list is empty and you cannot find the target square.

Step 3. Now you can save the path and work backward, starting from the target square, going to the parent square from each square you go, till it takes you to the starting square. You've found your path now.

4. **Steps for experiment/practical/Code:**

```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}              #store distance from starting node
    parents = {}        # parents contains an adjacency map of all nodes
    #distance of starting node from itself is zero
    g[start_node] = 0
    #start_node is root node i.e it has no parent nodes
    #so start_node is set to its own parent node
    parents[start_node] = start_node
    while len(open_set) > 0:
        n = None
        #node with lowest f() is found
        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v
        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m, weight) in get_neighbors(n):
                #nodes 'm' not in first and last set are added to first
                #n is set its parent
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                #for each node m,compare its distance from start i.e g(m) to the
                #from start through n node
                else:
                    if g[m] > g[n] + weight:
                        #update g(m)
                        g[m] = g[n] + weight
                        #change parent of m to n
                        parents[m] = n
                        #if m in closed set,remove and add to open
```

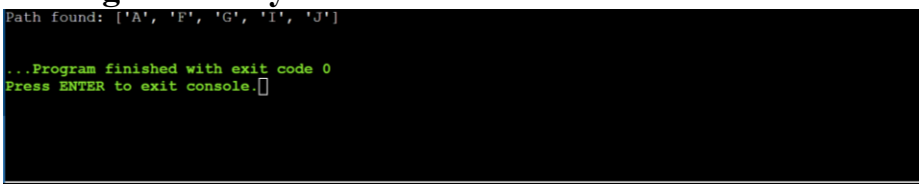
```
                if m in closed_set:
                    closed_set.remove(m)
                    open_set.add(m)
        if n == None:
            print('Path does not exist!')
            return None

        # if the current node is the stop_node
        # then we begin reconstructin the path from it to the start_node
        if n == stop_node:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(start_node)
            path.reverse()
            print('Path found: {}'.format(path))
            return path
        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected
        open_set.remove(n)
        closed_set.add(n)
    print('Path does not exist!')
    return None

#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None

        #for simplicity we ll consider heuristic distances given
#and this function returns heuristic distance for all nodes
```

```
def heuristic(n):
    H_dist = {
        'A': 1,
        'B': 4,
        'C': 6,
        'D': 7,
        'E': 8,
        'F': 9,
        'G': 5,
        'H': 3,
        'I': 2,
        'J': 0,
    }
    return H_dist[n]

#Describe your graph here
Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('A', 6), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 1), ('E', 5)],
    'D': [('B', 2), ('C', 1), ('E', 8)],
    'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],
    'F': [('A', 3), ('G', 1), ('H', 7)],
    'G': [('F', 1), ('I', 3)],
    'H': [('F', 7), ('I', 2)],
    'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],
}

aStarAlgo('A','J')
```

5. Observations/Discussions/ Complexity Analysis:

1. Record the execution time of the A* algorithm for each problem scenario.
2. Note the number of nodes expanded during the algorithm's execution.
3. Record the optimal path generated by the A* algorithm.
4. Evaluate the correctness of the generated path by comparing it with known optimal solutions, if available.
5. Analyze the efficiency and effectiveness of the A* algorithm based on the execution time, number of nodes expanded, and the quality of the generated paths.
6. Compare the performance of the A* algorithm with other algorithms or approaches, if applicable.

7. Result/Output/Writing Summary:



Learning outcomes (What I have learnt):

1.Learn About A* algorithm

2.Learn About how to find shortest path

3.Learn about python programing language

Evaluation Grid (To be created as per the SOP and Assessment guidelines by the faculty):

Sr. No.	Parameters	Marks Obtained	Maximum Marks
1.			
2.			
3.			