NATIONAL UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

ARTIFICIAL INTELLIGENCE LAB

NAME	Ayesha Imran
Class	CS-A
Lab	04
Course	Artificial Intelligence
Date	10-October-25
Submitted To	Lec. Ijlal Haider

IN LAB TASKS

TASK 01:

Write a Python function to perform Depth-First Search (DFS) on a given graph represented as a dictionary. The graph will be defined such that the keys are nodes, and the values are lists of neighbouring nodes. The function should print the nodes as they are visited.

SOLUTION:

```python
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()  # to keep track of visited nodes

    print(start, end=" ")  # print the node as it's visited
    visited.add(start)

    # recursively visit all unvisited neighbors
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)


# Example usage:
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

print("Depth-First Search starting from node 
A:")
dfs(graph, 'A')
```

OUTPUT:

```
Depth-First Search starting from node A:

A B D E F C
```

TASK 02:

Write a Python function to perform Uniform Cost Search (UCS) on a given graph. The graph will be represented as a dictionary, where each key is a node and the value is a list of tuples, each containing a neighbouring node and the cost to reach that neighbour. The function should find the least-cost path from a starting node to a goal node and print the path along with the total cost.

SOLUTION:

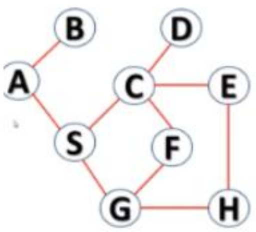
```
import heapq  # for priority queue (min-heap)

def uniform_cost_search(graph, start, goal):
    # Priority queue: stores (cost, path)
    queue = [(0, [start])]
    visited = set()

    while queue:
        # Get the path with the smallest total cost so far
        cost, path = heapq.heappop(queue)
        node = path[-1]

        # If goal is reached, print path and cost
        if node == goal:
            print("Least-cost path:", " → ".join(path))
            print("Total cost:", cost)
            return

        # Skip already visited nodes
        if node in visited:
            continue
        visited.add(node)

        # Explore neighbors
        for neighbor, edge_cost in graph.get(node, []):
            if neighbor not in visited:
                new_cost = cost + edge_cost
                new_path = path + [neighbor]
                heapq.heappush(queue, (new_cost, new_path))
    print("No path found from", start, "to",
goal)


# Example usage:
graph = {
    'A': [('B', 1), ('C', 4)],
    'B': [('D', 2), ('E', 5)],
    'C': [('F', 3)],
    'D': [('G', 1)],
    'E': [('G', 2)],
    'F': [('G', 2)],
    'G': []
}

print("Uniform Cost Search from A to G:")
uniform_cost_search(graph, 'A', 'G')
```

OUTPUT:

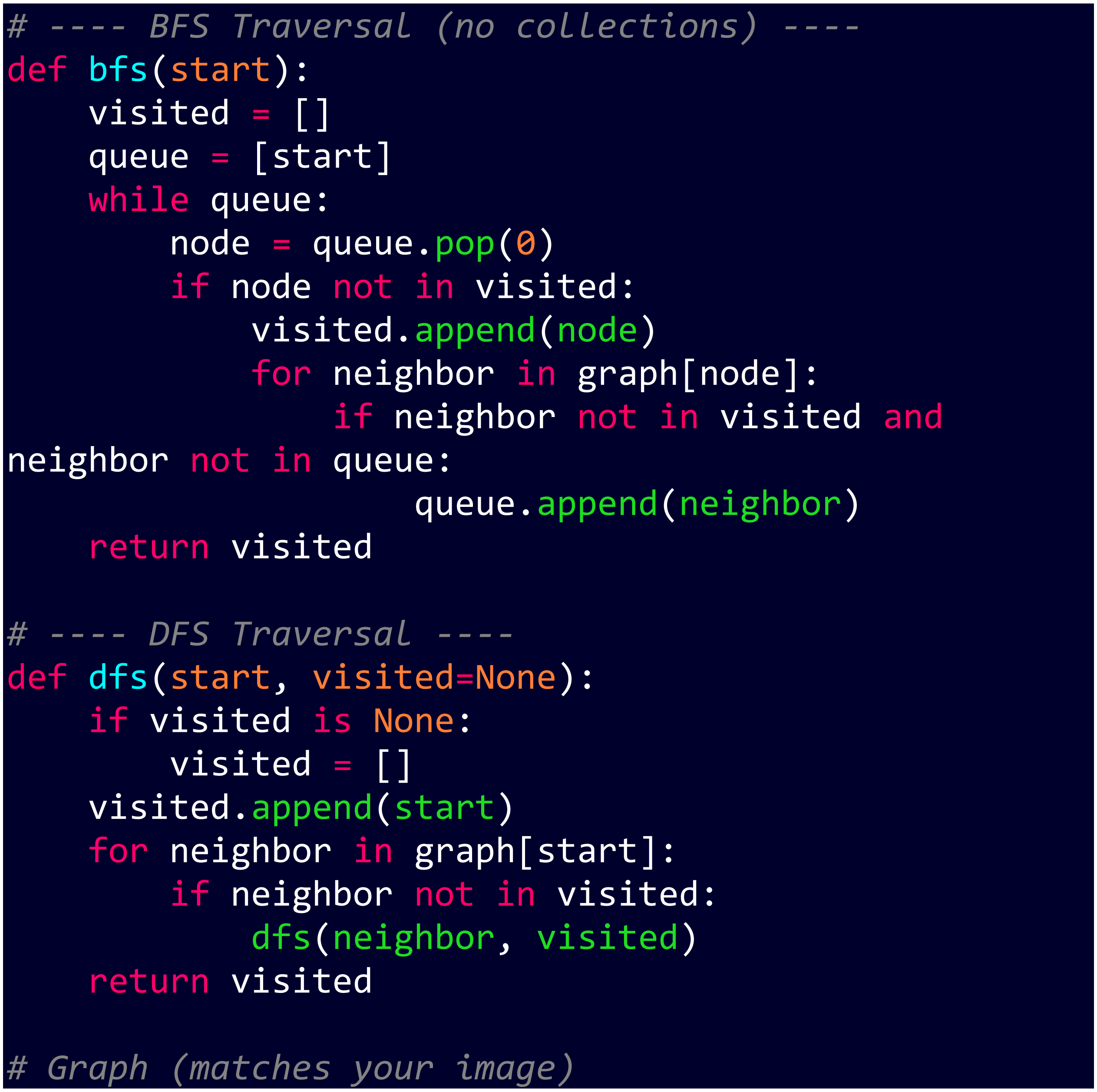
```
Uniform Cost Search from A to G:

Least-cost path: A → B → D → G

Total cost: 4
```

TASK 03:

Write Python code to traverse the following graph using BFS and DFS starting from node S. Print the nodes in the order they are visited.



SOLUTION:



```
# ---- BFS Traversal (no collections) ----
def bfs(start):
    visited = []
    queue = [start]
    while queue:
        node = queue.pop(0)
        if node not in visited:
            visited.append(node)
            for neighbor in graph[node]:
                if neighbor not in visited and
neighbor not in queue:
                    queue.append(neighbor)
    return visited

# ---- DFS Traversal ----
def dfs(start, visited=None):
    if visited is None:
        visited = []
    visited.append(start)
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(neighbor, visited)
    return visited

# Graph (matches your image)
```

```python
graph = {
    'A': ['B', 'S'],
    'B': ['A'],
    'S': ['A', 'C', 'G'],
    'C': ['D', 'E', 'F', 'S'],
    'D': ['C'],
    'E': ['C', 'H'],
    'F': ['C', 'G'],
    'G': ['S', 'F', 'H'],
    'H': ['E', 'G']
}

print("BFS Traversal starting from node S:")
print(bfs('S'))

print("\nDFS Traversal starting from node S:")
print(dfs('S'))
```

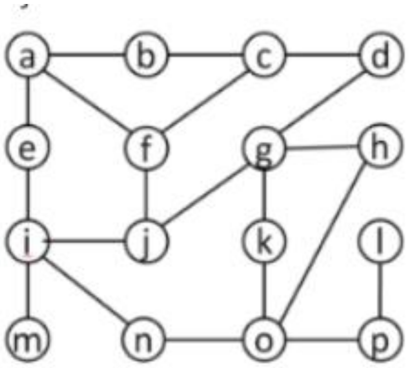
OUTPUT:

```
BFS Traversal starting from node S:

['S', 'A', 'C', 'G', 'B', 'D', 'E', 'F', 'H']


DFS Traversal starting from node S:

['S', 'A', 'B', 'C', 'D', 'E', 'H', 'G', 'F']
```

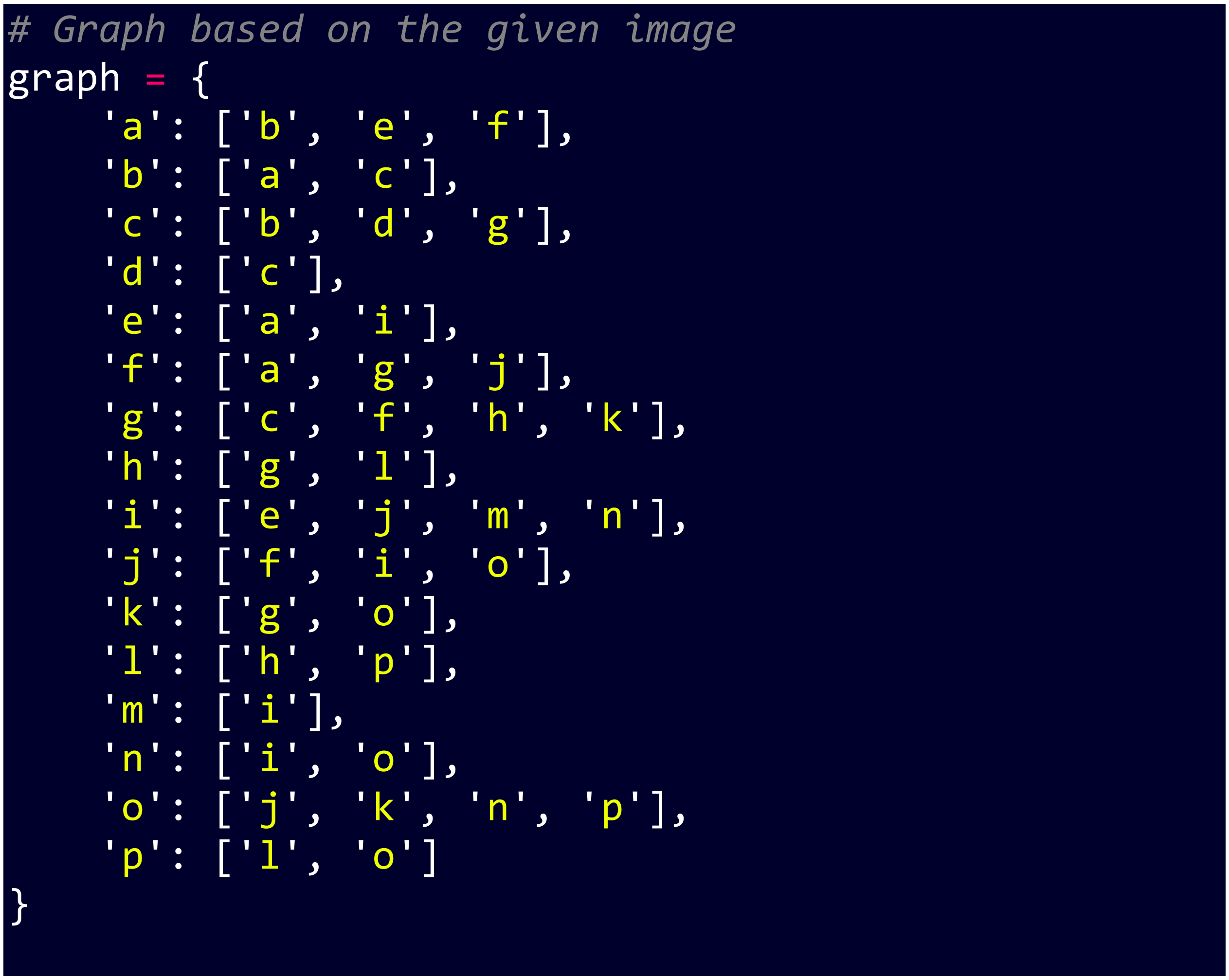
Post Lab Tasks

TASK 01:

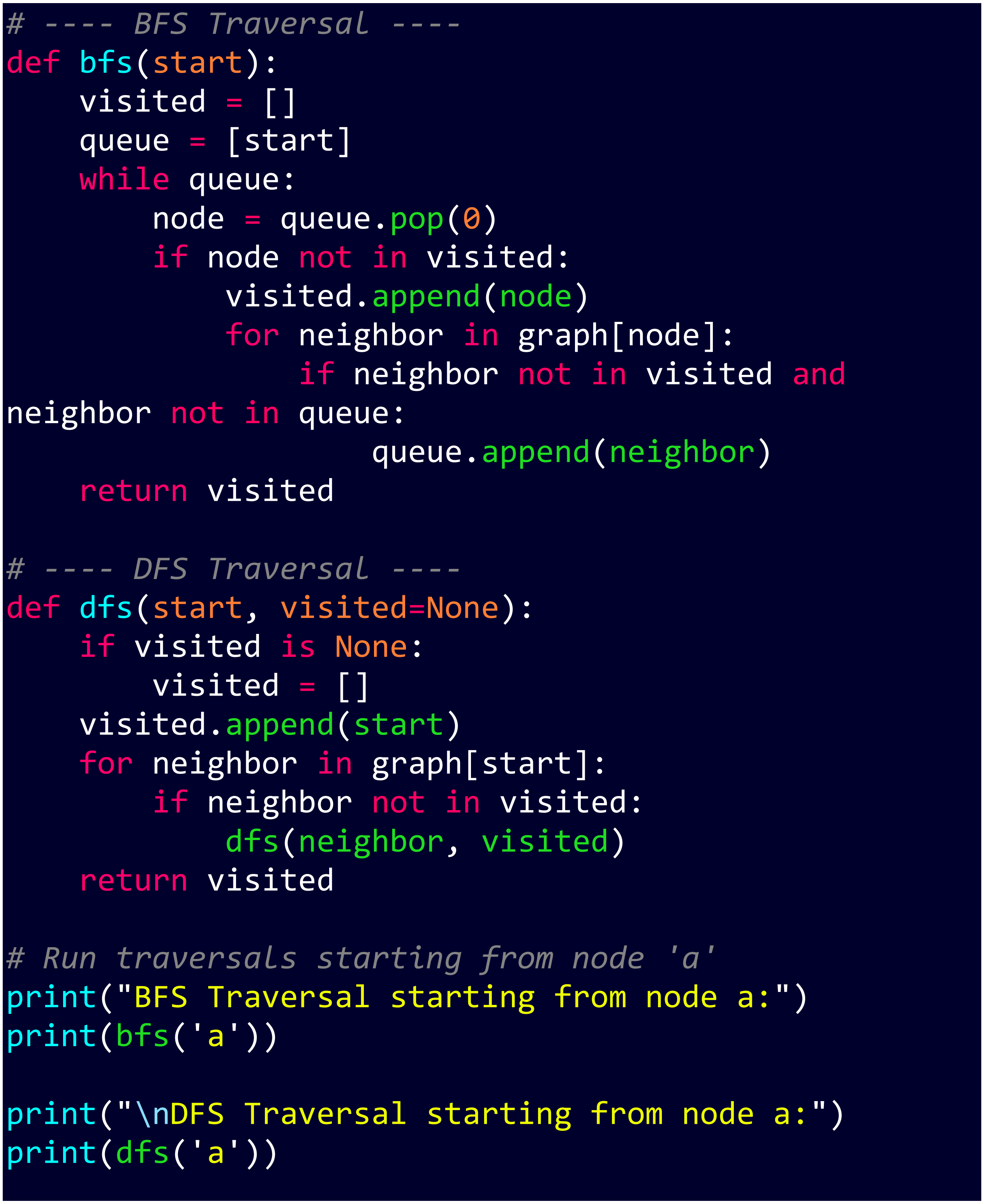
Write Python code to traverse the following graph using BFS and DFS starting from node a.Print the nodes in the order they are visited.



SOLUTION:



```
# Graph based on the given image
graph = {
    'a': ['b', 'e', 'f'],
    'b': ['a', 'c'],
    'c': ['b', 'd', 'g'],
    'd': ['c'],
    'e': ['a', 'i'],
    'f': ['a', 'g', 'j'],
    'g': ['c', 'f', 'h', 'k'],
    'h': ['g', 'l'],
    'i': ['e', 'j', 'm', 'n'],
    'j': ['f', 'i', 'o'],
    'k': ['g', 'o'],
    'l': ['h', 'p'],
    'm': ['i'],
    'n': ['i', 'o'],
    'o': ['j', 'k', 'n', 'p'],
    'p': ['l', 'o']
}
```



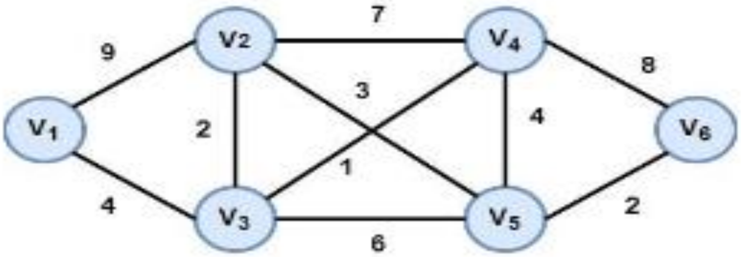
```python
# ---- BFS Traversal ----
def bfs(start):
    visited = []
    queue = [start]
    while queue:
        node = queue.pop(0)
        if node not in visited:
            visited.append(node)
            for neighbor in graph[node]:
                if neighbor not in visited and
neighbor not in queue:
                    queue.append(neighbor)
    return visited

# ---- DFS Traversal ----
def dfs(start, visited=None):
    if visited is None:
        visited = []
    visited.append(start)
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(neighbor, visited)
    return visited

# Run traversals starting from node 'a'
print("BFS Traversal starting from node a:")
print(bfs('a'))

print("\nDFS Traversal starting from node a:")
print(dfs('a'))
```

OUTPUT:

```
BFS Traversal starting from node a:
['a', 'b', 'e', 'f', 'c', 'i', 'g', 'j', 'd', 'm', 'n', 'h', 'k', 'o', 'l', 'p']


DFS Traversal starting from node a:
['a', 'b', 'c', 'd', 'g', 'f', 'j', 'i', 'e', 'm', 'n', 'o', 'k', 'p', 'l', 'h']
```

TASK 2:

For the given graph write a Python code to implement Uniform Cost Search (UCS) to find the shortest path from node V1 to node V6.



SOLUTION:

```
import heapq

# Graph representation
graph = {
    'V1': [('V2', 9), ('V3', 4)],
    'V2': [('V1', 9), ('V3', 2), ('V4', 7), ('V5',
3)],
    'V3': [('V1', 4), ('V2', 2), ('V5', 6), ('V4',
1)],
    'V4': [('V2', 7), ('V3', 1), ('V5', 4), ('V6', 8)],
    'V5': [('V2', 3), ('V3', 6), ('V4', 4), ('V6', 2)],
    'V6': [('V4', 8), ('V5', 2)]
}

def uniform_cost_search(graph, start, goal):
    visited = set()
    queue = [(0, start, [start])]  # (cost, current_node, path)

    while queue:
        cost, node, path = heapq.heappop(queue)

        if node == goal:
            return cost, path  # Found the goal with least cost

        if node not in visited:
            visited.add(node)

            for neighbor, edge_cost in graph[node]:
                if neighbor not in visited:
                    total_cost = cost + edge_cost
                    heapq.heappush(queue, (total_cost, neighbor, path + [neighbor]))

    return float("inf"), []  # If no path exists

# Run UCS
total_cost, path = uniform_cost_search(graph, 'V1', 'V6')
```

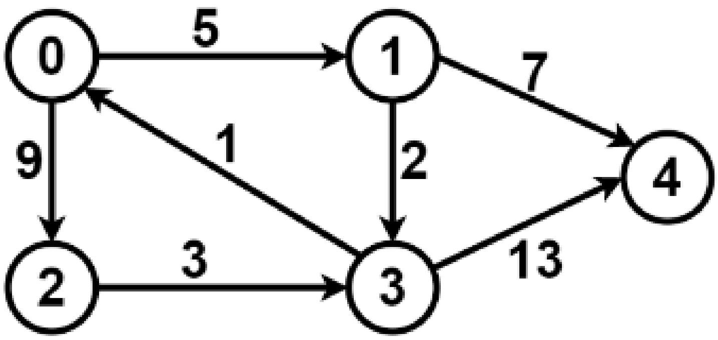
```
print("Shortest Path from V1 to V6:", " → ".join(path))
print("Total Cost:", total_cost)
```

OUTPUT:

```
Shortest Path from V1 to V6: V1 → V3 → V2 → V5 → V6

Total Cost: 11
```

TASK 03:

For the below graph write Python code to find the shortest path from node 2 to node 4 using UCS. Write complete order for the traversal.



SOLUTION:

```
# Step 1: Define the graph
graph = {
    0: [(1, 5), (2, 9)],
    1: [(2, 1), (3, 2), (4, 7)],
    2: [(3, 3)],
    3: [(4, 13)],
    4: []
}
```

```python
# Step 2: UCS function
def ucs(start, goal):
    visited = []
    queue = [(0, start, [])]  # (cost, current_node, path)

    while queue:
        queue.sort()  # sort by cost
        cost, node, path = queue.pop(0)

        if node in visited:
            continue

        visited.append(node)
        path = path + [node]

        if node == goal:
            print("Traversal order:", visited)
            print("Shortest path:", path)
            print("Total cost:", cost)
            return

        for neighbor, edge_cost in graph[node]:
            if neighbor not in visited:
                queue.append((cost + edge_cost, neighbor,
path))

# Step 3: Run UCS from node 2 to node 4
ucs(2, 4)
```

OUTPUT:

```
Traversal order: [2, 3, 4]

Shortest path: [2, 3, 4]

Total cost: 16
```