Report

1. Problem Formulation

I am interested in the level of activity on social networks, analyzing node density, node distance, etc. This data can help researchers determine the user connectivity on different platforms.

1.1 Dataset

I chose the first dataset from Stanford University's social network dataset collection, the anonymized social circles from Facebook. It is an undirected graph dataset. Similarly, I analyzed another densely populated dataset representing the degree of human connections, the social network of Github developers. In these datasets, the nodes represent users, and the edges are their connections.

1.2 Research Questions and Directions

To investigate the closeness of human connections, it is necessary to investigate the average and maximum distances of fixed points in the graph dataset. It is also necessary to calculate the Clustering Coefficient of the entire graph dataset. The Clustering Coefficient measures the degree of clustering of a graph. A local clustering coefficient of a node indicates the degree to which its neighboring nodes are also interconnected. The local clustering coefficient is calculated using twice the count of triangles as the numerator and the count of triplets as the denominator, to measure the number of triangles that appear against the number of triangles that could possibly appear. A higher clustering coefficient of a node indicates a higher degree of relationship among the nodes around it, indicating a higher degree of group aggregation, with a maximum value of 1, indicating that all nodes around a node are related.
The formulation is as follows:

$$LCC = \frac{2r_v}{d_v(d_v - 1)}$$

'In graph G, the degree of vertex v is the number of edges connected to v, denoted as d_v."

2. Data Acquisition and Clean

I downloaded the social network datasets from the large network dataset section of Stanford University's website, which are not clean by default.

The `read_file` function is used to load the data, taking `file_path: &str` and `max_nodes: usize` as parameters, creating a `graph` object. I set the maximum number of vertices to 15,000, and the remaining vertices were cleaned using Rust.

In the graph class, we have defined methods like `bfs`, `avg_path_length`, etc., which will be discussed later.

The `read_file` function works by reading a text file, extracting edges into a vector, then building the graph class based on these edges. It utilizes two functions from the BFS module, `new` and `add_undirectededge`, to create the graph data structure.

```
pub fn read_file(file_path: &str, max_nodes: usize) -> io::Result<Graph> {
    let file = File::open(file_path)?;
    let reader = BufReader::new(file);

    let mut edges = Vec::new();

    for (index, line) in reader.lines().enumerate() {
        if index >= max_nodes {
            break; // stop reading file after 4000 nodes to prevent excessive
run time
        }

        let line = line?;
        let mut iter = line.split_whitespace();

        if let (Some(a_str), Some(b_str)) = (iter.next(), iter.next()) {
            if let (Ok(a), Ok(b)) = (a_str.parse(), b_str.parse()) {
                edges.push((a, b));
            } else {
                eprintln!("Error parsing edge: {}", line);
            }
        }
    }

    let num_vertices = edges.iter().map(|&(a, b)| usize::max(a,
b)).max().map_or(0, |x| x + 1);
    let mut graph = Graph::new(num_vertices);

    for &(a, b) in &edges {
        graph.add_undirected_edge(a, b);
    }

    Ok(graph)
}
```

After constructing the Graph class, I implemented the methods `new` and `add_undirectededge`. The `new` method initializes a Graph object, creating an empty adjacency list for a specified number of vertices. The `add_undirectededge` method adds an undirected edge between two vertices and updates each vertex's adjacency list accordingly."

```
impl Graph {

    pub fn new(num_vertices: usize) -> Self {
        let adjacency_list = vec![Vec::new(); num_vertices];
        Graph {adjacency_list}
    }

    pub fn add_undirected_edge(&mut self, a: usize, b: usize) {
        self.adjacency_list[a].push(b);
        self.adjacency_list[b].push(a);
    }
    ...
```

3.Model:

3.1 BFS Introduction

To investigate the average and maximum distances between nodes in an undirected graph dataset, one could use the Floyd-Warshall algorithm or Breadth-First Search (BFS). However, for larger graphs, the Floyd-Warshall algorithm can be very time-consuming. Therefore, BFS is used to compute the shortest distances from each node to all others, averaging these and finding the longest of these shortest paths.

The principle of BFS in traversing graph datasets is to start from a specified start node and explore each layer's neighboring nodes while maintaining the order of nodes visited. Here is how BFS is generally implemented:

1. Choosing a Start Node: Select a start node for BFS from the graph.
2. Initializing Data Structures: Create a queue to store nodes to be visited and a hash table to record visited nodes.
3. Enqueue and Mark the Start Node as Visited: Add the start node to the queue and mark it as visited.
4. Perform BFS: Repeat the following until the queue is empty:
    - Dequeue a node.
    - Explore each of its unvisited neighboring nodes.
    - For each unvisited neighbor, mark it as visited and enqueue it.

5. Recording Paths or Other Information (Optional): If required, paths or other relevant information can be recorded during BFS.
6. Analysis of Results: After BFS, various operations such as computing shortest paths or checking connectivity between nodes can be performed based on the results.

The core idea behind BFS traversal is to ensure that each node is explored layer by layer from the start node, which effectively finds the shortest path to every other node in the graph. Due to its level-wise exploration, BFS is particularly useful for finding shortest paths and determining connectivity. BFS can also be applied to detect cycles, perform topological sorting, and other applications.

3.2 BFS Construction

The bfs function executes breadth-first search (BFS) starting from a given vertex and returns a HashMap, which contains the distances from the starting vertex to all other vertices in the graph.

```
pub fn bfs(& self, start_vertex: usize) -> HashMap<usize, usize> {
        let mut distances = HashMap::new();
        let mut queue = VecDeque::new();
        let mut visited = vec![false; self.adjacency_list.len()];

        queue.push_back((start_vertex, 0));
        visited[start_vertex] = true;
        distances.insert(start_vertex, 0);

        while let Some((current_vertex, distance)) = queue.pop_front() {
            for &neighbor in &self.adjacency_list[current_vertex] {
                if !visited[neighbor] {
                    queue.push_back((neighbor, distance + 1));
                    visited[neighbor] = true;
                    distances.insert(neighbor, distance + 1);
                }
            }
        }

        distances
    }
```

Subsequently, the `avg_path_length` function calculates the average distance between all pairs of nodes. This function uses the HashMap generated by the BFS function to calculate this average, reflecting the interconnectedness of the network.

```
pub fn avg_path_length(&self) -> f64 {
        let mut total_length = 0.0;
        let mut num_paths = 0;

        for start_vertex in 0..self.adjacency_list.len() {
            let distances = self.bfs(start_vertex);

            for &distance in distances.values() {
                total_length += distance as f64;
                num_paths += 1;
            }
        }

        if num_paths > 0 {
            total_length / num_paths as f64
        } else {
            0.0
        }
    }
```

The `bfs_max_distance` function calculates the maximum distance between the starting node and all other nodes in the graph, which helps to identify the most distant node pair. The `furthest_points` function finds the maximum distance between any two nodes in the graph, representing the network's diameter.

```rust
pub fn bfs_max_distance(&self, start_vertex: usize) -> usize {
        let mut max_distance = 0;
        let mut queue = VecDeque::new();
        let mut visited = vec![false; self.adjacency_list.len()];
        let mut distances = vec![0; self.adjacency_list.len()];

        queue.push_back(start_vertex);
        visited[start_vertex] = true;

        while let Some(current_vertex) = queue.pop_front() {
            for &neighbor in &self.adjacency_list[current_vertex] {
                if !visited[neighbor] {
                    queue.push_back(neighbor);
                    visited[neighbor] = true;
                    distances[neighbor] = distances[current_vertex] + 1;
                    max_distance = max_distance.max(distances[neighbor]);
                }
            }
        }

        max_distance
    }
pub fn furthest_points(&self) -> usize {
        let mut dist = 0;

        for start_node in 0..self.adjacency_list.len() {
```

```rust
            let individual_dist = self.bfs_max_distance(start_node);
            dist = dist.max(individual_dist);
        }

        dist
    }
```

The `local_clustering_coefficient` function calculates the clustering coefficient for a specific node. In a social network, this metric helps identify nodes that form a tight-knit community. A high local clustering coefficient indicates that a node's neighbors are also connected to each other, forming a closely-knit subgroup within the network.

```
pub fn local_clustering_coefficient(&self, node: usize) -> f64 {
        let neighbors: HashSet<usize> =
self.adjacency_list[node].iter().cloned().collect();
        let num_neighbors = neighbors.len();

        if num_neighbors < 2 {
            return 0.0;
        }

        let mut connected_neighbors = 0;

        for &neighbor1 in &neighbors {
            for &neighbor2 in &neighbors {
                if neighbor1 != neighbor2 && self.check_neighbors(neighbor1,
neighbor2) {
                    connected_neighbors += 1;
                }
            }
        }

        let clustering_coefficient = connected_neighbors as f64 / (num_neighbors
* (num_neighbors - 1)) as f64;

        clustering_coefficient
    }
```

Finally, the `local_clustering_coefficient` function measures the degree of local clustering for each node in the network.

```
 fn check_neighbors(&self, node1: usize, node2: usize) -> bool {
        self.adjacency_list[node1].contains(&node2) ||
self.adjacency_list[node2].contains(&node1)
```

3.3 Results

Running the program with `cargo run`, we analyzed three parameters for our datasets: one for Facebook users and one for Github users. The three parameters measured were the average distance within the graph dataset, the distance to the farthest node, and the network density."

```
Average distance between two nodes for Facebook: 3.354
Minimum distance between the two furthest nodes: 8
Average local clustering coefficient: 0.291

Average distance between two nodes for GitHub: 4.448
Minimum distance between the two furthest nodes: 12
Average local clustering coefficient: 0.004
```

4. Preliminary Conclusion

The analysis results can show that the average distance within the Facebook network is 1 less than in the GitHub network, and the distance between the two farthest nodes is 4 less in Facebook compared to GitHub. In terms of network density, Facebook's network density is significantly higher than GitHub's. GitHub appears sparsely connected, with many users having little to no connections with others, which indicates minimal contribution to the GitHub community. This suggests that GitHub might need to employ more strategies to actively engage more beginners in various projects.

5. test for model

```
#[test]
    fn test_bfs() {
        let graph = Graph {
            adjacency_list: vec![
                vec![1, 2], // interpretation: node 0 is connected to nodes 1
and 2
                vec![0, 2, 5],
                vec![0, 1, 3, 4],
                vec![2, 4],
                vec![2, 3],
                vec![1],
            ],
        };

        // check distances from node 0
        let distances = graph.bfs(0);
        assert_eq!(distances[&0], 0);
        assert_eq!(distances[&1], 1);
        assert_eq!(distances[&2], 1);
        assert_eq!(distances[&3], 2);
        assert_eq!(distances[&4], 2);
        assert_eq!(distances[&5], 2);
    }
```

After implementing the bfs function, this function runs for each node, performing a bfs for each, ensuring that the bfs function is not only applied to a single node, but all nodes in the network are included in the analysis.

```
#[test]
    fn test_avg_path_length() {
        let graph = Graph {
            adjacency_list: vec![
                vec![1, 2],
                vec![0, 2, 5],
                vec![0, 1, 3, 4],
                vec![2, 4],
                vec![2, 3],
```

```
                vec![1],
            ],
        };

        let avg_length = graph.avg_path_length();
        assert_eq!(avg_length, (50.0 / 36.0)); // Sum of the distances between
each pair of nodes / the total number of paths
    }

    #[test]
    fn test_furthest_points() {
        let graph = Graph {
            adjacency_list: vec![
                vec![1, 2],
                vec![0, 2, 5],
                vec![0, 1, 3, 4],
                vec![2, 4],
                vec![2, 3],
                vec![1]
            ],
        };

        let max_distance = graph.furthest_points();
        assert_eq!(max_distance, 3);
    }
```

The `test_avg_path_length` function tests the average path length between nodes. This function calls `avg_path_length` to ensure it accurately calculates the average path length between all node pairs in the graph. The `test_furthest_points` function tests the maximum distance between any two nodes. This function calls `furthest_points` to ensure it correctly finds the maximum distance between any two nodes in the graph. Both tests are run using `cargo test`, ensuring all network analysis functions are correctly implemented.

```
running 3 tests
test test_furthest_points ... ok
test test_bfs ... ok
test test_avg_path_length ... ok

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