# Project Write-Up

## Dataset Description

Link: https://snap.stanford.edu/data/soc-sign-bitcoin-alpha.html

I am planning to use the Bitcoin Alpha Trust Weighted Signed Network dataset from Stanford's SNAP repository. This dataset represents the network of Bitcoin transaction users on the Bitcoin Alpha platform, where each user is assigned a 'trust score' that ranges from negative values that indicate distrust to positive values that indicate trustworthiness (range from -10 to 10).

The dataset can be represented as a directed weighted graph. Each user can be represented as a node, and the trust score is formed as the directed, weighted edges between users. This structure provides an opportunity to apply various graph algorithms and explore how trust and distrust influence the overall network. Also, it is interesting for studying how trust and distrust between users and to find the relationship between different users with different trust scores and how can the relation evolve along the edges.

## Project Overview

My project conducts an analysis of graph-based data using a dataset of edges representing relationships between nodes. The main goal is to calculate various graph properties, including clustering coefficients, trust scores, and subgraph connectivity, and identifying key representative nodes (k representatives). It also analyze the clustering and centrality of high (nodes with trust score above 2.00) and low trust nodes (nodes with trust score below - 2.0). I choose -2.0 as low trust score boundary because there are a few users that have average score that is lower. With these result, I wish I can answer the question: In this dataset, do users clustered around users with high or low trust scores? Or is the clustering just random?

## How to Run

1. Place the input dataset file soc-sign-bitcoinalpha.csv in the directory: DS210Project/project (at the same layer with src, Cargo.toml …)
2. *Make sure to be on the directory DS210Project/project
3. Run the program using:
   cargo run / cargo run --release
4. Run the tests using:
   cargo test / cargo test --release

## Outputs

The program outputs the following information:

- **General Graph Information:** Number of edges, nodes and subgraphs etc.
- **Clustering and Centrality Analysis:** Statistics for high and low trust score nodes.
- **Representative Nodes:** A list of key representative nodes with associated clustering coefficients and trust scores.

The expected output will be shown after the description of each function in each module.

## File Structure and Functionality

There are a total of four major components in this project. They are:

- main.rs - wrap up calls
- data_loader module - clean data from raw csv file to structured data
- graph module - contain graph operation and algorithms.
- analyze module - contain methods that analyze the graph.

### Detail descriptions

1. **main.rs**
    - **Purpose:** Wrap-up all the functions and methods in the project to generate the final results.
    - **Input:** None (default CSV file path is hardcoded).
    - **Output:** Printed analysis results to the terminal.
    - **Process:**
        - Loads the CSV data.
        - Constructs the graph structure.
        - Computes graph metrics and results.
        - Displays results.
2. **data_loader/mod.rs**
    - **read_csv(file_path: &str) -> Vec<Edge>**
        - **Input:** File path to the CSV file. ("soc-sign-bitcoinalpha.csv")
        - **Output:** A vector of Edge structs.
        - **Purpose:** Reads the edge data from the CSV file and format and clean the data. (e.g. it removes the timestamp data in the csv file as it is unuseful)
        - **Process:**
            - Opens the CSV file.
            - Parses each line to extract from, to, and weight.
            - Constructs and returns a vector of edges. Where each edge is stored as an Edge struct that will be specified in the graph module.
3. **graph/mod.rs (graph module)**
    - **Structs:**
        - Edge: Represents a directed edge with:

        - from: source node
        - to: target node
        - weight: the trust score that the user represented by the source node gives to the user represented by the target node.
    - Graph: Represents the graph using a HashMap of nodes to adjacency lists.
    - NodeNeighbors: Captures input and output neighbors for a node.
- **Functions:**
    - **new(edge_lst: &Vec<Edge>) -> Graph**
        - **Input:** A list of edges.
        - **Output:** A Graph object.
        - **Purpose:** Constructs the graph from a list of edges.
        - **Process:**
            - Initializes a HashMap for adjacency lists.
            - Populates the map with edges, ensuring all nodes are included.
    - **get_neighbors(&self, node: usize) -> NodeNeighbors**
        - **Input:** Instance itself, A node ID.
        - **Output:** A NodeNeighbors object with input and output neighbors.
        - **Purpose:** Retrieves neighbors for a given node.
        - **Process:** Searches for edges connected to the node and then separates incoming and outgoing neighbors.
    - **get_degrees(&self) -> (HashMap<usize, f64>, HashMap<usize, f64>)**
        - **Input:** Instance itself
        - **Output:** HashMaps for in-degrees and out-degrees of nodes.
        - **Purpose:** Computes in-degrees and out-degrees for all nodes.
        - **Process:** Iterates through edges to count connections (both indegrees and out degrees) by using the get_neighbors method.
    - **clustering_coefficient(&self, node: usize) -> f64**
        - **Input:** Instance itself, A node ID.
        - **Output:** Clustering coefficient as a f64.
        - **Purpose:** Calculates how strongly the node's neighbors are connected. The clustering coefficient often indicates the density of connection around a specific node.
        - **Process:**
            1) Finds all neighbors of the node using get_neighbors.
            2) Counts edges between these neighbors.
            3) Computes the coefficient using the formula:
            (# of edges connecting two of the node’s neighbors) / (the total number of possible edges between the node’s neighbors)

        - **find_subgraphs(&self) -> Vec<Graph>**
            - **Input:** Instance itself.
            - **Output:** A vector of subgraphs.
            - **Purpose:** Identifies all connected subgraphs in the graph.
            - **Process:** It uses BFS algorithm in the algorithm module to traverse and isolate subgraphs. (BFS algorithm will be described below)
        - **get_trust_score(&self, node: usize) -> f64**
            - **Input:** Instance itself, A node ID.
            - **Output:** Trust score as a f64.
            - **Purpose:** Measures the node's trust based on incoming edge weights.
            - **Process:**
                1) Sums weights of incoming edges.
                2) Divides by the number of incoming edges.
4. **graph/algorithm.rs (algorithm module)**
    - **bfs(graph: &Graph, start_node: usize, visited: &mut HashSet<usize>) -> HashSet<usize>**
        - **Input:** A Graph object, the starting node, and a mutable visited set that stores the visited node by the bfs algorithm.
        - **Output:** A set of connected nodes.
        - **Purpose:** Explores connected nodes within a subgraph using BFS algorithm. (Similar one as we learned in class, but this one only records every nodes that are connected from the start_node).
        - **Process:**
            - Initializes a queue with the starting node.
            - Traverses all reachable nodes while marking them as visited.
5. **analyze/mod.rs (analyze module)**
    - **Structs:**
        - GraphInfo: Aggregates the computed graph metrics, including:
            1) in and outegrees
            2) clustering coefficients
            3) trust scores
            4) set of subgraphs
            5) The Graph represents the data.
    - **Functions:**
        - **get_info(graph: &Graph) -> GraphInfo**
            - **Input:** A Graph object.
            - **Output:** A GraphInfo object with graph metrics.

- **Purpose:** Initialize a new GraphInfo object that stores key data and properties of the graph.
- **Process:** Computes degrees, clustering coefficients, trust scores and identifies subgraphs.

- **analyze_clustering_centrality(&self, high_score: f64, low_score: f64) -> String**
  - **Input:** The instance itself, High and low trust score thresholds.
  - **Output:** A summary string of clustering and centrality analysis.
  - **Purpose:** Analyzes high and low trust nodes for clustering and centrality.
  - **Process:**
    1) Filters nodes by high and low trust score thresholds.
    2) Calculates percentages of high clustering and centrality in those two sets of nodes. Centrality is simply examined based on the number of indegree of the node. I choose indegree because it implies the frequency of people trading with this specific person, which might be useful for my analysis.
- **find_k_representatives(&self, k: usize) -> String**
  - **Input:** Number of representatives k.
  - **Output:** A summary string of representative nodes.
  - **Purpose:** Identifies key representative nodes based on metrics.
  - **Process:**
    1) Normalizes metrics for centrality and clustering by using the normalize helper function. This function uses the formula: normalize_val = (value - min) / (max - min) where min and max are the minimum and maximum value in the set. This min and max values are get by using another method: find_min_max.
    2) For each node, calculate a clustering_centrality score based on the formula: score = 0.7 * normalized_centrality + 0.3 * normalized_clustering_coefficient
    3) Sorts the nodes by combined scores.
    4) Selects the top k nodes.

## Tests

For each function / method there are 2-3 tests conducted to ensure correctness.

## Expected Output:

---------- General Info ----------

Total number of edges: 24186

Number of nodes in this data: 3783

Number of subgraphs in this data: 5

Number of nodes in sub graph 1: 3775

Number of nodes in sub graph 2: 2

Number of nodes in sub graph 3: 2

Number of nodes in sub graph 4: 2

Number of nodes in sub graph 5: 2

---------- Clustering and Centrality of nodes with high / low trust score ----------

Nodes with trust score >= 4: 216 nodes.

Percentage with high clustering: 25.93%.

Percentage with high centrality: 5.56%.

Nodes with trust score <= -2: 188 nodes.

Percentage with high clustering: 53.19%.

Percentage with high centrality: 10.64%.

---------- K representatives ----------

Selected Representatives: [1, 3, 2, 11, 177, 4, 7, 818, 1374, 471, 524, 736, 1023, 1209, 1359]

The number of representatives is 0.40% of total nodes.

Average trust score of representatives: [1.9045226130653266, 2.4302788844621515, 3.5853658536585367, 1.3940886699507389, 0.21717171717171718, 2.925373134328358, 1.8974358974358974, 2.5, 1.25, 5.666666666666667, 5.333333333333333, 3.6666666666666665, 2.3333333333333335, 1.6666666666666667, 1.6666666666666667]

Average clustering coefficient of representatives: 0.5503277563992436

# Analyze of the results:

## General Information:

## Clustering and Centrality of nodes with high / low trust score

A clustering coefficient measures the likelihood that a node's neighbors are connected. Nodes with high clustering coefficients tend to belong to densely connected communities, while lower coefficients indicate more distributed or relationships. (“high” clustering coefficients and centrality are defined in the program as above average. i.e. if node.cc > mean_cc then node.cc is high )

The results of the analysis shows that the 216 nodes with moderate to high trust scores ($\geq 4$) exhibit relatively low clustering (25.98%), and very low centrality (5.56% ) indicating that these nodes might act as connectors rather than isolated clusters around or the clusters around high trust score nodes are not very obvious. On the other hand, the 188 low-trust nodes ($\leq -2$) have a higher clustering percentage (53.19%) and relatively higher centrality (10.64%), suggesting that a considerable portion of these nodes might be a part of smaller, densely connected communities.

This result was not what I originally thought as I believe more transactions should be happening around people with high trust scores rather than the low ones. However, this result might not be promising since there might exist some well-clustered community around people with high trust scores, but due to the lack of transactions in the other high-trust-score users, this statistics is averaged down. Therefore, I conducted the k representative operation.

## k representatives

The k representatives are a small subset of nodes (I chose k = 15 in main.rs), selected based on their clustering coefficient and centrality measures.
The selected representatives have the characteristics:

- **Average Trust Score:**
  The average trust scores of the representatives ranged from **0.217** to **5.667**, with a mean of **2.430**.
- **Average Clustering Coefficient:**
  The representatives had an average clustering coefficient of **0.550**, indicating they are relatively well-connected within their local neighborhoods. (max is 1.0 where the neighbors of nodes are fully connected)

These 15 representatives actually align with my belief since they all have moderate to high trust scores that are strictly positive. This result is some what impactful as these representatives are

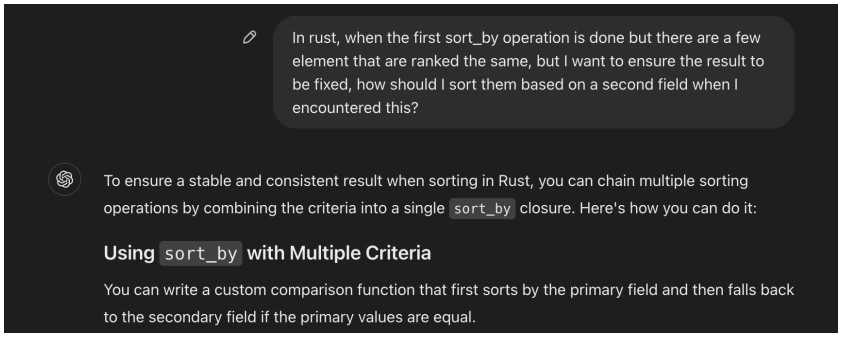
chosen based on clustering and centrality without the weight of trust_score itself. Therefore, at least for these 15 representatives that are relatively centered and well-clustered, they all got moderate to high trust scores.

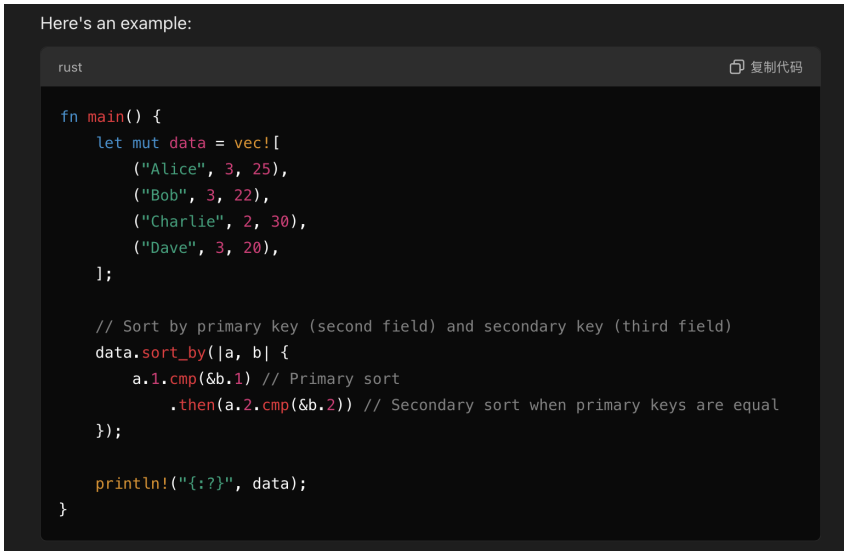
### Do users in this network cluster around high-trust score or low-trust score users?

From the k representative at least, I think the phenomenon “more transactions happened around high-trust score users rather than low-trust score users” does hold, as the 15 representative is strictly positive, with moderate to high trust score. And I have tried several other k values and I found out that the first 26 best representatives are all strictly positive.

The results in the **Clustering and Centrality of nodes with high / low trust score** section, however, suggests the contrasts, but this is probably due to localized interactions, some cases of one-time users or few-time users and the one that I discussed above.

## Chat GPT usage:



### Explanation

- `a.1.cmp(&b.1)` sorts based on the primary field.
- `.then(a.2.cmp(&b.2))` adds a fallback to sort by the secondary field if the primary field values are equal.

### When to Use `sort_by_key`

If your sorting criteria are simple, you can achieve the same result using `sort_by_key` by creating a tuple of the sorting keys:

rust 复制代码

```rust
data.sort_by_key(|item| (item.1, item.2));
```

This sorts the data by the second field first and then by the third field for ties.

### Important Notes

1. Rust's `sort_by` is **stable**, meaning it preserves the order of equal elements as they appear in the original list.
2. For performance with complex logic, `sort_by_key` can be more concise, but `sort_by` gives you more flexibility with custom logic.