SIT221 Data Structures and Algorithms

Task 8.1p

Part A

Since the graph has **unit edge weights**, we use **Breadth-First Search (BFS)**, which naturally computes the **shortest distance** from s to every node.

We modify BFS to also count the **number of distinct shortest paths** to each node. So,

While visiting neighbors during BFS:

- If the neighbor hasn't been visited before (first time reached), we:
    - Set its distance,
    - Copy the number of shortest paths from the current node.
- If the neighbor has already been visited **with the same distance**, we:
    - Add the number of current node's paths to the neighbor's path count.

PSEUDOCODE:

```
function countPaths(graph, source):
  for each node in graph:
    distance[node] = ∞
    paths[node] = 0

  distance[source] = 0
  paths[source] = 1
  queue = new Queue()
  queue.enqueue(source)

  while not queue.isEmpty():
    current = queue.dequeue()
    for neighbor in graph[current]:
      if distance[neighbor] == ∞:
        distance[neighbor] = distance[current] + 1
        paths[neighbor] = paths[current]
        queue.enqueue(neighbor)
      else if distance[neighbor] == distance[current] + 1:
```

paths[neighbor] += paths[current]

return paths

WHY THIS WORKS?

- **Shortest Distance**: BFS guarantees shortest paths in unweighted graphs since it expands nodes in layers.
- **Path Counting**:
    - When we first reach a node v, we set its shortest distance and inherit the number of paths from its parent node u.
    - If we later find another path to v with the same minimum distance, we add the number of paths to u to v's count.
- **Only shortest paths are considered** because we only update count[v] when dist[v] == dist[u] + 1

Time Complexity Analysis

- Initial setup: O(n)
- BFS traversal:
    - Each node enqueued/dequeued once → O(n)
    - Each edge examined at most twice → O(m)
- Total runtime: **O(n + m)**

Output Interpretation

The resulting count array gives the **number of shortest paths** from the train station s to every location v.
Miss Marple should choose the hotel v with the **highest count[v] value**.

PART B

The goal is to Determine **whether the graph is connected** — that is, if **every computer can reach every other computer**, either directly or via relays (intermediate nodes).

We are given that the network is described using an **adjacency list** representation. To meet the required time efficiency of **O(n + m)**—where **n** represents the number of vertices (computers) and **m** denotes the number of edges (connections)—a graph traversal strategy such as **DFS or BFS** is suitable.

To determine if the network is fully connected, we initiate a traversal from any arbitrary node and record which nodes can be reached. If, at the end of this process, the number of nodes visited

matches the total number of nodes in the graph, it confirms that the network is connected. If there are any unvisited nodes, it indicates a disconnected structure.

PSEUDOCODE FOR DFS

```
function isGraphConnected(graph):
    visited = new Set()

    function dfs(node):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                dfs(neighbor)

    start = any node from graph
    dfs(start)

    return len(visited) == total number of nodes
```

PSEUDOCODE FOR BFS

```
function countPaths(graph, source):
    for each node in graph:
        distance[node] = ∞
        paths[node] = 0

    distance[source] = 0
    paths[source] = 1
    queue = new Queue()
    queue.enqueue(source)

    while not queue.isEmpty():
        current = queue.dequeue()
        for neighbor in graph[current]:
```

```
        if distance[neighbor] == ∞:
            distance[neighbor] = distance[current] + 1
            paths[neighbor] = paths[current]
            queue.enqueue(neighbor)
        else if distance[neighbor] == distance[current] + 1:
            paths[neighbor] += paths[current]

    return paths
```

Time Complexity

- Each node is visited once → O(n)
- Each edge is examined at most twice (undirected) → O(m)
- **Total runtime = O(n + m)**

WHY THIS WORKS?

- The algorithm visits all nodes reachable from the starting node.
- If **all nodes are visited**, then **every node is reachable from any other** → the graph is **connected**.
- If some nodes are **not visited**, the graph is **disconnected**.

PART C

WHAT IS ADJENCY MATRIX ?

An **adjacency matrix** for a graph with n vertices is a **2D matrix** of size n×n.

Each cell A[i][j] contains:

- 1 (or a weight) if there is an edge from vertex iii to vertex j,
- 0 (or ∞) otherwise.

DFS with Adjacency Matrix

In DFS:

- You **visit each node once**.
- At each node u, you **scan all its neighbors** to check if there's an edge (u,v) and if v is unvisited.

But with an **adjacency matrix**, checking for neighbors takes:

- **O(n)** time per node (you must scan the entire row of the matrix).

So the total time is:

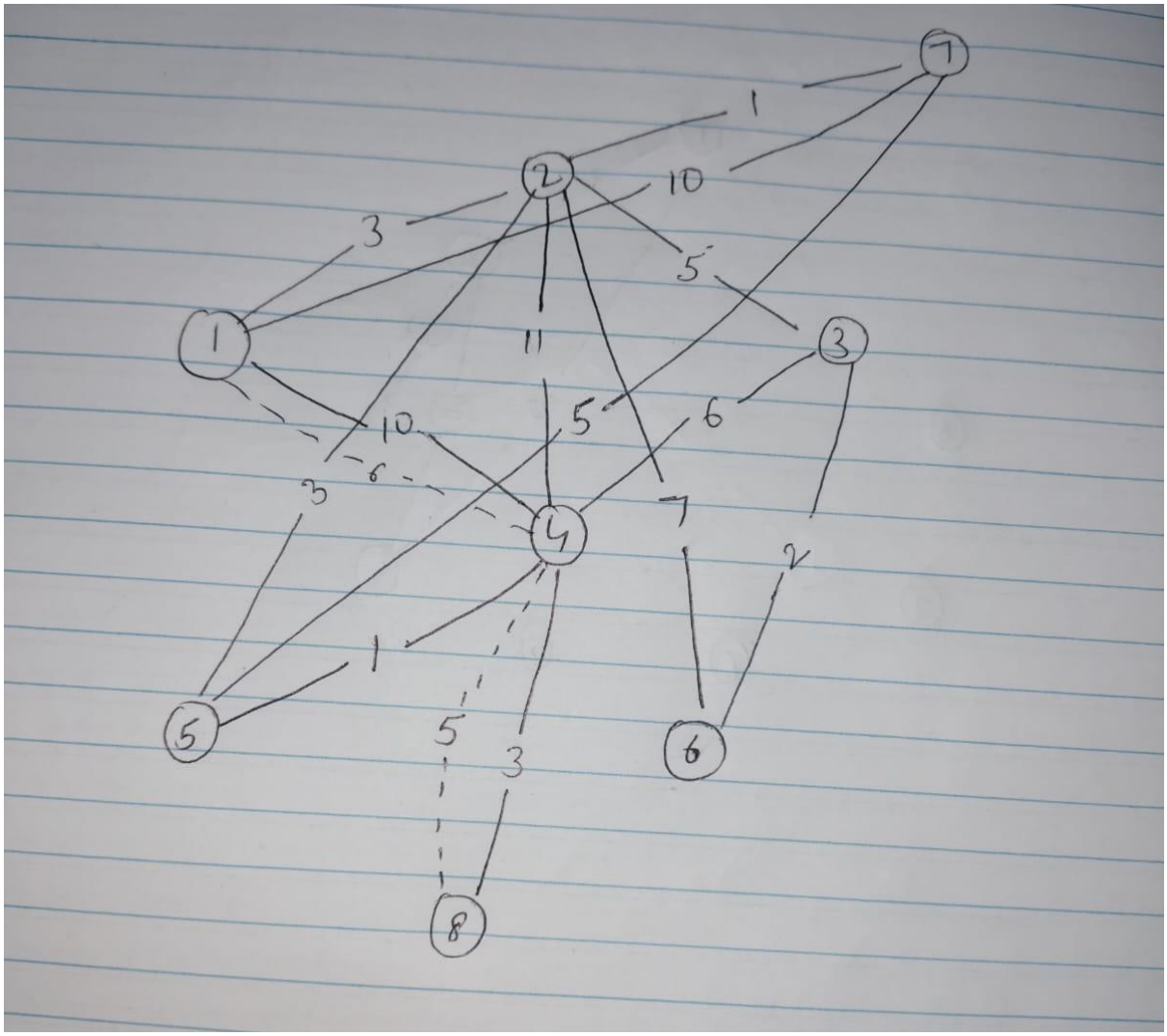
O(n) (for each of n nodes)= O (n^2)

The **runtime complexity** of DFS using an **adjacency matrix** is **O(n²)**.

Why Not O(n + m)?

- O(n + m) applies only when using an **adjacency list**, where you directly access the list of neighbors.
- In a matrix, even if a node has only 1 neighbor, you must **scan all n columns** to find it.

PART D :



Step-by-Step Queue Operations

1. Start at **Node 1**

- From 1 → 2: = 3
- From 1 → 6: = 10

2. Visit **Node 2** (lowest dist = 3)
    - 2 → 4: = 3+5 = 8
    - 2 → 5: = 3+11 = 14
    - 2 → 7: = 3+7 = 10
    - 2 → 8: = 3+1 = 4
3. Visit **Node 8** (dist = 4)
    - 8 → 3: = 4+10 = 14
4. Visit **Node 4** (dist = 8)
    - 4 → 6: = min(10, 8+1 = 9) → update to 9
5. Visit **Node 6** (dist = 9)
    - no better updates
6. Visit **Node 7** (dist = 10)
    - 7 → 6 = 2 (already better dist at 9)
7. Visit **Node 5** (14)
    - 5 → 10 = 14+3 = 17
8. Visit **Node 3** (14)
    - no updates
9. Visit **Node 10** (17)

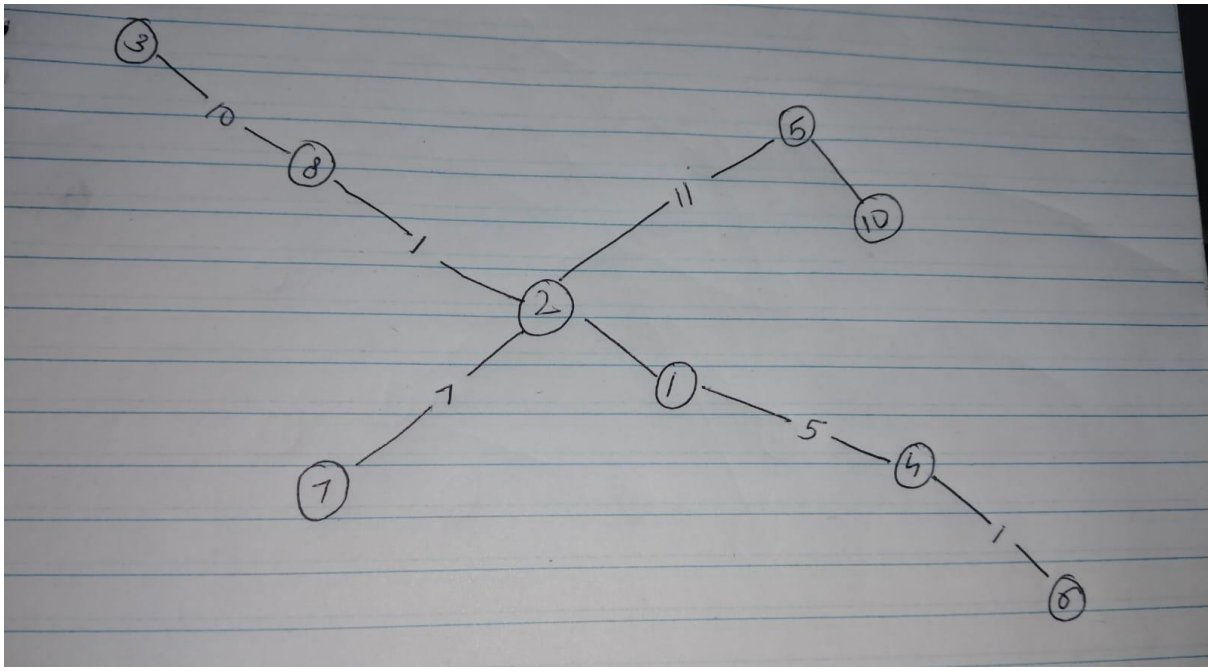
Final Output

Order of Node Removal (Priority Queue):

1 → 2 → 8 → 4 → 6 → 7 → 5 → 3 → 10

Node 9 is **not reached** from node 1.

SHORTEST PATH TREE



PART E

What Happens When Edge Weights Increase Uniformly?

- A path with **k edges** will have its total cost increased by **k**.
- Another path with **fewer edges** will be increased by **less**.

This means that **longer paths (more edges)** are **penalized more** than **shorter paths**, even if they were cheaper before.

This is because increasing each edge's weight by a fixed amount affects paths **proportionally to the number of edges** they contain, not just their total cost. A path that originally had more edges will have **more units of weight added** than a path with fewer edges. In other words, **the number of edges becomes a more significant factor** in the final cost of the path after the increase.

So even though path P was initially the shortest, **after increasing all edge weights by 1**, another path with **fewer edges** but a slightly higher original cost could now become **cheaper** than P.

No, the original shortest path does **not always remain** the shortest when all edge weights are increased by 1.

This is because **longer paths are penalized more** than shorter ones, which can cause a different path (with fewer edges but higher original weight) to become the new shortest path.