**1. Implement State Space Search using Breadth-First and Depth-First Search**  
**Description:** Write a Python program to define the initial and goal states of the graph. Implement two search algorithms—BFS and DFS—to find a path from the initial state to the goal state.

**Solution:**

**Breadth-First Search:**

from collections import deque

graph = {

'A':['B','C'],

'B':['D','E'],

'C':['F'],

'D':[],

'E':[],

'F':[]

}

queue = deque()

visited=set()

traversal\_order=[]

start\_node = 'A'

queue.append(start\_node)

visited.add(start\_node)

while queue:

current = queue.popleft()

traversal\_order.append(current)

for x in graph[current]:

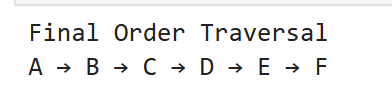
if x not in visited:

queue.append(x)

visited.add(x)

print("->".join(traversal\_order))

**Output:**



**Depth-First Search:**

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': [],

'F': []

}

visited = set()

traversal\_order = []

def dfs(node):

if node not in visited:

visited.add(node)

traversal\_order.append(node)

for neighbor in graph[node]:

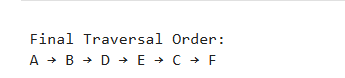
dfs(neighbor)

dfs('A')

print("\nFinal Traversal Order:")

print(" → ".join(traversal\_order))

**Output:**



**2. WAP for Best First Search and A\* search algorithm.**

**Solution:**

graph = {

'A': {'B': 1, 'C': 4},

'B': {'D': 5, 'E': 12},

'C': {'F': 2},

'D': {},

'E': {'F': 3},

'F': {}

}

heuristic = {

'A': 7,

'B': 6,

'C': 2,

'D': 1,

'E': 3,

'F': 0

}

def a\_star(start, goal):

# Each element: [node, path\_so\_far, cost\_so\_far]

open\_set = [[start, [start], 0]]

visited = []

while open\_set:

# Find the node in open\_set with the smallest f = g + h

lowest\_f\_index = 0

for i in range(1, len(open\_set)):

node, path, g = open\_set[i]

f = g + heuristic[node]

node\_lowest, \_, g\_lowest = open\_set[lowest\_f\_index]

f\_lowest = g\_lowest + heuristic[node\_lowest]

if f < f\_lowest:

lowest\_f\_index = i

current, path, g = open\_set.pop(lowest\_f\_index)

if current == goal:

return path, g # Return path and cost

visited.append(current)

# Explore neighbors

for neighbor in graph[current]:

if neighbor not in visited:

cost = graph[current][neighbor]

open\_set.append([neighbor, path + [neighbor], g + cost])

return None, float('inf')

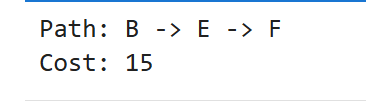
# Run and print results

path, cost = a\_star('B', 'F')

print("Path:", " -> ".join(path))

print("Cost:", cost)

**Output:**



**3. Write a program to implement AO\* Algorithm.**

**Solution:**

graph = {

'A': [['B', 'C'], ['D']],

'B': [['E'], ['F']],

'C': [['G']],

'D': [['H']],

'E': [], 'F': [], 'G': [], 'H': []

}

heuristic = {'A': 1, 'B': 1, 'C': 1, 'D': 2, 'E': 3, 'F': 2, 'G': 4, 'H': 3}

solution = {}

def ao\_star(node):

print(f"Processing: {node}")

if not graph[node]: return heuristic[node]

costs = []

for option in graph[node]:

cost = sum(heuristic[n] for n in option)

costs.append((cost, option)

best = min(costs, key=lambda x: x[0])

solution[node] = best[1]

for n in best[1]:

ao\_star(n)

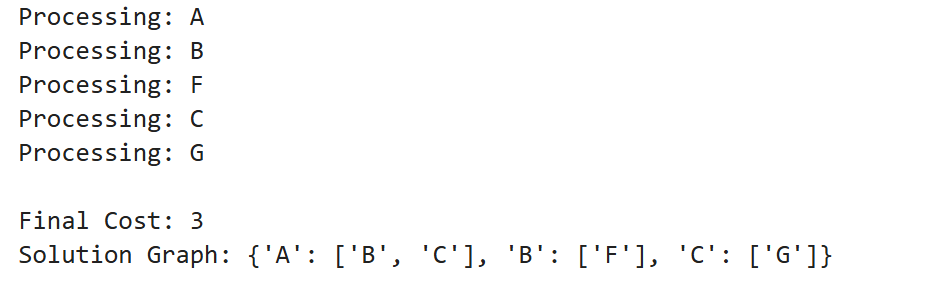
return heuristic[node] + best[0]

cost = ao\_star('A')

print("\nFinal Cost:", cost)

print("Solution Graph:", solution)

**Output:**



**4. Write a program to implement hill climbing.**