Amrita Kohli DS210 Final Project Report

For my project I aimed to answer the question, “How often are friends of my friends my friends?” by searching for the pair of friends with the most common friends and the pair of friends with the highest measure of similarity.

The data set I chose to use consisted of friends lists from Facebook. This data set is an ego network, or a personal network, which refers to the network of relationships surrounding an individual in a social network. The nodes in this ego network are individuals and the edges are undirected, representing the friends of the nodes. The data set contains 4039 nodes and 88234 edges. The original Facebook user-ids have been replaced with u32 values, which anonymizes the dataset. However, this limits feature analysis because features that originally conveyed specific information, like political affiliations, have been generalized to "anonymized feature 1".

I decided to use the petgraph crate because it provides efficient data structures suitable for social network analysis and it supports both undirected and directed graph representations. Since the data set I chose was undirected, I used the ‘Graph’ graph type as opposed to ‘DiGraph’, but it is helpful that petgraph allows the option to use either type. The nodes and edges in petgraph are represented by indices which allows for efficient indexing and retrieval of elements. This was useful in my project because it made finding and retrieving the pair of nodes with the most common friends or highest similarity more efficient.

For the structure of my code, I used 3 modules: graph.rs, common.rs and jaccard.rs. The graph.rs module uses the petgraph crate to build a graph from the data set. I created a struct to represent the SocialGraph I wanted to create, which used the ‘Graph’ type from the petgraph crate. The nodes are represented by u32 values to be consistent with the format of the data set. The node indices are represented by a HashMap in order to efficiently retrieve the NodeIndex for a given node value during operations. NodeIndex is a type provided by petgraph which represents an index into the graph’s nodes. The methods defined in the SocialGraph struct include a method which creates a new instance of SocialGraph and a method which loads the edges from the input file to the graph.

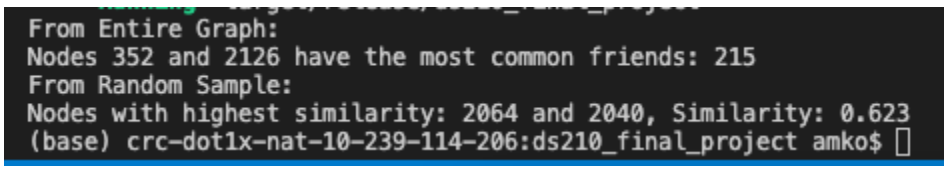
The common.rs module contains two methods: common_friends() and pair_most_common_friends(). The first method takes a graph as an argument, as well as 2 node indices. It then creates a HashSet of neighbors for the second node in order to optimize the process of checking whether another node is the neighbor of this second node. It then counts the number of common friends by retrieving an iterator over the neighbors of the first node. It then filters the iterator and keeps only the neighbors that were present in the HashSet of neighbors. To get the number of common friends, .count() is used to count the elements remaining in the filtered iterator. The second method iterates over all the pairs of nodes and calls the

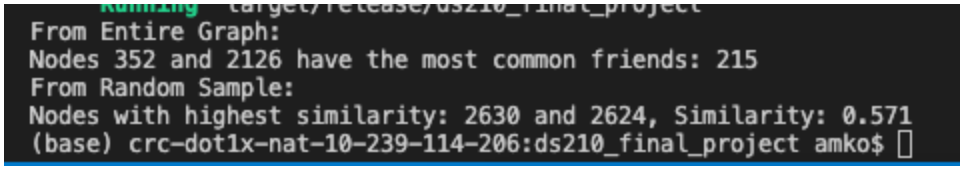
common_friends method() to calculate the number of common friends for each pair. A max_common_friends variable was initialized at the beginning of the method and set to zero. The calculated number of common friends was compared to the max_common_friends variable, and the max variable was updated accordingly. The method then returns a tuple which contains the node indices of thepair with the most common friends and the number of common friends for that pair.

The jaccard.rs module contains three methods: calc_sim(), jaccard_similarity(), and find_highest_sim(). I chose to use Jaccard similarity to measure similarity because it is useful for social networks and comparing the neighbors of a node. The Jaccard similarity provides a measure of how similar these neighbors are and indicates how many connections a pair of nodes share. The first method calculates the Jaccard similarity of two HashSets. It follows the formula for Jaccard similarity, which finds the intersection between the two sets and divides that value by the union size. The second method uses calc_sim() to find the Jaccard similarity between the neighbors of a pair of nodes in the SocialGraph. If one or both of the nodes don't have any neighbors, the Jaccard similarity return is 0.0. The third method finds the pair of nodes with the highest similarity from a random sample of the graph. In order to make the runtime more efficient, I selected a random sample of 1000 nodes from the graph. The method then iterated through all of the nodes in the sample and calculated the similarity between each pair. If the neighbor pair was empty, the similarity was not calculated. The max similarity and the corresponding node pair are kept track of and returned as Option at the then of the method.

The main.rs file utilizes the three modules as well as the methods to find the highest similarity and the pair with the most common friends. It contains a method which takes a SocialGraph and a sample size as its input and collects all the node indices from the graph. It then uses a random number generator, using the rand crate to choose the specified number of nodes randomly. It then clones these nodes and returns them as a vector of NodeIndex. This random sampling helped reduce the runtime significantly and optimized similarity calculations. In the main method, a SocialGraph is created using the methods from the graph.rs module. The pair of nodes with the most common friends is found using the pair_most_common_friends() method from common.rs, which is found based on the entire graph. The pair with the highest similarity coefficient is found by first calling the method to generate a random sample of nodes and then calling the find_highest_sim() method from jaccard.rs.

The output displays the following results:





Based on this output, Nodes 352 and 2126 have the most friends in common from the overall graph. The second screenshot shows that the nodes with the highest similarity differ each time the program is run due to the random sample. The advantage of using a random sample for the similarity calculation is that it optimizes the program and reduces the run time significantly. Prior to calculating the similarity based on a random sample, the program would take over 10 minutes to produce an output, even with cargo run –release. Now the program runs in about 30 seconds which is much more optimal.

Resources

Data Set: https://snap.stanford.edu/data/ego-Facebook.html
Similarity Measure:
https://neo4j.com/docs/graph-data-science/current/algorithms/node-similarity/#:~:text=Two%20nodes%20are%20considered%20similar,as%20the%20Szymkiewicz%E2%80%93Simpson%20coefficient.
Petgraph: https://docs.rs/petgraph/latest/petgraph/
ChatGPT was used to interpret error messages and resolve syntax issues.
Tests: https://doc.rust-lang.org/book/ch11-03-test-organization.html