CS 225 Final Report

Answering our Leading Question

- We wanted to explore the different communities in the Github world. Given a collection of frequent github users, who all have a relationship between each other? We calculated this based on the number of mutual followers. It is exciting to observe how, in such a large world, different people may unknowingly be related to each other!
- There are a variety of community detection algorithms online, and the one we used is called the Girvan-Newman algorithm which is a divisive algorithm that removes edges off of a given graph, allowing us to view the “communities” in a given network.
- This algorithm utilizes “edge betweenness centrality”, which is the number of shortest paths that go through an edge in a network. Computing the shortest paths can be done using breadth-first-search (BFS).
- Once we had our subgraphs, we performed Dijkstra’s algorithm to realize the relationship between Github users with the smallest mutual followers. We did this to further our analysis of how there can be surprising connections between a group of individuals in a community.

Ultimately, we were able to achieve our objective. We have explained how each algorithm worked as intended and detailed how we verified this using test cases ahead.

Dataset Acquisition

We used the dataset from the MUSAE paper http://snap.stanford.edu/data/github-social.html
It is a collection of github users who have at least 10 starred repositories, which helps ensure that the users in the dataset are relatively active. The graph is undirected. These are all existing features of the dataset so we do not need to derive any additional data. The weights for each edge can be derived by finding the number of mutual followers between 2 users. This information is already in the dataset so it is easy to access.

Algorithms

1. File I/O (Data Parsing and Cleaning)

Functionality:
For data parsing and data cleaning, we used a relatively simple algorithm. We wrote a generalized SplitString function that would parse data from a given file based on the separator that was provided. In our case, the separator was the newline on which each GitHub user’s ID was present. Furthermore, within our data parsing algorithm, we included an additional function to make an Adjacency Matrix for our graph.

We wrote several test cases for the same. The assumption we have made is that the relationships provided between nodes is correct. Therefore the only scope for “bad data” is when there are missing entries. Missing entries can be either in the ID, or in the weights column. If there is a missing ID, then we will ignore that relationship and move on, because there is no way to know which 2 users have a relationship if their IDs are not included. If there is a missing/negative weight, we will assign a weight of 1, as for 2 users to be connected they must

have at least one mutual follower. Additionally, via a try-catch structure, we have accounted for any malformed data that might be present too.

Runtime Analysis:
Since there are n nodes, loading the CSV file will take O(n) time. We will use a 2D vector to store node and edge data. Therefore the space complexity will be O(n) + O(m) where n and m are the number of nodes and edges respectively.

2. Breadth First Search (BFS)
Functionality:
While the BFS implementation to traverse a graph is well-known, we will implement BFS such that it incorporates “betweenness centrality” too. Betweenness centrality “captures how much of a given node is in-between others” i.e the number of shortest paths (from a given node to all other nodes) pass through the target node. Here, the implementation of BFS with betweenness centrality enables the highlighting of certain nodes - nodes (which represent GitHub users) that only have multiple repositories, but also that collaborate on multiple repositories. This enables us to identify prominent, highly active users effectively without significantly modifying the traditional BFS implementation.

Function Output:
The BFS implementation will output two things - a queue and a vector. The queue will effectively contain all the nodes of the graph which, during the implementation, were neighbors to the node being explored. As for the vector, it contains all the nodes that have been visited (and the order in which they have been visited). Two additional helper functions will be used for testing purposes. Namely, printVertices and printEdges.

The main thing we tested was the BFS pathing i.e given an adjacency matrix, does the BFS algorithm traverse the nodes in the correct order? Additionally, we included test cases to add edges to the graph and test that out too.

Runtime Analysis:
The total runtime of BFS will be the sum of the individual components of the implementation. First, marking every vertex in the graph as unexplored and later checking whether or not it is explored will take O(n) time each. Likewise, to mark every edge as unexplored it will take O(n) time. For the main traversal loops, O(n) will be taken to get through the queue in the worst case event that every vertex is added to it. Similarly, in the case of acquiring neighbors, the worst case will be the sum of the degrees of all the nodes which is O(2*m). Therefore, the total runtime will be the summation of the aforementioned which is effectively O(n + m). Improving or making further optimizations doesn’t seem feasible here. However, on the plus side, within this worst case runtime of O(n + m), we will also compute the betweenness centrality.

3. Dijkstra's Shortest Path

Functionality:

This is a very famous algorithm that is used to find the shortest path, given a node, to all nodes in a graph. We're using this algorithm to show users the shortest mutual follower connection between a user in a given community (we will perform community detection before this) and users in the same community.

The input to the function will be a graph and a nodeID. Any node's nodeID will be passed in. Note that all IDs are unique.

Function Output:

We plan to return a vector of nodes in a certain order. It contains the node that precedes each node in our shortest path so that we can reconstruct the shortest path for each node.

Runtime Analysis:

Dijkstra's algorithm has worst case complexity of O((|V| + |E|) log |V|) where V represents the node set and V represents the edge set. However, we improved the runtime performance of Dijkstra's algorithm using a priority queue, instead of a regular queue, to store the nodes that have been visited. The inner loop operations take O(V + E) and the decrease key method takes O(log(V)).

4. Girvan - Newman Algorithm

Functionality:

The Girvan–Newman algorithm is a greedy algorithm for finding communities in networks. It works by progressively removing edges from the original network until there are no more edges left. The communities are then the connected components of the remaining nodes. Edge centrality can be used in Girvan-Newman community detection by finding the edges that are most central to the community and using them to split the community into two.

The Girvan Newman algorithm we are using utilizes a graph as an input, on using this graph it finds the highest valuable edges and iteratively deletes these edges until set and discernible clusters are formed. We are using a graph input since the dataset has directed edges which can be utilized for edge betweenness centrality.

Function Output:

The function is a divisive algorithm and modifies the original graph, hence this is a void function. This does not return anything but essentially deletes edges within the original graph to form specific clusters or communities. Afterwards, different users can note the distinct relations within connected components within the main graph. The graph data structure will remain the same

Runtime Analysis:

There is no definitive answer to this question as the efficiency of the Girvan-Newman method can vary depending on the specific application.

The overall runtime of the Girvan-Newman algorithms varies greatly depending on the application, however, in our case we analyzed it with respect to our community detection implementation. Despite Girvan-Newman's popularity and quality of community detection, it has a high time complexity, increasing up to $O(m^2n)$ on a sparse graph having m edges and n nodes. As a result, Girvan-Newman is generally not used on large scale networks. Its optimal node count is a few thousand nodes or less. Hence we are utilizing this over our smaller dataset as opposed to a Louvain algorithm which would have been used for datasets with hundreds of thousands or even millions of data points.