

# Project Report: Analysis of Shortest Paths in a Social Network Using Rust

By: Melaku Mohammed

## Problem Statement

The objective of my project was to use the Breadth-First Search (BFS) algorithm to witness the shortest paths between nodes, representing users within a Facebook network graph. In doing so, I could gauge how closely related users are to one another. Firstly, the main goal was to calculate the sample's mean, median, and standard deviation of the shortest path length. Using the mean, I wanted to determine if the average degree of separation between pairs in a network is six or less. If so, we would be able to say that the network supports the "small world hypothesis." Secondly, I wanted to witness the percentage of the network that was reachable within six steps from any randomly chosen node. Thirdly, I wanted to calculate the average clustering coefficient of the network as well to get an additional measure of how tight this social network is altogether.

## Implementation

To start, I made two modules holding corresponding structs within my src file. The first was my GraphManager modules, which managed the construction and manipulation of the graph representation of my network. The second was my Statistics Struct, which helped me understand the distribution and variance of path lengths within the network, to help infer the network's connectivity.

### *graph\_manager.rs*

Within this module, I created the method "new", which initializes the GraphManager with an empty undirected graph and an empty hashmap. Within it, I used a graph data structure from an outside library called petgraph to represent the nodes and edges without direction. I then used a hashmap to map node identifiers to their node index within the graph. For my next method, "build\_graph", my main objective was to take a text file, which in this case was my network graph, and read it line by line to interpret. For each line within the file, the method splits the line into two nodes using the whitespace and takes the node identifiers, parsing them into usize. Each pair of these nodes within the file represents an edge between two nodes in a graph, and for each node, the method checks if it is already in our previously created hashmap or not. If it is not, we add it to the graph along with its node index being stored in said hashmap. For my third method, "get\_node\_indices," I collected all the node indices within the stored graph. This was done by collecting the values from the hashmap, which are all "NodeIndex" types, and cloning them to create a new collection of indices to reference. For my last method, "node\_count," I simply gave the size of the graph by counting the number of nodes.

#### *statistics.rs*

Within this module, I also created the method “new.” which initializes instances of the Statistics struct. This method helps to see the struct with an empty vector. This vector will then store the lengths of the paths when calculated to then use to find our mean, median, and standard deviation (SD). For my next method, I created the function “add\_length” which takes the shortest length between two randomly selected nodes and adds that length to our given vector. Lastly, the most important method, “compute” takes the vector of lengths and calculates the mean, median, and SD of said lengths. For the mean, the function sums all the lengths in the vector together and divides it by the number of paths. For the median, the function clones then sorts the length values in the vector using the `sort_unstable` function, a built-in method. If the number of points in the vector is even, the function takes the average of the two middle points and if it’s odd, it takes the value of the middle point. For the SD, we first used the found mean to calculate the average of the squared differences from the mean and then got the square root of said value to give an idea of how much the lengths deviate from the average.

#### *main.rs*

Finally, this program orchestrates the uses of the previously mentioned modules, which are now declared. After reading my file, I initiate the GraphManager and use the “build\_graph” method to parse the file and build the graph. For the Analysis of the built graph, the Statistics module is initialized to track and compute the statistics on lengths. I then randomly selected 1000 pairs of nodes using the “thread\_rng” function the “rand” created, a library in Rust. Now for each pair in my sample, I compute the shortest path between the two using the Breadth-First Search (BFS) function. This function works by marking the starting node. It then initializes the distance from the starting nodes as zero and places them in a queue to explore the next nodes. The function then enters a loop where each node is processed from the queue, and for each node, it checks if the node is the target node. If the node is the target node, the function returns the distance to this node as the shortest path. For the same current node though, the function explores all neighbors and checks if each neighbor has been previously visited to avoid redundancy. If a said neighbor has not been visited, the function marks it as visited, calculates its distance, and adds it to the queue to later explore the neighbor and its own neighbors as well. With this process, the loop continues until the queue is empty and there are no more nodes to explore anymore. The function then returns the shortest path length as a `u32` type, but if no path exists, it returns a `None` type instead. Now once this function identifies the shortest path lengths between 1000 randomly selected pairs of nodes in the network, the main program utilizes the Statistics module to analyze these path lengths. Each resulting length is added into a vector. I then use the compute method to calculate the average and median path length as well as the standard deviation of such. After calculating the average path length, I check if the result is less than or equal to 6. If it is, then we can confirm that the network supports the “Small World Hypothesis,” indicating a tightly connected network. If not, then we know that the networks are a bit separated.

Now for our reachability analysis, I wanted to see what percent of nodes are reachable within 6 steps. To do this, I used the “bfs\_reachability” function within the main program, which counts how many nodes can be reached from a given start node using a predetermined amount of steps. This function works by marking the start node for the BFS to begin and setting the maximum number of steps for the BFS to explore from the start node. A vector is also made to keep track of each node in the graph that has been visited to avoid repeat visits. Now to begin our BFS exploration loop, the start node is marked as visited with a distance of 0, and it’s added to a queue. The function then enters into a loop that explores all reachable nodes up to the maximum depth allowed. The function then returns the count of nodes that was able to be reached from the starting node. Now running this function with 1000 different starting nodes, I calculate how many nodes were reachable within the max depth and store these counts into a vector. I then took the average of the number of reachable nodes and divided it by the total number of nodes in the graph to get my percentage. With this percentage, I am now able to see just how much of the network is accessible within six steps.

Lastly, I wanted to get the Clustering Coefficient of the network, which gives a measure of the degree to which nodes in a graph tend to cluster together. To do this I had to get the coefficient for each node, by iterating over each one, retrieving its neighbors, and counting the actual edges between these neighbors. This coefficient for each node is then calculated by dividing the number of actual edges by the total possible edges. From there, I then average these values for each node to get a single value to represent the graph.

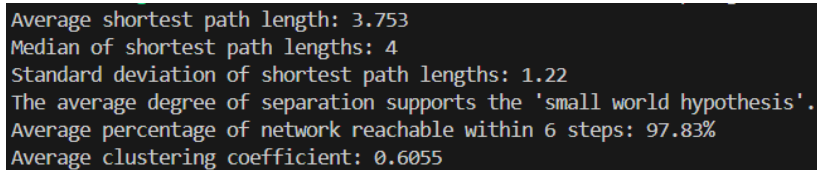
### *Tests*

For tests, I set up three crucial ones. The first of which tests the “bfs\_shortest\_path” function, making sure it correctly computes the shortest path between nodes in a graph. To do this, I set up a new, undirected graph with three nodes along with edges connecting node1 to node2 and node2 to node3. The test then calls said function and makes sure that it returns a value of 2, which is the expected result. For my second test, I wanted to ensure that the Statistics module correctly calculates the mean, median, and standard deviation of given data. To do this, I set an instance of the Statistics struct is created and populated with five path lengths (1, 2, 3, 4, 5). I then used the “compute” method to calculate the mean, median, and standard deviation of the added path lengths. From there I checked the corresponding values against the expected values. For the last test, I wanted to confirm that the “bfs\_reachability” function correctly calculates the number of nodes reachable within a specified number of steps from a start node. Similar to the first test, I created an undirected graph, but this time with four nodes linked sequentially instead of three. Then I called said function to witness how many nodes are reachable from n1 within 6 steps. Knowing that all nodes 4 nodes were within reach, I checked that the expected value matched the returned value (4).

## Output Analysis

Now to interpret the output, I'll discuss a single instance where I ran the main program. It should be noted that I have run the program multiple times to make sure that the values did not differ significantly. The average shortest path length resulted in 3.75, meaning that on average, any two nodes (representing individuals on Facebook) are separated by about 3.753 steps. The median value resulted in a 4, meaning that at least half of all user pairs are separated by 4 or fewer steps. The standard deviation resulted in 1.22, suggesting that most path lengths are closer together on average. These values, specifically the average, tell us that the network is highly connected, meaning information (like posts, news, etc.) can spread quickly from one individual to another through a small number of intermediate connections. With an average length of 3.75, we know that this network supports the "Small World Hypothesis," which claims that most people are connected through less than 6 steps. Such a finding is to be expected from social networks, where any person can be reached through a small number of connections. Next, we find that nearly all (97.83%) of the network's nodes can be reached from any other node within 6 steps. This high reachability percentage is also indicative of an inclusive network where almost every individual can connect or impact nearly every other individual within a few steps. Lastly, the average clustering coefficient is 0.605 on a scale from 0 to 1, where 1 means every node's neighbors are also neighbors with each other. This value tells us that this network also has a higher level of local tight-knit groups.

Below is a screenshot of the previously mentioned output:

A screenshot of a terminal window showing the output of a network analysis program. The text is as follows:

```
Average shortest path length: 3.753
Median of shortest path lengths: 4
Standard deviation of shortest path lengths: 1.22
The average degree of separation supports the 'small world hypothesis'.
Average percentage of network reachable within 6 steps: 97.83%
Average clustering coefficient: 0.6055
```

Now overall, it is clear that these values indicate a highly efficient, interconnected social network, which I would expect from a network like Facebook. This network's structure supports users to spread information widely and quickly within communities of other users, which seems to be critical for user engagement on social media platforms. Hence, one of the many reasons that Facebook has been and still is one of the largest social media platforms to ever exist.