**Report: Social Network Analysis Project**

**Abdullah Nadeem (22i-XXXX) , BS\_CYS\_A**

**CS-2009 Design and Analysis of Algorithms (Fall 2024)**

**PART-1:**

**1. Introduction**

The project is focused on finding the shortest path in a social network graph using the A\* algorithm. The dataset provided contains user connections with weights indicating the strength of these connections. The task is to generate a graph based on this dataset, apply A\* algorithm to find the shortest path, and visualize the path using Graphviz.

**2. Problem Understanding**

The problem involves the following tasks:

* Constructing an undirected graph from the dataset where the nodes represent users and edges represent their connections with weights.
* Implementing the A\* algorithm for finding the shortest path from one user to another.
* Visualizing the shortest path using Graphviz in a .dot file format.

**3. Algorithm Description**

***A Algorithm (Pseudocode)*\***

1. **Initialization**:
   * Let openSet be a priority queue (min-heap) that stores nodes along with their f(n) score, where f(n) = g(n) + h(n):
     + g(n) is the cost to reach node n from the start node.
     + h(n) is the heuristic estimate of the remaining cost from node n to the goal node (in this case, the number of direct connections for a node).
   * gScore[start] = 0 and fScore[start] = h(start) (heuristic value).
2. **Main Loop**:
   * While the openSet is not empty:
     + Pop the node current with the lowest f(n) score.
     + If current is the goal node, reconstruct the path by tracing back from the goal node to the start node.
     + For each neighbor of current, calculate the tentative gScore and update fScore accordingly.
3. **Path Reconstruction**:
   * Reconstruct the path by following the nodes from the goal back to the start.
   * For each edge in the path, store the edge weight.

**4. Code Explanation**

The C++ implementation follows these steps:

* **Graph Representation**: The graph is represented using an adjacency list, where each node points to its connected nodes with their respective weights.
* **Heuristic Function**: The heuristic function h(n) calculates the number of direct connections a node has, which is used in the A\* algorithm.
* ***A Search*\***: The shortest path is found using a priority queue and updating g(n) and f(n) for each node. If a path is found, the nodes and edge weights are written to a file.
* **Visualization**: The shortest path is visualized using Graphviz by creating a .dot file with the edges of the path marked in red with labels showing the weights.

The code also includes methods for:

* Writing the graph to a file.
* Writing detailed path information, including node and edge weights.
* Writing node connection information for nodes involved in the shortest path.

**5. Command to Generate Image**

To visualize the shortest path, the following command must be executed using Graphviz:

Copy code

**dot -Tpng shortest\_path\_visualization.dot -o shortest\_path\_visualization.png**

This command takes the .dot file generated by the program and outputs a PNG image of the graph, where the shortest path is highlighted.

**6. Sample Output**

* **Graph Output**: A file graph\_output.txt containing the graph’s node connections and weights.
* **Shortest Path**: A file a\_star\_shortest\_path.txt containing the detailed shortest path with edge weights.
* **Shortest Path Visualization**: The final graph visualization is generated in a .png format after executing the dot command.

**7. Time Complexity**

* **Graph Construction**: O(E) where E is the number of edges in the graph.
* ***A Algorithm*\***: O((V + E) \* log(V)), where V is the number of vertices and E is the number of edges.
  + The priority queue operations take O(log(V)) time, and we process each edge and node once.

**8. Conclusion**

This part successfully implements the A\* algorithm to find the shortest path in a social network graph, providing an efficient solution for this type of problem. The visualization of the shortest path helps in understanding the relationships between nodes in the network.

**Part-2:**

**Longest Increasing Influence Chain**

The task was to implement a solution for the "Social Network Analysis" problem, specifically focusing on finding the longest increasing chain of influence in a social network using dynamic programming. The dataset provided consists of a graph of users in a social network and a separate file containing the influence scores of each user.

My goal was to apply dynamic programming to find the longest increasing path in the network, where the influence score of a user in the chain is strictly greater than the previous user's influence score. The output required the maximum length of this chain and the sequence of users in the chain.

**Approach**

1. **Graph Representation**:
   * The social network was represented as an undirected graph where nodes represent users, and edges represent connections between them.
   * The influence scores of users were stored in a map, where each user’s ID was mapped to their influence score.
2. **Dynamic Programming Algorithm**:
   * We used a dynamic programming approach to find the longest increasing path in terms of influence scores.
   * We defined an array dp[] where dp[i] stores the length of the longest increasing path that ends at user i.
   * We iterated over each user, sorted them by their influence scores, and updated the dp[] values by comparing a user’s influence score with their connected neighbors' influence scores.
3. **Algorithm**:
   * Sort the users based on their influence scores in ascending order.
   * For each user, check all their neighbors in the graph. If a neighbor has a higher influence score, update the longest path ending at that neighbor.
   * Track the parent of each node to later reconstruct the longest increasing path.
4. **Output**:
   * After finding the longest path, we output both the length of the longest chain and the sequence of users in the chain.
   * Additionally, we also printed the influence score for each user in the longest chain.

**Pseudocode for Dynamic Programming Algorithm**

**1. Graph Creation and Influence Loading:**

function createGraph(inputFile):

Initialize an empty graph

For each line in the file:

Parse user ID, connection, and weight

Add the edge to the graph

Return the graph

function loadInfluenceScores(influenceFile):

Initialize an empty influenceMap

For each line in the influence file:

Parse user ID and influence score

Store the influence score in the map

Return the influenceMap

**2. Longest Increasing Path Algorithm:**

function findLongestInfluencePath(graph, influenceMap):

Initialize an array dp[] with 1 for each user (represents the length of the longest path ending at that user)

Initialize a parent[] array with -1 for each user (to store the parent node in the path)

Sort the users by their influence score in ascending order

For each user in sorted order:

For each neighbor of the user:

If influenceMap[user] < influenceMap[neighbor] and dp[user] + 1 > dp[neighbor]:

Update dp[neighbor] to dp[user] + 1

Update parent[neighbor] to user

Find the user with the maximum dp value (this represents the end of the longest path)

Initialize an empty path[]

While the current user is not -1:

Add the user to the path

Move to the parent of the current user

Reverse the path to get the correct order from start to end

Return the path and the length

**Time Complexity Analysis**

* **Graph Creation**:  
  The time complexity for reading the graph from the file is O(E), where E is the number of edges in the graph.
* **Influence Score Loading**:  
  The time complexity for loading influence scores is O(V), where V is the number of users (vertices in the graph).
* **Longest Increasing Path Calculation**:
  + Sorting the nodes by influence scores takes O(Vlog V).
  + The inner loop iterates over all edges in the graph, which gives a time complexity of O(E)O(E)O(E) for each user.
  + The overall time complexity for finding the longest increasing path is O(VlogV+E), which is dominated by the sorting step.
* **Final Path Construction**:
  + Reconstructing the longest path from the parent[] array takes O(V), since we may need to traverse all the vertices in the worst case.

Thus, the overall time complexity of the solution is O(VlogV+E), which is efficient for typical social network sizes.

**Results**

* The program correctly identifies the longest increasing path of influence in the social network, as described in the problem.
* The sequence of users in the longest chain is printed along with the influence scores for each user.
* The maximum length of the chain is also displayed.

**Example Output:**

Longest Chain Length: 18

User Sequence: 6595 14063 3243 21070 1389 1392 21445 26187 9799 212 28661 11490 20021 35737 21394 9318 20614 1860

Influence Scores for Each Node in the Sequence:

Node 6595: 3

Node 14063: 4

Node 3243: 5

Node 21070: 9

Node 1389: 15

Node 1392: 21

Node 21445: 25

Node 26187: 26

Node 9799: 30

Node 212: 40

Node 28661: 45

Node 11490: 50

Node 20021: 54

Node 35737: 60

Node 21394: 88

Node 9318: 89

Node 20614: 96

Node 1860: 100

**Conclusion**

The project successfully implements a dynamic programming solution to find the longest increasing path based on user influence in a social network. The solution meets the required specifications and is efficient enough to handle typical graph sizes in social networks. The algorithm is implemented in C++ as per the course requirements, and the results have been verified against the provided dataset.