Derrick Cnun
DS 210
Professor Chator
December 10, 2024

Final Project: Crypto Network Analysis

Main Hypothesis: Does higher trust lead to higher transactions?

Why I'm interested/motivation exploring trust dynamics in the in the Bitcoin network (Response for Project Proposal Questions):

As I experienced the pandemic and various economic situations that fluctuated stocks and Bitcoin, I started to wonder about the value of money. I realized that the value of money continues to decrease as time passes. Today's dollars are not worth as much as a dollar a few decades ago. The economic situation is unpredictable, and thus, many people started investing in stocks and Bitcoin.

The Bitcoin system is unique and separated from the government's control. The infrastructure is encrypted for reliability reasons. As an aspiring student who is interested in investment in stocks and Bitcoin, as well as company acquire and mergers; hoping to land a job role in investment banking or strategy consulting, I'm eager to leverage my programming skills learned in this course to use the R programming and coding algorithms to figure out the relationship between trust levels of users and the number of transactions they have in the Bitcoin network.

Unlike other financial datasets that usually record the number of transactions and their amount, this dataset has a social aspect of cryptocurrency trading. The dataset provides information about how users' trust in the Bitcoin network correlates to their willingness to do transactions, which ultimately determines network efficiency and provides insights to prevent fraud transactions in the Bitcoin network.

For this project, my goal is to investigate my main hypothesis: "Does higher trust lead to higher transactions?" The two graphs, a histogram and a scatter plot, that are debris from the Bitcoin Alpha dataset, will be analyzed in detail. The nodes will represent the users, and the edges will represent the trust scores assigned to every user. The degree distribution will represent the shortest path distances, and the average trust ratings will be calculated to investigate whether the transaction patterns correlate with trust levels.

Milestone:

1) Data cleaning: Parse the CSV file for further graph construction and verifying data
2) Computation of Metrics: Using BFS to find out the shortest paths, degree distributions, and the average trust values
3) Visualization and data analysis: Generating histograms and scatter plots (by using the Plotters library) to visualize the correlations and distributions
4) Testing for validation: Running the unit tests (test_trust_crrealaiotna and second_test_trsut_correlation) to make sure the calculations will be valid for the final analysis

My Dataset: **"**Bitcoin Alpha Trust Weighted Signed Network"
Link: https://snap.stanford.edu/data/soc-sign-bitcoin-alpha.html.

Citations of Dataset:
S. Kumar, F. Spezzano, V.S. Subrahmanian, C. Faloutsos. Edge Weight Prediction in Weighted Signed Networks. IEEE International Conference on Data Mining (ICDM), 2016.

Edge Weight Prediction in Weighted Signed Networks:
https://cs.stanford.edu/~srijan/pubs/wsn-icdm16.pdf

S. Kumar, B. Hooi, D. Makhija, M. Kumar, V.S. Subrahmanian, C. Faloutsos. REV2: Fraudulent User Prediction in Rating Platforms. 11th ACM International Conference on Web Searchand Data Mining (WSDM), 2018.

REV2: Fraudulent User Prediction in Rating Platforms:
https://cs.stanford.edu/~srijan/pubs/rev2-wsdm18.pdf

Main Statistics:
Nodes (Users): 3,783
Edge Weight Range: -10 to +10
Positive Edges: 93%

Dataset Format:
Source: Node ID of rater
Target: Node ID of the rater
Rating: Trust level (-10 to +10)
Time: Timestamp of the rating (seconds since Epoch)

Explanation of the Dataset:
The dataset represents a network called "who-trusts-whom" from Bitcoin Alpha, a Bitcoin platform for Bitcoin trading. I wanted to explore the trust rate of Bitcoin users trust rating and if that correlates to higher transactions of the users. Bitcoin users are anonymous, and reputation ratings are used to lessen the possibility of fraud in the Bitcoin trade. The rating is determined by the users rating other users from -10 to +10, as -10 represents the maximum distrust, and +10 represents the maximum trust. This dataset shows the explicitly weighted signed director network that studies network dynamics, trust, and fraud detection.

Launch Instructions:
(The dependencies require Rust C.1.77 or above to be installed on your machine)
(The plotting library has a lot of dependencies so the project might take some time (around 2 to 3 minutes) to compile)
1) Clone the repository: Use "git clone" command to down the repository onto the local machine.
2) Navigate to the project directory: "cd" into the "src" directory that contains the main.rs file and other modules.

3) Run the test: use "cargo test" to execute the project's test and to check that all functions are working.
4) Run the execution: use "cargo run" to compile and execute the project for the data visualizations and code output of analysis.
5) Run the execution (Option 2): If it takes more than a minute, please ^c and use "cargo run --release."
6) Finding PNG files: the "degree_distribution_histogram.png" and "trust_vs_degree.png" files can be found under project/src.

Description of the Code Project:

This project foresees a "who-trusts-whom" network from the Bitcoin Alpha platform. The Rust program utilized the relationship between the users, depending on the trust rating of the users in the Bitcoin network. The essential functionalities of the code include the average trust levels, calculating the correlations between trust levels, and the number of transactions in the network.

My Project (Overall in Result of what the Code does):

The Rust program analyses the weighted signed directed graph representing user trust on the Bitcoin Alpha. The trust rates, depending on the user interaction, are determined. First, the CSV file contains the trust ratings of all the users. The file includes source users, target users, ratings that range from -10 to +10, and the time timestamp. The data is utilized to construct weighted and directed graphs where the users are represented as nodes, and the trust relationships are represented with edges with weights that are assigned to each edge. The adjacency list for efficient data analysis is used for the graphs. The time field is not used in the code analysis because the main goal is to find the relationships between users depending on the trust ratings. Thus, the time data is useless for the current analysis, so for the code simplicity, it has been exempted.

The key metrics are computed by calculating the average shortest distance between the randomly selected nodes using the Breadth-First Search (BFS) algorithm to determine the connectivity between the users. The analyzed degree distribution of the nodes represents the number of connections for each user and executes the data visualization of the histogram. The project also explores the correlation between the user's average trust rating and degree, producing a scatter plot that explicitly shows the trends in the correlation between users' trust and their activity (transaction rate).

Data visualization is essential to showing the result of the code analysis in one PNG file graph. To reiterate, the histogram of the degree distribution and the scatter plot represent users' trust in their connectivity. Both graphs show analysis to explore the relationship between user behavior and trust rating on the Bitcoin Alpha network.

Additionally, to ensure the correctness of the code, unit tests are added to verify the known sets of edges that have the expected average trust values and degrees. The calculated results do match the theoretical results. The testing in the main.rs makes sure that the trust correlation is valid.

Code Methodology:

The main.rs coordinates the other files for code analysis and the visualization that generates the PNG files. The read_files function reads the CSV file called “soc-sign-bitcoinalpha_backup.csv,” which contains the dataset for the trust relationship among the users in the Bitcoin network. The file’s row consists of the source user (the rater), the target user (the rate), and the rating (trust core between -10 and +10). The read_files function parses each row into a tuple of (source, target, weight), and the rules are stored in a vector representing the graph’s edges.

The graph is generated by the Graph::new method, which uses the parsed edge data to create a representation of the adjacency list in the trust network. The adjacency list is stored as a Hashmap. In the Hashmap, each key is a user as a node, and the value is the vector of connected users, which is also known as edges. Thus, the data structure allows efficient and fast access to users’ connections.

The average_distance function calculates the average shortest path between the randomly selected node pairs. Breadth-First Search (BFS) determines the shortest distance between each pair. The BFS utilizes the VecDeque queue to traverse all the nodes by every layer to ensure no node is revisited through the visited map. Every sample node pair is selected through the rand library to calculate a large range of networks. This function executes the average distance across all the sampled pairs to calculate the network user’s connectivity numbers.

The degree_dsitributtion function computes the edges and the number of connections for each node by iterating over all the adjacency lists. The computed edges are visualized using a history using the plot_degree_distribution and the Plotters library. The histogram visualizes the frequency of nodes with specific degrees to show patterns in user connectivity.

The trust_correalation function investigates the relationship between a user’s average trust score and degree, also known as the user’s connectivity. The logic is separated into two functions: trust_correlation_data and trust_correlation. The trust_correlation_data computes and returns the average trust and degree data. And the trust_correlation produces the scatter plot.

The test_trust_correlation function and second_test_trust_correlation function verify the input data set to verify the average trust and degree calculation result and compare it with the expected values. The test step is essential to ensure that the analytic step in the code is constantly correct as the code runs.

In more detail, the function calculates the average trust rating for every user by adding their trust weight and dividing that by the number of connections. An average trust score is paired with the user’s degree and connectivity. Ultimately, the scatter plot visualizes how trust scores relate to user activity levels.

The code's final results are executed as PNG files. The first PNG image is degree_distribution_histogram.png, a histogram of the degree distributions. The second visualization is a scatter plot of trust correlation named trust_vs_degree.png.

The graph.rs is a graph struct used to model the trust network. The graph is stored as an adjacency list of (HashMap<i32>, Vec<i32>) where the keys represent the user IDs of nodes and the values are vectors of connected user IDs of edges. The Graph::new method in the module iterates through the list of edges and updates the adjacency list. The bidirectional edges are added for each relationship to ensure the graph can be traversed in and from either direction.

The Breadth-First Search Module (bfs.rs) function calculates the shortest path between the two nodes in the graph. The VecDeque queue is initialized with the start node and the distance 0. Nodes are dequeued one by one, and their neighbors are explored. The exploration stops when it reaches the target node and returns the distance. As mentioned before, the visited map ensures the nodes are not revisited to prevent unnecessary steps during the code execution. The function will return a None if no path exists between the nodes.

The analysis.rs includes the degree distribution function to calculate the number of edges and connections for each node by iterating over the adjacency list. The HashMap is utilized to count the frequency of each degree value. For example, it calculates the number of nodes with two or three connections. The distribution data is visualized as a histogram called plot_degree_distribution using the Plotters library in Rust. The trust correlation function calculates the average trust scores and connectivity to identify whether higher user activity correlates with higher user activity.

As an additional explanation, the CSV input file's timestamp, which is in seconds due to Epoch, is ignored since the project focuses on relationships, not temporal trends. The cargo.toml as rand is used to randomly select node pairs for distance calculations to ensure unbiased results. The plotters are incorporated in the dependency for generating visualizations like histograms and scatter plots using the Rust library.

In conclusion, the average distance is the numeric output indicating the average shortest path between random node pairs. The degree distribution is shown by a histogram indicating the distribution of user connections to visualize the patterns in user activity. The trust correlation is shown by a scatter plot to visualize the relationship between the trust scores and the connectivity of the users.

Output (Preview of the First TEN lines under):

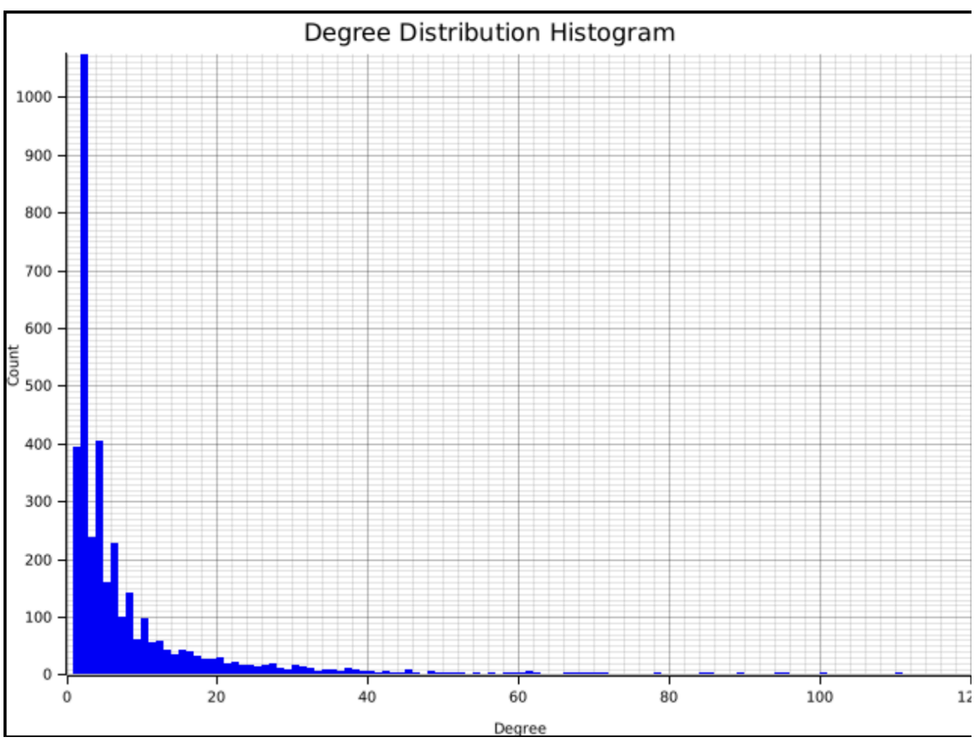
Vertex 1836: Degree = 2, Average Trust = 3.00
Vertex 3085: Degree = 1, Average Trust = 1.00
Vertex 1553: Degree = 3, Average Trust = 1.00
Vertex 7328: Degree = 8, Average Trust = -0.38
Vertex 2693: Degree = 1, Average Trust = 1.00
Vertex 2734: Degree = 1, Average Trust = 1.00
Vertex 3239: Degree = 1, Average Trust = 1.00
Vertex 543: Degree = 5, Average Trust = 3.00
Vertex 2086: Degree = 2, Average Trust = 1.00
Vertex 2532: Degree = 1, Average Trust = 7.00
Vertex 2456: Degree = 1, Average Trust = 1.00
Vertex 2192: Degree = 2, Average Trust = -0.50
Vertex 2583: Degree = 2, Average Trust = 4.00
Vertex 3237: Degree = 1, Average Trust = 1.00
Vertex 2112: Degree = 2, Average Trust = 1.00
Vertex 1949: Degree = 2, Average Trust = 1.00
Vertex 2854: Degree = 1, Average Trust = 1.00
Vertex 2735: Degree = 1, Average Trust = 1.00

Screenshot of first few lines of the Output:

```
Vertex 1836: Degree = 2, Average Trust = 3.00
Vertex 3085: Degree = 1, Average Trust = 1.00
Vertex 1553: Degree = 3, Average Trust = 1.00
Vertex 7328: Degree = 8, Average Trust = -0.38
Vertex 2693: Degree = 1, Average Trust = 1.00
Vertex 2734: Degree = 1, Average Trust = 1.00
Vertex 3239: Degree = 1, Average Trust = 1.00
Vertex 543: Degree = 5, Average Trust = 3.00
Vertex 2086: Degree = 2, Average Trust = 1.00
Vertex 2532: Degree = 1, Average Trust = 7.00
Vertex 2456: Degree = 1, Average Trust = 1.00
Vertex 2192: Degree = 2, Average Trust = -0.50
Vertex 2583: Degree = 2, Average Trust = 4.00
Vertex 3237: Degree = 1, Average Trust = 1.00
Vertex 2112: Degree = 2, Average Trust = 1.00
Vertex 1949: Degree = 2, Average Trust = 1.00
Vertex 2854: Degree = 1, Average Trust = 1.00
Vertex 2735: Degree = 1, Average Trust = 1.00
Vertex 2958: Degree = 1, Average Trust = 1.00
Vertex 2108: Degree = 2, Average Trust = 1.00
Vertex 2581: Degree = 1, Average Trust = 1.00
Vertex 284: Degree = 22, Average Trust = 0.91
Vertex 2606: Degree = 1, Average Trust = 1.00
Vertex 2094: Degree = 1, Average Trust = 1.00
Vertex 7435: Degree = 2, Average Trust = 1.00
Vertex 79: Degree = 69, Average Trust = 1.12
Vertex 1907: Degree = 1, Average Trust = 10.00
```

Graph:

a) **Histogram of the degree distributions**



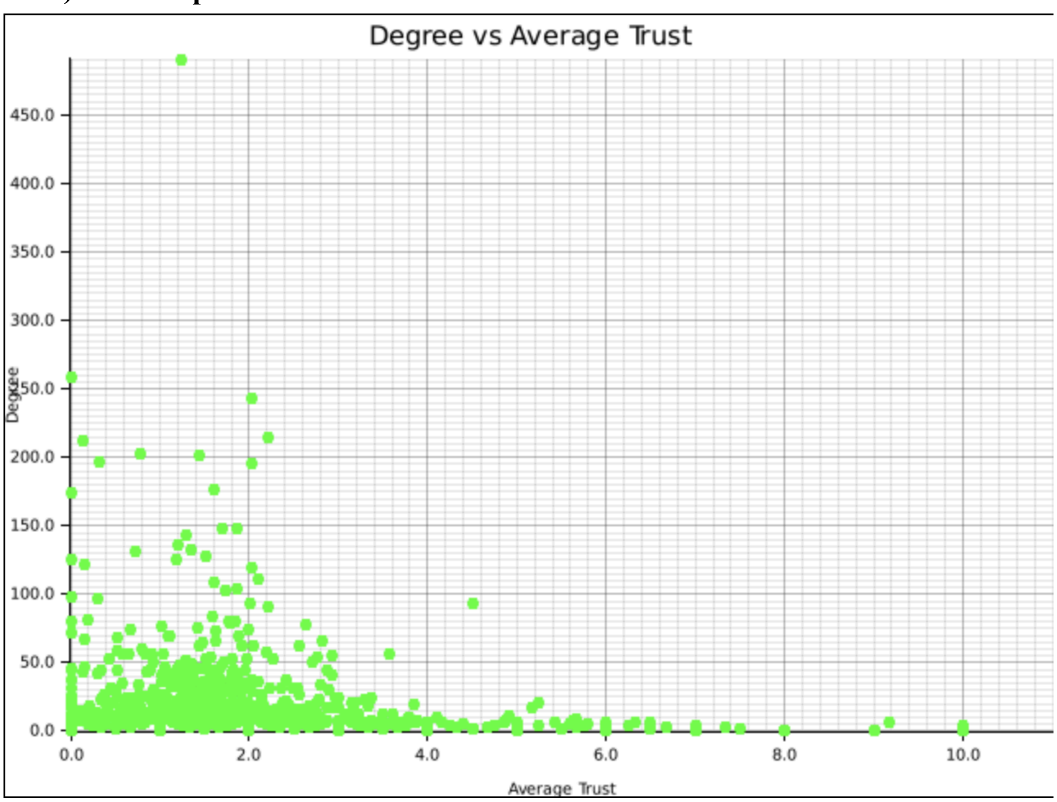
Histogram Explanation:

The graph (histogram) visualizes the degree distribution of nodes where the x-axis represents the degree, which is also known as the number of connections per node. The x-axis range for the degree distribution histogram is fixed from 1 to 120 to make the data more clearly seen in the graph since none of the data points were above 120. The y-axis represents the count (number of nodes with specific degrees). The graph shows a highly skewed distribution with a sharp peak at extremely low degrees and almost as close to zero. The long tail aims towards higher degrees. This skewed distribution shows that the majority of nodes and users in the network have only a small amount of connections, and smaller numbers of nodes have more connections. The long tail shows the few highly connected nodes known as hubs. The hubs are the central points in the network and connectivity. This correlates with real-world networks like social media or trust-based networks where connectivity across all nodes is not even.

The majority of the nodes have degrees less than 10, and few nodes have degrees exceeding 50 and fewer exceeding 100. These highly connected nodes facilitate transactions and relationships between less connected users. The dependence on a few highly connected nodes represents the network's vulnerability, and if removed, the network's efficiency would be reduced. Most nodes with low degrees represent users who have small amounts of interaction. On the other hand, their trust scores are likely to have a negative impact on the network dynamics.

In conclusion, the histogram shows that the Bitcoin Alpha trust network has an independent structure. The majority of the users have few connections, while a small number of highly connected users are essential in this analysis. The graph supports the hypothesis that trust helps the users exceed the network for higher connectivity.

b) Scatter plot of trust correlation



Scatter Plot Explanation:

The scatter plot above shows the relationship between the average trust score, which is shown on the x-axis, and the degree, which is the number of connections per node, all shown on the y-axis overall, representing the users in the Bitcoin Alpha trust network. The data visualization of the axes is adjusted so that the average trust is mapped to the x-axis and free to the y-axis.

Most nodes are clustered around the lower end of the axis, under 50 regarding the y-axis, with the average trust scores below 2. Thus, it could be concluded that most users in the network have low connections and relatively low trust scores. The graph's pattern shows the skewed degree distribution where most nodes have few connections.

A few nodes with high degrees, over 100 regarding the y-axis, are shown, but their average trust scores are mostly concentrated below 2. Thus, highly connected users tend to have moderate levels of trust. For example, a node with a degree above 450 has an average trust score under 2. Noes that have high average trust scores ($x > 5$) are densely distributed and have low degrees ($y<10$). This shows that users with high trust levels maintain small and strong connections. The plot does not show a strong linear relationship between average trust and degree. A slight trend exists where higher-level nodes are concentrated at lower average trust levels. However, the overall relationship is weak and spread across the axes.

The dense distribution of nodes with high average trust and low degree shows that users with high trust levels have selective or niche connections rather than broad and highly connected network relationships. Nodes with high degrees are the critical hubs, but their moderate trust scores show they are facilitators of interactions but not necessarily the most trusted individuals. The dense clustering of nodes in the low degree and low average trust score shows that trust is not concentrated, and users usually engage in lower amounts of interactions in the Bitcoin network.

Findings (Explanation/Analysis):

First, average trust and degree distribution are shown through the output, which shows a significant variation in both degrees: the number of connections against the average trust score for users in the Bitcoin network. For example, Vertex 450: Degree = 2, Average Trust = 6.5, and Vertex 833: Degree = 1, Average Trust = 10.00 are users with high trust but low degree. The two users above have extremely high trust scores, even though they have few connections, which shows evidence of the highly selective relationships of the users, who are vertex 450 and 833. Some users have high degrees but moderate trust, such as Vertex 106: Degree = 50, Average Trust = 1.28, and Vertex 91: Degree = 24, Average Trust = 3.00. These users have many connections but have moderate trust scores. Thus, broad and wide connectivity does not necessarily result in having the highest trust levels. Lastly, there are users with low degrees and negative trust, like Vertex 7398: Degree = 8, Average Trust = -1.50. This user shows that moderately connected nodes can correlate to distrust. Thus, having several interactions does not conclude or foster positive trust levels.

In conclusion, the results from several examples of vertex reveal that highly connected users tend to have moderate trust scores. Thus, it supports the idea that trust promotes broad engagement. On the other hand, users with low degrees often reveal extreme trust scores that are either positive or negative. As a result, it shows selective interactions or niche relationships between users in the Bitcoin network.

Second, the results show a positive correlation between trust and degree. The scatter plot of average trust against degree shows a weak positive trend. For example, Vertex 144: Degree = 21, Average Trust = 3.14, and Vertex 233: Degree = 12, Average Trust 2.58 show that users with relatively higher degrees tend to have slightly higher average trust scores. However, the trend is not relevant across all users. There are outliers like Vertex 450: Degree = 2, Average Trust = 6.5 and Vertex 833: Degree = 1, Average Trust = 10.00, where these outliers show that a small number of connections result in high trust. This pattern shows that trust facilitates higher degrees of interaction; it supports the hypothesis that trust motivates more transactions. However, the existence of outliers shows that the influence of selective relationships highly depends on trust levels.

Third, the output reveals outliers and niche patterns like the selective high trust of Vertex 833: Degree = 1, Average Trust = 10.00, where the node highlights a highly trusted individual with small interactions. This user shows a niche relationship that is built on strong trust. Vertex 7398: Degree = 8, Average Trust = -1.5 shows distrust but high connectivity. This user shows how distrust is possible even with numerous connections; it overall shows the complexity of the trust dynamics in the Bitcoin network. There is balanced connectivity with the example of user Vertex 91: Degree = 24, Average Trust = 3.00, showing high connectivity with moderate trust. This user shows a balance of high connectivity and trust as a stable user in the Bitcoin network. In the end, the outliers are an essential part of the analysis. Trust does promote transactions, but the quality of the connections may be unpredictable due to the complex nature of the interactions in the Bitcoin network. High trust in selective relationships does contrast with moderate levels of trust in a wider range of interactions.

To summarize, high trust initiates selective interactions. In more detail, users with few connections often show high trust scores. Focused relationships facilitate meaningful transactions and show that high trust drives selecting interactions. Also, moderate trust supports broader engagement. The users (vertex) with multiple connections have moderate trust scores. Lastly, distrust does not prevent interaction. There are nodes with negative trust scores but have moderate degrees. Thus, distrust does not automatically mean that users are excluded from network engagement.

In conclusion, the findings support the hypothesis that higher trust levels lead to high transactions. Higher trust levels foster more connectivity and engagement, which is revealed in the output and the analysis of users with high degrees and positive trust scores. At the same time, selective relationships built on concentrated trust demonstrate that low-degree nodes could also have a lot of interactions in the Bitcoin network. Some outliers have negative trust levels or extraordinarily high and positive trust scores that add nuance. This indicates that trust is not equally treated across all networks. The analysis shows that trust in decentralized networks like Bitcoin Alpha is important for maintaining healthy and trustworthy transactional activity and network health.

Citation (URL):

- https://stackoverflow.com/questions/34747464/implement-graph-like-data-structure-in-rust
- https://kuczma.dev/articles/rust-graphs/
- https://docs.rs/petgraph/latest/petgraph/
- https://sachanganesh.com/programming/graph-tree-traversals-in-rust/
- https://gist.github.com/vTurbine/16fbb99225ad4c0ac80b24855dd61a7c
- https://stackoverflow.com/questions/71189961/bfs-algorithm-tracking-the-path
- https://docs.rs/pathfinding/latest/pathfinding/directed/bfs/index.html
- https://docs.rs/graphrs/latest/graphrs/struct.Graph.html
- https://docs.rs/oxygraph/latest/oxygraph/bipartite/struct.BipartiteGraph.html
- https://stackoverflow.com/questions/8344565/how-do-i-calculate-weighted-degree-distributions-with-igraph-in-r
- https://yuudaiishihara.medium.com/using-rust-for-statistics-unlocking-high-performance-and-safety-with-sample-code-5490fb03a30d
- https://stackoverflow.com/questions/19671845/how-can-i-generate-a-random-number-within-a-range-in-rust
- https://stackoverflow.com/questions/57829511/how-to-create-a-random-sample-from-a-vector-of-elements
- https://users.rust-lang.org/t/rust-random-choice/6559
- https://docs.rs/graphrs/latest/graphrs/
- https://docs.rs/metrics/latest/metrics/
- https://docs.rs/metrics-runtime/latest/metrics_runtime/
- https://stackoverflow.com/questions/68050233/how-to-create-a-degree-correlation-matrix-for-a-network
- https://www.reddit.com/r/learnrust/comments/vlr7k7/how_to_implement_this_algorithm_in_rust_beginner/
- https://stackoverflow.com/questions/37586216/step-by-step-interactive-debugger-for-rust
- https://users.rust-lang.org/t/what-hapens-if-hash-and-partialeq-dont-match-when-using-hashmap/98052
- https://www.reddit.com/r/rust/comments/98hyq9/inserting_into_a_hashmap_in_a_match_errors_arms/
- https://stackoverflow.com/questions/74388236/expected-fn-item-error-when-storing-functions-in-a-hashmap-in-rust

GPT Citation (Screenshot & Explanation):

1) Citation 1

how can I implement a bfs algorithm in rust to calculate the shortest path between two nodes in a graph?

rust   Copy code

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

pub fn bfs(graph: &Graph, start: i32, target: i32) -> Option<usize> {
    let mut queue = VecDeque::new(); // Queue for BFS
    let mut visited = HashMap::new(); // Tracks visited nodes

    queue.push_back((start, 0)); // Start BFS from `start` node
    visited.insert(start, true); // Mark start node as visited

    while let Some((current_node, distance)) = queue.pop_front() {
        if current_node == target {
            return Some(distance); // Found the target, return distance
        }
```

```rust
    while let Some((current_node, distance)) = queue.pop_front() {
        if current_node == target {
            return Some(distance); // Found the target, return distance
        }

        if let Some(neighbors) = graph.adjacency_list.get(&current_node) {
            for &neighbor in neighbors {
                if !visited.contains_key(&neighbor) {
                    visited.insert(neighbor, true);
                    queue.push_back((neighbor, distance + 1)); // Visit neighbors
                }
            }
        }
    }

    None // Return None if no path exists
}
```

GPT Citation 1 Explanation:

I learned how to use a BFS algorithm to calculate the shortest path between two nodes in a graph. The algorithm uses a VecDeque queue to analyze every node and check that the shortest path is found in the unweighted graph. In the code, the hashmap tracks nodes to prevent redundancy. At the beginning of the code, it starts by adding the initial node with zero distance to the queue; then, it iterates to check whether the current node matches the target. If a match is found, the function returns the shortest path. When the target is not found, the current node's neighbors are enqueued with the adjusted distance, and it continues until the queue is empty. The implementation handles for no path exists by returning None. Thus, I learned how to manage graph traversal and optimize pathfinding algorithms using VecDeque and HashMap.

2) Citation 2

how do I generate and plot a histogram of node degrees in a graph using the plotters library in rust?

rust Copy code

```rust
use plotters::prelude::*;
use std::collections::HashMap;

pub fn compute_node_degrees(graph: &Graph) -> HashMap<usize, usize> {
    let mut degree_counts = HashMap::new();
    for edges in graph.adjacency_list.values() {
        let degree = edges.len();
        *degree_counts.entry(degree).or_insert(0) += 1;
    }
    degree_counts
}

pub fn render_histogram(degree_distribution: &HashMap<usize, usize>, output_file: &
    let drawing_area = BitMapBackend::new(output_file, (640, 480)).into_drawing_are
    drawing_area.fill(&WHITE).unwrap();
```

Copy code

```rust
    let max_degree = *degree_distribution.keys().max().unwrap_or(&0);
    let max_count = *degree_distribution.values().max().unwrap_or(&0);

    let mut chart = ChartBuilder::on(&drawing_area)
        .caption("Node Degree Histogram", ("Arial", 20).into_font())
        .x_label_area_size(40)
        .y_label_area_size(40)
        .build_cartesian_2d(0..max_degree + 1, 0..max_count + 1)
        .unwrap();

    chart.configure_mesh()
        .x_desc("Degree")
        .y_desc("Frequency")
        .draw()
        .unwrap();

    chart.draw_series(
        degree_distribution.iter().map(|(&degree, &count)| {
            Rectangle::new([(degree as i32, 0), (degree as i32 + 1, count)], BLUE.
        }),
    )
    .unwrap();
}
```

Key Steps:

1. **Compute Node Degrees**:
   - `compute_node_degrees` calculates the frequency of each degree by iterating through the graph's adjacency list.
   - The result is a mapping of degrees to their respective counts.
2. **Render the Histogram**:
   - `render_histogram` uses the Plotters library to create a histogram.
   - The x-axis represents node degrees, while the y-axis shows their frequency.
   - The final chart is saved to the file specified in `output_file`.

Usage:

rust Copy code

```rust
let degree_data = compute_node_degrees(&graph);
render_histogram(&degree_data, "node_degree_histogram.png");
```

This method generates a visual representation of the graph's degree distribution while keeping the implementation general and reusable.

GPT Citation 2 Explanation:

I learned how to generate and visualize a histogram of degrees and use the Plotters library. First, I need to compute the degree distribution and make the histogram. The compute_node_degrees function iterates over the adjacency list to calculate the frequency of every degree in a Hashmap and the degrees to their occurrence counts. The render_histogram function utilizes the Plotters library to create a histogram. The x-axis represents node degrees, and the y-axis shows the frequencies. Mesh configurations and rectangular bars count for each degree's count, and the code makes a graph saved to a file. I learned how to interpret graph-related metrics.