Project Write Up

Project Description:
For this final project, I chose GitHub MUSAE, a social network of GitHub users with user-level attributes, connectivity data and a binary target variable, with 37700 Instances and 4006 Features. This dataset contains information about a network of developers, allowing for the exploration of mutual follower relationships and the impact of these connections on the professional trajectory of each developer.

Project Execution:

1. I first read the file and store edges into tuples for the convenience of later analysis.
2. Then I read through each line of the data, using “,” as an indicator to signal where to split.
3. I created an adjacency list to find the shortest path by iterating over the vector of edges to populate the HashMap. For each edge, add each node to the map if it's not already there, appending its neighbor to the corresponding vector.
    a. Since it’s an undirected graph, I put in the key for node 1 if it's already there, add node 2 to its neighbors and do the inverse.
4. I use Brethfirst search to find the distance between nodes, starting with a distance of 0. For each neighbor of this node, if it hasn't been visited, add incremented distance and mark it as visited by adding it to the distance map
5. I then calculated the average distance function to find the average distance between the node and every other node, using the existing breadth_first_search function to get distances from the start node to all other nodes.
6. Calculate the average of the reachable nodes present in the distances map.
7. Then extending this method to the whole dataset by finding the average degree of separation for every single node.
8. Lastly, I used the data [(1, 2), (2, 3), (3, 1)] as a test.

Project Conclusion:
I investigated the average degree of separation is around 5.67. It means an average GitHub user is about 5 to 6 steps away from another use in the network.

This can be interpreted in the context of "Six Degrees of Separation" theory, which posits that all people are six or fewer social connections away from each other. This outcome aligns with the Six-Degree of Separation Theory, which according to Wikipedia: “... all people are six or fewer

social connections away from each other … a chain of "friend of a friend" statements can be made to connect any two people in a maximum of six steps.``

Code Snippet Appendix

```
use std::fs::File;
use std::io::{self, BufRead, BufReader};
use std::collections::{HashMap, VecDeque};

// storing edges to a vector of tupples
fn read_edges_from_file(path: &str) -> io::Result<Vec<(u32, u32)>> {
   let file = File::open(path)?;
   let reader = BufReader::new(file);
//setting up an empty vector for edges
   let mut edges = Vec::new();
//iterate through each line of the data
   for line in reader.lines() {
       let line = line?; //???
       let parts: Vec<&str> = line.split(',').collect();

       if parts.len() == 2 {
           if let (Ok(id_1), Ok(id_2)) = (parts[0].parse::<u32>(),
parts[1].parse::<u32>()) {
               edges.push((id_1, id_2));
           } else {
               eprintln!("Invalid data: {:?}", parts);
           }
       } else {
           eprintln!("Incorrect number of fields: {:?}", parts);
       }
   }

   Ok(edges)
}//check what "OK" means???

//Use adjacency list to find the shortest path
//Iterate over the vector of edges to populate the HashMap.
// For each edge, add each node to the map if it's not already there
// append its neighbor to the corresponding vector.
fn create_adjacency_list(edges: &[(u32, u32)]) -> HashMap<u32, Vec<u32>> {
   let mut adjacency_list = HashMap::new();
```

```
    for &(node1, node2) in edges {
        adjacency_list.entry(node1).or_insert_with(Vec::new).push(node2);
        adjacency_list.entry(node2).or_insert_with(Vec::new).push(node1);
    } //undirected graph: put in the key for node 1 if it's already there, add node 2
to its neighbors
    // do inverse

    adjacency_list
}
//Use Brethfirst search to find distance between nodes
//Start node with a distance of 0
//For each neighbor of this node, if it hasn't been visited,
// add incremented distance and mark it as visited by adding it to the distance map
fn breadth_first_search(start_node: u32, adjacency_list: &HashMap<u32, Vec<u32>>) ->
HashMap<u32, u32> {
    let mut distances = HashMap::new();
    let mut queue = VecDeque::new();

    distances.insert(start_node, 0); //taking starting node and giving it distance of 0
    queue.push_back(start_node);

    while let Some(node) = queue.pop_front() {
        let distance = distances[&node];

        if let Some(neighbors) = adjacency_list.get(&node) {
            for &neighbor in neighbors {
                if !distances.contains_key(&neighbor) {
                    distances.insert(neighbor, distance + 1);
                    queue.push_back(neighbor);
                }
            }
        }
    }

    distances
}

//Calc average distance function to find the average distance between the node and
very other nodes
//Use the breadth_first_search function to get distances from the start node to all
other nodes.
```

```rust
//Calc the avg
//only reachable nodes present in the distances map

fn calculate_average_distance(start_node: u32, adjacency_list: &HashMap<u32,
Vec<u32>>) -> (f64, f64) {
   let distances = breadth_first_search(start_node, adjacency_list);
   let sum: u32 = distances.values().sum();
   let count = distances.len() as f64;

   let average_distance = if count > 0.0 { sum as f64 / count } else { 0.0 };
   let average_degree_of_separation = if count > 1.0 { sum as f64 / (count - 1.0) }
else { 0.0 };

   (average_distance, average_degree_of_separation)
}

//find the average degree of separation for evrey single node
fn get_all_nodes(edges: &[(u32, u32)]) -> Vec<u32> {
   let mut nodes = edges.iter()
       .flat_map(|(a, b)| vec![*a, *b])
       .collect::<Vec<u32>>();
   nodes.sort_unstable();
   nodes.dedup();
   nodes
}



fn main() {
   let file_path = "musae_git_edges.csv";
   let edges = read_edges_from_file(file_path).expect("Failed to read the CSV file");
   let adjacency_list = create_adjacency_list(&edges);

   let nodes = get_all_nodes(&edges);
   let mut total_average_degree = 0.0;
   let mut count = 0.0;

   for node in nodes {
       let (_, average_degree_of_separation) = calculate_average_distance(node,
&adjacency_list);
       if average_degree_of_separation > 0.0 { // Ignore nodes with no connections
           total_average_degree += average_degree_of_separation;
           count += 1.0;
```

```
        }
    }

    let overall_average_degree = if count > 0.0 { total_average_degree / count } else {
0.0 };
    println!("The overall average degree of separation is: {}",
overall_average_degree);
}

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

    #[test]
    fn test_create_adjacency_list() {
        // make up a data
        let test_edges = vec![(1, 2), (2, 3), (3, 1)];

        // make adjacency list
        let mut expected_adj_list = HashMap::new();
        expected_adj_list.insert(1, vec![2, 3]);
        expected_adj_list.insert(2, vec![1, 3]);
        expected_adj_list.insert(3, vec![2, 1]);

        // populate adjacency list using the function
        let adj_list = create_adjacency_list(&test_edges);

        // match?
        assert_eq!(adj_list, expected_adj_list);
    }
}
```