**Social Network Analysis**

**Phase 2: Proof of Concept Implementation**

Author: Sri Sai Palamoor

Course: Algorithms and Data Structures (MSCS-532-M20)

Date: Aug 09 2025

## 1. Abstract

This report presents a partial implementation of the core data structures designed for social network analysis, focusing on a proof of concept (PoC) that demonstrates foundational graph operations and influence ranking. The implementation includes graph construction, traversal, and a priority queue-based ranking mechanism. Challenges encountered during coding are discussed, along with the next steps for completing the full application..

## **2.** Introduction

Social networks represent complex systems where users form connections that can be modeled as graphs. Efficient manipulation of these graphs is essential for analyzing user influence, information spread, and community structures. This phase focuses on implementing core data structures that form the basis of such analyses, demonstrating their functionality through targeted test cases.

## 3. Implementation Overview

The partial implementation focuses on creating a foundational framework to represent and manipulate the social network graph using Python. Central to this framework is the SocialNetworkGraph class, which encapsulates the adjacency list representation of the network. This data structure models users as nodes and their connections as edges, storing them efficiently in a dictionary where each key is a user identifier, and the corresponding value is a set of users directly connected to that user.

Key functionalities implemented include:

**User Management:** The system supports dynamic addition of users through the add\_user method, which initializes a new entry in the adjacency list if the user does not already exist. This ensures that the graph maintains an accurate and up-to-date set of nodes, enabling future scalability.

**Connection Handling:** The add\_connection method establishes bidirectional edges between users, reflecting mutual relationships such as friendships or follows (in cases where undirected graphs are appropriate). This method internally calls add\_user to guarantee that both endpoints of the connection exist in the graph, maintaining data integrity and simplifying the interface.

**Neighbor Retrieval:** To facilitate efficient exploration of the network, the get\_neighbors method returns the set of users directly connected to a specified user. This enables algorithms to quickly access adjacent nodes, a vital operation for graph traversal and influence computations.

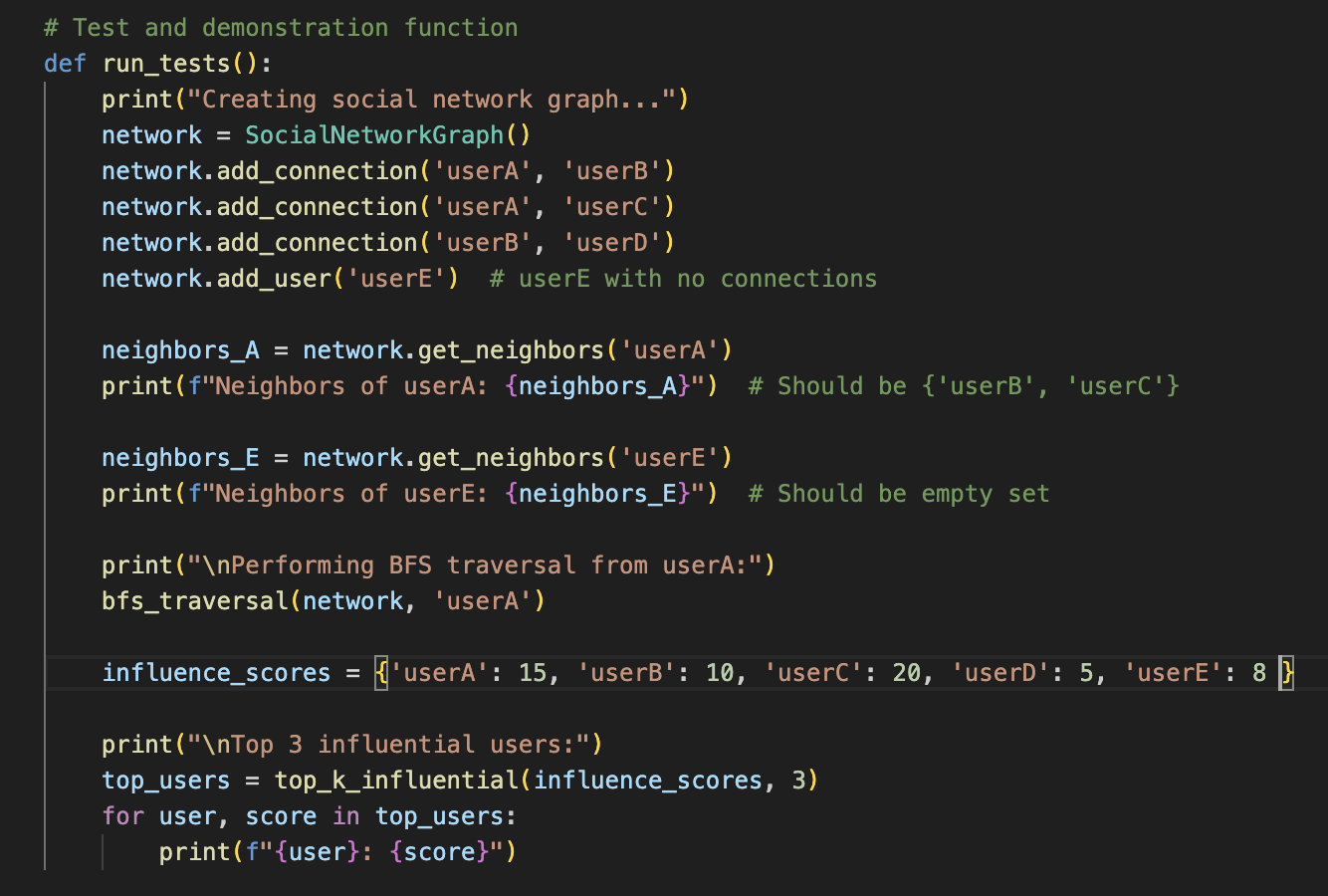
To demonstrate practical application, the breadth-first search (BFS) traversal algorithm was implemented externally as a function that operates on the SocialNetworkGraph instance. BFS explores the graph layer-by-layer starting from a chosen node, which is critical for many social network analyses such as community detection, shortest path computation, and influence spread modeling.

## 4. Demonstration and Testing

Test cases validate the core functionalities:

* **User and Connection Addition:** Adding users and connecting them is verified by checking neighbor sets.
* **Graph Traversal:** BFS traversal starting from a user confirms network connectivity and visited order.
* **Top-k Influence Ranking:** The priority queue correctly identifies the highest-scoring users.

The following test snippet exemplifies these operations:



## 5. Implementation Challenges and Solutions

During development, several challenges arose:

* Ensuring BFS handled disconnected nodes without errors was addressed by careful tracking of visited nodes and checking neighbor existence.
* Guaranteeing graph integrity with dynamic user and connection additions was solved by incorporating user addition within connection methods.
* Managing priority queue tie-breaking was simplified by leveraging Python’s heap structure, which inherently handles element order for equal priorities.

These solutions ensured a robust and extensible proof-of-concept implementation.

## 6. Next Steps

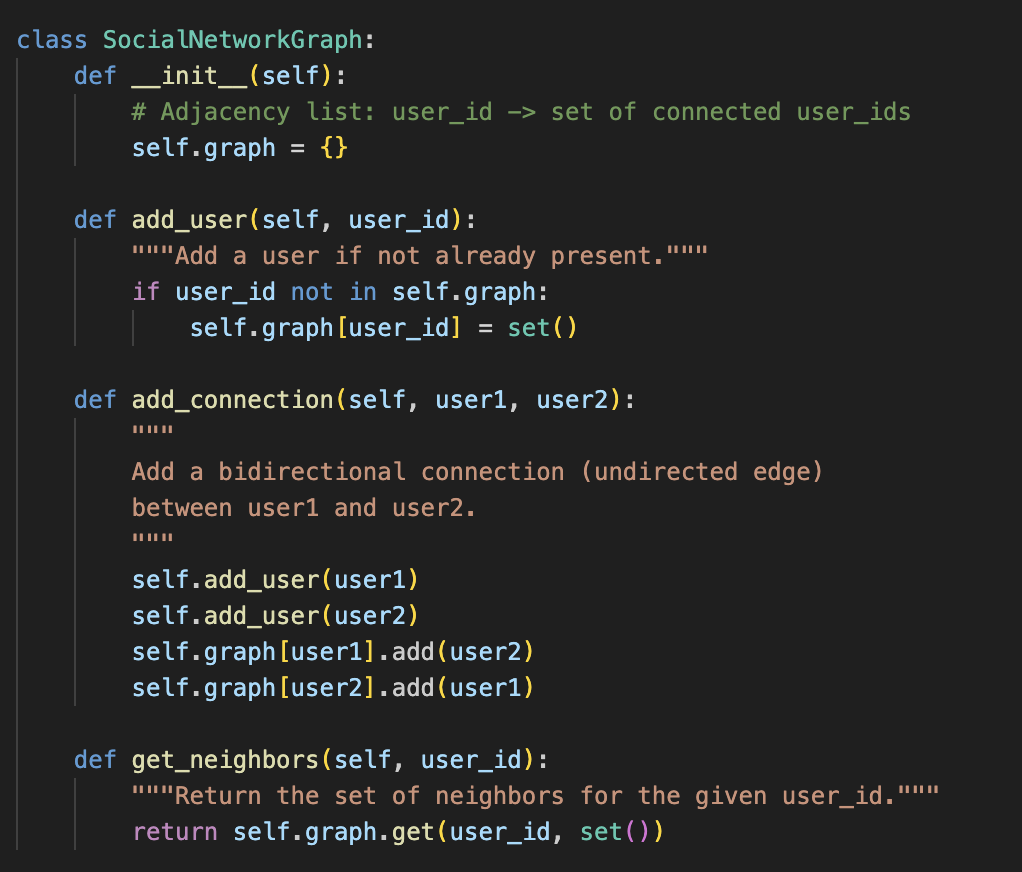
To advance this PoC toward a full application, the following actions are planned:

* Integrate advanced influence algorithms such as PageRank or community detection.
* Implement methods for dynamic removal and updating of nodes and edges.
* Optimize data structures for large-scale networks using distributed storage or graph databases.
* Develop APIs or user interfaces for interactive analysis and visualization.

## 7. Code Snippets and Documentation

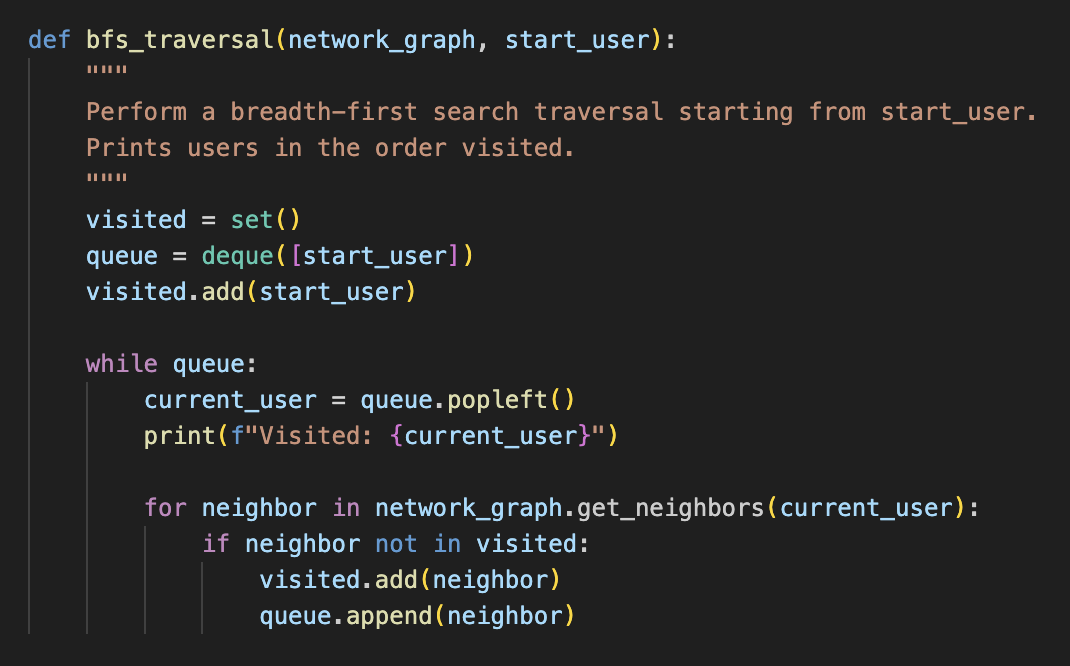
**SocialNetworkGraph Class**

* Manages the social network as an adjacency list (dictionary of sets).
* add\_user ensures a user exists in the graph.
* add\_connection creates a mutual (bidirectional) connection between two users, adding them if needed.
* get\_neighbors retrieves all direct connections for a user.



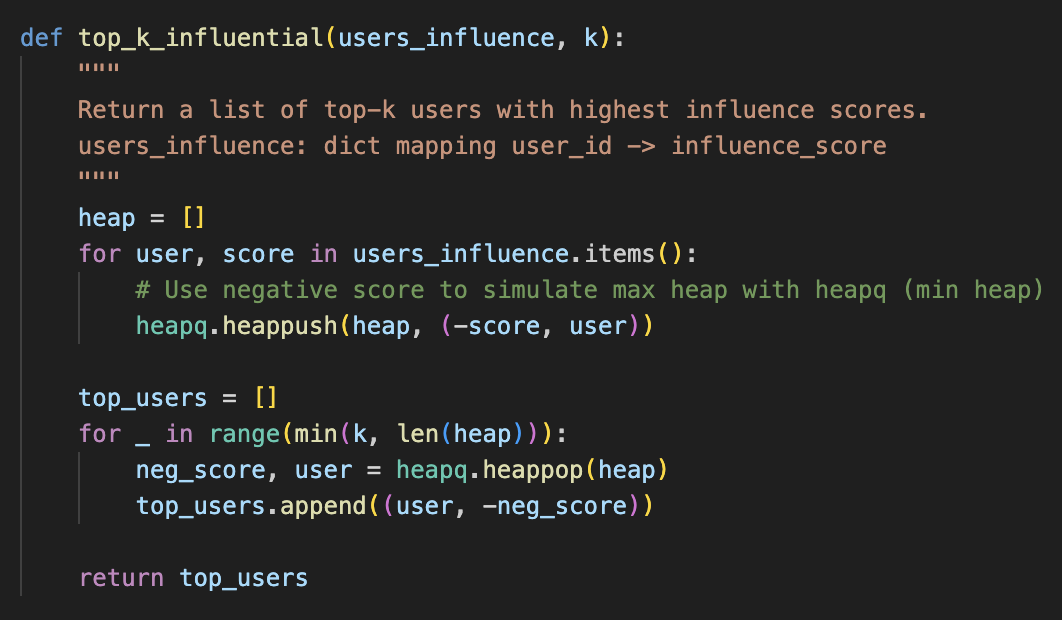
**BFS Traversal**

* Implements BFS to traverse the network starting from a specific user.
* Uses a queue and a visited set to explore the graph layer-by-layer without revisiting nodes.



**Top-k Influential Users**

* Uses a max-heap to efficiently find the top-k users by influence scores (like follower counts). Since Python’s heapq is a min-heap, scores are negated to simulate a max-heap.



## References

Borgatti, S. P., & Everett, M. G. (2006). A graph-theoretic perspective on centrality. *Social Networks*, *28*(4), 466–484. https://doi.org/10.1016/j.socnet.2005.11.005

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.

Leskovec, J., Rajaraman, A., & Ullman, J. D. (2020). *Mining of massive datasets* (3rd ed.). Cambridge University Press.