DS210 Final Project

Facebook Graph Analysis

Dataset: LINK

Github: LINK

Hassan Dawy

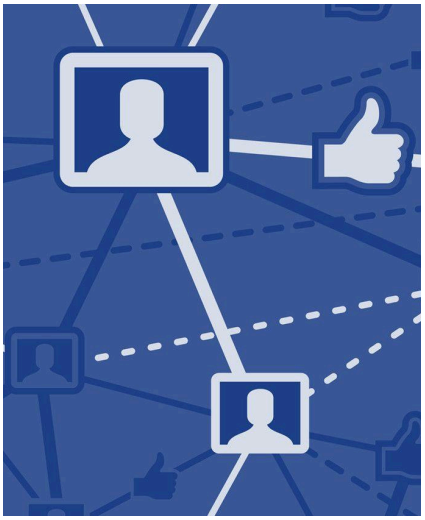
01. Overview

Project Overview: Facebook Ego Network Graph Analysis

My project explores the behavior of a real-world social network using graph theory, programmed on Rust. I will analyze a dataset from the Stanford SNAP Facebook ego networks (facebook_combined.txt.gz) to investigate how connections are formed between individuals. I will be applying multiple graph algorithms to get insights and observations about connectivity and influence patterns.

The Facebook network dataset is an undirected, and unweighted graph where each node corresponds to a user, and each edge represents a friendship. Basically, this dataset has ego networks which show each user and their immediate friends on facebook.

This allows us to easily find social dynamics. The main questions I will be answering are:



02. Question

How can we characterize the structure and relationships within a large social graph using measures of distance, centrality, and similarity?

a. What is the typical distance between nodes in the graph?
b. How are connections distributed across the network?
c. Which nodes are most central, and how do they compare to others?
d. How similar are friends of different people?

03. Solutions

Solutions to the questions in section 2 respectively

a. BFS Traversals
b. Degree Distribution
c. Closeness Centrality
d. Jaccard Similarity Index

04. Dataset

Here is the dataset I am using: LINK

As explained earlier, this dataset is undirected and unweighted and represents connections between users. As listed on the snap stanford website, there are 4039 nodes, and 88234 edges. Each node represents a user, and each edge is a neighbor. This dataset is structured in the following way:

facebook_combined.txt

```
0 1
0 2
0 3
0 4
0 5
0 6
0 7
0 8
0 9
0 10
```

Let's look at the first line of the dataset. The first entry, 0 in this example, represents the user 0. The second entry, 1 represents another user. This shows that user 0 and user 1 are friends on Facebook. And, since this network is undirected the friend users 0 and 1 have with each other is mutual. We can deduce, that from just the screenshot I attached, user

0 is friends with users 1-10. Once we're done with user 0's friends we move on to user 1's friends and so on. I also see that each line in this dataset has two integers separated by a space. This will be useful to know when I start with processing the dataset into a graph.

05. Code Breakdown

For my project, I split it up into three modules. The first one will be **graph.rs**. This file contains the core graph struct and methods for graph construction and basic degree analysis. The second module is **analysis.rs**, which implements the more complex graph algorithms and analysis, like BFS, closeness centrality etc. Lastly, we have the **main.rs** module, which serves as the entry point for our project. It loads the data, and runs the analyses so we get our results.

a. Graph.rs

I mainly handle the graph's core data structure in this file. We start off the code with:

```rust
use std::collections::{HashMap, HashSet};
use std::fs::File;
use std::io::{BufRead, BufReader};

#[derive(Debug)]
pub struct Graph {
    pub adj_list: HashMap<usize, HashSet<usize>>,
    pub num_nodes: usize,
    pub num_edges: usize,
}
```

We import the usual libraries for reading and processing the data, and we will go over how we used them as we proceed with our code breakdown. We then define our struct called *Graph* with *pub struct Graph*. The purpose of putting *pub* is to make this struct public, and visible to other files in our project. Since we are working with multiple modules, we will need to use *pub* if we want to use/refer to that struct in other files like main.rs. Within our struct we define the graph structure with an adjacency list and the number of nodes and the number of edges. The adjacency list uses a hashmap to map each node (user) to a set of its edges (neighbors).

```
impl Graph {
    pub fn new() -> Self {
        Self {
            adj_list: HashMap::new(),
            num_nodes: 0,
            num_edges: 0,
        }
    }
```

This method here creates an empty graph instance so I can later load the data in it.

```
    pub fn load_from_file(path: &str) -> Self {
        let file: File = File::open(path).expect(msg: "Failed to open graph file.");
        let reader: BufReader<File> = BufReader::new(inner: file);
        let mut graph: Graph = Graph::new();
```

The function I define here *load_from_file* is responsible for loading and parsing the Facebook dataset file. It begins by opening the file from the path I provide, and it wraps in bufreader for line-by-line reading. *Graph::new()* creates an empty graph.

```
for line: Result<String, Error> in reader.lines() {
    if let Ok(edge_line: String) = line {
        let parts: Vec<usize> = edge_line String
            .split_whitespace() SplitWhitespace<'_>
            .map(|x: &str| x.parse::<usize>().unwrap()) impl Iterator<Item = usize>
            .collect();
        if parts.len() != 2 {
            continue;
        }
```

After the empty graph is created I loop over every line of the dataset. As seen earlier, each line represents a friendship between two users. They're parsed into usize and stored in a vec. I accounted for error handling by making sure that if a line has more than two elements, then that line is skipped. This makes sure there is no dirty data that could cause us errors later on.

```
            let (u: usize, v: usize) = (parts[0], parts[1]);
            graph.adj_list.entry(key: u).or_default().insert(v);
            graph.adj_list.entry(key: v).or_default().insert(u);
            graph.num_edges += 1;
        }
    }

    graph.num_nodes = graph.adj_list.len();
    graph
}
```

Next, we add an undirected edge between users u and v (first and second user respectively) by doing *graph.adj_list.entry(u).or_default().insert(v);* and vice versa. The *or_default* initializes a new hashset if a key doesn't already exist. Also the last line in the if statement is *graph.num_edges += 1;* which as we can see adds 1 for every valid line in

the num_edges counter to count the number of edges. After processing all the lines of the dataset, we also set the number of nodes of the graph (*graph.num_nodes*) to the length of our adjacency list. *graph* just returns the instance of the graph we made.

```
    pub fn all_degrees(&self) -> Vec<(usize, usize)> {
        self.adj_list HashMap<usize, HashSet<usize>>
            .iter() Iter<'_, usize, HashSet<usize>>
            .map(|(&node: usize, neighbors: &HashSet<usize>)| (node, neighbors.len())) impl Iterator<Item
            .collect()
    }
} impl Graph
```

After loading our file, we have another function in the graph implementation which is *all_degrees*. This method simply returns a vector of tuples. Each tuple would have a node which is the user's ID and the number of friends that the node has (thats considered the degree). To do this in the function, we iterate over every key value pari in the adjacency list and then use *.map* to convert each pair into a tuple and use *.collect()* to put each tuple in a vector.

Next, I will go over analysis.rs where we will dive deeper into creating algorithms to find connections between users in the dataset. We saw a small preview of that with the last function defined in the graph implementation (*all_degrees*) which will help us find the degree distribution across the dataset.

b. Analysis.rs

```rust
use std::collections::{HashMap, HashSet, VecDeque};
use crate::graph::Graph;

pub fn average_distance(graph: &Graph) -> f64 {
    let mut total_distance: usize = 0usize;
    let mut count: usize = 0usize;
```

This is our analysis.rs module. As usual, we start our file with our imports which we will use later throughout the file. We define the function *average_distance* which initializes two variables *total_distance* and *count*. This function is meant to calculate the average shortest path length across all node pairs. I will show later on in the write-up of the main.rs file the output of these functions, but generally this helps us decide whether the theory of the six degrees of separation is true.

```
    for &start: usize in graph.adj_list.keys() {
        let distances: HashMap<usize, usize> = bfs_distances(graph, start);
        for &dist: usize in distances.values() {
            if dist > 0 {
                total_distance += dist;
                count += 1;
            }
        }
    }

    if count == 0 { 0.0 } else { total_distance as f64 / count as f64 }
}
```

This part of the function iterates through each node in the graph and runs bfs_distances (I defined that after this function). It accumulates the non zero distances (*if dist > 0*) and then at the end it returns the average path lenght by dividing the total distance by the number of valid paths. We account for error handling if the cont is 0 then it will return 0 since we can't divide by 0 (*total_distance as f64 / count as f64*).

```
pub fn bfs_distances(graph: &Graph, start: usize) -> HashMap<usize, usize> {
    let mut visited: HashSet<usize> = HashSet::new();
    let mut distance: HashMap<usize, usize> = HashMap::new();
    let mut queue: VecDeque<usize> = VecDeque::new();

    visited.insert(start);
    distance.insert(k: start, v: 0);
    queue.push_back(start);
```

Our next function is the breadth first search. This function performs a bfs from a given node. The 3 variables we initialized are visited, so we can avoid reprocessing nodes - distance which would be the shortest path from the start node to all reachable nodes - and queue is the bfs frontier. I used this source to understand some more about bfs: [link] and i also referred to the hw we had.

```
    while let Some(current: usize) = queue.pop_front() {
        let current_dist: usize = distance[&current];
        if let Some(neighbors: &HashSet<usize>) = graph.adj_list.get(&current) {
            for &neighbor: usize in neighbors {
                if !visited.contains(&neighbor) {
                    visited.insert(neighbor);
                    distance.insert(k: neighbor, v: current_dist + 1);
                    queue.push_back(neighbor);
                }
            }
        }
    }

    distance
}
```

This part of the function is a typical bfs while loop that pops the next node and adds al unvisited neighbors and then updates each neighbors distance with *current_dist + 1*. This function is useful to help us enable other metrics which we will see next (we already saw *average_distance*). It'll help us get the shortest path from one node to another, so the least amount of edges.

```rust
pub fn closeness_centrality(graph: &Graph) -> Vec<(usize, f64)> {
    let mut result: Vec<(usize, f64)> = vec![];
    for &node: usize in graph.adj_list.keys() {
        let dist: HashMap<usize, usize> = bfs_distances(graph, start: node);
        let sum: usize = dist.values().sum();
        let closeness: f64 = if sum > 0 {
            (dist.len() - 1) as f64 / sum as f64
        } else {
            0.0
        };
        result.push((node, closeness));
    }
    result.sort_by(compare: |a: &(usize, f64), b: &(usize, f64)| b.1.partial_cmp(&a.1).unwrap());
    result
}
```

Our next function is the *closeness_centrality* function. Basically what it does is that it runs *bfs_distances* to compute the shortest paths to all reachable nodes. “Closeness centrality indicates how close a node is to all other nodes in the network. It is calculated as the average of the shortest path length from the node to every other node in the network.” (source) This is different from our function above *average_distance* since *closeness_centrality* runs bfs from each node individually and for each node sums all shortest distances from that node to others. *Average_distance* runs *bfs_distances* from every node and returns the average path length across all nodes in the network. Again, we will dive deeper into this when we look at the results. Closeness centrality is computed with the following standard formula $C(u) = \frac{n-1}{\Sigma_v d(u,v)}$ (*dist.len() - 1) as f64 / sum as f64*) with *sum* being *(let sum: usize = dist.values().sum();)*.

```rust
pub fn jaccard_similarity(graph: &Graph, u: usize, v: usize) -> f64 {
    let a: Option<&HashSet<usize>> = graph.adj_list.get(&u);
    let b: Option<&HashSet<usize>> = graph.adj_list.get(&v);

    match (a, b) {
        (Some(set1: &HashSet<usize>), Some(set2: &HashSet<usize>)) => {
            let intersection: f64 = set1.intersection(set2).count() as f64;
            let union: f64 = set1.union(set2).count() as f64;
            if union == 0.0 { 0.0 } else { intersection / union }
        }
        _ => 0.0,
    }
}
```

The *jaccard_similarity* function is another one inspired by the ones you recommended to us in the project proposal. This is related to the question of “How often are friends of my friends my friends?” Jaccard similarity measures social similarity based on mutual friends. This is often used in the real world for tasks like friend recommendations. In our code, we define the function by firstly fetching the neighbors of two nodes (*let a = graph.adj_list.get(&u);* (and the same for b with …. .get(&v). Then we calculate the jaccard similarity index (inspired by this source). It follows the equation: $J(u,v) = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|}$. We account for error handling by return 0.0 if either node has no neighbors (*if union == 0.0 { 0.0 }*).

I initially didn't plan to add another function, but during my experimentation with jaccard_similarity, I felt the need to do so. My thought process is more closely looked at in Chapter 8 of this document, 'Reflection'.

```rust
pub fn most_similar_pairs(graph: &Graph, top_n: usize) -> Vec<((usize, usize), f64)> {
    let mut results: Vec<((usize, usize), f64)> = Vec::new();
    let nodes: Vec<usize> = graph.adj_list.keys().copied().collect();

    for i: usize in 0..nodes.len() {
        for j: usize in i + 1..nodes.len() {
            let u: usize = nodes[i];
            let v: usize = nodes[j];
            let neighbors_u: Option<&HashSet<usize>> = graph.adj_list.get(&u);
            let neighbors_v: Option<&HashSet<usize>> = graph.adj_list.get(&v);
            if neighbors_u.map_or(default: true, f: |n: &HashSet<usize>| n.len() <= 1) || neighbors_v.map_or(default: true, f: |n: &Has…| n.len() <= 1
                continue;
            }
            let sim: f64 = jaccard_similarity(graph, u, v);
            if sim > 0.0 {
                results.push(((u, v), sim));
            }
        }
    }

    results.sort_by(compare: |a: &((usize, usize), f64), b: &((usize,…| b.1.partial_cmp(&a.1).unwrap());
    results.truncate(len: top_n);
    results
}
```

Our function *most_similar_pairs* aims to use the *jaccard_similarity* function to find the relationships between close friend groups, and people with mutual friends. We firstly iterate through all unique node pairs in the graph. Then, we fetch their neighbors (*let neighbors_u = graph.adj_list.get(&u);* (same for v). After that, we filter out the nodes with 1 friend or less. I explain why I did that in my reflection also. After that, I finally computed the Jaccard similarity for each valid pair. (*let sim = jaccard_similarity(graph, u, v); -> if sim > 0.0 { results.push(((u, v), sim));*). We sort by similarity and then return the most similar user pairs.

06. Main.rs & Output Review

We will now be going over the main.rs file and the output we are getting. The reason we are doing this together is because the main.rs doesn't really have any new functions or algorithms, we're mainly using what we built in the other modules (analysis.rs and graph.rs) to get a result. Our general runtime for the code to run is around 20-30 seconds, which is not bad at all considering we are performing multiple algorithms on our dataset. I always run my code with *cargo run –release* command which makes it faster. According to ChatGPT it "Enables Optimizations, Disables Debugging Metadata, Performs Aggressive Compiler Tweaks". So, it basically takes a longer time to compile to enable compiler optimizations which will in turn make the runtime way slower.

```rust
mod graph;
mod analysis;
use graph::Graph;
use analysis::{average_distance, closeness_centrality, jaccard_similarity, most_similar_pairs};

```

For that case, we will be using *mod graph;* and *mod analysis;* since those are the other two files that we got code in. We will also be importing the functions we defined in the other files, from the graph.rs file we will import the Graph function, and for the analysis.rs we will import the above mentioned functions.

```rust
fn main() {
    let path = "data/facebook_combined.txt";
    let graph = Graph::load_from_file(path);
    println!("Loaded {} nodes and {} edges.", graph.num_nodes, graph.num_edges);
```

Our *fn main* starts with declaring the path to the dataset and then right after calls *Graph::load_from_file(path);* which reads the facebook dataset and constructs the adjacency list based graph. Next, we'll start with testing our analysis functions.

```rust
    println!("\nDegree Distribution:");
    for (node, degree) in graph.all_degrees().iter().take(10) {
        println!("Node {:>4}: Degree {:>3}", node, degree);
    }
    println!("             ");
```

The *all_degrees* function gives us the degree distribution of the nodes. This would be basically 10 random nodes from the dataset. This gives us an idea of a variety of users' friends and their social networks. It tells us how many friends these users have. Here is the output we get:

```
Degree Distribution:
Node  157: Degree   3
Node  961: Degree  13
Node  991: Degree  19
Node 2911: Degree  69
Node 1691: Degree  60
Node 3657: Degree  13
Node 1310: Degree  27
Node 2343: Degree  79
Node 1739: Degree  14
Node 3757: Degree  17
```

This output gives us an idea of what to expect with this sample of 10. It also helps me make sure that the code and dataset reading works just fine. For example, user 157 has 3 friends.

```rust
let avg_dist = average_distance(&graph);
println!("\nAverage Distance (Six Degrees): {:.2}", avg_dist);
println!("_____________");
```

Next, our *average_distance* function calculates the average shortest path length across the entire network. Here is the output:

```
Average Distance (Six Degrees): 3.69
```

This function was very important to test and highly relevant. It verifies the theory of the six degrees of separation. It is the idea that “all people are six or fewer social connections away from each other” (source). The average shortest path length across the entire network is 3.69 which confirms that facebook exhibits a very connected network. This means that everyone is on average 3.69 people away from everyone else in facebook. Up next is our function for closeness centrality.

```rust
println!("\nTop 5 Closeness Centrality Nodes:");
for (node, centrality) in closeness_centrality(&graph).into_iter().take(5) {
    println!("Node {:>4}: Closeness Centrality {:.4}", node, centrality);
}
```

The *closeness_centrality* function gives us the top 5 closeness centrality nodes.

```
Top 5 Closeness Centrality Nodes:
Node  107: Closeness Centrality 0.4597
Node   58: Closeness Centrality 0.3974
Node  428: Closeness Centrality 0.3948
Node  563: Closeness Centrality 0.3939
Node 1684: Closeness Centrality 0.3936
```

This function basically measures how close a node is to all other nodes in the network. So for instance, node 107 has a closeness centrality of 0.4597. That means that he is the most central user in the raph, and they’re the shortest distance away from all the other nodes on average. Implicitly, we can deduce that node 107 is friends with a lot of people in the dataset. They’re likely part of a densely interconnected network. This means that they could have a mutual friend across many different networks making him a central user. From an outsiders perspective, node 107 is also the most reachable person in the network. This means that others can get to them in the fewest steps in average. This can be important information for facebook because they can get in touch with those who have great influence in the network (people with high closeness centrality) and coordinate community building events with them. Our next function is the *jaccard_similarity*

```
println!("_____________");
let pairs = vec![(0, 1), (0, 2), (1, 3)];
println!("\nJaccard Similarities (Friends of Friends):");
for (u, v) in pairs {
    let sim = jaccard_similarity(&graph, u, v);
    println!("Nodes {} & {} → Similarity: {:.3}", u, v, sim);
}
```

Since the jaccard_similarity requires us to choose a pair of nodes to compare, i tested it out with 0 and 1, 0 and 2, and 1 and 3. It should print the social overlap between each pair, showing how similar they are in their social network.

```
Jaccard Similarities (Friends of Friends):
Nodes 0 & 1 → Similarity: 0.046
Nodes 0 & 2 → Similarity: 0.026
Nodes 1 & 3 → Similarity: 0.062
```

We had some pretty low similarity scores between these comparisons. This means that node 0 and 1 have very little similarity in their friends (not many mutual friends relative to how many friends they have). But this still doesn't mean 0, indicating that there exist some mutuals. To put the jaccard similarity function to use I was interested in what users are very similar to each other. And if there are any one to one similarities.

```
println!("_____________");
println!("\nTop Jaccard Similarities (Most Similar Friend Pairs):");
for ((u, v), sim) in most_similar_pairs(&graph, 5) {
    println!("Nodes {} & {} → Similarity: {:.3}", u, v, sim);
}
```

That's where our top jacccard similarities function comes (*most_similar_pairs*). We print out the top 5 most similar pairs.

```
Top Jaccard Similarities (Most Similar Friend Pairs):
Nodes 3147 & 2817 → Similarity: 1.000
Nodes 3854 & 3879 → Similarity: 1.000
Nodes 4006 & 4032 → Similarity: 1.000
Nodes 255 & 241 → Similarity: 1.000
Nodes 3372 & 2902 → Similarity: 1.000
```

Surprisingly, we get many cases where users have the EXACT same set of friends. This is highly beneficial for facebook to identify tight clusters of people or potential communicites. This can be very practical to suggesting friend recommendations. If a user has a high similarity with another user, then facebook could suggest friends from those users to each other. That's why I also made a lookup feature to search for exact nodes and who their friends are. For example, for node 3147, here is his inspection with this algorithm:

```
let reference = 3147; //CHECKING IF JACCARD WORKING
if let Some(friends) = graph.adj_list.get(&reference) {
    println!("Node {} has {} friends: {:?}", reference, friends.len(), friends);
```

```
Node 3147 has 5 friends: {2774, 1684, 3055, 3127, 3074}
```

```
Node 2817 has 5 friends: {3127, 1684, 2774, 3055, 3074}
```

This was helpful for me to understand and ensure that jaccard similarity was working. I would test both nodes and make sure they're the same. And they are.

07. Tests

Our tests are at the bottom of the analysis.rs and graph.rs files. I conduct a test on all the functions we defined. Starting with **graph.rs**,

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

We start with the usual basic imports for starting tests.

```
    #[test]
    fn test_all_degrees() {
        let mut graph = Graph::new();
        graph.adj_list.insert(1, HashSet::from([2, 3]));
        graph.adj_list.insert(2, HashSet::from([1]));
        graph.adj_list.insert(3, HashSet::from([1]));
        graph.num_nodes = graph.adj_list.len();
        graph.num_edges = 2;
        assert_eq!(graph.num_nodes, 3);
        assert_eq!(graph.num_edges, 2);

        let degrees = graph.all_degrees();
        assert!(degrees.contains(&(1, 2)));
        assert!(degrees.contains(&(2, 1)));
        assert!(degrees.contains(&(3, 1)));
    }
```

We then create a sample graph for this test. This is done instead of using the facebook dataset to make sure the testing is accurate, simple and efficient. This graph is a graph of 1 node that is connected to nodes 2 and 3. So this graph should have 3 nodes and 2 edges. Those are our *assert_eq!(graph.num_nodes, 3);* and

assert_eq!(graph.num_edges, 2); We are testing the function *all_degrees* here, and we are making sure the degree calculations that it does are accurate. So we also make sure that node 1 is connected to node 2, node 2 connected to node 1, and node 3 connected to node 1.

Moving forward to the **analysis.rs** tests. We start with:

```rust
#[cfg(test)]
mod tests {
    use super::*;
    use crate::graph::Graph;
    use std::collections::HashSet;
    fn small_graph() -> Graph {
        let mut graph = Graph::new();
        graph.adj_list.insert(0, HashSet::from([1, 2]));
        graph.adj_list.insert(1, HashSet::from([0, 2]));
        graph.adj_list.insert(2, HashSet::from([0, 1]));
        graph.num_nodes = 3;
        graph.num_edges = 3;
        graph
    }
```

The usual imports and test implementations. Then we create a small graph to test our analysis functions on. This graph has 3 nodes (0,1,2). 1 and 2 are connected, 0 and 2 are connected, and 0 and 1 are connected. They're all connected.

```rust
115
116     #[test]
117     fn test_bfs_distances() {
118         let graph = small_graph();
119         let distances = bfs_distances(&graph, 0);
120         assert_eq!(distances.get(&0), Some(&0));
121         assert_eq!(distances.get(&1), Some(&1));
122         assert_eq!(distances.get(&2), Some(&1));
123     }
124
```

Our first test is testing bfs distances function. We run bfs from node 0 on our triangle graph. So we are checking the bfs from each node to 0 (*assert_eq!(distances.get(&0)., Some(&0));*). Distance from 0 to 0 is 0. Distance from 1 to 0 is 1. Distance from 2 to 0 is 1.

```rust
#[test]
fn test_average_distance() {
    let graph = small_graph();
    let avg_dist = average_distance(&graph);
    assert!((avg_dist - 1.0).abs() < 0.0001);
}
```

For our next test we are testing our *average_distance* function. Here we do something a bit differently. So we're trying to compute the average shortest path length in our triangle, and the average distance should be 1.0 because every node is directly connected to the others. But because our result is a float number and not an integer, we

can't assert that the average distance is 1.0 since there may be some extremely small differences. That's why, we subtract the average distance by what we think the answer is which is -1.0: *assert!((avg_dist - **1.0**).abs() < 0.0001);* If the difference is less than 0.0001 that means the answer is pretty much equal and the test passes. We will use this logic for the next few tests too.

```
#[test]
fn test_closeness_centrality() {
    let graph = small_graph();
    let closeness = closeness_centrality(&graph);
    for &(_, centrality) in &closeness {
        assert!((centrality - 1.0).abs() < 0.0001);
    }
}
```

Here we test the *closeness_centrality* function. Each node should have the maximum possible closeness value (1.0).

```
    #[test]
    fn test_jaccard_similarity() {
        let graph = small_graph();
        let sim = jaccard_similarity(&graph, 0, 1);
        assert!((sim - (1.0 / 3.0)).abs() < 0.0001);
    }
}
```

Lastly, we have the test for the *jaccard_similarity* function. We will test the jaccard similarity between node 0 and 1. We know that they both share node 2 as a mutual friend so they share 1 out of 3 total friends. Therefore, the similarity should be 1.0 / 3.0.

To test all our functions we use *cargo test in our terminal*. Here are our results:

```
running 5 tests
test analysis::tests::test_bfs_distances ... ok
test analysis::tests::test_closeness_centrality ... ok
test analysis::tests::test_jaccard_similarity ... ok
test analysis::tests::test_average_distance ... ok
test graph::tests::test_all_degrees ... ok

test result: ok. 5 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

All passed

08. Reflection

This project has been very interesting and inspiring to me. It has changed my view on how I view social networks and the connections between them. It's always great to see what happens behind the scenes, and this has helped me understand briefly how influence and advertising could work through analyzing graphs with nodes and edges. During the process, I started with finding a dataset. My original dataset that I used for the proposal was too short so I had to change it. I used one of the recommendations you gave us and ended up going for the first link, the stanford one, and chose the facebook dataset. I started off with the graph.rs file then the analysis.rs file. While I was doing the main.rs file, I was constantly running my code to make sure it's working and that I can move on. And I to be honest, had a shorter analysis.rs file but as I saw the results in the main.rs file to the first few functions, I felt interested to learn more and explore more. I decided to add jaccard similarity because I was inspired by the friends of friends idea you mentioned in the piazza post for project ideas. When I did that, I realized that I can't really do much with the information if I had to hardcode the nodes I want compared. That's why I created the *most_similar_pairs* function which was a way to use jaccard similarity to dive deeper. When I saw the results, I was honestly shocked to see that the similarity was 1 across all 5 comparisons. I then created the lookup feature to see how many friends these nodes had, and most of them had just 1 friend, which made sense. Then, that's when I added the constraint on the most similar pairs function to give me comparisons of nodes that have more than one friend. I somehow still got the same similarity of 1.0 which was interesting. I guess there are more people than I expected with the exact same friend list. I thought a bit about it and it could be common within very close isolated families or communities/friend groups. When looking up how many friends the nodes with 1.0 similarity had, some of them reached 7 and maybe even more. One of the most difficult parts of this project to me was implementing some of the analysis functions. I had to do some research for some of them to make sure I fully understood the concept, and then I implemented from there. Some future considerations for this project would be to add charts to visualize the dataset. The reason I didn't do that was because I was so invested in finding solutions to all those different graph algorithm questions. Additionally, I wasn't very familiar with displaying visuals in Rust, but I'd love to visualize my graphs at some point in the future!