Experiment 4

Student Name: Rahul Saxena

UID: 24MCI10204

Branch: MCA (AI-ML)

Section/Group: MAM - 3(B)

Semester: II

Subject Name: Artificial Intelligence Lab [24CAP-674]

Aim: Implement A* Search Algorithm.

Steps:

1: Define the Graph

- The graph is represented as a dictionary where each key is a node, and the value is a list of tuples containing the neighboring nodes and their respective edge costs.

2: Define the Heuristic Function

- A heuristic dictionary is created to estimate the cost from each node to the goal node.

*3: Implement A Search Algorithm**

1. Initialize Data Structures
   - A priority queue (heapq) is used to keep track of nodes to explore.
   - A set (visited) is used to track explored nodes.
   - A dictionary (g_scores) stores the cost to reach each node from the start.
2. Push the Start Node into the Priority Queue
   - The priority queue stores tuples of (f-score, node, path).
   - The f-score is calculated as: f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n) where g(n) is the cost from the start node to n and h(n) is the heuristic cost.
3. Loop Until the Priority Queue is Empty
   - Extract the node with the lowest f-score.
   - If the node is the goal, return the path.
   - Otherwise, explore its neighbors.
4. Update Costs for Neighbors

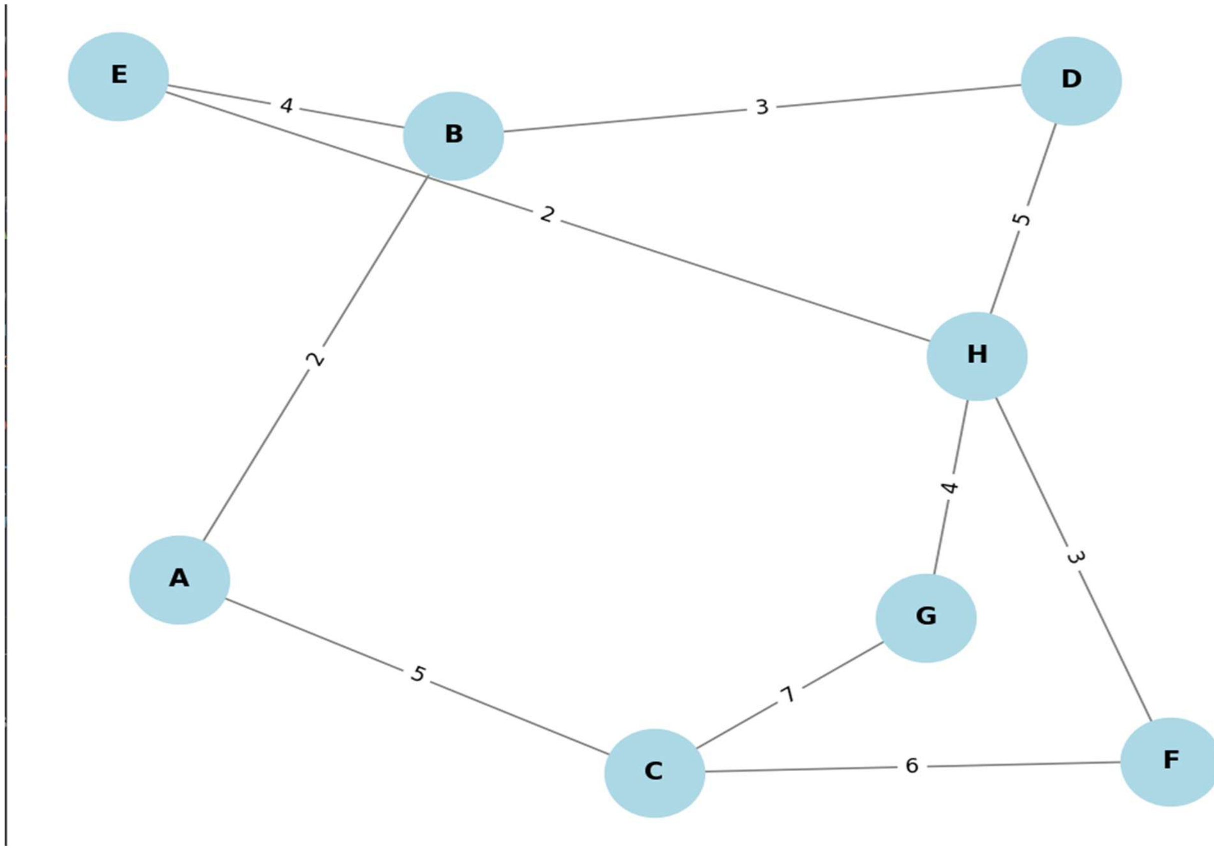
  - If a shorter path to a neighbor is found, update its g-score.
  - Compute its f-score and push it into the priority queue.
5. Continue Until Goal is Reached or No Path Exists
  - If the goal is reached, return the optimal path.
  - If no path is found, return None.

Input Code:

```
import heapq
import networkx as nx
import matplotlib.pyplot as plt
def a_star_search(graph, start, goal, heuristic):
    priority_queue = []
    heapq.heappush(priority_queue, (0 + heuristic[start], start, [start]))
    visited = set()
    g_scores = {start: 0}
    while priority_queue:
        f_score, current_node, path = heapq.heappop(priority_queue)
        if current_node in visited:
            continue
        visited.add(current_node)
        if current_node == goal:
            return path
        for neighbor in graph.neighbors(current_node):
            edge_cost = graph[current_node][neighbor]['weight']
            tentative_g_score = g_scores[current_node] + edge_cost
            if neighbor not in visited or tentative_g_score < g_scores.get(neighbor, float('inf')):
                g_scores[neighbor] = tentative_g_score
                f_score = tentative_g_score + heuristic[neighbor]
                heapq.heappush(priority_queue, (f_score, neighbor, path + [neighbor]))
    return None
G = nx.Graph()
edges = [
    ('A', 'B', 2), ('A', 'C', 5), ('B', 'D', 3), ('B', 'E', 4),
    ('C', 'F', 6), ('C', 'G', 7), ('D', 'H', 5), ('E', 'H', 2),
    ('F', 'H', 3), ('G', 'H', 4)
]
G.add_weighted_edges_from(edges)
heuristic = {
    'A': 7, 'B': 5, 'C': 6, 'D': 3,
    'E': 4, 'F': 2, 'G': 1, 'H': 0
}
start_node = 'A'
goal_node = 'H'
result_path = a_star_search(G, start_node, goal_node, heuristic)

print("A* Search Path:", result_path)
plt.figure(figsize=(8, 6))
pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_size=2000, node_color='lightblue', edge_color='gray', font_size=12, font_weight='bold')
edge_labels = {(u, v): d['weight'] for u, v, d in G.edges(data=True)}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
plt.title("Graph Representation of A* Search")
plt.show()
```

Graph:



Output:

```
PS D:\MCA\Semester 2\AI Practice> & C:/Users/saxen/AppData/Local/Programs/F
r.py"
A* Search Path: ['A', 'B', 'E', 'H']
```

Learning Outcome:

- Understanding A Search Algorithm*
  - Learn how A* search efficiently finds the shortest path using both path cost (g(n)) and heuristic (h(n)).
- Graph Representation in Python
  - Understand how to represent graphs using adjacency lists (dictionaries of lists).

- Working with Heuristic Functions
  - Learn the role of heuristic values in guiding the search process.
- Priority Queue with heapq
  - Gain hands-on experience using a priority queue to optimize searching in graphs.
- Implementation of Pathfinding Techniques
  - Understand how to track and update the best paths dynamically.