# EXPERIMENT 7

## AIM : Compare the different search algorithms.

## Theory : Short Note on BFS, DFS, UCS, and A* Search:

- **Breadth-First Search (BFS):**
  BFS explores nodes level by level, starting from the source. It uses a queue and is guaranteed to find the shortest path in an unweighted graph. It is complete and optimal for such graphs.
- **Depth-First Search (DFS):**
  DFS explores as far as possible along each branch before backtracking. It uses a stack (or recursion) and is memory efficient, but it may not find the shortest path and can get stuck in deep or infinite paths without precautions.
- **Uniform Cost Search (UCS):**
  UCS expands the node with the lowest total cost from the start. It is optimal and complete for graphs with positive edge weights, making it suitable for finding the least-cost path.
- **A* Search:**
  A* enhances UCS by using a heuristic to estimate the cost to the goal. It selects paths with the lowest combined actual cost and estimated future cost, making it efficient and optimal with an admissible heuristic.

**CODE :**

```
from queue import PriorityQueue

def bfs(graph, start, goal):
    queue = [(start, [start])]
    while queue:
        node, path = queue.pop(0)
        if node == goal:
            return path
        for neighbor in graph[node]:
            if neighbor not in path:
                queue.append((neighbor, path + [neighbor]))
    return None

def dfs(graph, start, goal, path=[]):
    path = path + [start]
    if start == goal:
        return path
    for neighbor in graph[start]:
        if neighbor not in path:
            new_path = dfs(graph, neighbor, goal, path)
            if new_path:
                return new_path
    return None
```

```
def ucs(graph, start, goal):
    queue = PriorityQueue()
    queue.put((0, start, [start]))
    while not queue.empty():
        cost, node, path = queue.get()
        if node == goal:
            return path
        for neighbor, weight in graph[node]:
            if neighbor not in path:
                queue.put((cost + weight, neighbor, path + [neighbor]))
    return None

def heuristic(node, goal):
    return abs(ord(node) - ord(goal))  # Example heuristic function

def a_star(graph, start, goal):
    queue = PriorityQueue()
    queue.put((0, start, [start]))
    while not queue.empty():
        cost, node, path = queue.get()
        if node == goal:
            return path
        for neighbor, weight in graph[node]:
            if neighbor not in path:
                total_cost = cost + weight + heuristic(neighbor, goal)
                queue.put((total_cost, neighbor, path + [neighbor]))
    return None

# Graph representation
graph_unweighted = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

graph_weighted = {
    'A': [('B', 1), ('C', 4)],
    'B': [('A', 1), ('D', 2), ('E', 5)],
    'C': [('A', 4), ('F', 3)],
    'D': [('B', 2)],
    'E': [('B', 5), ('F', 1)],
    'F': [('C', 3), ('E', 1)]
}

# Running algorithms
print("BFS:", bfs(graph_unweighted, 'A', 'F'))
print("DFS:", dfs(graph_unweighted, 'A', 'F'))
```

```
print("UCS:", ucs(graph_weighted, 'A', 'F'))
print("A*:", a_star(graph_weighted, 'A', 'F'))
```

---

```
BFS: ['A', 'C', 'F']
DFS: ['A', 'B', 'E', 'F']
UCS: ['A', 'B', 'E', 'F']
A*: ['A', 'C', 'F']
```