Community Detection from Betweenness Centrality Based on Random Walks
MATH76 Final Project Report

Zac Nelson-Marois, Matt Landry, and Ben Pable
Dartmouth College
Hanover, NH 03755
zachary.t.nelson-marois.24@dartmouth.edu matthew.r.landry.24@dartmouth.edu benjamin.e.pable.25@dartmouth.edu

Abstract

In this paper, we propose a novel approach for community detection which involves calculating the betweenness centrality based on random walks for each node. We considered the paths traversed by each random walk and identified nodes with high random-walk based betweenness centrality as potential community boundary nodes, or bridge nodes. We then ran a modified breadth-first search on each bridge node based on a specified threshold, where each breadth-first search formed independent communities by not visiting the other bridge nodes. To evaluate the success of this algorithm, we executed our algorithm on networks of varying sizes and types.

After implementing our initial algorithm, we discovered multiple issues, the primary of which being that the runtime of the betweenness calculation based on random walks was massive. We thus modified our algorithm to exclude calculating the betweenness for all nodes and furthermore reduced our network sizes before using the algorithm. We additionally implemented alterations to our algorithm to account for cases where communities overlapped and where communities had a lack of strong connectivity, thereby resulting in no community assignment whatsoever.

In comparing our finalized algorithm with another commonly used community detection algorithm, Louvain best partition, the results indicated our algorithm was undoubtedly capable of precisely identifying community structures in a similar manner to established techniques. This is due to the random walks capturing an increased proportion of localized structures (especially among larger networks) which were properly sorted due to using high betweenness to select community boundaries.

Our algorithm's ability to correctly select communities, particularly for larger networks, suggests that our methodology has many propitious implications as to warrant possible future exploration in the research area of community detection optimization and network analysis.

1. Introduction

1.1. Background

Community detection is a fundamental issue with regard to network analysis that looks to detect communities of nodes which display dense inter-community and sparse outer-community linkages. The multi-disciplinary applications of community detection are crucial to conceptualizing the structure, dynamics, and behavior of the world around us, and provide valuable insights for which individuals, groups, and corporations currently base their decisions upon. Community detection has vast implications, including uncovering patterns of interaction and information flow, ameliorating technological concerns involving personal recommendations and network security, and simplifying the visualization of large-scale networks. Community detection is paramount to various network domains, including:

- Social - co-authorships, online communications, sports/politics/individual interactions
- Biological - Food webs, brain networks, protein interactions, gene regulation
- Economic - investor correlations, stock networks, movie/song ratings, global companies
- Technological - electrical transmission networks, library dependencies
- Transportation - road/airport/public transport networks, traffic data

1.2. Purpose

Over the years, several community detection algorithms have been developed, each with its own strengths but also significant limitations. Some of the most common implementations we reviewed included:

- Louvain - Enhances the distinction between the observed and anticipated number of edges by utilizing modularity to measure the quality of node allocation. This approach assesses the density of connections within a given network relative to the level of connectivity expected in a random network. The Louvain algorithm is constrained by having a low resolution limit in that it cannot detect smaller communities within larger networks, partially as a product of relying

on modularity optimization. Furthermore, the Louvain algorithm is sensitive to the initial partitioning of the nodes, such that multiple runs with different initializations are often required and it often discovers internally disconnected, weak communities.

- Leiden - Identical to the Louvain algorithm but also utilizes a refinement phase which further splits the partitions in this added phase via merging nodes with randomly chosen communities. This functions as added randomness which allows for the discovery of the partition space with a larger scope. Furthermore, Leiden only visits nodes where the neighborhood has changed. The primary downside is that a modularity-based approach adds severe limitations which result in smaller communities not being detected accurately.
- Girvan-Newman algorithm - Identifies the most frequently occurring edges between pairs of nodes by computing betweenness centrality. It removes the edges with the highest betweenness and recalculates the betweenness for the remaining edges. However, this approach becomes impractical for large datasets due to its computational complexity of O(mn^2), where m represents the number of nodes and n represents the number of edges. As a result, the algorithm loses its usefulness when the number of nodes exceeds a few thousand.
- Infomap - Utilizes Huffman encoding to minimize a cost function by assigning varying length codes to nodes according to their visitation frequencies in an infinite-length random walk. The partitions are determined by the flow generated from the connectivity patterns within the network. While this approach yields high-quality communities, it lacks scalability when dealing with large graphs, making it impractical for processing sizable datasets.

As is noticeable with the aforementioned community detection algorithms and as is common for the majority of community detection algorithms that currently exist, there are constraints which limit the effectiveness of the results, the main issue being that smaller communities are incapable of being fully captured due to the sheer size of the networks being viewed. Our goal, therefore, is to mitigate this concern by creating our own algorithm that takes into account the favorable components of past community detection algorithms and minimizes the errors present within them as well.

1.3. Approach

Before beginning our algorithm implementation, we knew that we wanted to find primarily medium to large sized datasets for testing as to meet our goal of accurately representing smaller communities when there are a significant number of nodes and edges in a given dataset. We chose to incorporate four datasets in total for eventual testing, and when possible, identify ground-truth labels to allow for comparison between the predicted labels of our algorithm, as summarized below:

- Email-EU-Core Network (EEC) - This core email data from a European research institution consists of communications solely between members of the institution. The data includes emails exchanged among 42 different departments within the institution, which are considered as the ground-truth communities. These departments represent distinct administrative groups within the institution.
- Amazon Product Co-Purchasing Network (APC) - Consists of Amazon product categories and their associations based on the "Customers Who Bought This Item Also Bought" section. The dataset includes 5000 of the most prominent product categories, which are considered ground-truth communities. These categories represent various types of products available on Amazon, and they serve as a basis for understanding customer preferences and purchasing behavior. The co-purchasing network provides insights into the relationships between different product categories, indicating which categories are frequently purchased together by customers.
- Facebook Food Page Network (FFP) - Represents a collection of pages on Facebook dedicated to food-related content. We were not able to explicitly identify ground-truth labels without assigning them arbitrarily, as there were no obvious communities for assignment.
- Zachary's Karate Club Network (ZKC) - Used purely for the purpose of benchmark analysis. Even with it being such a small dataset, it provides a more simple visualization of how our algorithm is implemented in relation to other more common algorithms. Each edge in the dataset represents relationships, such as friendships or training partners. The network is so intriguing because there was a conflict between the instructor and club president which led to two communities being formed in an approximate halfway split.

Original Network Sizes

	EEC	APC	FFP	ZKC
Nodes	1,005	334,863	620	34
Edges	25,571	925,872	2,102	78

1.4. Scope

Our original intention was to use even larger datasets. For example, one of the original networks we looked into was a Youtube groups network with 1,134,890 nodes and 2,987,624 edges. However, after running our algorithm, it was evident that the run time was excessive and would take weeks to run. Generally, as the size of a network increases, the runtime of an algorithm increases in an exponential manner due to the algorithm processing more nodes and edges and thus increasing computational complexity. Thus, due to the potential room for error when considering the Youtube network, we omitted it from our data analysis. Furthermore, we were required to reduce the size of our Amazon network due to the colossal number of nodes and edges (which we will clarify in more detail later).

1.5. Structure

This paper will contain the following sections:

- Problem and Method
- Experiments and Results
- Conclusion
- Contribution
- Citations

2. Problem and Method

2.1. Problem

The problem that we explicitly wanted to address was how we could maximize the number of intricate communities within a given larger-sized network, using another well-known algorithm as a baseline for comparison (we chose the Louvain algorithm due to its commonality). In order to do so, we formulated an algorithm which considered components of past algorithms, specifically the utilization of betweenness centrality from the Girvan-Newman algorithm and the frequencies of visitations by random walks from the Infomap algorithm. As mentioned previously, this problem is significant because past algorithms have not been exceptionally helpful at determining the true number of communities within larger networks, which is very important for identifying key influencers, revealing more detailed insights about the structure of a given network, and detecting niche communities as well as anomalies. Based on this information, our proposed solution involves the utilization of betweenness centrality based upon random walks.

2.2. Betweenness Centrality

Our method uses betweenness centrality (based on random walks, which will be explained more later) in comparison to other metrics such as modularity or clustering coefficients. Betweenness centrality quantifies the degree to which a vertex occupies a strategic position along paths connecting other vertices in a network. Vertices with higher betweenness centrality have a greater capacity to control the flow of information between various parts of the network, thereby resulting in a more significant influence within the system. Removing these nodes from the network throws into disorder the communication between other vertices, as they occupy key positions along the largest number of utilized paths.

Betweenness centrality distinguishes itself from other centrality measures in that a vertex can have a low degree and also be connected to other vertices of low degree or have distant connections yet still have a high betweenness. Furthermore, bridge nodes, which serve as connectors between separate groups of vertices, possess high betweenness centrality, as numerous paths within each group rely on these bridges, even in cases of limited connectivity. This allows our algorithm to uncover subtle community structures by identifying important intermediary nodes, even when there is low overall connectivity, which most definitely is true for our reduced networks.

Betweenness centrality is based upon two fundamental assumptions: (1) all node pairs exchange information with equal probability, and (2) information transmission follows the unweighted shortest path between nodes, which is randomized when multiple shortest paths exist. This is yet again beneficial for our algorithm, as the networks we are considering relate to the information flow between members and identifying the vertices which most frequently control this information flow will allow us to create accurate communities.

The largest problem that we encountered using betweenness centrality is the run time. We employed a breadth-first search method starting from a single source vertex, which had a time complexity of $O(n + m)$ (where n represents the number of vertices and m represents the number of edges). However, when we repeated this process for all source vertices in the graph, the overall cost escalated to $O(n*(n + m))$. Considering that graphs

typically have a higher ratio of edges to vertices, the time complexity can be approximated as O(mn + n^2) or O(n^3) for a sparse graph, resulting in a quadratic cost. In order to deal with this, we reduced our network sizes to minimize this penalty, yet also made sure to keep the network size large enough to answer our intended question.

2.3. Random Walks

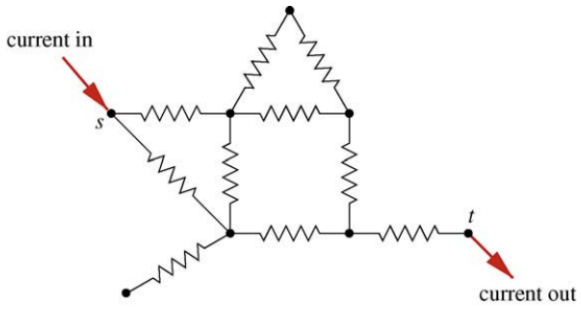
Our method simultaneously combines the concept of betweenness centrality with random walks. A random walk is a stochastics process used to determine the position of a point based upon randomized movements, for which each step has an equal probability of moving in a particular distance as well as direction. The process is characterized by a Markov process such that future movements are independent of past movements of a particular random walk.

Though betweenness centrality is effective with regard to measuring the influence of a node with regard to how much it is capable of spreading information, it relies on the hypothesis that information only spreads via these shortest paths, and doesn't consider all of the potential paths for nodes. However, we still want to assign more weight to the shorter paths, so the idea of betweenness centrality is heavily relied on in our algorithm, but this hypothesis is most certainly loosened.

Our method proposes a new measure of centrality quite similar to betweenness centrality, but of course, relying on random walks, which we can signify as random-walk betweenness. Specifically, the random-walk betweenness quantifies the number of times a random walk, starting at one vertex (s) and ending at another vertex (t), passes through the vertex of interest (i) along the way. This measure is averaged over all possible starting and ending vertex pairs. In a network where information spreads randomly until it reaches its intended destination, the random-walk betweenness centrality considers contributions from various paths that may not be optimal. However, shorter paths generally have a higher impact compared to longer ones because random walks are unlikely to become excessively long without reaching their target. So in essence, while the traditional betweenness centrality measures the number of shortest paths passing through a vertex, the random-walk betweenness centrality takes into account the number of times a vertex is encountered during random walks between all pairs of vertices, which is significant for handling sparse connectivity and capturing indirect influence of vertices.

2.4. Algorithm Implementation

Our algorithm closely resembles that proposed by M.E.J. Newman in his paper "A Measure of Betweenness Centrality Based on Random Walks". Newman defines random-walk betweenness according to a current flow analogy, in which we assume a given network is represented by an electrical circuit with which each edge contains a unit resistance, as depicted simplistically below.



We assume that for a single random walk, a single unit of current enters the network at s and exits at t, by which the betweenness is calculated according to the amount of current which flows through i averaged over all possible starting and target nodes.

If we let V_i represent the voltage at node i, if we apply engineering circuit logic, by Kirchhoff's Voltage Law, the total current flow in and out of a vertex is zero s.t.:

$$\sum_j A_{ij}(V_i - V_j) = \delta_{is} - \delta_{it} \qquad \delta_{ij} = \begin{cases} 1, \text{ if } i = j, \\ 0, \text{ otherwise.} \end{cases}$$

where A is represented by an adjacency matrix. Since the sum of the adjacency matrix is equivalent to the degree for each vertex, we can rewrite the expression as **(D - A)*V = s**, where D is the diagonal matrix with the degrees represented along the diagonal and s is a source vector as given by:

$$s_i = \begin{cases} +1, \text{ for } i = s, \\ -1, \text{ for } i = t, \\ 0, \quad \text{otherwise} \end{cases}$$

Due to the singularity of D-A, we must remove any equation from the system in order to invert the matrix and obtain V. It is easiest to do so if we determine the voltages according to some standardized vertex v and then remove the vth row and column of D-A, resulting in an (n-1) x (n-1) matrix s.t. $\mathbf{V} = (\mathbf{D}_v - \mathbf{A}_v)^{-1} \cdot \mathbf{s}$. In this analogy, the voltage of the removed vertex v is equivalent to zero, which we now must add back to the inverted D-A matrix (we denote this as matrix T). Then, given the equation for the source vector, where all i=s are 1 and all i=t are -1, we have:

$$V_i^{(st)} = T_{is} - T_{it}$$

and furthermore, the current flowing through the ith vertex must be the sum of the currents along the edges for a specified vertex, with the absolute value taken since it is

the net change and multiplied by ½ to account for interactions between vertices being counted twice, once for each vertex being involved with an edge:

$$I_i^{(st)} = \frac{1}{2}\sum_j A_{ij}|V_i^{(st)} - V_j^{(st)}| = \frac{1}{2}\sum_j A_{ij}|T_{is} - T_{it} - T_{js} + T_{jt}|, \quad \text{for } i \neq s, t.$$

It is also important to note that the source and target vertices are set to 1 to take into consideration the current added and taken away from the network.

Thus, our betweenness can be calculated as the mean current flow over all of the source and target combinations:

$$b_i = \frac{\sum_{s<t} I_i^{(st)}}{(1/2)n(n-1)}$$

Why calculate random-walk betweenness according to this analogy? Well, if we think about a circuit, current must flow along all potential paths, yet by nature will apply more voltage to the shortest paths as a result of less resistance by Ohm's Law, which states that current is directly proportional to the voltage applied to it and inversely proportional to the resistance (V = IR). Thus, paths with lower resistance will have a higher current flow and therefore voltage drop (more voltage applied to them). Additionally, if the path never reaches particular nodes, then there can be no betweenness (although this is an issue we must address later).

The current flow analogy is numerically identical to social network applications, where we instead assume the voltage is equivalent to a message beginning at s which is being sent to t, with no prior knowledge of where t is such that it is passed randomly until t is reached. The betweenness is set to be the net number of times a walk passes through a vertex i such that walks which pass back and forth or ultimately average to be equally likely to go in multiple directions from a given vertex are effectively "canceled".

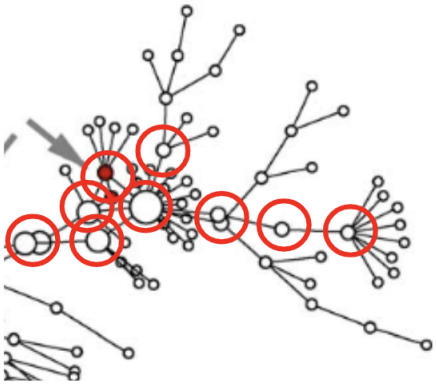
In summary, there are four steps for our algorithm:

1) Construct **D-A**
2) Remove any row or column
3) Insert new matrix and replace removed row and column with zeros
4) Calculate betweenness using the betweenness formula above, as well as using the values of I_i

3. Experiments and Results

3.1. Algorithm

Our algorithm idea was to calculate the betweenness based on random walks of each node and then mark the nodes that pass a certain betweenness as bridge nodes. We wanted to run the modified breadth-first search on each bridge node, which would not visit other bridge nodes. Each of these nodes encountered by the breadth-first search would be marked as its own community. In that sense, by adjusting the bridge threshold to be smaller, we could find even smaller communities. Notice in the figures below how the marked bridge nodes with high betweenness are responsible for forming communities (in theory):



```
def calc_T_matrix(graph, n, nodes):

    n = n - 1

    A_matrix = nx.to_numpy_array(graph, nodelist=nodes)
    D_matrix = np.diag(np.sum(A_matrix, axis = 1))

    L_matrix = D_matrix - A_matrix

    modified_L = np.delete(L_matrix, obj = n, axis = 0)
    modified_L = np.delete(modified_L, obj = n, axis = 1)
    modified_L = np.flip(modified_L)

    row_zeros = np.zeros(shape=(1,n))
    col_zeros = np.zeros(shape=(n+1,1))


    T_matrix = np.append(modified_L, row_zeros, axis=0)
    T_matrix = np.append(T_matrix, col_zeros, axis=1)

    return A_matrix, T_matrix, D_matrix, L_matrix
```

In the first portion of our code, we calculate the T matrix as described in our algorithm implementation. The function takes in a NetworkX graph, the number of nodes,

and a list of nodes to consider. We convert the graph to an adjacency matrix and calculate the degree matrix by summing up the rows of the adjacency matrix. We calculate the Laplacian matrix and delete the last row and column, afterwards concatenating a row and column of zeros to replace the lost values. We return all of these matrices for future reference.

```
def calc_rand_walk_betweenness(n, threshold, A_matrix, T_matrix, D_matrix, min_degree=0, max_degree=99999):

    betweenness_list = np.zeros(n)

    bridge_nodes = set()

    for i in range(n):

        if D_matrix[i][i] > min_degree and D_matrix[i][i] < max_degree:

            betweenness = 0
            print(i)

            for s in range(n):
                #print(s)

                for t in range(n):
                    if s < t:
                        I = 0
                        for j in range(n):
                            I += A_matrix[i,j] * np.absolute(T_matrix[i,s] - T_matrix[i,t] - T_matrix[j,s] + T_matrix[j,t]

                        I = I * (1/2)

                        if i == s or i == t:
                            I = 1


                        betweenness += I

            betweenness = betweenness * (1 / ((1/2) * n * (n-1)))

            betweenness_list[i] = betweenness

            if betweenness > THRESHOLD:
                bridge_nodes.add(i)


    return(betweenness_list, bridge_nodes)
```

In the second portion of the code, we calculate the random-walk betweenness centrality for each node in the graph. The function takes the number of nodes, the adjacency/modified Laplacian/diagonal matrix, and a minimum/maximum degree threshold. We check if the degree of a node is within the specified range as denoted by the minimum and maximum threshold by iterating through the diagonal matrix (which contains the degrees). We iterate over all pairs of s and t and sum the absolute difference of the entries multiplied by the adjacency matrix and divided by 2. We adjust the current node to 1 if at s or t. We normalize by multiplying by 1/(½*n*(n-1)), and add to the bridge nodes if the betweenness is greater than a specified threshold. We return the valid betweennesses and the bridge nodes.

```
def community_bfs(graph, start, bridge_list, marked_nodes):
    queue = deque([start]) # Queue for BFS
    community = set()

    node = queue.popleft()

    community.add(node)


    for neighbor in graph.neighbors(node):
        queue.append(neighbor)

    while queue:

        node = queue.popleft()  # Get the first node from the queue

        if node not in community and node not in bridge_list and node not in marked_nodes:

            community.add(node)  # Mark the current node as visited
            marked_nodes.add(node) # Mark node in master list

            # Add all the unvisited neighbors of the current node to the queue
            for neighbor in graph.neighbors(node):
                if neighbor not in community and neighbor not in bridge_list and neighbor not in marked_nodes:
                    queue.append(neighbor)

    return community, marked_nodes
```

In the third portion of the code, we perform a breadth-first search to identify communities within a given graph. The function takes in the graph, the starting node, the list of bridge nodes, and a set of nodes which have already been visited. We initialize a queue, where the first node is "popped" to the left side of the queue and assigned as the node for which to be added to the community. We iterate over the neighbors and add them to the end of the queue. We then check that the queue is not empty, and if so, if a node is not already in the community, bridge list, and marked nodes, we add this node to the community and mark it as visited. We again iterate over the neighbors and add to the queue if it meets the conditions mentioned previously. The function returns the community and all nodes that were marked as visited.

```
def bridge_calc(betweenness_list, threshold):
    bridge_nodes = set()
    for i in range(len(betweenness_list)):
        if betweenness_list[i] >= threshold:
            bridge_nodes.add(i)

    return(bridge_nodes)
```

In the fourth portion of the code, we identify bridge nodes based upon a threshold and set of random-walk betweenness centrality scores, which are our input parameters. We iterate over all of the nodes for which the betweenness was calculated and check if the betweenness score is greater than or equal to a threshold as specified by the input parameter. If the threshold is met, we identify that particular node as a bridge node. We return a list of all bridge nodes.

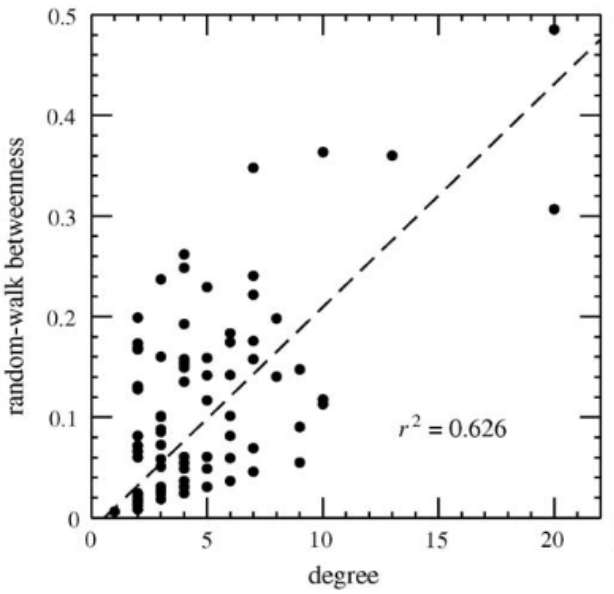
```
def create_communities(graph, betweenness_list, threshold_list, min_community_size=6, max_community_size=40):
    final_communities = []
    marked_in_final = set()

    for threshold in threshold_list:
        bridges = bridge_calc(betweenness_list, threshold)

        marked_nodes = marked_in_final

        communities = []


        for bridge in bridges:

            new_community, updated_marked_nodes = community_bfs(graph, bridge, bridges, marked_nodes)
            communities.append(new_community)
            marked_nodes = updated_marked_nodes


        for community in communities:

            if len(community) >= min_community_size and len(community) <= max_community_size:
                final_communities.append(community)

                for node in community:
                    marked_in_final.add(node)


    no_community_nodes = []

    for node in graph.nodes:
        included = 0

        for community in final_communities:
            if node in community:
                included = 1

        if included == 0:
            no_community_nodes.append(node)


    while no_community_nodes:
        node = no_community_nodes.pop(0)

        done = 0

        for neighbor in graph.neighbors(node):
            if done == 0:
                for community in final_communities:
                    if neighbor in community and done == 0:
                        community.add(node)
                        done = 1
        if done == 0:
            no_community_nodes.append(node)

    return(final_communities)
```

In the fifth portion of the code, we perform community detection by iteratively identifying communities based upon bridge nodes and threshold values. We take the graph, betweenness centrality scores, threshold values, and minimum/maximum community sizes. We iterate over all thresholds and identify bridge nodes for a particular threshold, and then after further iterating over each bridge node, we perform a breadth-first search to return a given community. After we iterate over all of the bridge nodes, we iterate over each community and check that it falls within the specified minimum and maximum

community sizes, which are taken as input parameters, and if satisfied, it is saved to a new list of communities and marked nodes as well as identifying nodes which aren't added to any community (which is added to another separate list). After we have our list of nodes with no community, we assign these values as a node iteratively until a neighbor is present in a community for which the node can be added to. The process is repeated until all nodes have been added to a community, and a final communities list can be returned.

3.2. Adjustments

We encountered multiple issues when developing the algorithm. The first and most noteworthy problem encountered was with regard to run time. Our original betweenness calculation considered all possible nodes in a given network for calculating the betweenness centrality, which increased the run time. However, according to Newman's paper, we found that there is a somewhat strong correlation between random-walk betweenness centrality and degree, as depicted in the graph below:



Thus, if approximately 62.6% of the variability in random-walk betweenness can be explained by degree, there is a clear moderately positive linear relationship which we can use to exclusively calculate the betweenness of nodes with higher degrees. Thus, we adjusted our original code as depicted below to check that the degree falls within a given range before iterating through to find the bridge nodes. Below is our original code:

```
def calc_rand_walk_betweenness(n, threshold, A_matrix, T_matrix):

    betweenness_list = np.zeros(n)

    bridge_nodes = set()

    for i in range(n):
        betweenness = 0
        print(i)

        for s in range(n):

            for t in range(n):
                if s < t:
                    I = 0
                    for j in range(n):
                        I += A_matrix[i,j] * np.absolute(T_matrix[i,s] - T_matrix[i,t] - T_matrix[j,s] + T_matrix[j,t])

                    I = I * (1/2)

                    if i == s or i == t:
                        I = 1


                    betweenness += I

        betweenness = betweenness * (1 / ((1/2) * n * (n-1)))

        betweenness_list[i] = betweenness

        if betweenness > THRESHOLD:
            bridge_nodes.add(i)


    return(betweenness_list, bridge_nodes)
```

To adjust, we added the line **if D_matrix[i][i] > min_degree and D_matrix[i][i] < max_degree** before iterating through the source and target nodes to check this condition.

However, this change was not enough, as our Amazon and Email-EU-Core networks were still quite large. As a result, running our algorithm was taking upwards of days to weeks to run in full. Therefore, we decided to reduce our network sizes to 1000 and 500 nodes respectively so that we could interpret our results on a smaller scale and eliminate potential overfitting errors. Furthermore, by reducing our networks, we were better able to understand the limitations of our algorithms, which could be due to the number of nodes, edges, or the unique structures of particular networks.

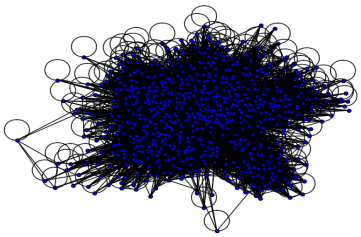
One important consideration that we needed to ensure when reducing our networks was that the structure was not compromised substantially (the uniqueness had to be preserved). Our preliminary results indicated that community detection for networks with a limited number of nodes and edges were relatively successful, as demonstrated by Zachary's Karate Club. Thus, we decided the best method for network reduction was to eliminate nodes with the lowest degree centrality. This method was much more efficient and less time intensive than the other methods that we looked into. Our code for network reduction is as follows:

```
def reduction(edgelistpath, labellistpath, edgestopath, labeltopath, num_nodes):

    #reads the edgelist, creates the graph to be edited, and creates the variable to read the current number of nodes in the netw
    Graph = nx.read_edgelist(edgelistpath)
    pruned_graph = Graph
    currNumNodes = pruned_graph.number_of_nodes()

    #loop to reduce the number of nodes in the network
    while currNumNodes > num_nodes:
        # Calculate node importance based on node degree centrality
        centrality = nx.degree_centrality(pruned_graph)
        # Sort nodes based on centrality in descending order
        sorted_nodes = sorted(centrality, key=centrality.get, reverse=True)
        # Remove the least important nodes (nodes with the lowest degree centrality), until the square root of the
        #target network node count has been removed, or until the network is the desired size.
        #Using degree centrality and removing multiple nodes before recalculating centrality was a compromise
        #between network shape retention and efficiency.
        i = 0
        while i < sqrt(num_nodes) and currNumNodes > num_nodes:
            pruned_graph.remove_node(sorted_nodes[-i])
            i = i + 1
            currNumNodes = pruned_graph.number_of_nodes()
        #finds the largest connected component and makes that into the graph, in case a component was separated due to node remov
        Gcc = sorted(nx.connected_components(pruned_graph), key=len, reverse=True)
        pruned_graph = nx.Graph(pruned_graph.subgraph(Gcc[0]))

    #writes reduced graph to an edgelist
    nx.write_edgelist(pruned_graph, edgestopath)
```

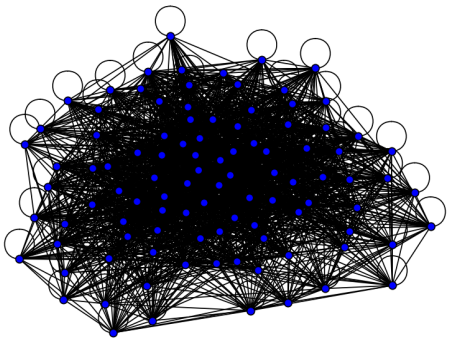
The code loops through the nodes to see if it is greater than a particular threshold, and if not the case, it iterates through based upon degree centrality and removes nodes of the smallest degree centrality until the square root of the target network node count is removed or until the desired network size is reached. We additionally find the largest connected component and add that to the graph since the use of degree centrality and removing nodes before calculating the centrality compromises the network shape and efficiency.

As shown below, our reduction worked well at reducing the graph while maintaining the general uniqueness and still has a significant number of connections:

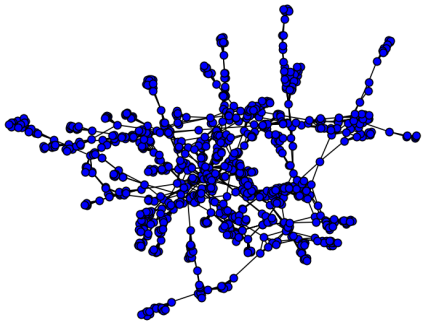
Original Amazon Graph



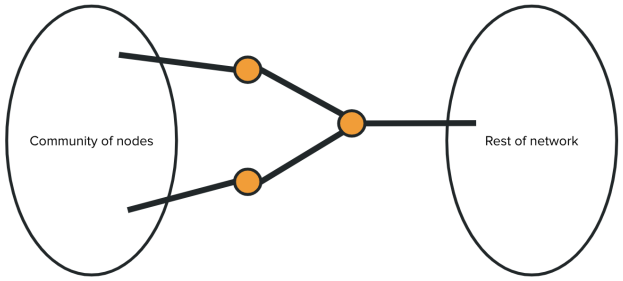
Updated Amazon Graph



Updated Email-EU Core Graph



The second issue which we encountered was nodes being placed in multiple communities. Unfortunately, a drawback of community detection is it often allows for overlapping communities, where nodes often belong to more than one community due to their connectivity and interactions with other nodes. In the figure below, note how the given set of nodes connecting the community of nodes to the rest of the network could hypothetically be assigned to more than one community, even though in actuality they should be apart of the same community:



Our original code looked as follows:

```python
def bfs(graph, start, bridge_list):
    queue = deque([start]) # Queue for BFS
    community = set()

    node = queue.popleft()

    community.add(node)

    for neighbor in graph[node]:
        queue.append(neighbor)

    while queue:
        node = queue.popleft()  # Get the first node from the queue

        if node not in community and node not in bridge_list:

            community.add(node)  # Mark the current node as visited

            # Add all the unvisited neighbors of the current node to the queue
            for neighbor in graph[node]:
                if neighbor not in community and neighbor not in bridge_list:
                    queue.append(neighbor)

    return community
```

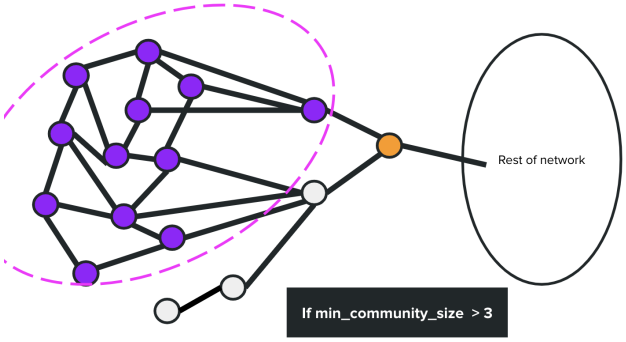
Notice how our original implementation does not include a marked nodes list. After adding this to our algorithm, we were able to control the number of duplicate visits by ensuring that nodes have already been visited and making sure that they are not revisited during

BFS. This in turn also improves the run time and validity of our communities.

The third issue which we encountered was that some nodes were not being placed into any community at all. This is likely due to their generally low degree, especially with the newly reduced network sizes. Nodes with few connections likely do not contain the structural cues to clearly be assigned to a community on the first iteration. Thus, to account for this, we changed our original implementation:

```
def create_communities(graph, betweenness_list, threshold_list, min_community_size=6, max_community_size=40):
    final_communities = []
    marked_in_final = set()

    for threshold in threshold_list:
        bridges = bridge_calc(betweenness_list, threshold)

        marked_nodes = marked_in_final

        communities = []

        for bridge in bridges:
            new_community, updated_marked_nodes = bfs_2(graph, bridge, bridges, marked_nodes)
            communities.append(new_community)
            marked_nodes = updated_marked_nodes

        for community in communities:
            if len(community) >= min_community_size and len(community) <= max_community_size:
                final_communities.append(community)

                for node in community:
                    marked_in_final.add(node)

    return(final_communities)
```

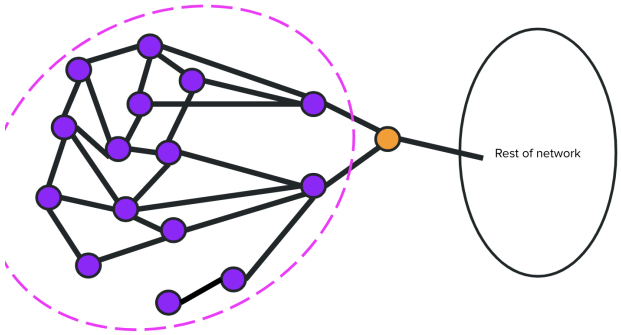
In the figure below, note how when the minimum community size is set to be greater than 3, we simply disregard the set of three nodes which branch from the bridge node, and do not assign it to any community as it is a community which is equal to 3, so it does not meet the specifications for a community. However, it is apparent visually that the entire collection of these nodes should be assigned to the same community.



To fix this problem, we added a new section of code which finds neighbors for the nodes which aren't a part of a community and adds the collection of these nodes to communities iteratively (such that it keeps running until all nodes have been assigned to a community):

```
no_community_nodes = []

for node in graph.nodes:
    included = 0

    for community in final_communities:
        if node in community:
            included = 1

    if included == 0:
        no_community_nodes.append(node)


while no_community_nodes:
    node = no_community_nodes.pop(0)

    done = 0

    for neighbor in graph.neighbors(node):
        if done == 0:
            for community in final_communities:
                if neighbor in community and done == 0:
                    community.add(node)
                    done = 1
    if done == 0:
        no_community_nodes.append(node)

return(final_communities)
```

Under this new implementation, we see that all nodes are included in the same community from our earlier example, as it notes the bridge node as a neighbor and likewise the set of nodes as neighbors, while still maintaining a minimum community size greater than three by its addition of these three nodes to the entire community:



3.3. Results

Our results look promising. We compared our networks with the Louvain best partition algorithm, and according to visual interpretation, our model does significantly better at creating smaller communities for these relatively large datasets.

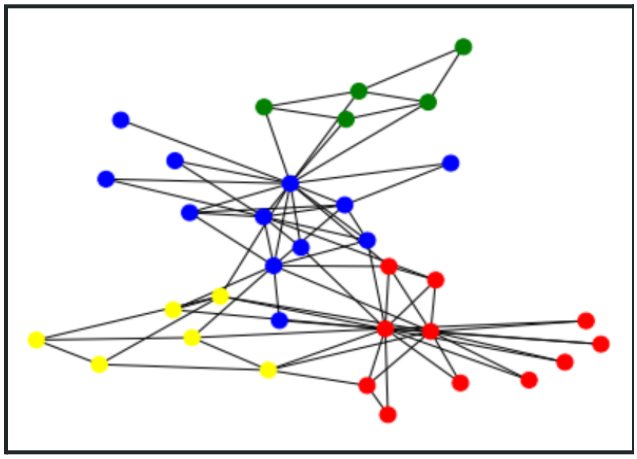
For reference, here is an updated table indicating our network sizes including our network reductions:

Updated Network Sizes

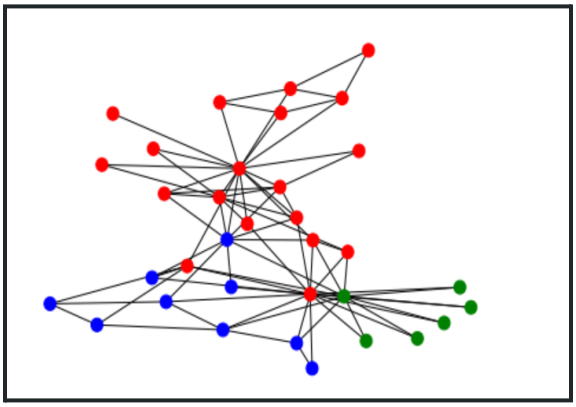
	EEC	APC	FFP	ZKC
Nodes	500	1,000	620	34
Edges	12,651	2,763	2,102	78

For a baseline, we tested the Karate Club Network:

Louvain Best Partition



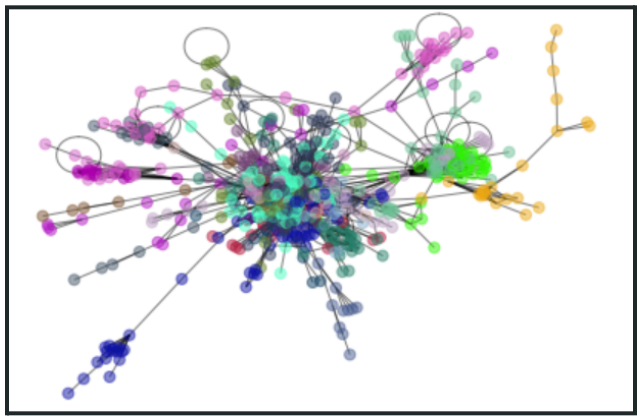
Our Model



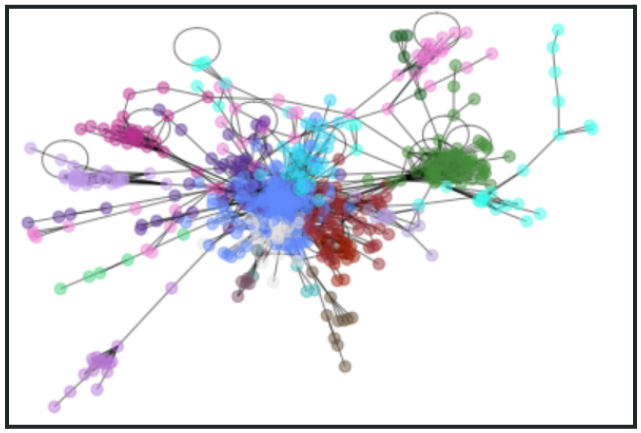
We know Karate Club has two true "communities", so given that our algorithm forms two distinct large communities and one smaller one is in line with what we might expect. Furthermore, we know our algorithm is biased to perform even better with larger datasets.

Next, we tested the Facebook Food Product network:

Our Model



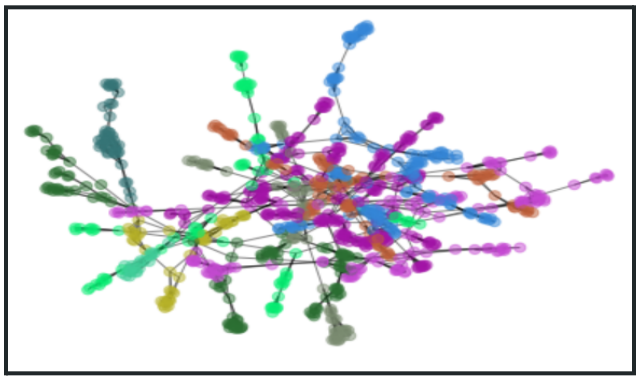
Louvain Best Partition



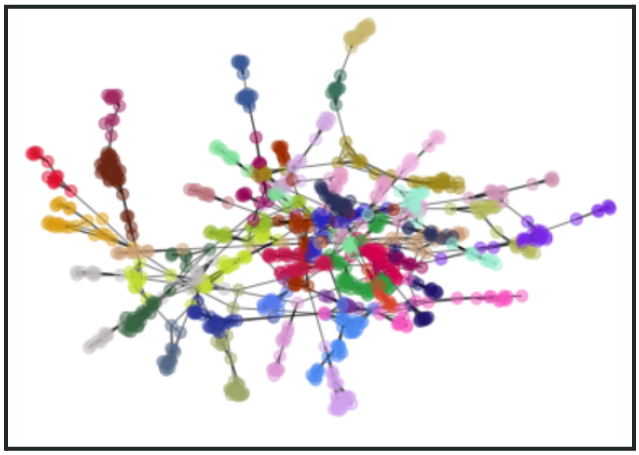
Though we don't have access to the ground-truth labels for this community, when we compare our algorithm to Louvain, it is apparent that there is a significant difference in the number of communities. Each of the communities appears to be densely clustered and exhibits strong interactions, yet also appear to be more densely connected internally as compared to outside of the groupings. We also note that both algorithms can recognize and form communities for sparsely connected communities as well, as the outer edges are similarly grouped together. The fact that such cohesive subgroups exist is a great sign that our algorithm is making accurate predictions for smaller subgroups.

Additionally, we tested the Amazon network:

Our Model



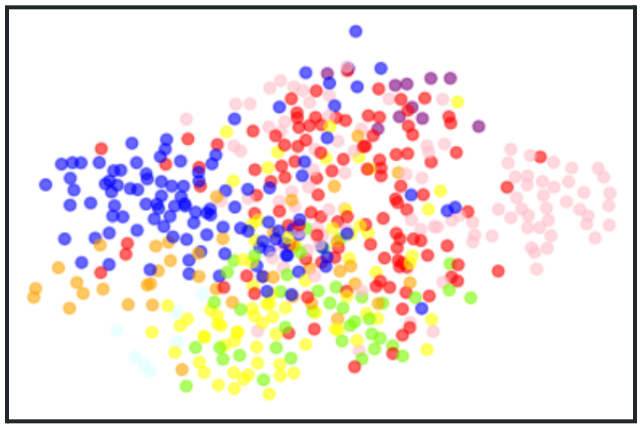
Louvain Best Partition



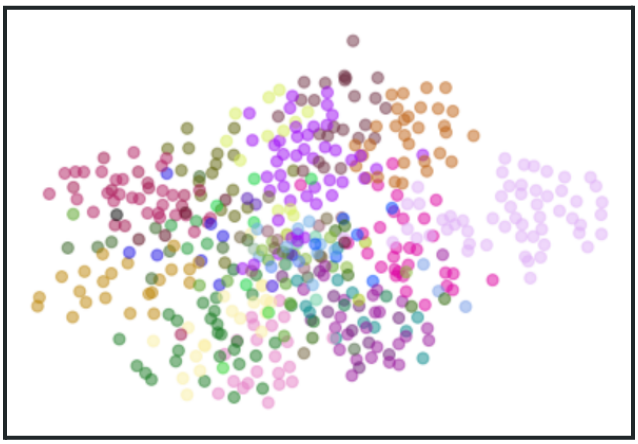
As before, there are evident structural patterns as well as a high number of connections among nodes within communities and limited number of connections for nodes outside of the communities. This is yet another indication that our algorithm is performing as expected for larger networks. Despite there being less communities formed for our algorithm in comparison to Louvain best partition, this is actually preferred since the Louvain algorithm appears to generate an unnecessarily high number of communities for the number of connected nodes, whereas our algorithm appears to be fine-tuned to account for more realistic connections between networks in this rather sparse network.

However, one issue that we could not resolve is the issue of when there are too many edges. When we ran our tests for the Email-EU-core network, we found that the large number of edges made the network too complex to create accurate communities. Here is our results compared to the ground-truth communities (with the edges themselves eliminated for visualization purposes):

Our Model



Ground-Truth Communities



We obtained an NMI score of 0.33, which indicates a low to moderate agreement between the clustering results. Though some subgroups clearly align to some extent, there are also distinct variations in the community formation.

4. Conclusion

Our objective was to effectively generate accurate representations of communities, particularly for larger networks which often do not capture smaller communities. We found that after taking steps to minimize the run time and deal with edge cases, our model is quite effective, with the exception of when there are too many edges. However, this is not a problem unique to our algorithm, so it is not of much concern.

Despite these positive findings, there is still much room for improvement. We could consider running the betweenness centrality calculation for the rest of the nodes in the networks and then cluster the communities with all of the calculated betweennesses, although we expect this would take a very long time to run. We could also modify the algorithm further such that users can select the desired number of communities, yet this would

require noticing patterns with the current parameters. Lastly, we could consider testing on even larger networks, such as our originally proposed Youtube network.

Our findings indicate that Newman's methodology of random-walk betweenness centrality has vast implications, particularly for when dealing with larger networks. The key difference is that it considers all paths due to the spontaneity of the algorithm, whereas other algorithms tend to omit nodes and edges at an early stage. Therefore, we conclude that random-walk betweenness centrality is an effective method of community detection that should be explored further to contribute to the field of network analysis.

5. Contributions

All authors provided feedback when necessary, proposed the experiments, and contributed to the results/analysis. Matt took the lead on the algorithm design and implementation, Ben took the lead on reducing our network sizes to manageable forms, and Zac took the lead on preliminary research as well as the writing of this final report, though we each took part in all parts of the process as mentioned above, as well as other components such as the midterm and final presentation. All authors are in agreement that we contributed equally to this project.

Citations

[1] *3D Simulation of a Random Walk in a Solution. - Researchgate*. https://www.researchgate.net/figure/3D-simulation-of-a-random-walk-in-a-solution_fig2_342851703.

[2] "Betweenness Centrality." *Betweenness Centrality*, www.sci.unich.it/~francesc/teaching/network/betweeness.html. Accessed 6 June 2023.

[3] "Chapter 12 - Random Walks." *Chance*, https://chance.dartmouth.edu/teaching_aids/books_articles/probability_book/Chapter12.pdf.

[4] "Community Detection Algorithms." *Neo4j Graph Data Platform*, neo4j.com/developer/graph-data-science/community-detection-graph-algorithms/#:~:text=What%20are%20community%20detection%20algorithms,supports%20many%20different%20centrality%20algorithms. Accessed 6 June 2023.

[5] *Department of Mathematics | The University of Chicago*. https://www.math.uchicago.edu/~lawler/srwbook.pdf.

[6] *Index of Complex Networks*, icon.colorado.edu/#!/networks. Accessed 6 June 2023.

[7] Jayawickrama, Thamindu Dilshan. "Community Detection Algorithms." *Medium*, Towards Data Science, 1 Feb. 2021, https://towardsdatascience.com/community-detection-algorithms-9bd8951e7dae#:~:text=on%20the%20domain.-,Community%20Detection%20Techniques,edge%20to%20the%20weaker%20edge.

[8] M.E. J. Newman. "A Measure of Betweenness Centrality Based on Random Walks." *Social Networks*, North-Holland, 10 Feb. 2005, https://www.sciencedirect.com/science/article/pii/S0378873304000681?casa_token=jXYD1tHwdKsAAAAA%3AQR84V1Sv-SnjIkrrNo6td0e3F-pR1pjswjvzy6a2jO_GR56lSfeAQTSzc_y5QaBnu4578hnqP3g.

[9] Kasper, London. "Girvan-Newman v. Louvain for Community Detection." *Medium*, 30 Apr. 2022, medium.com/smucs/girvan-newman-v-louvain-for-community-detection-33988baab55b.

[10] Kivimäki, Ilkka, et al. "Two Betweenness Centrality Measures Based on Randomized Shortest Paths." *Nature News*, 1 Feb. 2016, www.nature.com/articles/srep19668.

[11] "Random Walk--2-Dimensional." *From Wolfram MathWorld*, https://mathworld.wolfram.com/RandomWalk2-Dimensional.html#:~:text=Amazingly%2C%20it%20has%20been%20proven,number%20of%20steps%20approaches%20infinity.

[12] Rita, Luis. "Infomap Algorithm." *Medium*, 12 Apr. 2020, towardsdatascience.com/infomap-algorithm-9b68b7e8b86.

[13] "The One-Dimensional Random Walk." *Random Walk*, https://galileo.phys.virginia.edu/classes/152.mf1i.spring02/RandomWalk.htm.

[14] Traag, V. A., et al. "From Louvain to Leiden: Guaranteeing Well-Connected Communities." *Nature News*, 26 Mar. 2019, www.nature.com/articles/s41598-019-41695-z.

[15] Waibel, Jeremy. "Community Detection — Girvan Newman." *Medium*, 14 Apr. 2022,

medium.com/@jeremywaibel/girvan-newman-introuction-1a296c67d89e.

[16] Wattage, Motoki. “Exercise for Chapter 6: Centrality.” *Social Sciences*, 1998, www.sscnet.ucla.edu/soc/faculty/mcfarland/soc112/cent-ans.htm.

[17] Wu, Xueping. “Network Analysis of Zachary’s Karate Club.” *Information Visualization*, 22 Apr. 2019, studentwork.prattsi.org/infovis/visualization/network-analysis-of-zacharys-karate-club/.

[18] Zeng, Jianping, and Hongfeng Yu. “A Distributed Infomap Algorithm for Scalable and High-Quality Community Detection.” *Cse.Unl.Edu*, cse.unl.edu/~yu/research/nsf17_graph/paper/2018.Distributed%20Infomap%20Algorithm%20for%20Scalable%20and%20High-Quality%20Community%20Detection.pdf. Accessed 6 June 2023.