Design and Analysis of Algorithms

Project Report

Submitted By:

Usman Haroon 22i-1177

Hussain Mehmood 22i-1706

CY(D)

Submitted To:

Maam Amina Siddique

PART 1(Dijkstra Algorithm):

Pseudo-Code

```
Function Dijkstra(Graph, source, destination):
    // Initialize distances to all nodes as infinity except the source node
    Set distances[source] = 0
    For each node in Graph:
        If node != source:
            Set distances[node] = infinity

    // Create a priority queue to store (distance, node) pairs
    Create a priority queue pq

    // Add the source node with distance 0 to the priority queue
    pq.push((0, source))

    // Initialize a map to store the previous node for the shortest path
    Initialize previous[node] = -1  // No previous node initially

    While pq is not empty:
        // Get the node with the smallest distance (currentNode)
        (currentDistance, currentNode) = pq.pop()

        // Skip if this node's current distance is not optimal
        If currentDistance > distances[currentNode]:
            Continue

        // Explore all neighbors of currentNode
        For each neighbor (neighborNode, edgeWeight) in Graph[currentNode]:
```

```
        // Calculate the distance to the neighbor
        newDist = currentDistance + edgeWeight

        // If a shorter path to the neighbor is found, update distance and
previous node
        If newDist < distances[neighborNode]:
          distances[neighborNode] = newDist
          previous[neighborNode] = currentNode
          pq.push((newDist, neighborNode))

  // Reconstruct the shortest path from source to destination
  Initialize an empty list path
  Set currentNode = destination

  // Traverse back from destination to source using the previous node map
  While currentNode != -1:
    Add currentNode to path
    Set currentNode = previous[currentNode]

  // Reverse the path to get it from source to destination
  Reverse(path)

  // Print the results
  If distances[destination] == infinity:
    Print "No path exists."
  Else:
    Print "Shortest Path Distance:", distances[destination]
    Print "Path:", path
```

<u>Time Complexity:</u> O((V+E)LogV)

<u>Explanation:</u>

- **V** is number of vertices
- **E** is number of edges in graph
- Initialization: O(V) (As loop is used to initialize each node)
- Queue Operation:O(E Log V) (As insertion take Log n operation to propagate all the element)
- Path Reconstruction: O(V) (Simply use loop to print path of nodes)
  So overall time complexity is O((V+E)LogV)

PART 2(Longest Chain of Influence):

<u>Pseudo-Code</u>

```
Function DFS(node, Graph, influences, memo, predecessor):
   // If the result for this node is already computed, return it
   If node is in memo:
      Return memo[node]

   // Initialize the longest chain length to 1 (the node itself)
   maxLength = 1

   // Explore all neighbors of the current node
   For each neighbor in Graph[node]:
      // Only consider neighbors with higher influence scores
      If influences[neighbor] > influences[node]:
         // Recursively find the longest chain starting from the neighbor
```

```
            chainLength = 1 + DFS(neighbor, Graph, influences, memo, predecessor)

            // If this is the longest chain found so far, update maxLength
            If chainLength > maxLength:
                maxLength = chainLength
                predecessor[neighbor] = node  // Track the predecessor for path reconstruction

    // Memoize the result
    memo[node] = maxLength

    // Return the longest chain length for this node
    Return maxLength

Function FindLongestChain(Graph, influences):
    // Initialize memoization map and predecessor map
    memo = empty map
    predecessor = empty map

    maxChainLength = 0
    startNode = -1

    // Traverse all nodes to find the longest chain
    For each node in Graph:
        // Call DFS for each node and get the length of the longest chain starting from it
        chainLength = DFS(node, Graph, influences, memo, predecessor)

        // Keep track of the maximum chain length
```

```
        If chainLength > maxChainLength:
            maxChainLength = chainLength
            startNode = node
Function FindLongestChainBetweenTwoNodes(startNode,endNode,
Graph, influences, predecessor):

// Initialize memoization map for DFS
memo = empty map

 // Get the longest chain starting from the start node
longestChainFromStart = DFS(startNode, Graph, influences, memo,
predecessor)

// Clear memoization for the second DFS calculation
memo.clear()

// Get the longest chain starting from the end node
longestChainFromEnd = DFS(endNode, Graph, influences, memo,
predecessor)

 // Return the shortest chain length between the two nodes
RETURN min(longestChainFromStart, longestChainFromEnd)

    // Reconstruct the longest chain sequence
    chain = empty list
    While startNode is not -1:
        chain.push(startNode)
        startNode = predecessor[startNode]
```

```
    // Reverse the chain to get it from the starting node to the ending node
    Reverse(chain)

    // Output the longest chain length and sequence
    Print "Longest Chain Length:", maxChainLength
    Print "Chain Sequence:", chain
```

Time Complexity: O(V*(V+E))

Explanation:

- **V** is number of vertices
- **E** is number of edges in graph
- Outer Loop over Nodes: O(V) (Explores the nodes)
- DFS :O(V+E) (Performs DFS on each edge)
- Longest Influence in between Node: O(V+E) (As It uses same function as above for it working)
  So overall time complexity is O(V*(V+E))