Rationale:

For my DS 210 final project, I decided to implement a modified random-surfer implementation of the EigenTrust algorithm, based on its description from the paper *The Eigentrust algorithm for reputation management in P2P networks* on the Bitcoin OTC trust weighted signed network dataset. First, I decided to explore the Bitcoin OTC trust weighted signed network dataset, because it seemed super interesting to analyze a social network, but I didn't want to pick a social network that typically comes to mind like facebook friends. The bitcoin dataset was a unique social network that definitely received less attention from more traditional social network datasets, which made it more alluring to work with. Additionally, the dataset itself had a good amount of nodes at 5,881 implying that it was complex enough for this project without being overly complex and large.

When analyzing a social network that is specifically a trust network, like in the case of the chosen dataset, it seems intuitive to want to analyze the dataset in the context of trust. After looking around for some algorithms, I came across the EigenTrust algorithm which, when given a weighted trust graph, can provide the trust values for all nodes in the graph. However, the native algorithm utilized linear algebra and a stochastic matrix to do so. However, a modified, probabilistic, version of the algorithm could be run that would be more of a graphing algorithm. This algorithm was chosen for the project.

The Algorithm:

The modified EigenTrust algorithm works as follows:

1. An adjacency matrix is initialized based on the graph
2. The normalized trust values are calculated for each node: this is the sum of the ratings given by the node to all other nodes.
3. Divide the ratings of all nodes by their normalized trust values. These new ratings are called the normalized ratings.
4. Initialize a random surfer beginning at a random node, and have it jump to random nodes, with the probability that it jumps to node x being the normalized rating. Have it jump a substantial number of times.
5. Repeat step 4 a substantial number of times.

The steps are fairly vague, and a good amount of modifications (which will be discussed in the next section) were made to the above steps which were outlined in the paper for a probabilistic implementation of the EigenTrust algorithm. The idea is that the random surfers tend to end up in more trusted nodes than less trusted nodes, so over time you can rank the trustworthiness of nodes by the amount of times random surfers ended up there.

The Code:

The code begins with reading in the dataset. The function read_file in file.rs parses the csv file, first turning it into a vector of &str where each element is the line. The elements of this vector are then iterated through and each element is then split once again to create a new vector whose elements are each number within the line. The first three numbers are then parsed into type unsigned integer 32 (since they are type &str when split), packaged into a tuple, and pushed into the result vector which the function then outputs. Note that there is a match statement that catches empty lines (as these are in the dataset).

Then, a custom structure was created called an adjacency matrix with two main fields: normalized_trust_vector and obj. obj is the adjacency matrix, while normalized_trust_vector has the sum of all ratings above zero for all nodes (the ith index of normalized_trust_vector corresponds to the ith + 1 node, and same for the adjacency matrix). A normalized adjacency matrix could not be created, as the resultant error created from storing fractions as floats was too large (of the magnitude of >30%). In other words, instead of each row's sum of ratings greater than zero being one, it was less than 0.7 for some rows. There is one associated method with the adjacency matrix, which is the method to create it. In order to do so, it initializes a matrix of the desired size, and fills it up with an offset value (which in this case is 0). Then, it loops through the list of edges and updates the matrix's rater-1 row and ratee-1 column (since all indices are node name - 1) to the weight. After, each row is iterated through and summed in such a way to create the normalized trust vector. Though an optimization could be in order here, and the normalized trust vector could be created simultaneously while the adjacency matrix is created, it adds little time and comes at a tiny performance cost compared to the simplicity of having it in a separate loop.

Now that the adjacency matrix and normalized trust denominator have been created, the algorithm can finally be run. First, the random_walk function was created, and given an adjacency matrix and iters, the number of steps within this walk, the function generates a random starting node, traverses the specified number of nodes (steps), and returns the last node. It does this by generating a random die roll from 0 to the normalized trust vector, and iterating through the rating row of the current node, adding up trust values greater than zero until the die roll number is reached. Whatever node it is currently on will be the node it jumps to next. Then, the supplementary function all_walks was created, and it just runs random_walk the specified number of times and returns a vector with its answers. These functions are implemented using multi-threading in the main thread, with two function parameters cloned and one passed in using an Arc pointer. Each one of these threads returns a result internal to

the thread handle, which is their all_walks result. The function flatten in the eigentrust.rs file turns these multiple vectors into one vector, which is then tossed into the analysis function. The analysis function essentially outputs, using the vector of end nodes, the most trustable and least trustable vectors based on this 'flattened' vector. Finally, a scatter plot of the results are created where the x value is the node name and the y value is the amount of times it was the ending node for each random walk. Interestingly, the nodes with numerically lesser names have higher trust values while those with numerically larger names have lower trust values. The visualization is stored in the out.html file.

A couple of cons to note about this version of the algorithm. The proper EigenTrust algorithm essentially finds the steady-state of the normalized adjacency matrix (after modifying it a couple of times to turn it into a regular stochastic matrix). As such, it is able to account for negative trust values assigned to nodes while the current random-surfer version of the EigenTrust algorithm cannot account for negative trust values, and treats them the same as zero. A possible solution could be to change the offset value to 10, thus making the possible range of trust values from 0 to 20. However, this makes it much harder for the data to converge, and thus unless the number of iterations and cycles are increased to an extremely high number the output is noise and the most trusted nodes change each time. Another feature of the algorithm is that it only considers nodes that a random surfer landed on. If a random surfer did not land on a node, it could be because it has all negative or zero ratings, in which case the algorithm doesn't necessarily need to alert the operators that said node is untrustworthy. There do exist some of these nodes (with all zero or negative ratings) that exist within the system, and including them would also make the output of the algorithm way less interesting.

Running the Program:

First, download the github repository. It includes all files including the cargo toml, lock, and dataset. Running cargo tests will ensure that all files, modules, and crates are properly set up and imported. Then, cargo build –release can be run to run the main file, and the program will print part of the output and generate an "out.html" file which has a visualization. Note that the first time cargo build –release is run will take a long time because many dependencies for a crate required for the code, plotly, will have to be installed (there could be something else going on, I am not too sure). However, after this compiling and running the code will become quicker. In the multithreading section, underneath the multithreading comment, there are various parameters that you can change including thread count, the number of random walks, and the steps per walk. However, the code itself already has some predefined value that produces a good distribution and takes a slightly long but manageable amount of time.

Findings:

As mentioned before, the algorithm outputs the top five trusted nodes, the least trusted five nodes that were picked up by the algorithm, and a graph showing the distribution of trust values. The top five trusted nodes are pretty much the same each time, with little variation, and even in the graph it is clearly visible that the top trusted nodes have trust values significantly higher than the rest of the nodes. However, it is also visible in the output that the top trusted nodes are close together, and that's probably why the top five change occasionally, but the top trusted nodes remain the same. However, with the untrustworthy nodes, there are so many that have appeared only once at the end of 100,000 random walks that the bottom five least trustworthy nodes change each time. There are also many more, as explained before, that nobody has rated ever so they are never even picked up by the algorithm. However, I am certain that the algorithm works as the top five most trustworthy nodes are generally very consistent with few changes. This can be understood more by looking at the visualization, which can be accessed via the "out.html" file. Note that for the visualization the x-axis doesn't mean anything, and simply sorts the trust values from least to greatest.

Overall, the findings are interesting and show that the random-surfer implementation can assure that a node is trustworthy, but struggles to definitively identify untrustworthy nodes and rank the least trustworthy against each other.