Max Schultz
Professor Chator
DS210: Programming for Data Science
May 1st, 2024

Final Project Report

Project Idea: Given a social network, can we recommend a user (a node) the top k users based on connectivity and links? This is meant to somewhat model "friend suggestions" — a popular app feature utilized by various social media platforms to suggest closely related "mutual" friends.

To begin this project, I followed a simple data reading pipeline that we explored in class. Once I was able to load the facebook data, I then created my modified_bfs() function. This aspect of the project was particularly interesting. At first, I tried to implement a bfs function that would find the distance from all nodes to all; however, this was inefficient and trivial in the grand scheme of my project idea. Indeed, since my project was aiming to find relationships and user recommendations for a user-specific node, there was no need to actually implement this feature. Moreover, the recursive bfs function would have only added to the complexity of my code, and it would have inevitably led to a longer compile/ run time. In crafting this function, I utilized a VecDeque — which would output the elements in the same order — and a while loop which would cease once it reached None. I then finished crafting the algorithm — making sure it only visited unseen neighbors — and then pushed all of these results into a vector of tuples and sorted based on the second element of the tuple — i.e the degrees of separation. With this result, it was then time to craft the core of the project: the recommend_friends() function…

The recommend_friends function takes the graph, node, and max difference as arguments and returns a HashSet of friends recommendations — to avoid duplicates. To accomplish this, I had to call bfs function on the starting user and then iteratively call a degree centrality function — which, in its essence, just returns a float representation of how many nodes are connected to a user — for each result of the bfs function. I then had to create constraints so that not all close friends were returned… as this would be an immense amount, considering the amount of nodes. In this regard, my constraint was set so that the only nodes that were pushed to a vector were the nodes that had a degree centrality higher than that of the starting node… and this exception works off of an interval based on a max difference allowed. Next, I added a final constraint at the end which specifies that the true friend recommendations centrality in the graph must be greater than that of the starting node itself. You may initially think that these two constraints are the same, and thus trivial; however, these two work together to ensure that the final recommended friends not only have a path connection to the starting node, but also have centrality values within a specified range relative to the starting node. Finally, I stored these results in a hashset and selected only 10 using the .take() method — which shouldn't be a problem since hashsets are not inherently ordered.

The last two functions are merely for cosmetic purposes, as they work together in determining if a user is, perhaps, introverted or extroverted (shy or popular). The influencer function iteratively calls the degree centrality function on each node in the graph, and pushes a tuple of (node, centrality) to a vector. It then sorts the vector in ascending order, based on the centrality metric, and takes only the top three nodes which have the highest centrality. The user_importance function determines if a user is extroverted or introverted by comparing their centrality to the top three node's centrality — and this works on an interval of ±10% within the range of the top user's centralities.

In the end (i.e. in main), we end up returning user recommendations based on specified constraints, whether or not the user is introverted or extroverted, and the top three users of the network.