Saif Alqubaisi

Graph Analysis with Breadth-First Search, Degree Distribution, and Clustering Coefficients

Introduction:

In today's interconnected world, networks play a crucial role in various domains, from social media platforms to biological systems and transportation networks. Analyzing these networks provides valuable insights into their structure, behavior, and optimization opportunities. This project focuses on studying a social network graph derived from Facebook data to uncover its structural properties and key metrics.

The dataset used for this analysis represents friendships as undirected edges, making it an ideal candidate for testing graph algorithms on real-world data. By examining key graph properties, such as degree distribution, clustering coefficients, and connectivity through breadth-first search (BFS), the project sheds light on phenomena like the small-world effect and community structures. These insights have practical applications in improving network resilience, understanding information flow, and designing targeted marketing strategies.

Using an adjacency list for graph representation ensures computational efficiency and scalability for handling large datasets. Rust, chosen for its performance and safety, further enhances the reliability and speed of the analysis. This writeup outlines the methodologies, challenges, results, and broader implications of this project.

Project Goals:

The primary goals of the project were:

1. **Data Preparation**: Load and clean the dataset to ensure accurate representation of the graph.
2. **Graph Representation**: Construct the graph in a way that supports efficient computations.
3. **Graph Analysis**: Perform the following analyses on the graph:
    - Degree distribution to understand node connectivity.
    - Shortest path calculations using Breadth-First Search (BFS).
    - Clustering coefficient calculations to evaluate local connectivity.
4. **Code Quality**: Ensure modular, maintainable, and well-tested code to meet academic and professional standards.

Methodology:

Data Preparation:

The project uses the facebook_combined.txt dataset, representing undirected edges between Facebook users. The dataset was cleaned by removing duplicate edges and ensuring a consistent representation of nodes. This cleaning process was critical to avoid biases in degree distribution and clustering coefficient calculations.

Graph Representation:

The graph is represented using an adjacency list due to its efficiency in space and traversal operations, especially for sparse graphs. This choice ensures that operations like breadth-first search (BFS) and degree calculations remain computationally efficient. Each node's neighbors are stored in a hash set, enabling constant-time lookups and updates.

Algorithmic Analysis:

1. **Breadth-First Search (BFS)**: BFS was employed to calculate the shortest paths from a starting node to all other nodes. This method helps measure the network's connectivity and identify distances between nodes.
2. **Clustering Coefficient**: The clustering coefficient quantifies the tendency of nodes to form tightly connected communities. This was computed for individual nodes and averaged for the entire graph to understand local and global clustering tendencies.
3. **Degree Distribution**: The degree of each node, representing the number of connections it has, was calculated. Statistical analysis of the degree distribution, including minimum, maximum, and average degrees, provides insights into the network's topology.
4. **Testing**: Rigorous unit tests were implemented to validate graph operations, such as edge addition, BFS traversal, and clustering coefficient calculations. These tests ensured code reliability and accurate results.

Programming Environment:

The project was developed in Rust, chosen for its performance, safety, and rich ecosystem for handling large datasets. The code adheres to Rust's best practices, ensuring memory safety and concurrency where applicable.

Challenges Faced:

A major challenge was ensuring the portability of the dataset across different environments. Initially, the program relied on absolute paths, which limited flexibility. By implementing a path-checking mechanism, the program now locates the dataset dynamically, improving usability and ensuring the project can run seamlessly on other systems.

1. **Data Cleaning**: The dataset included duplicate edges and inconsistencies, which required implementing a robust cleaning mechanism. A hash set was used to normalize and remove redundant edges efficiently.
2. **Portability**: Ensuring the dataset path was correctly resolved across different environments presented a challenge. This was addressed by implementing a path-checking mechanism that dynamically selects the appropriate dataset location.
3. **Algorithmic Optimization**: Balancing computational efficiency with accuracy was critical, particularly for clustering coefficient calculations. Iterative testing and profiling were conducted to optimize these algorithms.
4. **Testing Robustness**: Writing meaningful unit tests to cover all edge cases required significant effort. For example, testing edge cases for graphs with no nodes or highly connected nodes highlighted potential edge conditions.

Why Adjacency Lists for Sparse Graphs?

The adjacency list representation was chosen for its efficiency, especially when dealing with sparse graphs like the Facebook dataset. Sparse graphs are characterized by having far fewer edges than the maximum possible number of edges. In such cases:

- **Memory Efficiency**: Adjacency lists use memory proportional to the number of edges (O(V + E)), whereas adjacency matrices require O(V^2) memory regardless of edge count. This makes adjacency lists far more space-efficient.
- **Traversal Speed**: Adjacency lists allow for quick traversal of a node's neighbors, as only connected nodes are stored. In contrast, adjacency matrices require scanning entire rows to identify neighbors, which is less efficient.
- **Flexibility**: Adjacency lists can easily adapt to changes, such as adding or removing edges, with minimal computational overhead compared to matrices.

In this project, where the graph has thousands of nodes but a relatively smaller number of edges, an adjacency list ensures both low memory usage and fast execution for graph algorithms like BFS and clustering coefficient calculations.

Testing for Accuracy and Reliability

Rigorous unit tests were implemented to ensure the correctness of the graph algorithms and data handling methods. These tests were designed to cover edge cases and verify expected behaviors:

- **Breadth-First Search (BFS)**: Tests validated that BFS correctly calculates shortest paths from a starting node to all other nodes in small, known graphs. For instance, a manually constructed graph was used to confirm that distances were calculated accurately.
- **Clustering Coefficients**: Tests ensured that the clustering coefficient algorithm produced correct results for various graph configurations, including fully connected nodes (expected coefficient = 1.0) and isolated nodes (expected coefficient = 0.0).

- **Data Cleaning**: Tests confirmed that duplicate and unordered edges were removed during the cleaning process. For example, input datasets with duplicate edges were checked to ensure only unique edges remained.
- **Degree Statistics**: Calculations of minimum, maximum, and average degree were tested against small graphs with known properties to verify accuracy.

These tests not only validated the functionality of individual components but also ensured the robustness and reliability of the overall implementation.

Results:

The analysis of the dataset yielded several key insights into the structure and properties of the social network graph.

1. **Degree Distribution**: The degree distribution revealed a wide variance in node connectivity, with the minimum degree being 1 and the maximum degree reaching 1045. This significant disparity indicates the presence of "hub" nodes—nodes that act as critical connectors within the network. These hub nodes have high degrees and likely represent users who are highly connected within the social graph, such as influencers or central figures in communities. The average degree of approximately 43.69 suggests that, on average, nodes in the network maintain a moderate number of connections, but the distribution is highly skewed due to the presence of these hubs.
2. **Clustering Coefficients**: The clustering coefficient analysis provided insights into the community structure of the network. The clustering coefficient for node 0 was calculated to be 0.0420, indicating a low tendency for its immediate neighbors to form connections with one another. However, the average clustering coefficient across the entire graph was 0.6055, suggesting a strong overall community formation tendency in the network. This aligns with the idea that social networks often exhibit clusters of tightly-knit groups.
3. **Breadth-First Search (BFS)**: The BFS traversal demonstrated the ability to calculate distances between nodes in the graph efficiently. For example, starting from node 0, distances to other nodes varied widely, with nodes located as close as 1 hop away and others requiring up to 5 hops. This highlights the small-world property often observed in social networks, where most nodes are relatively close to each other through a few connections.
4. **Insights**:
    - The presence of hub nodes emphasizes the network's robustness, where communication or connectivity can still be maintained even if smaller nodes are removed.
    - A high clustering coefficient reflects the existence of tightly-knit groups or communities, which is a characteristic feature of real-world social networks. These groups could represent specific interest-based communities or friend circles.
    - The small-world effect, as evidenced by the BFS results, aligns with the notion that any two users in a social network are likely connected by a relatively short path, further demonstrating the efficiency of information flow within the network.

Clustering Coefficient and Community Structures

The high average clustering coefficient (0.6055) aligns with established studies of social networks, which often exhibit strong community structures. This indicates that nodes (users) tend to form tightly-knit groups, such as friend circles or interest-based communities. These clusters play a critical role in the propagation of information, as ideas or messages can spread rapidly within a tightly-connected subgroup before reaching the broader network. The low clustering coefficient of node 0 (0.0420), on the other hand, suggests that some nodes act as bridges, connecting disparate communities with limited internal connections.

Degree Distribution and Scale-Free Networks

The observed degree distribution demonstrates a significant skew, with most nodes having a low degree and a few nodes (hubs) exhibiting extremely high connectivity, such as the maximum degree of 1,045. This behavior is consistent with "power law" distributions seen in many real-world networks, often referred to as "scale-free" networks. Such networks are robust against random failures, as the majority of nodes have minimal connectivity, while targeted removal of hub nodes could lead to network fragmentation. This characteristic has significant implications for understanding resilience and vulnerability in social networks, such as the potential for disrupting misinformation campaigns by targeting influential nodes.

Small-World Effect

The BFS results revealed that most nodes are separated by only a few hops, typically between 1 to 5 edges. This reflects the small-world property commonly observed in social networks, where any two nodes are likely to be connected through a short path. This phenomenon enables efficient communication and rapid dissemination of information across the network, emphasizing the interconnected nature of the graph.

Overall, the results showcase the complex yet structured nature of the social network graph, highlighting the importance of key nodes and the prevalence of community formations within the network.

Dataset Handling

The dataset, facebook_combined.txt, represents an undirected social network graph where nodes correspond to users, and edges represent friendships between them. This dataset is ideal for testing graph algorithms because of its high density and scale. By cleaning duplicate edges and ensuring consistent formatting, we maintained the integrity of the data while preparing it for analysis. Social network datasets like this are critical for studying phenomena such as information diffusion, community detection, and the spread of influence. This specific dataset reflects real-world relationships, making it a valuable resource for network analysis.

The dataset consists of edges representing connections between nodes. For efficient processing:

- Loading: The dataset is read line-by-line into a vector of tuples.
- Cleaning: Duplicate and unordered edges are removed to maintain graph consistency.

Relevant code:

```rust
/// Load the dataset from a file and return edges as tuples of (source, target)
pub fn load_dataset(file_path: &str) -> Result<Vec<(String, String)>, Box<dyn Error>> {
    // Open the file at the given path
    let file: File = File::open(file_path)?;
    let reader: BufReader<File> = BufReader::new(inner: file); // Wrap the file in a buffered reader for efficient reading

    let mut edges: Vec<(String, String)> = Vec::new(); // Create a vector to store the edges

    // Iterate over each line in the file
    for line: Result<String, Error> in reader.lines() {
        let line: String = line?; // Unwrap the line or return an error

        // Split the                Type:        usize
                                    Coerced to: &usize
        let nodes: Ve                                  whitespace().collect();
        if nodes.len() == 2 {
            // If there are exactly two parts, treat them as an edge
            edges.push((nodes[0].to_string(), nodes[1].to_string()));
        }
    }

    Ok(edges) // Return the vector of edges
}
```

```rust
/// Clean the dataset by removing duplicate edges
pub fn clean_dataset(edges: Vec<(String, String)>) -> Vec<(String, String)> {
    let mut seen: HashSet<(String, String)> = std::collections::HashSet::new();
    let mut unique_edges: Vec<(String, String)> = Vec::new();

    for (a: String, b: String) in edges {
        // Ensure edges are stored in a consistent order (smallest first)
        let edge: (String, String) = if a < b { (a.clone(), b.clone()) } else { (b.clone(), a.clone()) };
        if seen.insert(edge.clone()) {
            unique_edges.push(edge);
        }
    }

    unique_edges
}
```

Graph Representation

Social network datasets like this are critical for studying phenomena such as information diffusion, community detection, and the spread of influence. This specific dataset reflects real-world relationships, making it a valuable resource for network analysis. An adjacency list was selected as the data structure for graph representation in this project because of its efficiency in memory usage and flexibility in handling large, sparse graphs. Unlike adjacency matrices, which require O(V^2) memory regardless of the number of edges, adjacency lists scale linearly with the number of edges, making them more space-efficient for graphs like the Facebook dataset. Additionally, adjacency lists allow efficient traversal of a node's neighbors, which is crucial for algorithms like BFS and clustering coefficient calculations implemented in this project.

A graph is implemented as an adjacency list:

- **Nodes**: Stored as keys in a HashMap.

- **Edges**: Stored as HashSets, ensuring quick lookups and no duplicate connections.

This structure supports efficient graph traversal and querying.

Relevant code:

```rust
// Define the Graph structure
4 implementations
pub struct Graph {
    // Adjacency list to store nodes and their neighbors
    pub adjacency_list: HashMap<String, HashSet<String>>,
}

impl Graph {
    /// Create a new, empty graph
    pub fn new() -> Self {
        Graph {
            adjacency_list: HashMap::new(),
        }
    }

    /// Add an edge to the graph
    /// Since this is an undirected graph, add both directions
    pub fn add_edge(&mut self, source: String, target: String) {
        // Insert the target node into the source node's neighbor list
        self.adjacency_list HashMap<String, HashSet<String>>
            .entry(key: source.clone()) Entry<'_, String, HashSet<…>>
            .or_insert_with(default: HashSet::new) &mut HashSet<String>
            .insert(target.clone());

        // Insert the source node into the target node's neighbor list
        self.adjacency_list HashMap<String, HashSet<String>>
            .entry(key: target) Entry<'_, String, HashSet<…>>
            .or_insert_with(default: HashSet::new) &mut HashSet<String>
            .insert(source);
    }
```

Graph Analysis Algorithms

Degree Distribution

The degree of a node is the number of connections it has. This metric helps identify influential nodes and the overall network structure.

Metrics:

- **Minimum degree**: Nodes with the least connections.
- **Maximum degree**: Most connected nodes (potential hubs).
- **Average degree**: Overall connectivity.

Relevant code:

```rust
impl Graph {
    /// Calculate the degree of each node and return as a HashMap
    pub fn degree_distribution(&self) -> HashMap<String, usize> {
        let mut degree_map: HashMap<String, usize> = HashMap::new();

        for (node: &String, neighbors: &HashSet<String>) in &self.adjacency_list {
            degree_map.insert(k: node.clone(), v: neighbors.len());
        }

        degree_map
    }

    /// Calculate the min, max, and average degree
    pub fn degree_statistics(&self) -> (usize, usize, f64) {
        let degree_map: HashMap<String, usize> = self.degree_distribution();
        let mut min_degree: usize = usize::MAX;
        let mut max_degree: usize = usize::MIN;
        let mut total_degree: usize = 0;

        for &degree: usize in degree_map.values() {
            if degree < min_degree {
                min_degree = degree;
            }
            if degree > max_degree {
                max_degree = degree;
            }
            total_degree += degree;
        }

        let average_degree: f64 = total_degree as f64 / degree_map.len() as f64;

        (min_degree, max_degree, average_degree)
    }
} impl Graph
```

Breadth-First Search (BFS)

BFS explores all reachable nodes from a given starting node, calculating distances to each node. This is crucial for understanding network reachability.

Example output:

Distances from node 0 to the first 10 nodes:

Node: 1082, Distance: 2

Node: 3961, Distance: 4

Node: 106, Distance: 1

```rust
impl Graph {
    /// Perform BFS from a given start node
    pub fn bfs(&self, start: &String) -> HashMap<String, usize> {
        use std::collections::{HashMap, VecDeque};

        let mut distances: HashMap<String, usize> = HashMap::new();
        let mut queue: VecDeque<String> = VecDeque::new();

        distances.insert(k: start.clone(), v: 0);
        queue.push_back(start.clone());

        while let Some(current: String) = queue.pop_front() {
            if let Some(neighbors: &HashSet<String>) = self.adjacency_list.get(&current) {
                for neighbor: &String in neighbors {
                    if !distances.contains_key(neighbor) {
                        distances.insert(k: neighbor.clone(), v: distances[&current] + 1);
                        queue.push_back(neighbor.clone());
                    }
                }
            }
        }

        distances
    }
} impl Graph
```

Clustering Coefficient

The clustering coefficient measures how likely a node's neighbors are to connect, indicating tightly-knit communities.

Metrics:

- **Node-specific coefficient**: Proportion of actual edges to possible edges among a node’s neighbors.
- **Graph-wide coefficient**: Average of all node-specific coefficients.

Relevant Code:

```rust
impl Graph {
    /// Calculate the clustering coefficient for a specific node
    pub fn clustering_coefficient(&self, node: &String) -> f64 {
        // Get neighbors of the node
        if let Some(neighbors: &HashSet<String>) = self.adjacency_list.get(node) {
            let neighbors: Vec<&String> = neighbors.iter().collect();
            let neighbor_count: usize = neighbors.len();

            // If the node has less than 2 neighbors, clustering coefficient is 0
            if neighbor_count < 2 {
                return 0.0;
            }

            // Count the edges between neighbors
            let mut edges_between_neighbors: i32 = 0;
            for i: usize in 0..neighbor_count {
                for j: usize in (i + 1)..neighbor_count {
                    if let Some(neighbor_edges: &HashSet<String>) = self.adjacency_list.get(neighbors[i]) {
                        if neighbor_edges.contains(neighbors[j]) {
                            edges_between_neighbors += 1;
                        }
                    }
                }
            }

            // Total possible edges between neighbors
            let possible_edges: usize = neighbor_count * (neighbor_count - 1) / 2;

            // Clustering coefficient
            return edges_between_neighbors as f64 / possible_edges as f64;
        }

        // Return 0 if the node does not exist
        0.0
    } fn clustering_coefficient
```

What the Code Does

This project implements a comprehensive graph analysis pipeline using the Rust programming language. The primary goal is to analyze a social network graph derived from Facebook data, revealing insights into its structural properties. Below is an explanation of the key components of the code and their functionalities:

Graph Representation

The graph is represented using an adjacency list, implemented as a HashMap where each node is a key, and its value is a HashSet of its neighbors. This choice ensures memory efficiency and allows for fast traversal of nodes. It is particularly suitable for sparse graphs, like social networks, where most nodes are not fully connected.

Dataset Handling

The dataset is loaded using the load_dataset function, which reads the facebook_combined.txt file and parses each line into edges. To ensure accuracy, the clean_dataset function removes duplicate edges and standardizes edge representation by storing node pairs in a consistent order. This preprocessing step avoids biases in subsequent calculations, such as degree distribution and clustering coefficients.

Degree Distribution Analysis

The degree_distribution function computes the degree of each node (i.e., the number of connections it has). The degree_statistics function further summarizes this data by calculating the minimum, maximum, and average degree across all nodes. These metrics provide insights into the network's topology, including the presence of "hub" nodes with significantly higher degrees.

Breadth-First Search (BFS)

BFS is implemented to calculate the shortest paths from a starting node to all other nodes in the graph. This traversal helps uncover the "small-world" property of social networks, where most nodes are separated by a small number of hops. The BFS output includes distances from the starting node to the first 10 nodes, illustrating the network's reachability.

Clustering Coefficient Calculations

The clustering coefficient is computed to quantify the likelihood of a node's neighbors being interconnected. The clustering_coefficient function calculates this metric for individual nodes, while the average_clustering_coefficient function computes it for the entire graph. These metrics provide insights into the formation of tightly-knit communities within the network.

Testing and Validation

Unit tests are implemented for all major functions to ensure the correctness and reliability of the code. These tests cover key graph operations, such as edge addition, BFS traversal, and clustering coefficient calculations. By handling edge cases, such as graphs with no nodes or highly connected nodes, the tests validate the robustness of the implementation.

Error Handling and Portability

To make the project portable, the code dynamically checks for the dataset's location in either the root directory or the src folder. This flexibility ensures that the program runs seamlessly on different systems without requiring manual path adjustments.

Summary

This project combines data preprocessing, graph representation, and algorithmic analysis to provide a detailed exploration of a social network graph. The code is modular, efficient, and robust, supporting large-scale graph processing and yielding meaningful insights into the network's structural properties. By leveraging Rust's features, the implementation achieves a balance between computational efficiency and code safety, making it a reliable tool for graph analysis.

Usage Instructions:

1. Ensure that Rust is installed

2. Clone the project repository and navigate to the project directory.

3. Ensure the dataset file, facebook_combined.txt, is located in either the root project folder or the src directory.

4. Compile the project using:

“cargo build”

5. Run the project with:

“cargo run”

The program will load the dataset, clean it, construct the graph, and perform the analyses described above. Outputs such as degree distribution, clustering coefficients, and BFS distances will be displayed in the terminal.

Conclusion:

This project successfully analyzed the structure and properties of a social network graph using efficient graph representations and algorithms. By employing an adjacency list to represent the network, we achieved an optimal balance between memory usage and computational efficiency, enabling the analysis of large datasets like the Facebook social graph. The implementation of degree distribution analysis, clustering coefficient calculations, and breadth-first search (BFS) traversal provided critical insights into the network's characteristics, including the prevalence of hub nodes, community structures, and the small-world effect.

These findings have broader implications for understanding real-world networks. For instance, the identification of hub nodes could be vital for optimizing information dissemination or improving network resilience. Similarly, the observed clustering tendencies align with the expectation that real-world social networks are highly community-driven.

Limitations and Future Directions

1. **Assumptions about the Dataset:**
    - This analysis assumed undirected edges and unweighted connections, simplifying the graph structure but potentially overlooking nuances in user interactions.
    - The dataset's representativeness of real-world social networks was not critically evaluated, which could affect the generalizability of the results.
2. **Static Graph Limitation:**

- The analysis was performed on a static snapshot of the social network. Future work could extend this approach to dynamic graphs, capturing temporal changes in relationships and interactions.

3. **Potential Enhancements:**
    - Incorporating weighted edges or additional metadata (e.g., interaction frequency or relationship strength) could enrich the analysis.
    - Exploring alternative algorithms or parallelized approaches could improve scalability for even larger datasets.

Broader Impact

The methodologies and insights gained from this project have applications beyond social network analysis. The techniques employed here could be adapted for studying biological networks, transportation systems, and communication networks, among others. By extending the implementation to weighted or directed graphs, researchers can address more complex problems, such as predicting network growth or identifying influential nodes in various domains.

In conclusion, this project demonstrates the power of graph theory and algorithmic approaches to uncover hidden structures and relationships within complex networks. It lays the foundation for further exploration of dynamic and heterogeneous networks, contributing to the broader field of network science.