DS210 Final Project: Twitter Connectivity Analysis

By Nicholas Thompson

Abstract

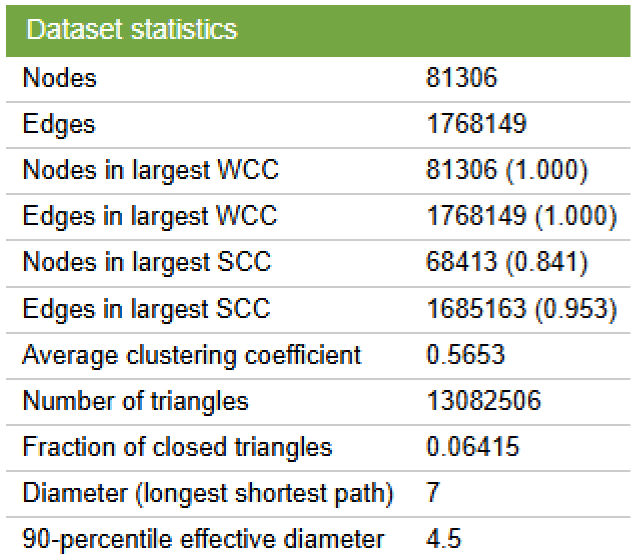
This project aims to provide insight into the connectivity of Twitter as a social media platform via several graph analysis methods in Rust, providing the user with information about the structure of Twitter and details about user profiles. The dataset is taken from the Stanford Network Analysis Project (SNAP), entitled "Social circles: Twitter" obtained from the following URL: https://snap.stanford.edu/data/ego-Twitter.html. Containing 81,306 node features (profiles) and 1,768,149 edges, this dataset provides useful insight into real world social media platforms, allowing for structure analysis to understand their connectedness, enabling users to determine whether or not they are useful in connecting like minded individuals. The functions implemented in this project, which will be discussed later in this report, provide structural properties of the Twitter graph such as average degrees of separation, as well as provide useful details about specific user profiles and generate recommendations of other profiles users should follow based on shared neighbors.

Dataset Properties and Downloading Instructions

The dataset used for this project was taken from the Stanford Network Analysis Project (SNAP), entitled “Social circles: Twitter” obtained from the URL: https://snap.stanford.edu/data/ego-Twitter.html

Properties:

The dataset consists of circles from Twitter taken from public sources and includes node features (profiles), ego networks, and circles. Twitter was specifically chosen, as it provides a more interesting array of profiles. For example, many people get news from Twitter, whereas they likely would not use a platform like Instagram as a news source, as it is more focused on connecting small hubs of already acquainted people. Some of the dataset statistics described on the SNAP website include 81,306 Nodes. Each of these nodes are connected to a node ID, which can be understood as individual Twitter profiles. Connections between nodes are made when different profiles follow each other, allowing the derivation of many key statistics. Some of the other dataset statistics are as follows:



Dataset statistics	
Nodes	81306
Edges	1768149
Nodes in largest WCC	81306 (1.000)
Edges in largest WCC	1768149 (1.000)
Nodes in largest SCC	68413 (0.841)
Edges in largest SCC	1685163 (0.953)
Average clustering coefficient	0.5653
Number of triangles	13082506
Fraction of closed triangles	0.06415
Diameter (longest shortest path)	7
90-percentile effective diameter	4.5

J. McAuley and J. Leskovec. Learning to Discover Social Circles in Ego Networks. NIPS, 2012.

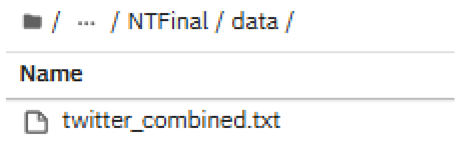
The data is available from the SNAP website, Facebook, or Google+ and the instructions to download it are as follows:

Step 1: Go to the SNAP website - https://snap.stanford.edu/ - and navigate to the “Stanford Large Network Dataset Collection” on the front page.

Step 2: Under “Social Networks” find the dataset with the name “ego-Twitter” - it should be the third selection on the list with 81,306 nodes and 1,768,149 edges.

Step 4: Under “Files” at the bottom of the page download the “twitter_combined.txt.gz” file. This should automatically download and will allow you to import the dataset into the program.

Note - The dataset is already imported into the program in the “data” folder of the repository. If this for some reason does not work, simply create this folder yourself and import the “twitter_combined.txt.gz” file inside of it. It is important to note that the .gz file extension is not needed, and inside the data folder should only be “twitter_combined.txt” as such:


/ ··· / NTFinal / data /

Name

twitter_combined.txt

Feature Implementation and Output

This project employs an array of features used to analyze the structure of the graph to provide insights into Twitter as a social media platform. In order to fully outline the output and purpose of each function, I will explain chunks of code from each of the files inside the src folder of the repository beginning with the graph.rs file, then the main.rs file, then the analysis.rs file, and finally the tests.rs file.

Graph.rs

```
use std::collections::{HashMap, HashSet};
use std::fs::File;
use std::io::{self, BufRead};
use std::path::Path;
use rand::seq::IteratorRandom;
```

This chunk of code simply imports the correct libraries needed for each of the functions. The libraries are used to create a hashmap representing the graph, read the dataset file, and randomly sample nodes from the dataset to create a sample graph.

```
fn read_lines<P>(filename: P) -> io::Result<io::Lines<io::BufReader<File>>>
where
   P: AsRef<Path>,
{
   let file = File::open(filename)?;
   Ok(io::BufReader::new(file).lines())
}
```

This function is simply a utility function to help read lines from a file, allowing for easier readability of the graph dataset. It uses the **use std::io::{self, BufRead};** library imported at the top of the file.

```
pub fn read_graph(file_path: &str) -> HashMap<usize, HashSet<usize>> {
   let mut graph = HashMap::new();
   if let Ok(lines) = read_lines(file_path) {
      for line in lines {
         if let Ok(edge) = line {
            let nodes: Vec<usize> = edge.split_whitespace()
               .filter_map(|s| s.parse::<usize>().ok())
               .collect();
            if nodes.len() == 2 {
               graph.entry(nodes[0]).or_insert_with(HashSet::new).insert(nodes[1]);
               graph.entry(nodes[1]).or_insert_with(HashSet::new).insert(nodes[0]);
            }
         }
      }
   }
   graph
}
```

This function is used to read the graph from the dataset, representing it as an adjacency list stored in a hashmap. It takes a file path for a graph dataset of edges and reads the file line by line, parsing each line into two nodes (two profiles), and adds the edge to the adjacency list. It allows for connections to be made between nodes and their neighbors and stores the graph structure for other functions to perform graph analysis.

```
pub fn sample_graph(
    graph: &HashMap<usize, HashSet<usize>>,
    sample_size: usize,
) -> (HashMap<usize, HashSet<usize>>, HashSet<usize>) {
    let sampled_nodes: HashSet<usize> = graph.keys().cloned()
        .choose_multiple(&mut rand::thread_rng(), sample_size)
        .into_iter()
        .collect();
    let mut sampled_graph = HashMap::new();
    for &node in &sampled_nodes {
        if let Some(neighbors) = graph.get(&node) {
            let filtered_neighbors: HashSet<usize> =
neighbors.intersection(&sampled_nodes).cloned().collect();
            if !filtered_neighbors.is_empty() {
                sampled_graph.insert(node, filtered_neighbors);
            }
        }
    }
    (sampled_graph, sampled_nodes)
}
```

This function takes form the original graph and randomly samples nodes and their edges, creating a sample graph. This is used for convenience, as the dataset is large and may take a long time to run on certain machines, so allowing the user the option to randomly sample the graph avoids issues of the program crashing or having a long runtime. If no sample size is selected by the user, then the entire dataset is used and no random sampling takes place.

```
pub fn analyze_graph(
    graph: &HashMap<usize, HashSet<usize>>,
) -> (usize, usize, f64, f64) {
    let num_nodes = graph.len();
    let num_edges: usize = graph.values().map(|neighbors| neighbors.len()).sum();
    let avg_degree = num_edges as f64 / num_nodes as f64;
    // Compute average degrees of separation
    let random_node = *graph.keys().next().unwrap();
    let avg_degrees_of_separation = compute_avg_degrees_of_separation(graph,
random_node);

    (num_nodes, num_edges, avg_degree, avg_degrees_of_separation)
}
```

This function computes statistics about the graph, allowing users to infer certain properties about Twitter as a social media platform. It computes and displays the number of nodes, edges, average degree (number of edges per node) and the average degrees of separation. It is important to note the function leaves out nodes that are not connected to any other nodes, as they represent inactive profiles and don't contribute to the overall production of Twitter as a platform. The aim is to determine the connectivity of Twitter when used properly, so inactive accounts are excluded. This may result in a discrepancy between the number of sampled nodes and the number of nodes displayed in the output of this function's calculation. This is extremely useful in understanding the structure of Twitter. Firstly, the number of nodes provides a measure of how many profiles are represented in the dataset. The number of edges provides a measure of the total number of connections between accounts, revealing how often and how many users interact with one another. The average degree represents the average connections per node. This is extremely useful in understanding the connectedness of Twitter, as it provides an average value for how many profiles each person follows and interacts with. A higher average degree would represent a greater number of connections and follower relationships being made, thus, more interaction between profiles. The average degrees of separation represents the average shortest path between one node and every other node in the graph. This is extremely useful in understanding the overall connectedness of Twitter as a whole, highlighting its global connectivity. A lower value represents a more closely knit platform, revealing the small world phenomenon observed in many social media networks. This small world phenomenon suggests that people are more connected than they might think, as every person is connected through a small number of acquaintances. Overall, this function is the crux of the project, detailing the most important statistics relating to the complete connectedness of Twitter as a platform.

```
pub fn compute_avg_degrees_of_separation(
    graph: &HashMap<usize, HashSet<usize>>,
    start: usize,
) -> f64 {
    let mut visited = HashSet::new();
    let mut queue = std::collections::VecDeque::new();
    let mut total_distance = 0;
    let mut num_reachable_nodes = 0;

    queue.push_back((start, 0));
    visited.insert(start);

    while let Some((current, distance)) = queue.pop_front() {
        total_distance += distance;
        num_reachable_nodes += 1;

        if let Some(neighbors) = graph.get(&current) {
            for &neighbor in neighbors {
                if !visited.contains(&neighbor) {
                    visited.insert(neighbor);
                    queue.push_back((neighbor, distance + 1));
                }
            }
        }
    }

    total_distance as f64 / num_reachable_nodes as f64
}
```

This function simply calculates the average degrees of separation that was discussed in the explanation for the previous function. It calculates the average distance required to reach all other nodes from the starting node.

```
pub fn degree_centrality(graph: &HashMap<usize, HashSet<usize>>) ->
HashMap<usize, usize> {
    let mut centrality = HashMap::new();

    for (&node, neighbors) in graph {
        // Degree centrality is number of neighbors
        centrality.insert(node, neighbors.len());
    }

    centrality
}
```

This function calculates the degree centrality of each node in the graph, representing the number of neighbors for each node. This is important because it provides a direct measure of how many follower relationships a specific profile has, and is used to determine the most influential (most neighbors) Twitter profiles in the dataset. The nodes with the highest degree centralities are the nodes which are most connected to the rest of the graph, representing their vast influence. This can also be useful for advertising companies, as they can identify what are the most popular users and who can reach the most people.

```
pub fn most_shared_neighbors(
    graph: &HashMap<usize, HashSet<usize>>,
    selected_node: usize,
) -> Vec<(usize, usize)> {
    let binding = HashSet::new();
    let selected_neighbors = graph
        .get(&selected_node)
        .unwrap_or(&binding);

    let mut shared_counts: Vec<(usize, usize)> = graph
        .iter()
        .filter_map(|(&node, neighbors)| {
            if node != selected_node {
                let shared_neighbors = selected_neighbors
                    .intersection(neighbors)
                    .count();
                if shared_neighbors > 0 {
                    Some((node, shared_neighbors))
                } else {
                    None
                }
            } else {
                None
            }
        })
        .collect();

    shared_counts.sort_by(|a, b| b.1.cmp(&a.1)); // Sort by shared neighbor count,
descending
    shared_counts.truncate(5); // Keep top 5
    shared_counts
}
```

This function takes a reference to a node ID and finds which other nodes share the most common neighbors with it. This is extremely useful, as it can take a profile and provide suggested accounts for them to follow based on the amount of other profiles they each follow. For example, if profile x follows profile y, and profile z also follows profile y, then profile x is suggested to follow profile z. This can help aid in further connecting like minded individuals, further improving the overall connectedness of the dataset and Twitter as a whole. It is used in this project to provide the Top 5 most recommended profiles to follow (the 5 profiles with the highest number of shared neighbors). A future implementation of this project might further develop this function to allow for community detection, suggesting a user follow specific profiles based on their shared interests or follow a community based on a certain topic followed by shared neighbors.

That is all the code contained inside the graph.rs function. Each of these functions are implemented into the main.rs function to contribute to the final output.

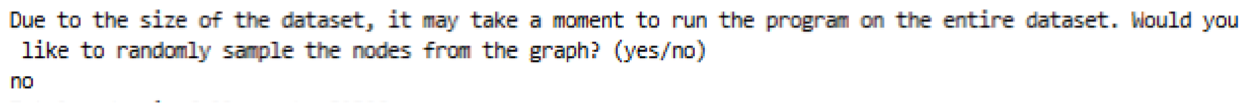
Main.rs

```
use std::io::{self};
use rand::prelude::SliceRandom;
mod graph;

const DATA_FILE: &str = "data/twitter_combined.txt";
```

This chunk of code simply imports the correct libraries and crates needed for the program to run. It also imports the graph module from our graph.rs file allowing for references to those functions defined earlier. Lastly, it defines DATA_FILE as the path to the dataset making for easier implementations.

```
fn main() {
    // Step 1: RANDOMLY SAMPLE. Ask the user if they want to randomly sample the graph for quicker analysis
    let mut sample_size: Option<usize> = None;

    println!("Due to the size of the dataset, it may take a moment to run the program on the entire dataset. Would you like to randomly sample the nodes from the graph? (yes/no)");
    let mut input = String::new();
    io::stdin().read_line(&mut input).expect("Failed to read line");
    let mut input = input.trim().to_lowercase();
```

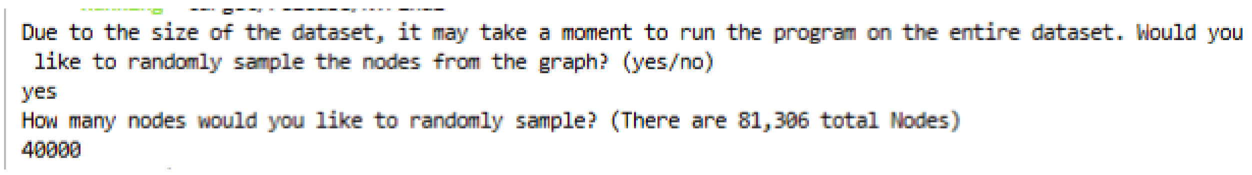
This code prompts the user to determine whether or not they would like to randomly sample the graph. Since the dataset is large and some of the methods (shared neighbors) have to iterate over each node, the program can have a long runtime on some machines. This avoids those issues and allows the user to randomly sample the graph.



```
Due to the size of the dataset, it may take a moment to run the program on the entire dataset. Would you
 like to randomly sample the nodes from the graph? (yes/no)
no
```

```
// Step 2: HOW MANY NODES. If yes, how many nodes
    if input == "yes" {
        println!("How many nodes would you like to randomly sample? (There are
81,306 total Nodes)");
        input.clear();
        io::stdin().read_line(&mut input).expect("Failed to read line");


        // Parse the user input as a number, if invalid number, default to full dataset
        if let Ok(number) = input.trim().parse::<usize>() {
            sample_size = Some(number);
        } else {
            println!("Invalid number. Proceeding with the full dataset.");
        }
    }
```

This code continues the sampling process by asking the user how many nodes they would like to sample if they answered yes. This allows for full user control of sampling, allowing the user to tailor the program to their machine's ability. If an invalid number is inputted (outside of the range of 1-81,306), the program defaults to the full dataset.



```
Due to the size of the dataset, it may take a moment to run the program on the entire dataset. Would you
 like to randomly sample the nodes from the graph? (yes/no)
yes
How many nodes would you like to randomly sample? (There are 81,306 total Nodes)
40000
```

// Step 3: READ THE GRAPH. Read the full graph
let graph = graph::read_graph(DATA_FILE);
let total_nodes = graph.len();
println!("Total nodes in full graph: {}", total_nodes);

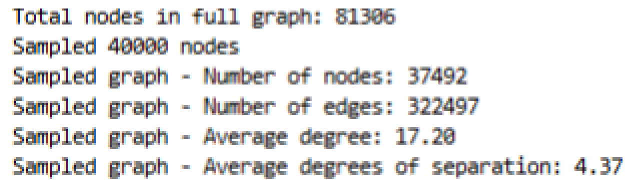
// Step 4: SAMPLE WITH DESIRED SAMPLE SIZE. Sample the graph if sample size provided, otherwise use the full graph

```
    let (sampled_graph, sampled_nodes) = if let Some(size) = sample_size {
        graph::sample_graph(&graph, size)
    } else {
        (graph.clone(), graph.keys().cloned().collect())
    };

    let sampled_nodes_count = sampled_nodes.len();
    println!("Sampled {} nodes", sampled_nodes_count);
```

This code simply reads the full graph using the read_graph function outlined earlier and takes the user's desired sample size and completes the sampling process. This displays some of the basic graph properties and how many nodes were sampled.

```
    // Step 5: GRAPH ANALYSIS. Analyze the graph
    let (num_nodes, num_edges, avg_degree, avg_sep) =
graph::analyze_graph(&sampled_graph);
    println!("Sampled graph - Number of nodes: {}", num_nodes);
    println!("Sampled graph - Number of edges: {}", num_edges / 2);
    println!("Sampled graph - Average degree: {:.2}", avg_degree);
    println!("Sampled graph - Average degrees of separation: {:.2}", avg_sep);
```

This code prints the complete graph analysis, utilizing the functions defined earlier in the graph.rs file. It prints the number of nodes, edges, average degree, and average degrees of separation along with the basic graph properties.



```
Total nodes in full graph: 81306
Sampled 40000 nodes
Sampled graph - Number of nodes: 37492
Sampled graph - Number of edges: 322497
Sampled graph - Average degree: 17.20
Sampled graph - Average degrees of separation: 4.37
```

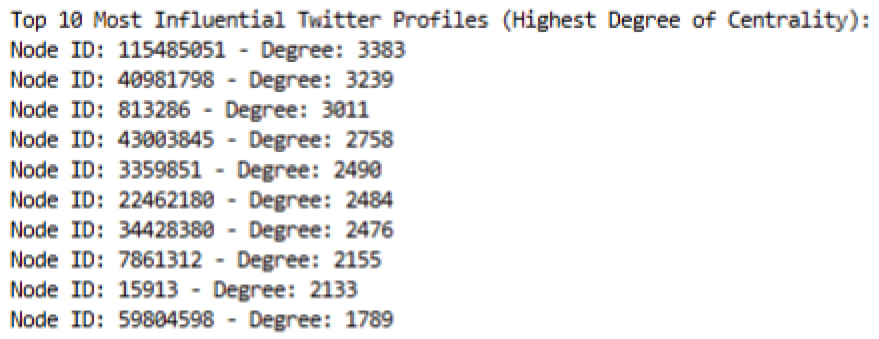
// Step 6: MOST INFLUENTIAL PROFILES. Calculate the degree centrality for high influence users (higher degree centrality means more connections / followers)

let top_10_centrality = graph::degree_centrality(&sampled_graph);

// Step 7: SORT FOR TOP 10. Sort the degree centrality values by degree in descending order (highest degree first)
let mut degree_vec: Vec<_> = top_10_centrality.iter().collect();
degree_vec.sort_by(|a, b| b.1.cmp(a.1));

// Step 8: PRINT TOP 10. Print the top 10 most influential nodes (highest degree)
println!("\nTop 10 Most Influential Twitter Profiles (Highest Degree of Centrality):");
for (node, degree) in degree_vec.iter().take(10) {
println!("Node ID: {} - Degree: {}", node, degree);
}

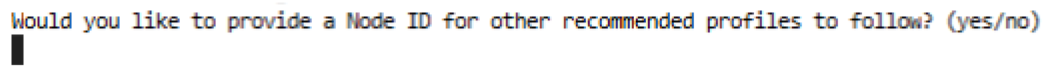
This code implements the degree centrality function defined in the graph.rs file earlier in order to calculate the most influential Twitter profiles. First, it calculates the centrality of each node in the graph, then sorts them in descending order and keeps the top 10. This displays only the top 10 most influential Twitter profiles.



```
Top 10 Most Influential Twitter Profiles (Highest Degree of Centrality):
Node ID: 115485051 - Degree: 3383
Node ID: 40981798 - Degree: 3239
Node ID: 813286 - Degree: 3011
Node ID: 43003845 - Degree: 2758
Node ID: 3359851 - Degree: 2490
Node ID: 22462180 - Degree: 2484
Node ID: 34428380 - Degree: 2476
Node ID: 7861312 - Degree: 2155
Node ID: 15913 - Degree: 2133
Node ID: 59804598 - Degree: 1789
```

// Step 9: PROFILE SUGGESTIONS. Ask user for a Node ID or select a random one
println!("Would you like to provide a Node ID for other recommended profiles to follow? (yes/no)");
let mut input = String::new();
io::stdin().read_line(&mut input).expect("Failed to read line");
let mut input = input.trim().to_lowercase();

This code prompts the user to determine if they would like to provide a specific Node ID for profile recommendations. This is useful, as discussed earlier in the function breakdown, in order to determine the most recommended profiles to follow based on the highest number of shared neighbors.

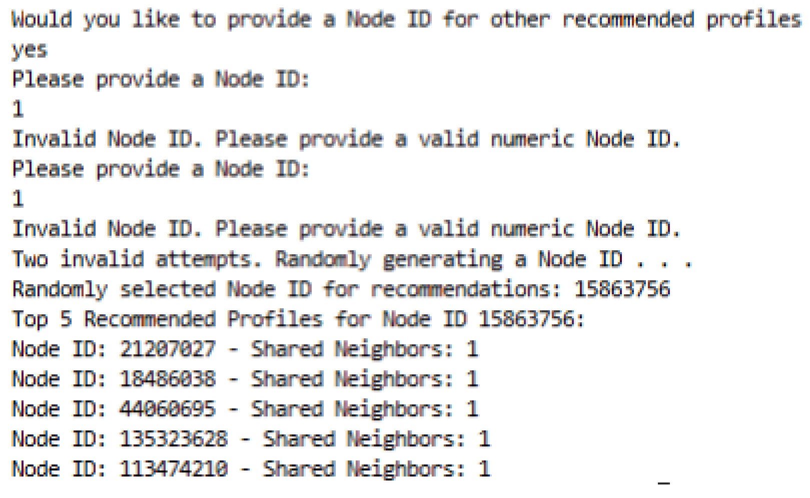


```
Would you like to provide a Node ID for other recommended profiles to follow? (yes/no)
█
```

```
    // WHAT PROFILE. Ask for the Node ID
    if input == "yes" {
        let mut attempts = 0;
        let mut valid_node_found = false;
        while attempts < 2 && !valid_node_found {
            println!("Please provide a Node ID:");
            input.clear();
            io::stdin().read_line(&mut input).expect("Failed to read line");

            // Try to parse the input as a usize
            let node_id: Option<usize> = input.trim().parse().ok();

            if let Some(id) = node_id {
                // If the user input is valid, proceed with recommendations
                if sampled_graph.contains_key(&id) {
                    // If the node exists, proceed with recommendations
                    let suggestions = graph::most_shared_neighbors(&sampled_graph,
id);

                    // TOP 5 PROFILES TO FOLLOW. Display the top 5 recommendations
                    if suggestions.is_empty() {
                        println!("Node {} has no shared neighbors in the sampled graph.",
id);
                    } else {
                        println!("Top 5 Recommended Profiles for Node {}:", id);
                        for (neighbor, shared_count) in suggestions {
                            println!("Node {}: {} shared neighbors", neighbor, shared_count);
                        }
                    }
                    valid_node_found = true;  // Valid input, stop asking
                } else {
                    // Handle invalid Node ID input. Give the user another chance to enter
valid ID (up to two chances).
                    println!("Invalid Node ID. Please provide a valid numeric Node ID.");
        }
     } else {
        println!("Invalid input. Please provide a valid numeric Node ID.");
     }
     attempts += 1;
}

  if !valid_node_found {
           // If the user failed both attempts, randomly select a Node ID.
           println!("Two invalid attempts. Randomly generating a Node ID . . .");

           // RANDOM NODE ID SAMPLING. Randomly select a node ID
           let random_node_id = {
              let mut rng = rand::thread_rng();
              let nodes: Vec<usize> = sampled_graph.keys().cloned().collect();
              *nodes.choose(&mut rng).expect("Failed to choose random
  node")
           };

           println!("Randomly selected Node ID for recommendations: {}",
  random_node_id);

           // PROFILE RECOMMENDATIONS ON SHARED NEIGHBORS (if three
  profiles I follow all follow the same account, suggest that I follow that
  account as well). Get recommendations for the random node
           let suggestions = graph::most_shared_neighbors(&sampled_graph,
  random_node_id);

           // TOP 5. Display the top 5 recommended profiles (nodes with most
  shared neighbors)
           println!("Top 5 Recommended Profiles for Node ID {}:",
  random_node_id);
           for (id, shared_neighbors) in suggestions.iter().take(5) {
              println!("Node ID: {} - Shared Neighbors: {}", id,
  shared_neighbors);
           }
        }
```

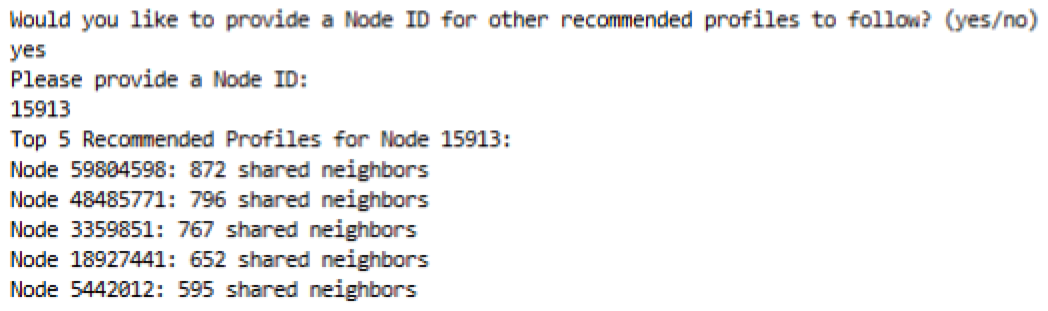
This code handles the “yes” case for profile recommendations, prompting the user to enter a specific Node ID. If the user enters a valid Node ID it displays the top 5 recommended profiles. If the user enters an invalid Node ID the program returns **"Invalid Node ID. Please provide a valid numeric Node ID."** The user then has another chance to enter a valid Node ID. If the second Node ID they enter is valid, it displays the top 5 recommended profiles to follow. If the user again inputs an invalid Node ID the program returns **"Two invalid attempts. Randomly generating a Node ID . . ."**. Then, the program randomly generates a Node ID and provides its top 5 recommended profiles to follow.

Two invalid attempts:



```
Would you like to provide a Node ID for other recommended profiles
yes
Please provide a Node ID:
1
Invalid Node ID. Please provide a valid numeric Node ID.
Please provide a Node ID:
1
Invalid Node ID. Please provide a valid numeric Node ID.
Two invalid attempts. Randomly generating a Node ID . . .
Randomly selected Node ID for recommendations: 15863756
Top 5 Recommended Profiles for Node ID 15863756:
Node ID: 21207027 - Shared Neighbors: 1
Node ID: 18486038 - Shared Neighbors: 1
Node ID: 44060695 - Shared Neighbors: 1
Node ID: 135323628 - Shared Neighbors: 1
Node ID: 113474210 - Shared Neighbors: 1
```

Valid attempt:



```
Would you like to provide a Node ID for other recommended profiles to follow? (yes/no)
yes
Please provide a Node ID:
15913
Top 5 Recommended Profiles for Node 15913:
Node 59804598: 872 shared neighbors
Node 48485771: 796 shared neighbors
Node 3359851: 767 shared neighbors
Node 18927441: 652 shared neighbors
Node 5442012: 595 shared neighbors
```

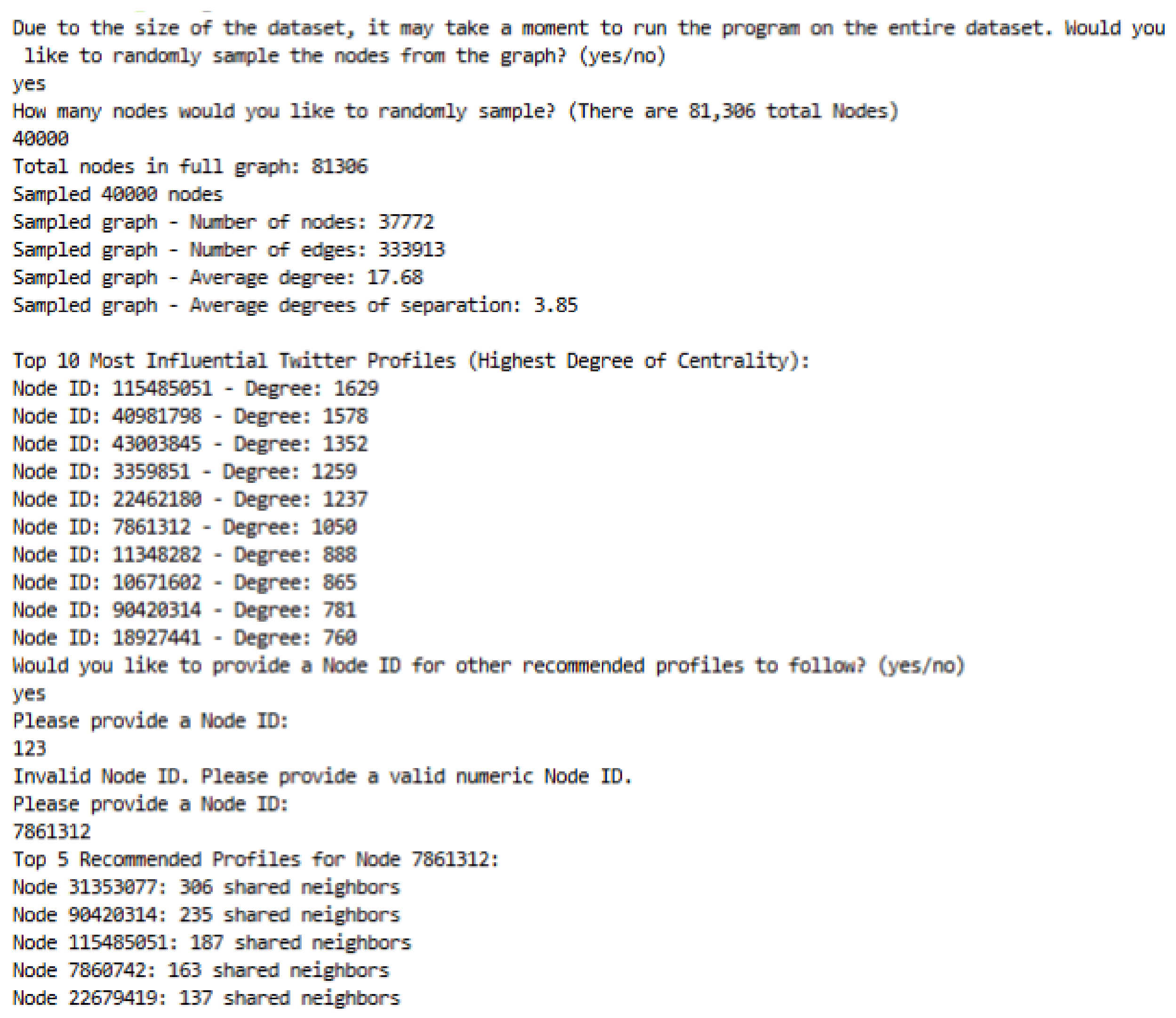
```
} else if input == "no" {
        // NO NODE ID. Handle the case where the user doesn't want to provide a
Node ID
        let random_node_id = {
            let mut rng = rand::thread_rng();
            let nodes: Vec<usize> = sampled_graph.keys().cloned().collect();
            *nodes.choose(&mut rng).expect("Failed to choose random node")
        };

        println!("Randomly selected Node ID for recommendations: {}",
random_node_id);

        // Get recommendations for the random node
        let suggestions = graph::most_shared_neighbors(&sampled_graph,
random_node_id);

        // Display the top 5 recommended profiles (nodes with most shared
neighbors)
        println!("Top 5 Recommended Profiles for Node ID {}:", random_node_id);
        for (id, shared_neighbors) in suggestions.iter().take(5) {
            println!("Node ID: {} - Shared Neighbors: {}", id, shared_neighbors);
        }
    } else {
        println!("Invalid input. Please enter 'yes' or 'no'.");
    }
}
```

This code handles the “no” case for profile recommendations. If the user enters “no”, the program randomly generates a Node ID and provides its top 5 recommended profiles to follow. This is extremely useful in connecting Twitter users, as explained in the function breakdown in the graph.rs file.

```
#[cfg(test)]
mod tests;
```

This code simply allows for testing of our functions.

Overall, a sample final output of the program might look something like this:



```
Due to the size of the dataset, it may take a moment to run the program on the entire dataset. Would you
 like to randomly sample the nodes from the graph? (yes/no)
yes
How many nodes would you like to randomly sample? (There are 81,306 total Nodes)
40000
Total nodes in full graph: 81306
Sampled 40000 nodes
Sampled graph - Number of nodes: 37772
Sampled graph - Number of edges: 333913
Sampled graph - Average degree: 17.68
Sampled graph - Average degrees of separation: 3.85

Top 10 Most Influential Twitter Profiles (Highest Degree of Centrality):
Node ID: 115485051 - Degree: 1629
Node ID: 40981798 - Degree: 1578
Node ID: 43003845 - Degree: 1352
Node ID: 3359851 - Degree: 1259
Node ID: 22462180 - Degree: 1237
Node ID: 7861312 - Degree: 1050
Node ID: 11348282 - Degree: 888
Node ID: 10671602 - Degree: 865
Node ID: 90420314 - Degree: 781
Node ID: 18927441 - Degree: 760
Would you like to provide a Node ID for other recommended profiles to follow? (yes/no)
yes
Please provide a Node ID:
123
Invalid Node ID. Please provide a valid numeric Node ID.
Please provide a Node ID:
7861312
Top 5 Recommended Profiles for Node 7861312:
Node 31353077: 306 shared neighbors
Node 90420314: 235 shared neighbors
Node 115485051: 187 shared neighbors
Node 7860742: 163 shared neighbors
Node 22679419: 137 shared neighbors
```

Analysis.rs

The code inside the analysis.rs file is not actually implemented in the output. This is because it requires iteration over every single node, and this can take a very long time for the program to run. Even though the code is not included in the output, it is useful to include and understand, as it contributes to the overall analysis of the graph.

```
use std::collections::{HashMap, HashSet};

#[allow(dead_code)]
pub fn clustering_coefficient(graph: &HashMap<usize, HashSet<usize>>) ->
HashMap<usize, f64> {
   let mut coefficients = HashMap::new();
```

```
    for (&node, neighbors) in graph {
        let mut triangles = 0;
        let mut possible_triangles = 0;

        // Check pairs of neighbors for possible triangles
        for &neighbor in neighbors {
            for &other_neighbor in neighbors {
                if neighbor != other_neighbor && graph.get(&neighbor).map_or(false, |n|
n.contains(&other_neighbor)) {
                    triangles += 1;
                }
                possible_triangles += 1;
            }
        }

        // Clustering coefficient: number of triangles / possible triangles
        if possible_triangles > 0 {
            coefficients.insert(node, triangles as f64 / possible_triangles as f64);
        } else {
            coefficients.insert(node, 0.0);
        }
    }

    coefficients
}
```

This code first imports the hashmap and hashset collections. Then, it defines the function **clustering_coefficient**, which computes the clustering coefficient for each node. This is important in measuring the connectedness of the graph because of the nature of social media platforms. Most of these platforms have hubs of tightly knit groups and many outsider nodes that are not involved in any connections. Clustering coefficients can represent the users who are participating in these “hubs”. This could be extremely useful in a future implementation of this project, as these clustering coefficients and hubs can be used to automatically determine specific communities on social media platforms pertaining to a specific topic. For example, a “Boston Sports” community which is a hub of profiles that interact with each other mainly about the Celtics or Red Sox. This would require further research and implementation of methods along with its long runtime, so it is excluded from the output, however, it is a very interesting calculation and concept that could be detailed in a future study.

```
#[allow(dead_code)]
pub fn graph_diameter(graph: &HashMap<usize, HashSet<usize>>) -> usize {
    let mut max_distance = 0;

    for &start in graph.keys() {
        let mut visited = HashSet::new();
        let mut queue = std::collections::VecDeque::new();
        let mut distances = HashMap::new();

        visited.insert(start);
        queue.push_back(start);
        distances.insert(start, 0);

        while let Some(current) = queue.pop_front() {
            let current_distance = *distances.get(&current).unwrap();
            for &neighbor in graph.get(&current).unwrap() {
                if !visited.contains(&neighbor) {
                    visited.insert(neighbor);
                    queue.push_back(neighbor);
                    distances.insert(neighbor, current_distance + 1);
                }
            }
        }

        // Get the farthest node distance from start
        if let Some(&max) = distances.values().max() {
            max_distance = max_distance.max(max);
        }
    }

    max_distance
}
```

This code implements the function **graph_diameter** which computes the diameter of a graph, in this case, the Twitter dataset. This is done by calculating the longest shortest path between any two nodes by first performing a breadth-first search from each node to calculate this path. It returns the three longest distances. This is very useful in determining just how tightly knit these “hubs” are, as it can identify profiles who might be able to bridge different hubs of profiles. A larger diameter would represent more isolated clusters.

Tests.rs

```
use std::collections::{HashMap, HashSet};
use crate::graph::{compute_avg_degrees_of_separation, sample_graph,
degree_centrality};

// Create a simple test graph
fn create_test_graph() -> HashMap<usize, HashSet<usize>> {
    let mut graph = HashMap::new();
    graph.insert(1, HashSet::from([2, 3]));
    graph.insert(2, HashSet::from([1, 3]));
    graph.insert(3, HashSet::from([1, 2, 4]));
    graph.insert(4, HashSet::from([3]));
    graph
}
```

This code imports the correct crates and uses **compute_avg_degrees_of_separation, sample_graph,** and **degree_centrality** from the graph.rs file. It creates a simple test graph to test these functions.

```
#[cfg(test)]
mod tests {
    use super::*;

    // Graph sampling
    #[test]
    fn test_sample_graph() {
        let graph = create_test_graph();
        let (sampled_graph, sampled_nodes) = sample_graph(&graph, 2);

        // Check that two nodes are sampled
        assert_eq!(sampled_nodes.len(), 2);
        // Check that the sampled graph contains only the sampled nodes
        assert!(sampled_graph.len() <= 2);
        for node in &sampled_nodes {
            assert!(graph.contains_key(node));
        }
    }
```

This code tests the functionality of the **sample_graph** function described earlier in the graph.rs file. It checks that the sampled nodes and graph size are correct, pivotal in our analysis. If the graph is not being sampled properly, the analysis will be innacurate.

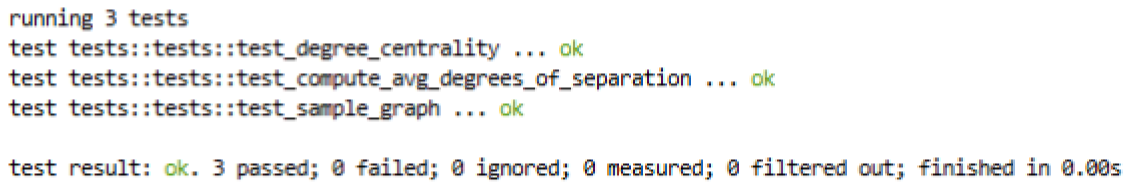
```
    // Degree centrality
    #[test]
    fn test_degree_centrality() {
        let graph = create_test_graph();
        let centrality = degree_centrality(&graph);

        // Verify the degree centrality of specific nodes
        assert_eq!(centrality[&1], 2);
        assert_eq!(centrality[&3], 3);
    }
```

This code tests the functionality of the **degree_centrality** function, assuring that the calculation is being made properly. The function is used with the test graph and the result is stored in the centrality variable. This centrality variable is then tested to make sure it is returning the right values. This is important because if not working properly, advertisers might target the wrong profiles to advertise their products, which could lead to financial losses.

```
    #[test]
    fn test_compute_avg_degrees_of_separation() {
        let graph = create_test_graph();
        let avg_separation = compute_avg_degrees_of_separation(&graph, 1);

        // Check that the average degrees of separation is within expected range
        assert!(avg_separation >= 1.0 && avg_separation <= 2.0);
    }
}
```

This code tests the functionality of the **compute_avg_degrees_of_separation** function in order to assure that it is working properly. This is a pivotal part in the analysis of the graph, as the average degrees of separation reveals the overall connectedness of Twitter as a whole, highlighting its global connectivity. If this function is not implemented properly, our analysis will be inaccurate.

All three of these tests are passed, assuring us that the program is working properly:



```
running 3 tests
test tests::tests::test_degree_centrality ... ok
test tests::tests::test_compute_avg_degrees_of_separation ... ok
test tests::tests::test_sample_graph ... ok

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

Conclusion

This project aimed to analyze and understand the structure of the “Social circles: Twitter” dataset, providing a greater understanding of how Twitter operates in connecting users as a social media platform. The average degrees of separation obtained in this project shows that Twitter is relatively efficient in connecting individuals and spreading information. This project can be used as a baseline analysis for social media platforms, and further analysis of how social networks aim to achieve their purpose of connecting individuals. This project interacts with the user, allowing them to determine their desired inputs, tailoring the program to their needs. As shown throughout this report, the project demonstrates exceptional code complexity, methodology, and has a clear, defined purpose in real-world applications. A future implication of this study might use the clustering coefficients function and the other graph analysis tools in order to automatically generate communities of profiles that share common interests.