Nicholas Tan, U68220750

DS210 Final Project Report

Overall

This project aims to map out a graph of Bitcoin (BTC) Alpha trust networks between users (vertices) based on past interactions and transactions. The weighted edges represent trust scores between -1 and 1, with -1 being complete distrust while 1 is full trust. 0 represents neutrality. The code outputs a graph with unidirectional edges, as the rater and ratees are specified in the dataset. A graph with bidirectional edges is ***not*** used as the rater and ratee may have different ratings for each other.

Methodology

This code first computes the average trust rating of each user using the Eigenvector centrality index, before looking at the likelihood that users separated by at least one mutual trust each other as well. This is done by identifying mutual connections before allocating a mutual trust score to them. The TrustGraph struct is represented by an adjacency matrix generated organically. This combination allows for the most optimal way to display the graph and the connectedness between nodes.

Dataset

The dataset utilized is the Stanford Network Analysis Project (SNAP) dataset with 3,783 nodes representing BTC Alpha users and 24,186 edges, representing the trust ratings.

Eigenvector Centrality

Eigenvector centrality is a measure of the influence a node has on a network (Shaw, 2019). If a node is pointed to by many nodes (which also have high eigenvector centrality) then that node will have high eigenvector centrality. Relative scores are assigned to all nodes in the network based on the concept that connections to high-scoring nodes contribute more to the score of the node in question than equal connections to low-scoring nodes. In other words, it not only counts the direct connections a node has but also considers the influence of the nodes to which it's connected (Golbeck, 2013).

$$x_i = \frac{1}{\lambda} \sum_j A_{ij} x_j$$

where:

- A_{ij} is the adjacency matrix entry representing an edge from node i to node j,
- λ is a constant (the largest eigenvalue of the adjacency matrix).

The PageRank algorithm used by Google's search engine is a variant of Eigenvector Centrality, primarily used for directed networks (Hansen, Shneiderman, 2020).

Graph Representation: **Adjacency Matrix**

The ‘DMatrix’ is constructed explicitly for the eigenvector calculation. This approach is required because linear algebra operations are simpler with a matrix representation as data processing is more efficient. The ‘build_adjacency_matrix’ function creates a ‘DMatrix’ object (from the ‘nalgebra’ crate) to represent the graph as a matrix. The matrix is populated by iterating through all the edges in the graph and setting the appropriate matrix entries according to the edges' weights. Using an adjacency matrix is beneficial for this specific computation because matrix operations (like eigenvector calculations) require it.

Modularization and Implementation

The code is split into five parts, the csv, centrality, relationships, main and test.rs files.

1. The ‘csv.rs’ crate reads the BTCAlpha Trust Graph csv, and then constructs the necessary structs of ‘TrustGraph’ and ‘TrustRelation’. It implements TrustGraph as a method by first constructing the HashMap and adjacency list from the ‘petgraph::DiGraph’ object, using the functions ‘add_relation’ and ‘add_relations’. It then uses the ‘build_adjacency_matrix’ function to build on DiGraph to construct the adjacency matrix via the DMatrix object from the nalgebra crate by analyzing the data from the csv file and then using it to populate the graph.
2. The ‘centrality.rs’ crate computes the Eigenvector centrality score (as described above) as a metric for determining the trustworthiness of each node/user/vertex. It takes advantage of a trait and implementation methods to iterate through the TrustGraph and look for convergences by determining if they are within the tolerance threshold. It then uses this to calculate the centrality score represented by a HashMap.
3. The ‘relationships.rs’ crate analyzes the mutual relationship between various vertices separated by a singular vertex and then attempts to calculate a score for the strength of their mutual trustworthiness. It also takes advantage of the trait and implementation blocks, creating a vector and then pushing nodes through it should they be identified as neighbors with a mutual connection. The mutual trust score is then computed via the formulation: If two vertices A and B share a mutual neighbor C, then the trust score between A and B is the average of the scores between A and C and between B and C.

4. The ‘main.rs’ crate runs all the above crates based on the BTC Alpha csv file data containing the information of the rater, ratee and trust score. It then outputs the TrustGraph, the centrality score as well as the mutual score between nodes.
5. The ‘test.rs’ crate runs tests for generating the graphs, Eigenvector centrality as well as the mutual scores using pre-set values to see if the functions work as intended.

Output

The output thus comprises the TrustGraph, the centrality score for each node as well as the mutual score between nodes.

References

Hansen, Derek, and Ben Shneiderman. “Analyzing Social Media Networks with NodeXL | ScienceDirect.” www.sciencedirect.com, 2020. Analyzing Social Media Networks with NodeXL | ScienceDirect.

Golbeck, Jennifer. “Analyzing the Social Web | ScienceDirect.” www.sciencedirect.com, 2013. Analyzing the Social Web | ScienceDirect.

Shaw, Alan. “Understanding the Concepts of Eigenvector Centrality and Pagerank.” Strategic Planet, July 13, 2019. Understanding The Concepts of Eigenvector Centrality And Pagerank.

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

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 Search and Data Mining (WSDM), 2018.

Appendix

petgraph::DiGraph use case

The ‘petgraph::DiGraph’ object is an adjacency list graph implementation. It keeps edges organized by their connections and supports efficient traversal, searching, and pathfinding. It is more memory-efficient than an adjacency matrix for .arge yet sparse graphs which have relatively few edges per node compared to the total number of possible connections.