DS Final Project Report:

Project Overview

- The goal of this project is to analyze Facebook's profiles, circles, and ego networks–which are also referred to as their node features. Through use of centrality (degree, closeness, and betweenness) measures and analysis, the program identifies the most influential nodes. It also figures out which are most connected and accessible from other nodes and is anonymized.
- I used Stanford University's SNAP website to access their Facebook Combined dataset, which uses data retrieved from Facebook's survey participants. The dataset is undirected with 4,039 nodes and 88,234 edges.

Data Processing

- I downloaded the dataset off of the SNAP website and extracted the file. Then, I uploaded it to the Rust project file's src folder on Visual Studio Code. I imported BufReader, and std::fs::File in order to read in the information from the dataset and load it in order to perform other operations. I parsed through each line of the file and stored the information in an adjacency list (HashMap<usize, Vec<usize>>). This data depicted an undirected edge between two user IDs–the hashmap's keys represented the node IDs, and the hashmap's values stored the list of the neighboring nodes.
- I did not do any cleaning or transformation on the dataset because it was already clean and organized. It seemed very usable and easy to work with, which was one of the reasons I chose it.

Code Structure

- I did not use any structs, enums, or trait for my program, but rather, I used a type alias Graph, which was a HashMap that I dedicated to storing data of usize, and Vec<usize>. The purpose of this was to handle the graph, which was used in all of my other functions and helped work with the adjacency list.
- The graph.rs module is specifically designed to load in the data from the facebook_combined.txt file and construct the graph.
- The centrality.rs module's stores the three main functions used in the program–degree_centrality(), closeness_centrality(), and betweenness_centrality(). By maintaining these three functions in the same module, it makes the code clearer and easier to reuse, as each module has a specific function.
- These two modules are implemented in the entry point of my program (main.rs) through the mod graph; and mod centrality; statements at the beginning of my main.rs file.
- I used pub type Graph = HashMap<usize, Vec<usize>>; to construct an undirected graph with an adjacency list, which would map each node idea to their neighbors. This was implemented in all three of the centrality computations, as well as in the function that loaded in the graph.
- The load_graph() function took in a &str input and output Graph, which was a HashMap. The function's purpose was to read in the list of edges from the file and make a type Graph. The idea behind it was to open the file, parse through each line and separate it into two node IDs and store the edge in both directions (node 1 is connected to node 2 & node 2 is connected to node 1).
- The degree_centrality() function takes in a reference of Graph, meaning it allows access to the data (borrowing). This function returns a HashMap<usize, f64), which is

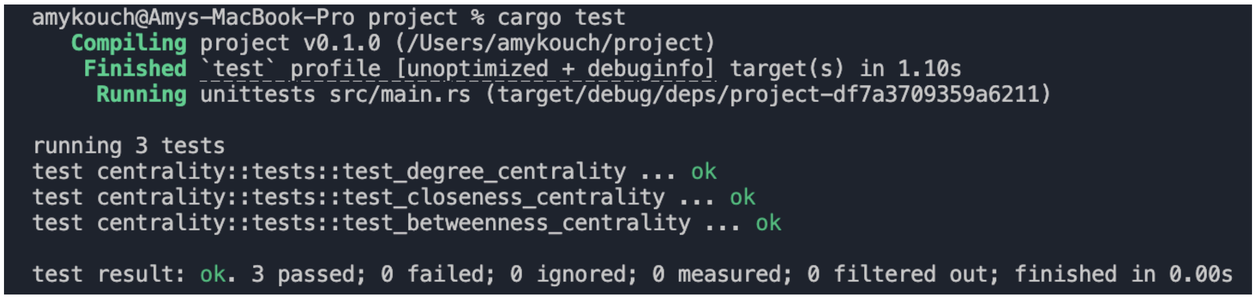
essentially a map from a node to its closeness score. Through use of for loops, it iterates through the nodes and counts its adjacency list's length (number of direct connections) and uses that to determine its degree count.

- The closeness_centrality() function takes in a reference to Graph, and returns HashMap<usize, f64>, which maps each node to their closeness centrality. I made use of the breadth-first search to find the shortest path from a node to every other reachable one. Then, the distances are added to compute closeness and find the shortest distance to the other nodes from a specified starting node. Then I calculated the inverse in order to make it more readable and comprehensible. Higher scores indicate higher closeness scores–meaning that the code is close to many others in the dataset.
- The betweenness_centrality() function takes in a reference to Graph, and again, returns a HashMap<usize, f64>, which maps each node to their betweenness value. This function finds specific nodes that neighboring nodes may use as bridges to other nodes. It performs the breadth-first search and finds the nodes with the shortest distances. If many of these shortest paths cross the same node, then it indicates that it's serving as a bridge in the dataset for nodes to access other nodes. Then, the function starts from the furthest nodes and checks each node based on the previous metric in order to assign its betweenness score.
- My main.rs file is where all of the functions are used in execution. First, the load_graph function is used to load in the facebook_combined.txt file, then each centrality measure is computed, and finally, everything is printed. For each centrality measure, the program correctly identifies the top five nodes for each category–sorting it and printing it.
- As an overview, my graph module helps read in the data and load it, while my centrality module is where the rest of the functions are maintained. These functions compute the centrality scores that I am focused on identifying and analyzing. This is also where I implemented my test cases, which can be shown if cargo test is used in the terminal. I further discuss them in the next section. Together, these three main functions interact in the main function, where I am able to turn my raw data into meaningful information.

Tests

- I incorporated three tests under the centrality.rs module to run test cases for the three primary functions and centrality measures and verify their results. First, I created a sample graph with fn sample_graph(), which would implement the three functions to test them. It creates a triangle, consisting of three connected nodes. The first test evaluates the degree_centrality() function. It checks whether each node has two connections, which is expected in a triangle. As a result, it confirms that the function is able to correctly implement degree centrality logic and correctly count the nodes. The second test calls the closeness_centrality() function and retrieves a node's closeness value. Then using assert! It checks its range to ensure that it is within 0 and 1–which would indicate that the function runs correctly and without erroring. The last test calls the betweenness_centrality() function and ensures that all the scores are positive or 0 (a negative number would be invalid).

Testing output:

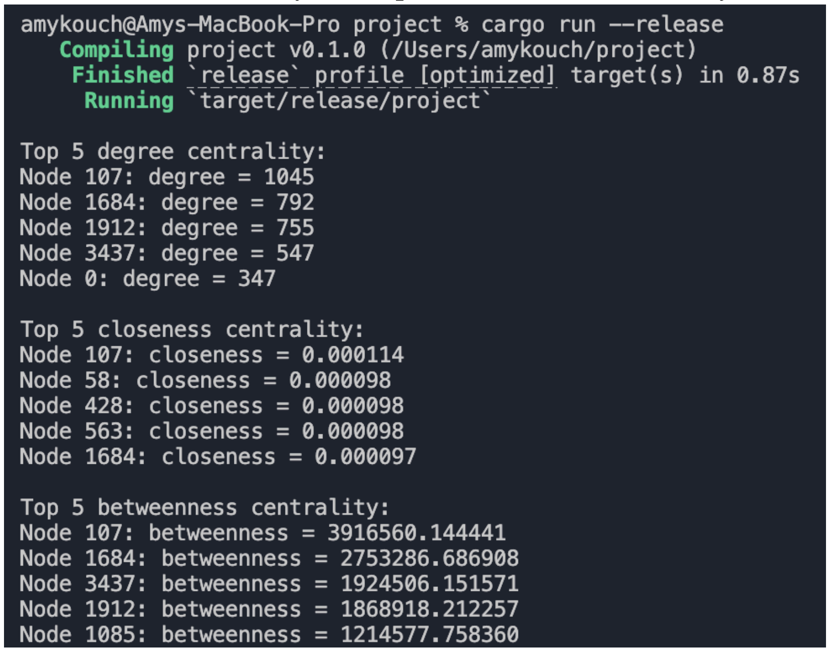


```
amykouch@Amys-MacBook-Pro project % cargo test
   Compiling project v0.1.0 (/Users/amykouch/project)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 1.10s
     Running unittests src/main.rs (target/debug/deps/project-df7a3709359a6211)

running 3 tests
test centrality::tests::test_degree_centrality ... ok
test centrality::tests::test_closeness_centrality ... ok
test centrality::tests::test_betweenness_centrality ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

- The testing output shows that the three functions correctly perform what the test functions ask of them–finishing in 0.0 seconds.

Results

- The program outputs the top 5 degree centrality nodes, then top 5 closeness centrality nodes, and lastly, the top 5 betweenness centrality nodes.



```
amykouch@Amys-MacBook-Pro project % cargo run --release
   Compiling project v0.1.0 (/Users/amykouch/project)
    Finished `release` profile [optimized] target(s) in 0.87s
     Running `target/release/project`

Top 5 degree centrality:
Node 107: degree = 1045
Node 1684: degree = 792
Node 1912: degree = 755
Node 3437: degree = 547
Node 0: degree = 347

Top 5 closeness centrality:
Node 107: closeness = 0.000114
Node 58: closeness = 0.000098
Node 428: closeness = 0.000098
Node 563: closeness = 0.000098
Node 1684: closeness = 0.000097

Top 5 betweenness centrality:
Node 107: betweenness = 3916560.144441
Node 1684: betweenness = 2753286.686908
Node 3437: betweenness = 1924506.151571
Node 1912: betweenness = 1868918.212257
Node 1085: betweenness = 1214577.758360
```

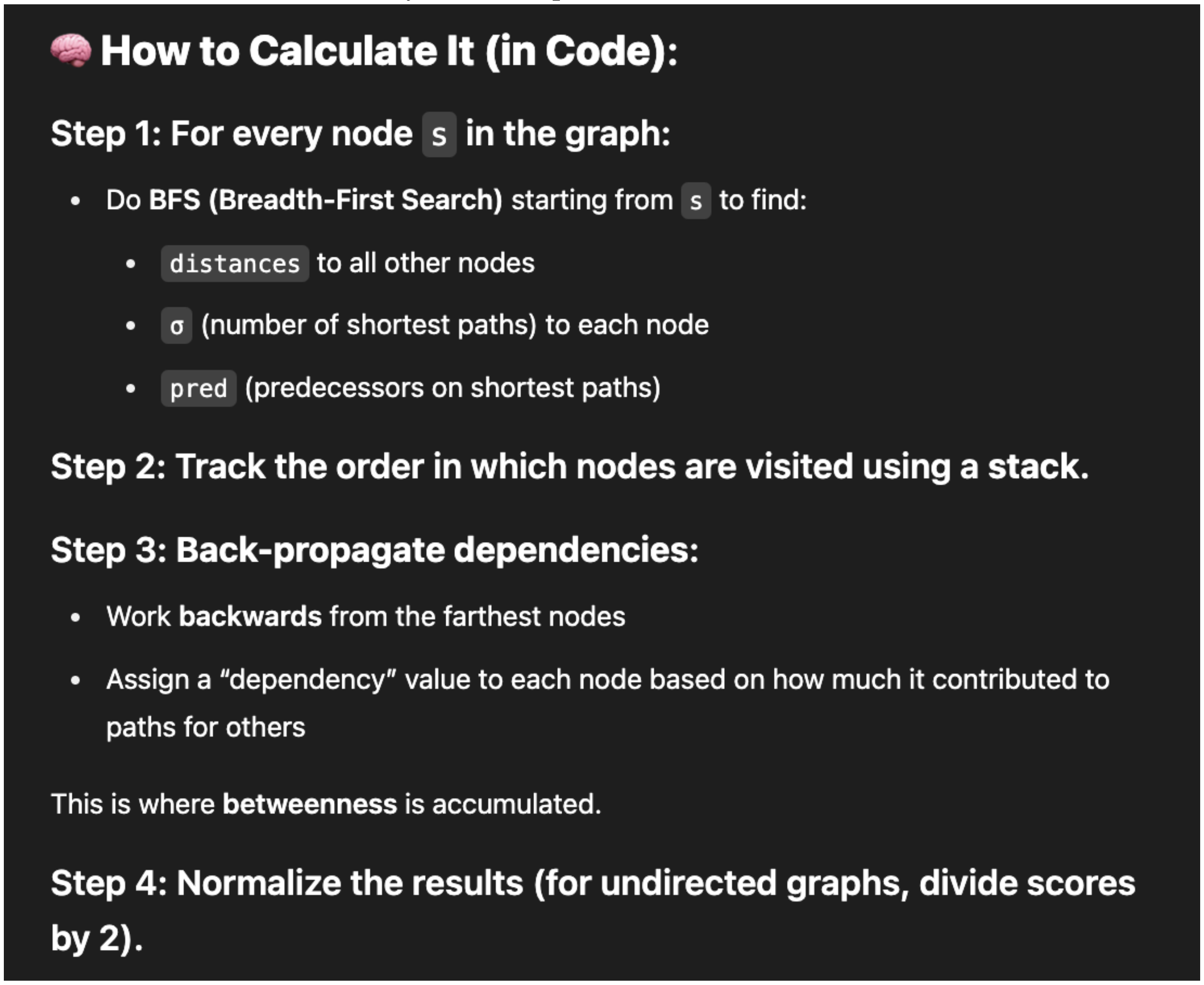
- As the output conveys, Node 107 seems to rank highest in all three centralities–degree, closeness, and betweenness. This indicates that out of the entire dataset, it is the most connected to other nodes. However, other nodes like Node 1684, Node 3437, and Node 1912 appear frequently within the top 5 for the categories as well–which implies they are also central, but not to the same level as Node 107, which outperforms the other nodes in all categories. Node 107's high degree centrality score suggests that it is well-connected to other users (directly). Its high closeness value reveals that it has a shorter average path compared to other nodes, and its high betweenness score shows that different nodes are able to use it as a bridge to other nodes frequently. In context of the dataset, this depicts that Node 107 has many friends (direct connections), knows many people across the platform (is located centrally in terms of profiles, less profiles to search through before reaching someone), as well as serves as a mutual friend to many (serving as a bridge between two people).

Usage Instructions

- The program is easy to run–just type cargo run into the terminal. The degree centrality runs super fast (0.95s), whereas when calculating closeness centrality and betweenness centrality, it takes much longer. As a result, I use cargo run –release when running the program because it optimizes it and runs it quicker. The top degree centralities are calculated in (0.85s), the closeness centrality and betweenness centrality are both calculated slightly slower, but significantly faster than when –release is not used. The results are all printed in the terminal window. There are no command-line arguments or user interactions in the terminal aside from the initial execution of the project.

AI Assistance Disclosure

- ChatGPT was used in order to help clarify issues and errors within my code. I had lots of trouble with the betweenness function, resulting in lots of questions on ChatGPT. I also had trouble with my test cases, which ChatGPT also clarified.
- I used AI to help me calculate betweenness centrality, as I was initially unsure of how to calculate it within my project. ChatGPT helped refresh me on how to also implement the breadth-first search, as I was rusty on that implementation.



🧠 How to Calculate It (in Code):

Step 1: For every node `s` in the graph:

- Do **BFS (Breadth-First Search)** starting from `s` to find:
  - `distances` to all other nodes
  - `σ` (number of shortest paths) to each node
  - `pred` (predecessors on shortest paths)

Step 2: Track the order in which nodes are visited using a stack.

Step 3: Back-propagate dependencies:

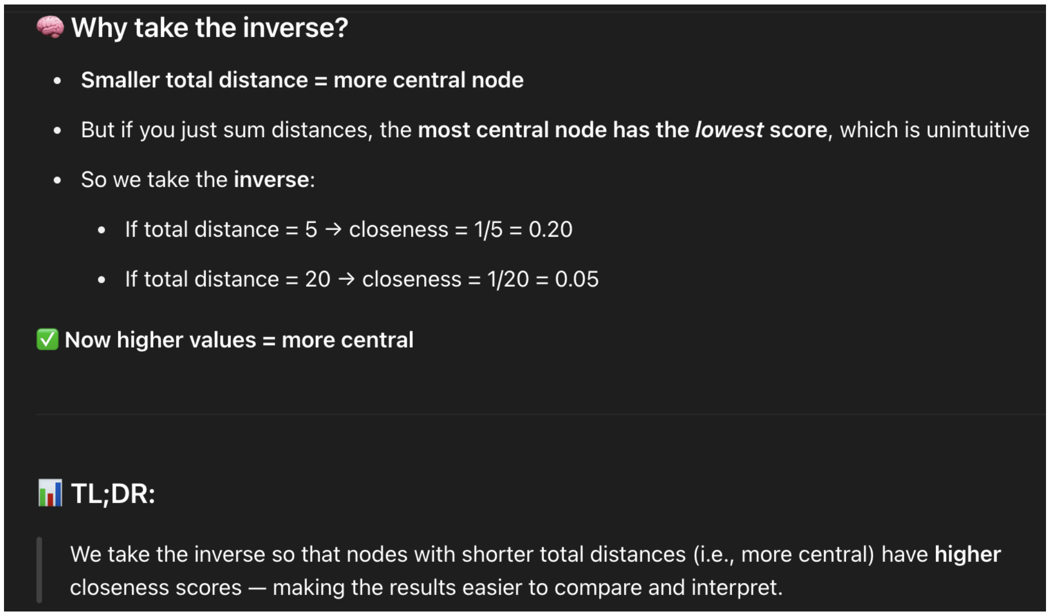
- Work **backwards** from the farthest nodes
- Assign a “dependency” value to each node based on how much it contributed to paths for others

This is where **betweenness** is accumulated.

Step 4: Normalize the results (for undirected graphs, divide scores by 2).

- I had no idea what back-propagate dependency means, so in order to implement my betweenness-centrality function, I had ChatGPT simplify the measures I needed to take in order to craft the function.
- After giving it context of my issue and code, it helped me better display my results. For closeness centrality, I had to take the inverse to make it more

readable, as ChatGPT explained that this would make it easier to view and understand.



🧠 Why take the inverse?

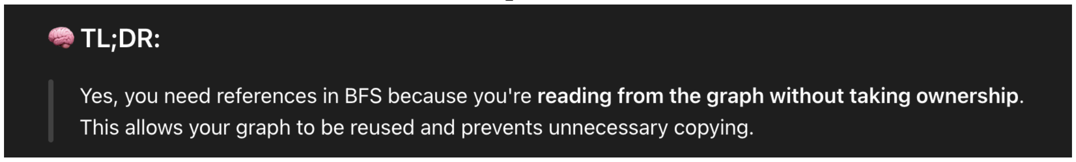
- **Smaller total distance = more central node**
- But if you just sum distances, the **most central node has the *lowest* score**, which is unintuitive
- So we take the **inverse**:
  - If total distance = 5 → closeness = 1/5 = 0.20
  - If total distance = 20 → closeness = 1/20 = 0.05

✅ **Now higher values = more central**

📊 TL;DR:

> We take the inverse so that nodes with shorter total distances (i.e., more central) have **higher** closeness scores — making the results easier to compare and interpret.

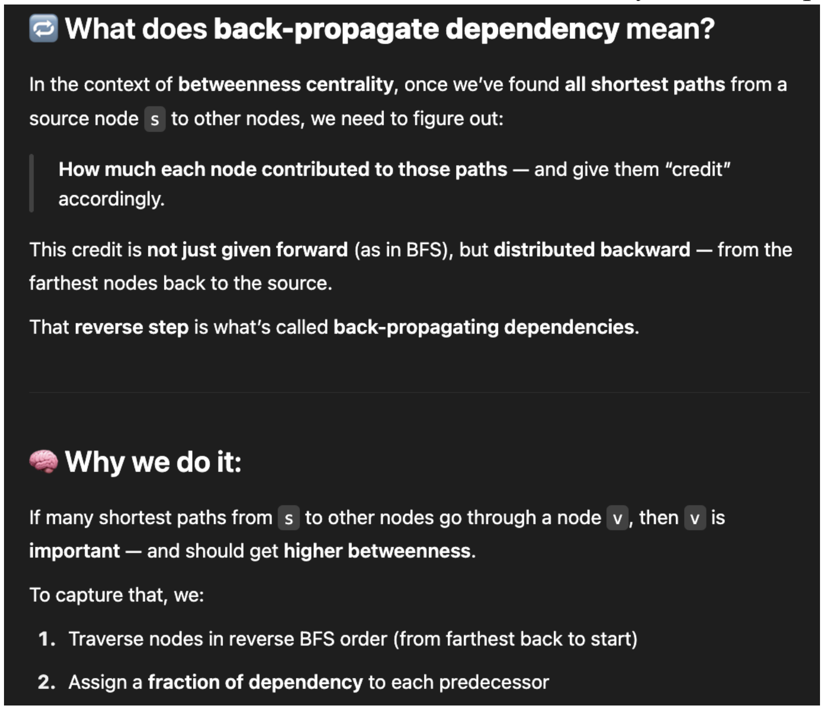
- I had lots of difficulties determining when to use references, and when not to, which ChatGPT also helped me answer.



🧠 TL;DR:

> Yes, you need references in BFS because you're **reading from the graph without taking ownership**. This allows your graph to be reused and prevents unnecessary copying.

- Through many small conversations with the AI bot, I was able to correctly calculate the betweenness centrality values and print out the top 5 highest scores.



🔁 What does **back-propagate dependency** mean?

In the context of **betweenness centrality**, once we've found **all shortest paths** from a source node `s` to other nodes, we need to figure out:

> **How much each node contributed to those paths** — and give them "credit" accordingly.

This credit is **not just given forward** (as in BFS), but **distributed backward** — from the farthest nodes back to the source.

That **reverse step** is what's called **back-propagating dependencies.**

🧠 Why we do it:

If many shortest paths from `s` to other nodes go through a node `v`, then `v` is **important** — and should get **higher betweenness.**

To capture that, we:

1. Traverse nodes in reverse BFS order (from farthest back to start)
2. Assign a **fraction of dependency** to each predecessor