Analysis of Community Detection Algorithms

I. INTRODUCTION

Context

Caroline Craig

Social media platforms have become increasingly important spaces for consuming and sharing news. Though they are often thought of as a modern, digital ‘public square’, in reality these spaces are networks that are structured and mediated. I am interested in understanding the dynamics that impact the spread of information online. For example, under what conditions is a piece of news more likely to become viral? One such dynamic involves the existence of communities in social media networks. The first step in studying the formation of communities is detecting them. I hope that learning about community detection algorithms through this project will shed light on questions such as, what do detected communities look like within a graph? Do they occupy a particular position that shapes the role they play in the spread of information?

Faith Nassiwa

Community detection from a graph perspective is the process of identifying vertices that are closely linked to each other forming subgroups of clusters/communities in a graph. Due to the existence of communities in multiple real-world applications like social networks, and biological networks, there is a need for efficiently identifying communities to perform targeted tasks on them. For example, in social media applications, community detection helps in identifying communities to target advertisements based on common interests or traits which can reduce advertising costs while maxing profits for companies as they reach out to very specific communities that are most likely to use or buy their products. I am excited to work on this topic as it will show me the applicability of what we have been studying including but not limited to the application of graph theory and algorithm strategies used in designing community detection algorithms.
For the project, I have used the Louvain method for detecting communities in the social circle dataset of Facebook from the Stanford Network Analysis

Anshul Rao

With the ever-increasing size of social media data, finding communities within social networks is an arduous task that has a wide range of applications. For example, it is a part of many recommendation algorithms in order to discover people with similar interests. This is based on the assumption that like-minded people would prefer to watch similar movies, buy the same kind

of items, etc. This was my motivation behind working on this project as finding communities in a network is one of the most basic and critical problems that need to be tackled in implementing many advanced algorithms like targeting customers, finding conniving groups in a social network, etc.

Question

In simple terms, finding communities is equivalent to locating "locally dense connected subgraphs" [Barabási, 2015, section 9.2] in networks. For this project, we will analyze, implement and compare different algorithms for finding communities in social networks. To validate our results, we will visualize them and qualitatively assess the density of community partitions.

Scope

In this project, we plan to focus on three community detection algorithms, namely, Girvan–Newman algorithm, Louvain method and hierarchical agglomerative clustering, using the average link method to calculate inter-cluster distances. We will implement these algorithms in Python and compare the performance of all three on the same social network data. The data that we have used is the social circle dataset of Facebook from the Stanford Network Analysis Project. It is an undirected graph that has around 4k nodes and 88k edges.

Name	Type	Nodes	Edges	Description
ego-Facebook	Undirected	4,039	88,234	Social circles from Facebook (anonymized)

Method

The members picked one algorithm each as follows:

- Anshul Rao: Girvan-Newman Algorithm
- Caroline Craig: Hierarchical Agglomerative Clustering, using average-linkage distance
- Faith Nassiwa: Louvain Method implemented using the Python-Louvain Module

Each member has individually implemented the algorithm in Python.
The implementation can be found here: https://gitlab.com/cs5800-algorithms/community-detection.

II. ANALYSIS

1. Girvan-Newman Algorithm

Algorithm:

Step#1: Compute the edge-betweenness centrality values of all the edges.

Step#2: Remove the edge(s) with the highest edge-betweenness centrality values from the network.

Step#3: Find connected components in the network using DFS (Depth-First Search).

Step#4: Repeat Step#1 to Step#3 until the graph breaks into two or more connected components, i.e., communities.

Variations:-

Step#4: After removing edge(s) with highest edge-betweenness centrality values, calculate modularity Q (fraction of edges within communities), which lies between the range [-0.5,1] and keep repeating Step#1 to Step#3 till the modularity keeps increasing.

= = OR = =

Step#4: Step#1 and Step#2 are repeated until no edges remain in the graph.

The end result is a dendrogram produced from the top down (i.e. the network splits up into different communities with the successive removal of links). The leaves of the dendrogram are individual nodes.

How is edge-betweenness centrality value computed?

1. Select a node as root and perform BFS to find the number of shortest paths from root to every other node. Let these values be stored in a dictionary paths_count.
2. Starting from the leaf nodes we calculate edge score as:

   $$((\mathbf{1} + \sum \boldsymbol{edge\ score\ to\ node\ u}) \,/\, \boldsymbol{paths\_count}[\boldsymbol{u}]) * \boldsymbol{path\_counts}[\boldsymbol{v}]$$

   where u is the source/child node and v is the destination/parent node.

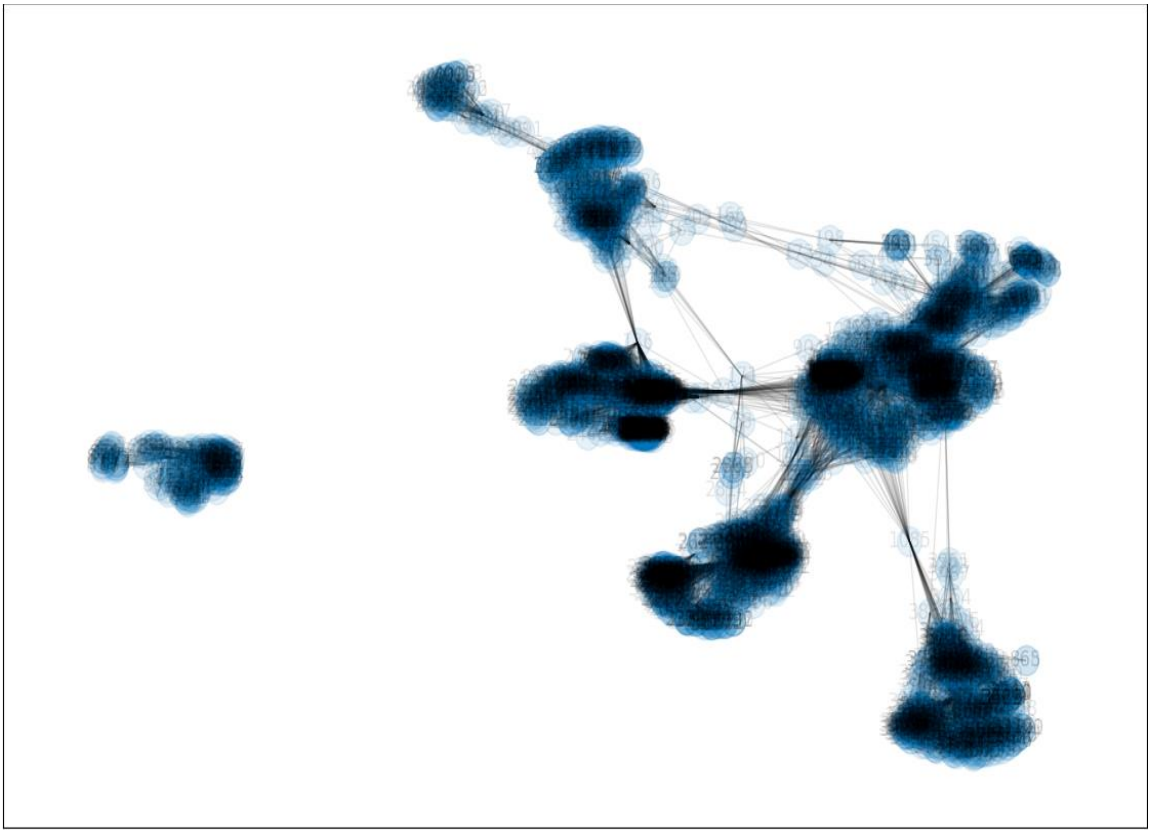
3. Sum of edge scores for all edges considering every node as the root once and then divide them by 2.

4. The resultant edge scores are edge-betweenness centrality scores.

Runtime Analysis: One run from Step#1 to Step#3 takes $O(V^2 + EV)$. In the worst case, we will remove one edge at a time in a complete graph until it breaks into at least two components which makes this algorithm very slow. Due to the high time complexity of this algorithm (and the need to recompute edge-betweenness values) sometimes greedy algorithms are preferred over this since although they might not give most accurate results, they will still be faster and more efficient.

The source code for Girvan-Newman algorithm can be found here: https://gitlab.com/cs5800-algorithms/community-detection/-/tree/main/Girvan-Newman

Facebook Social Network broken into two communities using the Girvan-Newman Algorithm implemented as part of this project:-



2. Hierarchical Agglomerative Clustering

Given that there are many implementations of hierarchical agglomerative clustering, we chose to refer to the Ravasz algorithm, which was developed for use in networked data, specifically to model the hierarchical structure of "topologic modules" in metabolic networks [Ravasz et al. 2002]. In addition to following the algorithm's steps, we also borrowed its formula for computing similarity within a network:

$$O_{\mathrm{r}}(i,j) = \frac{J_{\mathrm{n}}(i,j)}{min(k_{\mathrm{i}}, k_{\mathrm{j}})}$$

where O is the "topological overlap" or similarity matrix, i and j are nodes, k is their degree, and J is the number of common neighbors between nodes i and j (to which we add one if there is a direct link between i and j). Following [Barabási 2015, section 9.3], we added smoothing to this formula:

$$O_{\mathrm{r}}(i,j) = \frac{J_{\mathrm{n}}(i,j)}{min(k_{\mathrm{i}}, k_{\mathrm{j}}) + 1 - \Theta(A_{\mathrm{ij}})}$$

where Theta(x) is the Heaviside function, or Theta(x) is zero if x is lesser than or equal to 0, and one if x is greater than 0.

Algorithm

Step #1: Build the similarity matrix, using the formula provided above
The matrix is a symmetric nxn matrix, where n is the number of nodes in the network.

Step #2: Convert the similarity matrix to a distance matrix
The range of values in the similarity matrix range from 0 to 1, with many ties for most nodes. In hierarchical agglomerative clustering, once a cluster has been formed, it cannot be unformed. Therefore, it is important to choose the best merging at the outset, taking into account the similarity not only between two individual nodes, but also between the two clusters' similarities to the other nodes in the network. To this end, we treated each column of the similarity matrix as a vector representing each node's multi-dimensional similarity with every node in the network, and computed distances between nodes in this high-dimensional space. We used the cosine distance as distance metric and SciPy's implementation:
`sklearn.metrics.pairwise.pairwise_distances(similarity_mat,metric='cosine')`

Step #3: Perform hierarchical clustering
Assign each node to its own cluster and merge nodes that are closest to each other into a single cluster.

Step #4: Recompute the distance matrix between each pair of new clusters, using the average-linkage method
The average-linkage method computes the distance between clusters A and B by first computing the distance between every pair of nodes in (A, B) and then taking the average of these distances. It is less sensitive to noise and outliers, when compared to the single-linkage method (distance between the closest pair of nodes) or the complete-linkage method (distance between the farthest pair of nodes).

Step #5: Repeat steps #3 and #4 until all nodes have been merged into a single cluster and keep track of the hierarchical relationships between clusters in a tree structure

Runtime Analysis
Step #1: $O(n^2)$ on average, given that the number of edges per node is not close to n, $O(n^3)$ in the worst case
We built the similarity matrix using the following objects:
dict_neighbors: {'node':[neighbor1, neigbor2, ...]}. For every node, go through list of edges and extract its neighbor nodes. Runtime: $O(n^2)$
common_neighbors: nxn matrix with number of common neighbors per (node1, node2) pair. Runtime: $O(n^2)$
degree_array: 1-D array with degree per node. Runtime: O(n)
To build the similarity matrix: for every pair of nodes, extract their common neighbors, check whether there is a direct link between them, and get each node's degree from degree_array. Runtime: O(n*n*num_edges)

Step #2: for each pair of node similarity vectors, compute the cosine distance between them. Runtime: $O(n^2)$

Step #3: for every node, find the minimum distance in its distance vector, then merge the nodes that are closest to each other. If there are ties, merge all nodes into one cluster at once. Runtime: O(n)

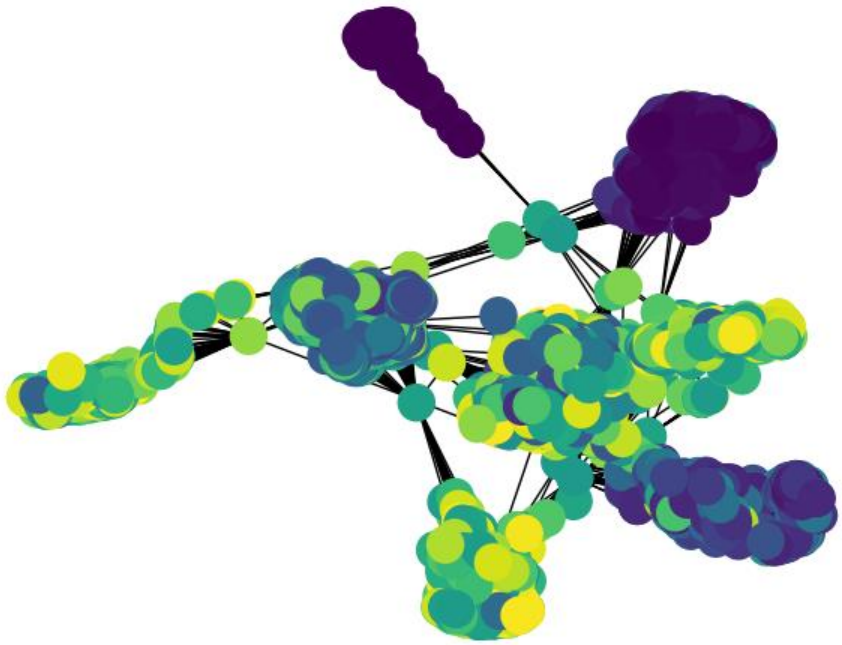
Step #4: for every pair of clusters (A,B), compute the distance (a, b) between every node a in A and node b in B, then take the average of these distances. Runtime: $O(num\_clusters^2)$, which is less than $O(n^2)$ given that the number of clusters continually decreases with every clustering

pass. With each clustering pass, save the new clusters in an array containing a dict per cluster run ({‘cluster_label’: [cluster_members]}).

Step #5: repeat steps #3 and #4. Runtime: O(height_tree * (n+ n^2)) = O(n^2), since the height of the tree is not close to n

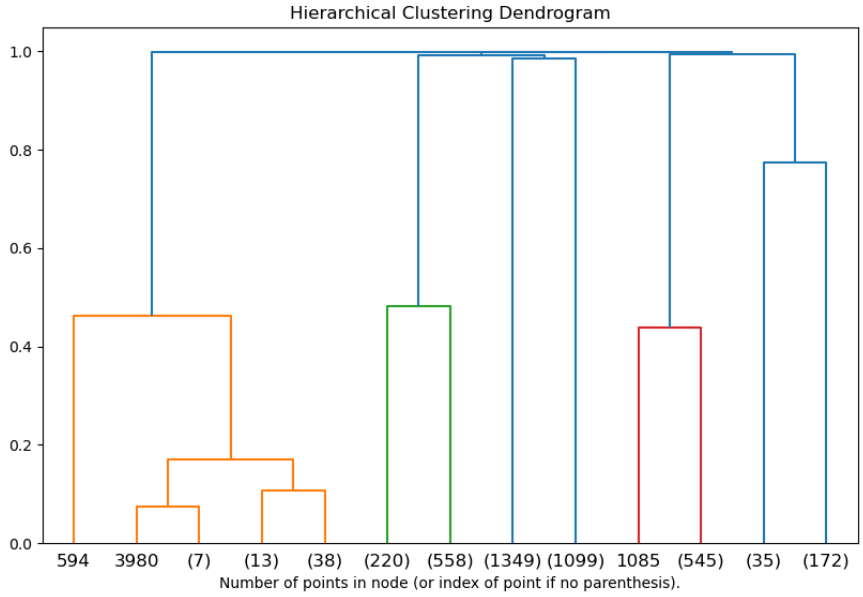
Total runtime: O(n^2 + n^2 + n + n^2 + n^2) = O(n^2)

Please note that we could not get our code for Step #3 to work, due to issues with the tree object that is returned after every clustering pass. In order to produce a completed algorithm, we therefore passed our distance matrix into Scikit-learn's agglomerative clustering module.1 Our original code can be viewed on our GitLab here: https://gitlab.com/cs5800-algorithms/community-detection/-/blob/main/hierarchical_agg_clustering/hac.py

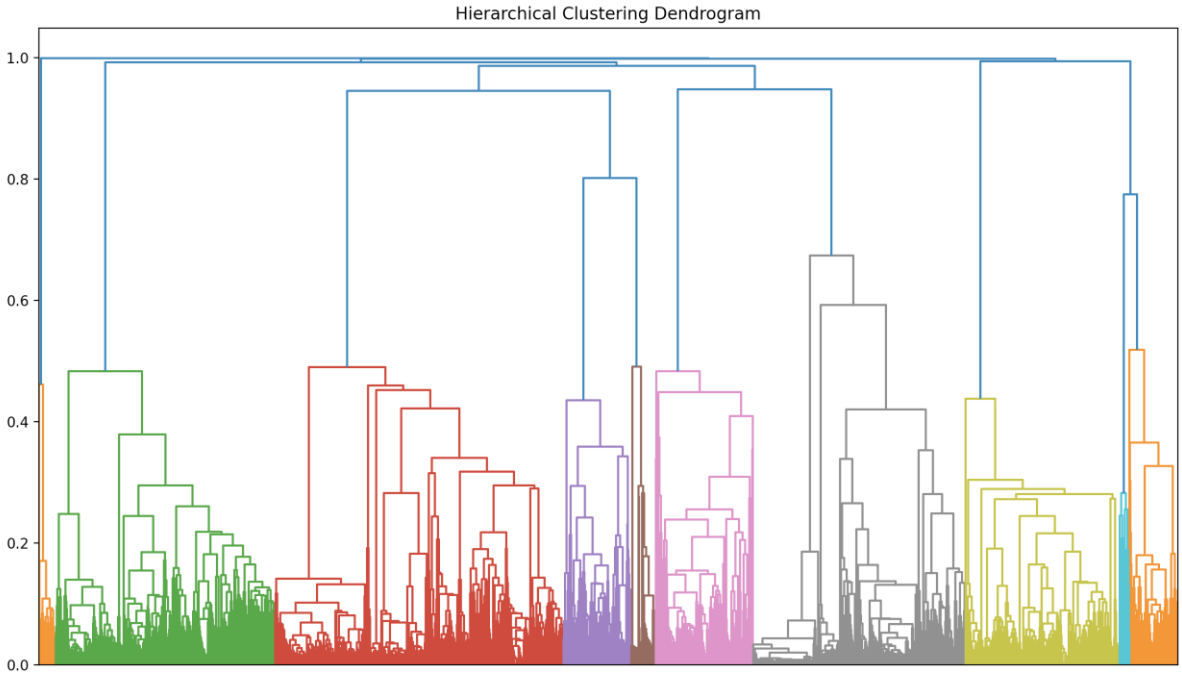


Dendogram showing top three levels of the tree:

[1] sklearn.cluster.AgglomerativeClustering(n_clusters=2, *, affinity='euclidean', memory=None, connectivity=None, compute_full_tree='auto', linkage='ward', distance_threshold=None, compute_distances=False



Dendogram showing all levels of the tree:



3. Louvain Method

The Louvain method is an unsupervised hierarchical clustering method/algorithm for extracting communities from large networks created by Blondel et al. from the University of Louvain (the source of this method's name). The method is a greedy optimization method that maximizes the modularity of a community partition by continuously moving nodes until an optimal community partition is obtained. Modularity is a value ranges(-1, 1), that measures the density of links inside communities compared to links between communities.
Its runtime complexity on the worst case is loglinear O(n logn) but can also be implemented to be linear O(m + n) where n is the number of nodes in the network and m is the number of edges.

The algorithm works in mainly 2 steps:-

Step 1

It initializes every node to be in its own community e.g. {'community_1': node_1, 'community_2': node_2, … 'community_n': node_n} and then for each node it tries to find the maximum positive modularity gain by moving each node to all its neighbor communities. If no positive gain is achieved, the node remains in its original community. This continues until no individual move can improve the modularity.
The modularity of moving an individual node i into a community C is computed using the formulas below: -

- For undirected graphs

$$\Delta Q = \frac{k_{i,in}}{2m} - \gamma \frac{\Sigma_{tot} \cdot k_i}{2m^2}$$

Where Q is the modularity, m is the size(sum of edge weights) of the graph, $k_{i,in}$ is the sum of the weights of the links from i to nodes in C, k_i is the sum of weights of the links incident to node i, $\sum_{tot}$ is the sum of the weights of the links incident to nodes in C and γ is the resolution parameter.

- For directed graphs

$$\Delta Q = \frac{k_{i,in}}{m} - \gamma \frac{k_i^{out} \cdot \Sigma_{tot}^{in} + k_i^{in} \cdot \Sigma_{tot}^{out}}{m^2}$$

Where k_i^{out}, k_i^{in} are the outer and inner weighted degrees of node i and Σ_{tot}^{in}, Σ_{tot}^{out} are the sum of in-going and out-going links incident to nodes in C.

Step 2

This step is for building a new network whose nodes are now the communities found in step 1. It merges the nodes of the same community and builds a new network whose nodes are communities.
The two steps are repeated iteratively until no additional modularity gain is achieved or is less than the threshold.

The output of the algorithm gives several partitions. The partition found after the first step typically consists of many communities of small sizes and as it progresses to subsequent steps, larger and larger communities are found due to the aggregation mechanism.

What I did and why?
I decided to use python modules to detect communities in the dataset since they are well developed, optimized, and rigorously tested to give optimal results.
Below is the list of the key modules that I used:

- Pandas – to read the dataset file and put it in an easy to manipulate form.
- Python-Louvain – to detect communities using the Louvain method.
- Networkx – to create and analyze a standard network graph.
- Matplotlib – to visualize the data.

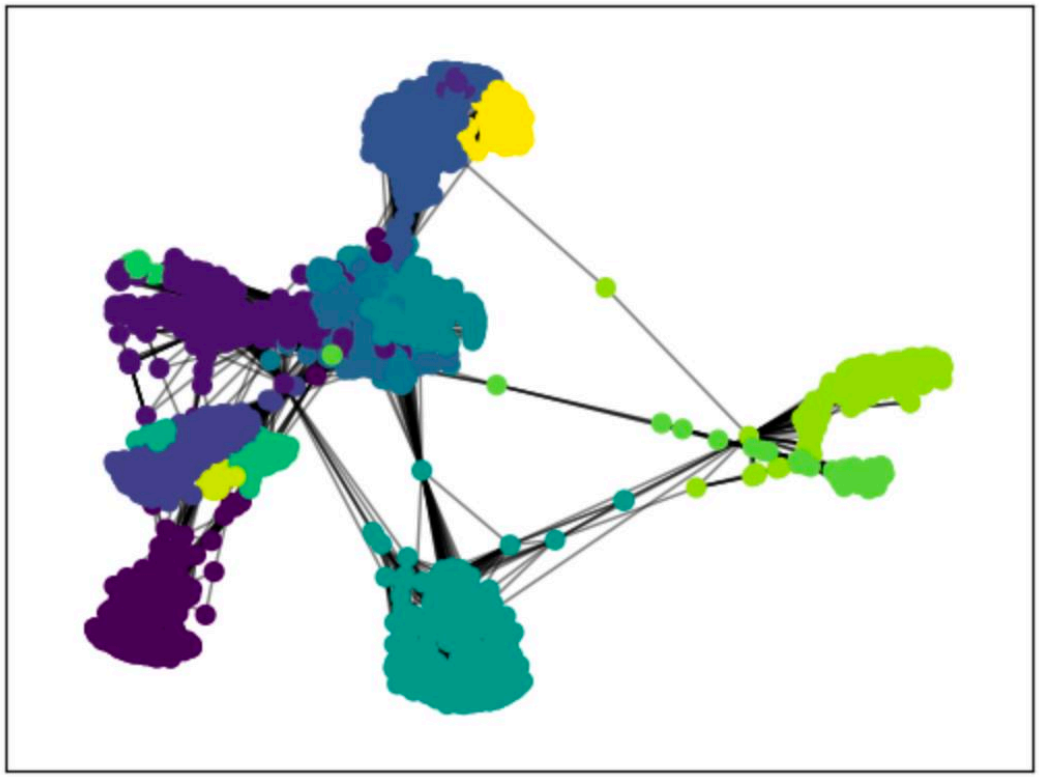
Clear steps throughout every step of my analysis and reference to specific topics or modules covered in the course

1. Generating the network graph, I wrote a function to read the Facebook text data file and extract a list of edges which I used to create a graph/original network. This function runs in O(c + r) time where c is the number of columns and r is the number of rows in the data frame.

2. Partitioning the network, I used the best.partition() method from the python-louvain module to get partitions from the original network. The module implements the Louvain method using a greedy strategy of maximizing the modularity at every iteration when moving nodes and merging them into communities.
   The average computation runtime of the partition_graph() function on the social circle dataset of Facebook is approximately 2 seconds and the module

3. Analyzing and visualizing the new network, I implemented the communities_count method to get the number of communities detected, graph_modularity() to get the maximum modularity obtained, graph_density() to get the density of the original network and visualize_network() to plot the communities detected in the graph.

The algorithm identified 16 communities in approximately 2 seconds in the network with a maximum modularity of approximately 0.834. Below is the visualization of the communities in the new network graph.



I applied the knowledge gained in the Complexity, Data structures and Graphs, Greedy Algorithms and Network Flows modules throughout the analysis and implementation of my python program/module and during the literature review and report documentation.

Program / Source code

The project source code that implements the Louvain method for detecting communities in the social circle dataset of Facebook can be accessed publicly here: https://github.com/faithNassiwa/louvain_community_detection/blob/main/Louvain/main.py.

The above python file can be run in your favorite integrated development environment(IDE), but you need to install all the required modules to not get any issues with missing module. A list of all the dependencies can be accessed here.

III. CONCLUSION

The Louvain algorithm takes fewer computation time per iteration compared to the other two algorithms at a worst time complexity of O(n log n).

Despite Girvan-Newman's popularity it is computationally expensive and takes a long time to run and hence is not generally used on large scale networks. While it does give accurate results but due to its high time complexity, faster (though less accurate) algorithms like Louvain method are preferred over Girvan-Newman especially for larger datasets with millions of nodes.

Hierarchical clustering has several strengths. Firstly, it does not require the user to make any a priori assumptions about the optimal number of clusters. Second, when present, it uncovers a hierarchical structure between nodes in a graph. This is a feature that can be particularly interesting in the context of social networks in that it can help identify the specific roles that different nodes play in the network, for example in the sharing of viral information online. On the other hand, its runtime is slow, which is a limitation when studying large networks.

We would like to use benchmarks to quantitatively evaluate our results, such as the Lancichinetti-Fortunato-Radicchi (LFR) benchmark and the Girvan-Newman benchmark [Barabási, 2015, section 9.6].

Faith Nassiwa

I learned the usage of two new python modules that I had never used before which are the Networkx and Python-Louvain. I plan on continuing to use them for graph related work and want to explore more functions in them. I also want to further analysis the communities that were detect as I did not get to complete this task due to a tight project deadline. I will get to do it either as a personal project or in another future course.

Caroline Craig

This project allowed me to learn about different community detection algorithms. I plan to continue to explore these techniques while applying them to the study of information sharing in social networks. In particular, I would like to learn how to combine community detection with studying the specific role that different nodes in a network play in the sharing of information. One idea is to apply the Louvain algorithm for community detection on a Twitter user network and a hierarchical, density based clustering algorithm on the text of their tweets. This would allow me to leverage the speed of the Louvain algorithm while still obtaining information about any hierarchical structure in the network.

Anshul Rao

While working on this project I was able to learn about different community detection techniques that exist and the pros and cons of each. I am sure that the algorithms and other nuances I have learned in this project will be advantageous for me when I work on any application of community detection like building a recommendation system. I would be in a much better position to select the most suited community detection algorithm for my use case or rather even implement my own algorithm.

REFERENCES

1) Albert-László Barabási, *Network Science*. 2015. Accessed July 22, 2022. http://networksciencebook.com/

2) 2021. *Girvan–Newman algorithm.* July 22. Accessed July 21, 2022. https://en.wikipedia.org/wiki/Girvan%E2%80%93Newman_algorithm.

3) 2022. *Louvain method.* June 27. Accessed July 21, 2022. https://en.wikipedia.org/wiki/Louvain_method.

4) Ravasz, Erzsébet, Audrey Somera, D A Mongru, Zoltán N. Oltvai and A.-L. Barabási. "Hierarchical Organization of Modularity in Metabolic Networks." Science 297 (2002): 1551 - 1555.

5) 2022. Louvain Communities. Accessed August 17, 2022. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.community.louvain.louvain_communities.html - rd180461fb6d3-1

6) 2022. Louvain Method: Finding communities in large networks. Accessed August 17, 2022. https://sites.google.com/site/findcommunities/

APPENDIX

https://gitlab.com/cs5800-algorithms/community-detection