Algorithm Analysis Report

Part 1: Shortest Path Algorithm (Dijkstra's)

Pseudocode

```
DIJKSTRA(graph, start, goal):
    Initialize priority queue Q
    Initialize distance map dist[v] = ∞ for all vertices
    Initialize previous map prev[v] = undefined for all vertices
    dist[start] = 0
    Q.push((start, 0))

    while Q is not empty:
        current = Q.pop()

        if current == goal:
            return construct_path(prev, start, goal)

        for each neighbor of current:
            tentative_dist = dist[current] + weight(current, neighbor)
            if tentative_dist < dist[neighbor]:
                dist[neighbor] = tentative_dist
                prev[neighbor] = current
                Q.push((neighbor, tentative_dist))

    return empty_path
```

Time Complexity Analysis

- Let V = number of vertices, E = number of edges
- Main operations:
  1. Priority queue operations: O(log V)
  2. Each vertex is processed once: O(V)
  3. Each edge is examined once: O(E)
- Overall complexity: O((V + E) log V)
- Space complexity: O(V) for the priority queue and distance maps

Part 2: Longest Increasing Path (Dynamic Programming)

Pseudocode

```
LONGEST_INCREASING_PATH(node):
    if node in memo:
        return memo[node]

    maxLength = 1
```

```
    maxPath = [node]

    for each neighbor of node:
        if influence[neighbor] > influence[node]:
            result = LONGEST_INCREASING_PATH(neighbor)
            if result.length + 1 > maxLength:
                maxLength = result.length + 1
                maxPath = [node] + result.path

    memo[node] = (maxLength, maxPath)
    return memo[node]

FIND_MAXIMUM_INFLUENCE_CHAIN():
    globalMaxLength = 0
    globalMaxPath = []

    for each node in graph:
        result = LONGEST_INCREASING_PATH(node)
        if result.length > globalMaxLength:
            globalMaxLength = result.length
            globalMaxPath = result.path

    return (globalMaxLength, globalMaxPath)
```

Time Complexity Analysis

- Let V = number of vertices, E = number of edges

1. For each node:
   - Each node is processed exactly once due to memoization
   - Each edge from that node is examined once
   - Path construction takes O(V) in worst case
2. Overall complexity:
   - Base operation: $O(V \times E)$ for visiting all nodes and their edges
   - Memoization reduces repeated computations
   - Total complexity: $O(V \times E)$

- Space complexity: O(V) for memoization table and storing paths

Implementation Details

Part-1.cpp

- Uses adjacency list representation
- Priority queue for efficient vertex selection
- Implements standard Dijkstra's algorithm

Part-2.cpp

- Uses adjacency list with influence scores
- Implements dynamic programming with memoization
- Maintains path information along with lengths

Correctness

- Both algorithms guarantee polynomial time solutions
- Part 1 guarantees shortest path between vertices
- Part 2 guarantees longest increasing sequence of influence scores
- Both implementations handle edge cases (disconnected graphs, no valid paths)

Space-Time Tradeoff

- Both implementations prioritize time efficiency while maintaining reasonable space complexity
- Memoization in Part 2 trades space for improved time complexity by avoiding recomputation
- Data structures (priority queue, hash maps) chosen for optimal access times

The implementations satisfy the polynomial time requirement while providing efficient solutions to the given problems.