**Name: AJAL RC**

**Student ID: 005039456**

**Algorithms and Data Structures (MSCS-532-B01)**

**Deliverable 3: Optimization, Scaling and Final Evaluation**

**Optimization of Data Structures**

After building the basic social network in Phase 2, I realized that my previous implementation worked well for several users. However, things started to slow down and become messy when the data size increased. So, for this phase, I focused on making the code faster and more reliable, especially for larger graphs and datasets. Here are some of the upgrades that happen in this new approach.

* **Adjacency List Upgrade:** I implemented a standard Python dictionary for the adjacency list in my previous deliverable. Every time I added a new user in this implementation, I had to check if they already existed. For a big social network, this gets annoying and slows things down. To make it more efficient, I switched to using a defaultdict(set). This way, if I ever reference a user who is not in the graph yet (like adding a connection to a new user), Python creates the entry automatically. This makes my code shorter, avoids possible KeyErrors, and saves time (Cheng, Yu, & Qin, 2013).
* **Profile Lookups and Updates:** Since I implemented a dictionary for the user profile, I implemented .get() and. setdefault() everywhere to prevent crashes from missing keys.  This also allowed me to update profiles on the fly, just like modern key-value stores optimize lookups (Hao et al., 2018).
* **Centrality and Ranking:** Calculating the most “influential” users by sorting everyone’s connection count is fine for a dozen users, but it gets slow when you have thousands. Instead, I now use Python’s heapq.nlargest() to find the top-k users with the most connections. This approach is efficient and is used in large-scale graph analytics (Sahu et al., 2017).
* **Caching Degree Centrality:** My code first recalculated the degree centrality whenever needed. Now, I store the result (cache) and only update it when the network changes. This simple memoization saves lots of time, especially when running repeated analytics.

**Impact of these optimizations:**

All these changes make the core social network code run smoother and more reliably, particularly as the number of users and connections grows. Adding users or connections is instant, analytics for top influencers are nearly instant, and error handling is almost automatic.

A screen shot of a computer program

AI-generated content may be incorrect.Following is the snapshot of the new implementation.

**Scaling for Large Datasets**

Once the optimizations were in place, I wanted to see how my network would perform with "realistic" data sizes, not just a few friends, but for a massive dataset with a complex network of connections. I based my scaling and testing approach on recommendations from real-world graph processing research (Cheng, Yu, & Qin, 2013; Hao et al., 2018).

1. **Efficient Data Structures:** I decided to use defaultdict and sets for user connections, allowing my code to use memory for what is present in the graph. This approach will not involve any unnecessary pre-allocation or wasted space.
2. **Batch Loading:** I wrote a function to quickly add thousands of users or connections from lists or files. This approach allowed me to simulate big networks, like what we usually do in large data-loading for scalable analytics systems (Sahu et al., 2017).
3. **Lazy Analytics and Memory Management:** The program saves time and keeps memory usage in check by only calculating centrality/analytics when needed and caching results. Furthermore, I investigated using libraries like NetworkX or Neo4j for better mapping or larger networks and integrating a real database for storage and processing.

**Challenges faced:**

Python shows limitations when the network gets extremely large, especially with large users and networks. The program might slow down or run out of memory at that large scale. My previous implementation was exemplary for up to thousands of nodes, but beyond that, the pain points started to be visible.

**Advanced Testing and Validation**

Testing was crucial to ensure my optimizations did not just look good on the code, but improved things in practice. I implemented unit and stress tests alongside validation strategies to verify the optimized code.

1. **Unit Tests:** I tested every method, including adding/removing users, making connections, finding top influencers, updating/retrieving profiles, and traversing the network with BFS. Furthermore, I tested edge cases (like self-loops, users with no connections, and duplicate users) to be sure the system handled “weird” situations and failed gracefully without crashing the whole project. All the tests passed successfully. You can see the implementation in the screenshot below as well.  
     
   A screen shot of a computer program

   AI-generated content may be incorrect.  
     
   Here is an example of what you can see from the console for the unit test. A screen shot of a computer

   AI-generated content may be incorrect.
2. **Stress Tests:** To mimic more realistic large networks, I performed stress tests creating connections for 10,000 and 50,000 users. For each size, I measured a couple of metrics: how fast users and connections were added, how quickly top-k influencers were found, and finally, how BFS performed on dense, sparse, and disconnected graphs.

A screen shot of a computer program

AI-generated content may be incorrect.

For this stress test, the console output looks like all users and connections were added instantly, in less than a second. The same was for finding the top 10 influencers and finding a random user, which is very quick. A computer screen with white text

AI-generated content may be incorrect.

1. **Correctness Validation:** For validation testing, I targeted edge cases like self-loops, duplicate connections, connections with non-existing users, and disconnected users. The code below looks like this. A computer screen shot of a program code

   AI-generated content may be incorrect.

A computer screen with white text

AI-generated content may be incorrect.The response below shows that the code can handle all the above mentioned scenarios without crashing the system or any severe warnings.  Here we are adding a self-loop with Alice, and duplicates with Bob, but there were no crashes. Similar was trying to add a non-existent user ‘eve’.

**Final Evaluation and Performance Analysis**  
I visualized my analytics scaled with network size regarding the final evaluation and performance analysis. Using my benchmark script, I tested networks with 1,000, 5,000, 10,000, 20,000, and 100,000 users, and plotted how long it took to run degree centrality, find top influencers, and perform BFS. The graph was created using the matplotlib module, and it looks like the graph below:

A computer screen shot of a black screen

AI-generated content may be incorrect.

A graph with a green line

AI-generated content may be incorrect.

(Graph shows that degree centrality, influencer ranking, and BFS all scale efficiently with increasing network size. No sudden spikes—suitable for real-world use!)

**From the graph, you can observe that as** the network size increased, the time for each operation grew slowly and predictably, indicating good scalability for in-memory analytics. Similarly, Centrality calculation and influencer analytics scaled nearly linearly with network size.

BFS was also fast unless the starting user was part of a huge component, which was the case for 100,000 users, and expected in real networks. This is also confirmed from the console output and the graph above. Additionally, there were no crashes or runaway memory usage even for large networks.

Based on the graphs and benchmark results, you can see that this optimized solution is far better than the one presented in deliverable 2. Some of the learnings from this approach are as follows:

**Optimized vs. Initial Implementation:**

* The original code in deliverable 2 worked well for small networks but struggled with increasing data size. To be specific, analytics performed slow and sometimes failed with errors.
* After optimization, adding users/connections is faster and more reliable, and analytics (like top influencers) run almost instantly, even for networks with tens of thousands of users.
* Memory was one of the major issues before, but now memory and performance scale predictably. Also, edge cases were handled cleanly.

**Trade-Offs:**

* One flaw with this approach is that caching analytics results speeds up repeated queries but uses more memory.
* Using Python’s built-in structures makes the code readable, but a specialized database or library would be better for even larger datasets.
* All results are accurate, but approximations could make things even faster with extensive networks.

**Strengths & Limitations:**

* Strengths: Robust, efficient, and easy to extend; suitable for real-world-sized data.
* Limitations: In-memory Python has scaling limits but lacks advanced analytics and parallelism.
* Future work: Move to a graph database, add parallel analytics, and implement more metrics.

Hence, the optimizations to the code made the system much more scalable and reliable. This is proven by the results, which can be seen in the screenshots above. Finally, analytics are fast for large networks, and the system is ready for further extension.

**GitHub**

The detailed code for this implementation can be found in the GitHub: <https://github.com/ucajalrc/Deliverable-3>

**References**

Cheng, J., Yu, J. X., & Qin, L. (2013). Graph computation: Techniques and applications. Proceedings of the VLDB Endowment, 6(12), 1458–1459. <https://vldb.org/pvldb/vol6/p1458-cheng.pdf>

Hao, J., Li, J., Li, H., & Wu, K. (2018). Optimizing large-scale hash tables for key-value stores. IEEE Transactions on Computers, 67(4), 543–555. <https://ieeexplore.ieee.org/document/8049519>

Sahu, A. K., et al. (2017). The who, what, and how of high-performance graph processing. Proceedings of the 2017 Symposium on Cloud Computing, 475-489. <https://dl.acm.org/doi/10.1145/3127479.3131624>