Fall 2020

Visualization of Large Networks Using Recursive Community Detection

Xinyuan Fan
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the [Other Computer Sciences Commons](#)

Recommended Citation

Fan, Xinyuan, "Visualization of Large Networks Using Recursive Community Detection" (2020). *Master's Projects*. 965.

DOI: <https://doi.org/10.31979/etd.atus-pbv9>

https://scholarworks.sjsu.edu/etd_projects/965

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Visualization of Large Networks Using Recursive Community Detection

CS298 Report

Presented to
Department of Computer Science
San Jose State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Xinyuan Fan
December 2020

© 2020

Xinyuan Fan

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Visualization of Large Networks Using Recursive Community Detection

by

Xinyuan Fan

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

December 2020

Dr. Katerina Potika Department of Computer Science

Dr. Teng Moh Department of Computer Science

Prof. Kevin Smith Department of Computer Science

ABSTRACT

Networks show relationships between people or things. For instance, a person has a social network of friends, and websites are connected through a network of hyperlinks. Networks are most commonly represented as graphs, so graph drawing becomes significant for network visualization. An effective graph drawing can quickly reveal connections and patterns within a network that would be difficult to discern without visual aid. But graph drawing becomes a challenge for large networks. Ambiguous edge crossings are inevitable in large networks with numerous nodes and edges, and large graphs often become a complicated tangle of lines. These issues greatly reduce graph readability and makes analyzing complex networks an arduous task. This project aims to address the large network visualization problem by combining recursive community detection, node size scaling, layout formation, labeling, edge coloring, and interactivity to make large graphs more readable. Experiments are performed on five known datasets to test the effectiveness of the proposed approach. A survey of the visualization results is conducted to measure the results.

Keywords: Networks, Visualization, Graph Drawing, Community Detection, Louvain, Node Size, Graph Layout, Labeling, Edge Coloring, Interactivity

ACKNOWLEDGMENTS

I would like to thank Dr. Katerina Potika for serving as the advisor for this project. Dr. Potika's continued guidance and feedback have been invaluable towards the completion of this project. I would also like to thank committee members Dr. Teng Moh and Prof. Kevin Smith for taking an interest in this project. I am very appreciative of their time and efforts. Lastly, I would like to thank my family and friends for their encouragement and support.

TABLE OF CONTENTS

CHAPTER

1	Introduction	1
2	Related Works	4
2.1	Network Tools	4
2.2	Network Embedding	4
2.3	Network Nucleus	5
2.4	Layouts	5
2.5	Edge Coloring	6
3	Methodology	8
3.1	Community Detection	8
3.1.1	Louvain Method for Community Detection	10
3.1.2	Recursive Community Detection	10
3.2	Node Size Scaling	16
3.2.1	Minimum Node Size Scaling	16
3.2.2	Logarithmic Node Size Scaling	17
3.2.3	Capped Node Size Scaling	19
3.3	Layouts	20
3.3.1	Static Layouts	20
3.3.2	Interactive Layouts	21
3.4	Labeling	24
3.5	Edge Coloring	25

3.5.1	Node Coloring	25
3.6	Interactivity	27
4	Setup of Experiments	29
4.1	Datasets	29
4.2	Technologies	30
4.3	System Requirements	31
4.4	User Survey	31
5	Results	34
5.1	Recursive Community Detection	34
5.2	Node Size Scaling	37
5.2.1	Minimum Node Size Scaling	37
5.2.2	Logarithmic Node Size Scaling	38
5.2.3	Capped Node Size Scaling	39
5.3	Layouts	40
5.4	Labeling	43
5.5	Edge Coloring	43
5.6	Interactivity	44
6	Conclusion and Future Work	48
6.1	Conclusion	48
6.2	Future Work	49
	List of References	51

APPENDIX

Data Visualization User Survey Questions	55
A.1 Email Iteration 1	56
A.2 Email Iteration 2	57
A.3 DBLP Iteration 1	58
A.4 DBLP Iteration 2	59
A.5 Amazon Iteration 1	60
A.6 YouTube Iteration 1	61
A.7 YouTube Iteration 2	62
A.8 Wikipedia Iteration 1	63

CHAPTER 1

Introduction

A network is an interconnected system of people or things. Networks are ubiquitous, as everyone has their own social network of friends, colleagues, and other acquaintances. Furthermore, with the rise of online social networks, such as Facebook and Twitter, millions of people can connect from across the world. With these rapidly growing networks, visualization becomes more important than ever to comprehending complex networks. Network analysis provides valuable information about network organization and community structure. These insights can be most clearly seen by representing networks as graphs. Actors correspond to nodes, and relationships are depicted as edges. A simple example of a social network in graph format can be seen in Figure 1.

Figure 1: Social network graph [1].

A visual graph representation can reveal details about a network from just a glance. This makes understanding network connections quick and easy compared to trying to analyze raw data. However, as networks grow larger, they become increasingly difficult to visualize. With so many nodes and overlapping edges, a graph can become a giant hairball, impossible to extract valuable information from. This project aims to address this problem by combining various visualization techniques and proposing some for the first time in order to improve the readability of large graphs.

The objective of this research is to create a method to effectively visualize large networks. This is achieved by

- proposing a recursive community detection approach,
- exploring various node size scaling approaches,
- applying various layout formations,
- applying node labeling to convey information on community size,
- incorporating edge coloring for the inter-connections between communities,
- and experimenting with interactive visualization capabilities.

Community detection is necessary for identifying closely related groups within a network [2]. Node size scaling ensures nodes are sized proportionately compared to other communities based on how many nodes from the original network are contained in the community. Visualizations are drawn in different layouts [3, 4] depending on the graph characteristics. Communities are labeled with the number of nodes contained in the community. This labeling enables the viewer to quickly gather insights on

community size. Edge coloring [5] is applied to edges between communities, interconnections, to minimize confusion from edge crossings. Finally, making the graphs interactive allows for useful capabilities such as zooming, dragging, and highlighting node connections.

These topics have been individually researched, but there has yet to be a visualization method that utilizes all of these features together to generate optimal network visualizations. Combining these methods for graph visualization is the primary innovation and challenge of this project. Additionally, a user survey on different visualizations of five known datasets for community detection is used to rate the experimental results.

CHAPTER 2

Related Works

There have been various research efforts towards developing new graph analysis and visualization techniques. This project draws inspiration from some of these existing solutions to produce a comprehensive graph visualization approach.

2.1 Network Tools

Many network analysis tools have arisen in recent years with the boom of social networks. These tools provide community detection and mining capabilities to extract useful information from networks. However, with so many different tools available, it can be difficult to determine which tool is best for a job. Amin, Ahmad, and Choi [1] hoped to address this issue by comparing four major network analysis tools: Paek, Gephi, IGraph, and NetMiner. They tested network metrics, such as degree centrality and network diameter, as well as different community detection algorithms across the four platforms. Amin, Ahmad, and Choi also compared the execution time of algorithm features on each of these tools. This project utilizes different networks tools, NetworkX and D3.js (D3), but the metrics from [1] can be applied to these platforms as well. Also, [1] serves as a good reference if additional tools are required for future work.

2.2 Network Embedding

One of the challenges in network mining is encoding graphs to be exploitable by traditional machine learning algorithms. Gutierrez-Gomez and Delvenne [6] proposed an unsupervised learning method for graph embeddings. The model would learn graph

embeddings from a collection of networks and capture the overall structure of network data in order to distinguish between different types of networks. Gutierrez-Gomez and Delvenne evaluated their model in the areas of graph clustering, classification, and visualization. They compared their approach with other feature extraction and network comparison techniques such as graph distances and graph kernels and created synthetic random datasets for their experiments. Machine learning is not a focus area for this project but can be useful for future work. Graph embedding from [6] may prove helpful for layout classification in graph drawing.

2.3 Network Nucleus

Community detection is a well studied area of network analysis, but little focus has been put on finding the core structure, or nucleus, of a network. The nucleus holds parts of the network, or constituent communities, together. Dumba and Zhang [7] researched uncovering the nucleus of networks using k-shell decomposition. For datasets, they used an autonomous systems graph and nine social network graphs, including an email communication network, Facebook friendship network, and a jazz musician collaboration network. Dumba and Zhang analyzed the network core structure using core path length, core centrality, and core removal metrics. This project uses the standard Louvain method for community detection by Blondel et al. [2], but finding the nucleus of graphs could be a logical next step for experiments beyond community detection.

2.4 Layouts

Graphs can be arranged in different layouts such as force-directed, hierarchical, or circular to name a few. Viewing the same graph in different layouts can result in

diverse visualizations and reveal different information. Thus, it is important to select the best layout for a task. Aesthetic metrics are commonly used to select a good layout, but it is computationally expensive to calculate multiple layouts and their corresponding aesthetics. Kwon, Crnovrsanin, and Ma [8] proposed using graph kernels to calculate the topological similarity of graphs for a machine learning approach to large graph visualization. Their approach can display graphs in different layouts and calculate their associated aesthetic metrics. They used 3,700 graphs from the University of Florida Sparse Matrix Collection and studied aesthetic metrics such as minimizing the number of edge crossings and maximizing the angle between incident edges. To evaluate layout aesthetics in this project, a user survey is conducted to rate visualizations in different layouts.

Wang et al. [9] are also interested in finding optimal graph layouts. They noticed that graph drawing functions often require users to adjust parameters and find the best graph layout for their needs through trial and error. Wang et al. decided to address this by applying deep learning techniques to graph drawing with DeepDrawing. DeepDrawing is a long short-term memory based model trained to extract layout characteristics from graphs. It can then draw graphs in a similar layout for new networks. Wang et al. synthetically generated graphs for their experiments and tested DeepDrawing on four kinds of layouts: grid, star, ForceAtlas2, and PivotMDS. This project borrows [9]’s concept of adjusting layout parameters based on graph characteristics to produce the best looking visualization.

2.5 Edge Coloring

Edge crossings in a graph can cause confusion and decrease readability. Hu, Shi, and Liu [5] proposed an edge coloring algorithm to differentiate edges crossings in a

graph with contrasting colors. It uses a branch-and-bound method on color gamut space decomposition. The approach first creates a sparse dual collision graph and then colors the nodes of the dual graph to maximize color differences. Finally, the node colors from the dual graph are used to color the edges of the original graph. They tested their results on a combination of six graphs from the University of Florida Sparse Matrix Collection and test graphs in Graphviz. They also conducted a user study using the Zachary's Karate Club social graph to measure the effectiveness of their edge coloring approach. Edge coloring proves to be useful for small to medium sized graphs and is used in this project to distinguish edge crossings between communities.

CHAPTER 3

Methodology

This project aims to develop a method for visualizing large networks. This is achieved through the following steps:

1. Input the original network data.
2. Perform recursive community detection on the network.
3. Scale the node sizes proportionately.
4. Draw the graph in different layouts.
5. Label the nodes with the number of nodes from the original graph and the number of nodes from the previous community detection iteration.
6. Add edge coloring to help distinguish overlapping edges.
7. Incorporate interactive capabilities such as zooming, panning, and highlighting node connections.

These steps are outlined in Figure 2.

3.1 Community Detection

When a network has over hundreds of nodes, plotting every node results in a visualization that is overcrowded and difficult to extract insights from. This project proposes a recursive community detection approach to contract the graph size until the number of communities cannot be further reduced.

Figure 2: Methodology.

3.1.1 Louvain Method for Community Detection

The community detection algorithm used in each recursive iteration is the Louvain method for community detection. The Louvain method created by Blondel *et al.* [2] is a well known community detection technique that aims to extract communities through modularity optimization. Modularity is the number of edges within groups minus the expected number of edges within groups of an equivalent network where the node degrees are the same but the edges are randomly placed [10]. The degree of a node is the number of edges connected to that node. A high modularity value means that there are many edge connections between nodes of the same community (intra-community connections) but few edges between nodes from different communities (inter-community connections) [11]. The range for modularity lies within $[-\frac{1}{2}, 1]$ [12], and modularity between (0.3, 0.7) suggests notable community structure [13].

The Louvain method is used over other community detection algorithms such as Girvan-Newman [14] because Louvain is better suited for large networks. Louvain runs in time $O(n \log n)$ where n is the number of nodes [15]. In contrast, Girvan-Newman has time complexity $O(m^2 n)$ for a network with n nodes and m edges [16]. This makes the Girvan-Newman algorithm inefficient for networks with more than about one thousand nodes.

3.1.2 Recursive Community Detection

This project proposed an algorithm that recursively applies the Louvain community detection algorithm to the input network until Louvain cannot further condense the communities in order to make the graph smaller and probably visually more pleasant. Louvain partitions the input graph nodes into communities. From these partitions, a new graph is created where each community is represented by a single

node. An edge is added between community nodes A and B if there is a node from the input graph partitioned into community A that connects to a node from the input graph partitioned into community B . This new graph is then used as the input graph for the next iteration of community detection.

The recursion ends when running Louvain on the input graph results in the same number of nodes as the input. Louvain uses the modularity measure to form communities, and modularity is calculated based on the network edges. Thus, once there are no more edges in the graph, the nodes cannot be further grouped using the Louvain algorithm. There are two possible final iteration graphs that can result from the recursive community detection:

1. A graph with only one node.
2. A graph with multiple nodes which all have degree zero.

This project is interested in testing how many rounds of community detection are required before the minimum number of communities is reached. The following algorithms explain the recursive community detection method and show how it is used to create new graphs for every iteration.

Algorithm 1 is the recursive community detection function. The details of the iterative step procedures are explained in sub-function Algorithm 2.

Algorithm 1: Recursive Community Detection

Input: G : NetworkX input graph,
nodeSizeList: list of node sizes for graph G ,
prevG: input graph from previous iteration, default = None if this is the first iteration

Output: This function does not return any values.

Result: Creates graphs for all community detection iterations of the initial input dataset.

Function recursiveCommunityDetection(G , *nodeSizeList*, *prevG*):

```

    if (len( $G$ .nodes()) == len(prevG.nodes())) then
        // base case = input graph has the same number of nodes
        // as the input graph from the previous iteration
        return;
    else
        // iterative case
        newInput = doCommunityDetection( $G$ , nodeSizeList);
        newG = newInput[0];
        newNodeSizeList = newInput[1];
        recursiveCommunityDetection(newG, newNodeSizeList,  $G$ ,
            nodeSizeList);
    end

```

Algorithm 2 presents the iterative step for Algorithm 1. It uses Louvain community detection [2] and creates a new graph from the results. The graph creation and node size scaling methods are detailed in helper functions Algorithm 3 and Algorithm 4 respectively.

Algorithm 2: Recursive Community Detection Iterative Step

Input: G : NetworkX input graph,

$nodeSizeList$: list of node sizes for graph G

Output: $results$: A list containing the new graph and node size list after community detection.

Result: Performs Louvain community detection [2] on G and saves/draws the new graph results.

Function doCommunityDetection(G , $nodeSizeList$):

```
/* partition = dictionary where:
   left hand side (LHS) = node number in input graph,
   right hand side (RHS) = community number LHS node was
   assigned to (starting from 0) */
partition = Louvain(G);
partitionValues = [partition.get(node) for node in G.nodes()];
communityNodes = set(partitionValues);
if ( $len(communityNodes) == len(G.nodes())$ ) then
    | return [G, nodeSizeList];
end
newG = createGraph(G, partition, communityNodes);
newNodeSizes = calculateNodeSize(G, partitionValues, nodeSizeList);
newNodeSizeList = [];
foreach  $nodeSizeGroup$  in  $list(newNodeSizes.values())$  do
    | newNodeSizeList.append(nodeSizeGroup[0]);
end
// write results to JSON file for interactive visualizations
writeJson(newG, newNodeSizes);
// draw static visualizations with NetworkX
draw(newG, newNodeSizeList);
return [newG, newNodeSizeList];
```

Algorithm 3 creates a new graph from the community detection partition results of Algorithm 2.

Algorithm 3: Create Graph

Input: G : NetworkX input graph,
 $partition$: dictionary where LHS = node number in input graph and
 RHS = community number LHS node was assigned to,
 $communityNodes$: set of unique community node numbers
Output: $newG$: new NetworkX graph created from $communityNodes$.
Result: Creates a new graph structure for Algorithm 2's community
 detection partition results.

Function createGraph(G , $partition$, $communityNodes$):

```

    newG = NetworkX.Graph();
    newG.add_nodes_from(communityNodes);
    communityEdges = set();
    foreach  $edge$  of  $G.edges()$  do
        community1 = partition[edge[0]];
        community2 = partition[edge[1]];
        if ( $community1 \neq community2$ ) then
            communityEdges.add((community1, community2));
        end
    end
    newG.add_edges_from(communityEdges);
    return newG;
```

Algorithm 4 calculates how many nodes from the original dataset graph are contained in each of the new communities formed in Algorithm 2. It also counts the number of nodes from the previous community detection iteration graph contained in each community. The results of this algorithm are used for node size scaling and labeling when generating the visualizations.

Algorithm 4: Calculate Node Size

Input: G : NetworkX input graph,
 $partitionValues$: list of community numbers for each node in G ,
 $nodeSizeList$: list of node sizes for graph G

Output: $nodeSize$: dictionary where
LHS = community number,
RHS = list with two elements:
(index 0) number of nodes from the original dataset graph
(index 1) number of nodes from the previous iteration graph

Result: Calculates how many nodes from the original dataset graph and how many nodes from the previous community detection iteration graph are contained in each community from Algorithm 2's community detection partitions.

Function `calculateNodeSize( $G$ ,  $partitionValues$ ,  $nodeSizeList$ )`:

```
    for ( $i = 0$ ;  $i < range(G.number\_of\_nodes()); i++$ ) do
        communityNum = partitionValues[i];
        if ( $communityNum$  not in  $nodeSize$ ) then
            nodeSize[communityNum] = [nodeSizeList[i], 1];
        else
            nodeSize[communityNum][0] += nodeSizeList[i];
            nodeSize[communityNum][1] += 1;
        end
    end
    return nodeSize;
```

3.2 Node Size Scaling

When drawing a graph, node sizes should be representative of how large a community is compared to other communities. This can be measured by a node's raw node size, or how many nodes from the original dataset graph are contained in the community node. Since the datasets contain up to millions of nodes, the raw node sizes can be very large. If visualizations were to be drawn using the raw node sizes, a single node could take up the entire screen. Thus, node size scaling methods are required to best resize the nodes for visualization.

This project explores three different node scaling methods in search of one which would generate optimally sized nodes for graph drawing:

1. Minimum Node Size Scaling
2. Logarithmic Node Size Scaling
3. Capped Node Size Scaling

3.2.1 Minimum Node Size Scaling

Minimum node size scaling divides all raw node sizes by the minimum raw node size. This results in a list where the smallest node has a node size value of one. However, a node with node size equal to one is too small to be easily visible in the graph drawing. Thus, after division, all node sizes are then multiplied by a constant factor of fifty so that the smallest nodes are still visible in the visualization. Algorithm 5 presents the minimum node size scaling function.

Algorithm 5: Minimum Node Size Scaling

Input: *nodeSizeList*: raw node size list

Output: *scaledMinNodeSizeList*: node size list scaled using the minimum node size scaling method

Result: Creates a scaled node size list by dividing the raw node sizes by the minimum node size and multiplying the results by a constant value.

Function *minNodeSize*(*nodeSizeList*):

```
minNodeSize = min(nodeSizeList);
minNodeSizeList = [];
foreach nodeSize in nodeSizeList do
    minNodeSizeList.append(nodeSize / minNodeSize);
end
scaledMinNodeSize = [];
foreach nodeSize in scaledMinNodeSize do
    scaledMinNodeSize.append(nodeSize * 50);
end
return scaledMinNodeSize;
```

3.2.2 Logarithmic Node Size Scaling

Raw node sizes can span over a large range. For instance, the same graph may contain raw node sizes in the single digits as well as raw node sizes in the thousands. The logarithmic node size scaling approach attempts to even out the difference by taking the log of all raw node size values. Logarithmic scaling ensures that small node sizes are not drawn too small and large node sizes are not drawn too big in the visualization.

Logarithmic node size scaling has a special case when the raw node size value is one. Since $\log(1) = 0$, the logarithmic node size value is zero when the raw node size value is one. A node size value of zero means that these nodes cannot be displayed in

the graph drawing. Logarithmic node size values of zero are thus replaced with 0.1 so that all nodes can be represented in the graph.

Finally, similar to minimum node size scaling, logarithmic scaling requires that all node sizes be multiplied by a fixed number, fifty, so that the node sizes are not be too small to see. Algorithm 6 shows the full details of the logarithmic node size scaling method.

Algorithm 6: Logarithmic Node Size Scaling

Input: *nodeSizeList*: raw node size list

Output: *scaledLogNodeSizeList*: node size list scaled using the logarithmic node size scaling method

Result: Creates a scaled node size list by taking the log of the raw node sizes and multiplying the results by a constant value.

Function `logNodeSize(nodeSizeList):`

```

logNodeSizeList = NumPy.log(nodeSizeList);
foreach index, nodeSize in enumerate(logNodeSizeList) do
    if (nodeSize == 0) then
        | logNodeSizeList[index] = 0.1;
    end
end
scaledLogNodeSize = [];
foreach nodeSize in scaledLogNodeSize do
    | scaledLogNodeSize.append(nodeSize * 50);
end
return scaledLogNodeSize;

```

3.2.3 Capped Node Size Scaling

Capped node size scaling defines a constant, MAX_NODE_SIZE , as the maximum node size to be used in the visualization. All raw node sizes are scaled in accordance to the MAX_NODE_SIZE . For instance, the largest raw node size is re-scaled to MAX_NODE_SIZE . The scale factor is equal to MAX_NODE_SIZE divided by the maximum raw node size. All raw node sizes are then multiplied by the scale factor to get the capped node sizes. The predefined MAX_NODE_SIZE value varies depending on the visualization library. Static visualizations created with NetworkX require a MAX_NODE_SIZE value of 1000. Interactive visualizations created by D3, on the other hand, use a MAX_NODE_SIZE value of 8000. The capped node size scaling method is summarized in Algorithm 7.

Algorithm 7: Capped Node Size Scaling

Input: *nodeSizeList*: raw node size list

Output: *cappedNodeSizeList*: node size list scaled using the capped node size scaling method

Result: Scales the raw node sizes based on a predefined constant MAX_NODE_SIZE .

Function *cappedNodeSize*(*nodeSizeList*):

```
    MAX_NODE_SIZE = 8000;
    CAP_SCALE_FACTOR = MAX_NODE_SIZE / max(nodeSizeList);
    cappedNodeSizeList = [];
    foreach nodeSize in nodeSizeList do
        cappedNodeSizeList.append(nodeSize * CAP_SCALE_FACTOR);
    end
    return cappedNodeSizeList;
```

3.3 Layouts

This project explores visualizations in both static and interactive environments. Static versus interactive visualizations use different technologies and libraries and thus have different layout modules. Static visualizations are created with NetworkX, a Python network library. NetworkX hosts a variety of different graph layouts. Four of these layouts best suited for this project are used in the visualization experiments. Interactive visualizations use D3, a JavaScript visualization library. D3 offers a force function which can be customized with different parameters to create varying force-directed layouts. Three different force settings are used to generate interactive layouts in the experiments of this project.

All layouts, with the exception of NetworkX’s Random layout, use force-directed graph drawing algorithms. Force-directed layouts are frequently used for network visualization since they tend to produce aesthetically pleasing results based on criteria such as even node distribution, uniform edge lengths, and symmetry [17, 18]. Force-directed algorithms usually assign forces among the graph nodes and edges. Spring-like forces based on Hooke’s law attract nodes to each other while repulsive forces like those of electric particle repulsion based on Coulomb’s law push nodes apart [19].

3.3.1 Static Layouts

NetworkX produces a still image of graphs and provides several different layout options [3] to determine where nodes should be drawn. Experiments were conducted on four of these layouts:

1. **Random:** The Random layout plots nodes uniformly at random [20]. This layout is used as the baseline to compare the other layouts to.

2. **Kamada-Kawai:** This layout positions nodes using the Kamada-Kawai path-length function [21]. The Kamada-Kawai algorithm is a force-directed graph drawing approach that aims to create an optimal layout positioning by minimizing the total spring energy of the network. There is a spring force between each pair of vertices. The spring lengths are proportional to the nodes' graph-theoretic distance, or the desired Euclidean distance between two nodes in a graph drawing [22].
3. **Spring:** NetworkX's Spring layout draws nodes using the Fruchterman-Reingold force-directed algorithm [23]. The Fruchterman-Reingold algorithm calculates attractive and repulsive forces under the principle that nodes should be drawn towards, but not too close to, each other if they are connected by an edge [18].
4. **NetworkX:** This is NetworkX's default drawing method. It positions nodes using the Spring layout but adds default labels to each of the nodes as well. The nodes are labeled an integer from $[0, n - 1)$ where n is the number of nodes in the graph [24].

3.3.2 Interactive Layouts

Interactive visualizations are created with D3 and use D3's force layout with different parameter settings based on how many nodes are in the graph. D3's force module simulates physical forces on particles using Verlet integration, a method for integrating Newton's equations of motion [4, 25].

D3's force function can be customized through various parameter settings. Small versus large networks require different parameters to produce the most user friendly

interactive visualizations for the given graph. For this project, a small graph is defined as a graph with fifty or fewer nodes. A large graph is defined as a graph with over fifty nodes. There are four parameter values manipulated to achieve the optimal force settings for different sized graphs:

1. **Link Distance:** Link distance is the distance between edges [4]. Large graphs can benefit from longer link distances to minimize overlapping nodes.
2. **Charge:** Charge measures the level of attraction or repulsion between nodes. Nodes are repelled from each other if the charge value is negative and attracted to each other if the value is positive. It is recommended that a negative value be used for graph layouts [4]. Since small graphs have only a few nodes, they can use stronger repulsive forces than large graphs, which may become too spread out by strong negative charges.
3. **Gravity:** Gravity acts as a virtual spring between each node and the center of the visualization [4]. A low gravity value means that the nodes are more dispersed and take longer to settle into position since the gravitational pull towards the visualization center is weak. In contrast, a high gravity value quickly converges towards the center due to the strong gravitational pull. The larger the graph, the longer it takes for the force simulation to settle into position. Thus, a higher gravity value is used for large graphs in attempt to make the graph converge faster.
4. **Friction:** Particles, or the nodes of the graph, have velocities in a force simulation. Friction is the measure by which the particle velocity decays each step of the simulation. Friction values lie in the range $[0, 1]$, where friction equal to zero freezes all particles in place, and friction equal to one simulates a friction-less

environment [4]. A lower friction value allows the force simulation to stabilize faster, which is more desirable for large graphs which tend to stay in motion for an extended period of time.

Three different force settings were tested for the visualization layouts in this project:

1. **Default:** This layout uses all of the default D3 force attribute values and serves as a baseline to compare the other layouts against.

- Link Distance: 20
- Charge: -30
- Gravity: 0.1
- Friction: 0.9

2. **Small:** The Small layout is a customized to create the best interactive visualizations for graphs with fifty or fewer nodes.

- Link Distance: 200
- Charge: -2000
- Gravity: 0.5
- Friction: 0.7

3. **Large:** The Large layout uses custom parameters to create interactive visualizations optimized for graphs with over fifty nodes.

- Link Distance: 250

- Charge: -600
- Gravity: 0.7
- Friction: 0.5

3.4 Labeling

Labeling the nodes in a network visualization is useful for quickly conveying useful information about the graph to the user. However, early experiments quickly showed that labels are not particularly effective in static visualizations. This was gathered from static visualizations created using the NetworkX layout, which automatically adds node number labels. When there are multiple nodes in near proximity to one another, the labels tend to overlap, making the labels difficult to read. The overlapping labels make static visualizations less aesthetically pleasing and fail to convey useful information to the user. Thus, poor label readability in static visualizations leads to undesirable visualizations, and the following labeling procedures are only applied to interactive visualizations. The static NetworkX layout drawings with labels are kept for comparison with the identical Spring layout drawings, which do not have labels. The rest of the static visualizations do not include labels to maximize graph readability.

In interactive visualizations, this project hopes to communicate information on community size through node labeling. Nodes are labeled with a number, x , or the number of nodes from the original graph contained in each community. If the graph is the result of more than one iteration of community detection, another number, y , is appended to the node label. y is the number of nodes from the previous iteration graph partitioned into the corresponding community node of the current visualization. The label is displayed in the format " x, y ".

For example, suppose a network has ten nodes total. The first iteration of community detection creates three community nodes, A , B , and C . Community A contains one node, community B contains two nodes, and community C contains seven nodes. Communities A , B , and C are labeled "1", "2", and "7" respectively, according to the number of nodes from the original network contained in each community. In the second iteration of community detection, communities A and B are combined into a single node, community D . Community D 's label is "3,2" because it contains three nodes from the original graph and two nodes from the previous community detection iteration.

3.5 Edge Coloring

Following edge paths in a graph is a challenging task if all edges are the same color. It is especially difficult to differentiate edges if multiple edges cross at small angles. Inspired by Hu, Shi, and Liu's [5] work on edge coloring, this project incorporates random edge coloring to differentiate edge crossings. This project randomly colors visualization edges with colors from a predefined color scale to create visual differentiation between edges. Interactive visualizations use D3's "category20" color scale, an ordinal scale with a range of twenty colors [26]. Static visualizations use Matplotlib's "tab20" color map, Matplotlib's equivalent of the same twenty color scale.

3.5.1 Node Coloring

However, even with edge coloring, node distinctions are not always clear when nodes overlap. This is because all nodes are the same color, and NetworkX does not provide a method to draw node outlines. Thus, static visualizations are also drawn

with node coloring instead of edge coloring to make node differentiation more clear. When node distributions are not understandable from the edge coloring visualization results, the equivalent node coloring visualization can be used for clarification and comparison. Different node colors are applied according to the Matplotlib's "jet" color-map, a spectral color mapping based on jet simulations [27]. An example of edge versus node coloring is shown in Figure 3.

Figure 3: Static visualizations which demonstrate edge coloring (left) versus node coloring (right) on the Amazon dataset.

Node coloring is not necessary in interactive visualizations since node differentiation can be achieved with node borders in D3. Figure 4 shows how node borders help distinguish overlapping nodes in interactive visualisations. Without the issue of differentiating overlapping nodes, edge coloring provides more benefits than node coloring since it can help differentiate edges crossings. Furthermore, node coloring can give the false impression that nodes of similar colors are related, which is not the case in this project.

Figure 4: Interactive visualization with edge coloring and node borders on the Wikipedia dataset.

3.6 Interactivity

Adding interactivity to graphs gives users many helpful capabilities such as zooming, dragging, and highlighting node connections [28]. These features can significantly enhance the visualization experience. Zooming and panning allows users to explore the graph at their desired level of detail. Users can zoom into an area of interest or zoom out to view the graph as a whole. Interactive visualizations do not restrict nodes to a fixed position. Thus, if a node happens to be obstructing other nodes or edge connections that the user is interested in, the user can simply drag the node to another position. This makes dragging particularly useful for differentiating overlapping nodes. Dragging can also help single out a particular node the user is looking for. Node highlighting can further improve node focus by highlighting the selected node and its connections. When a node and its connections are highlighted, all other nodes

and edges are faded in the background to draw emphasis to the selection. This is especially beneficial for quickly identifying inter-community relationships in a graph. Figure 5 shows an example of how a node and its connections are highlighted in the Email dataset.

Figure 5: Node highlighting example on Email dataset.

CHAPTER 4

Setup of Experiments

4.1 Datasets

The experiments are conducted on five datasets of varying sizes from the Stanford Network Analysis Project (SNAP) Stanford Large Network Dataset Collection [29]. These are networks with ground truth communities that can be used for comparison. Table 1 summarizes the characteristics of each dataset.

Table 1: SNAP Networks with Ground Truth Communities [29].

Name	Nodes	Edges	Communities	Description
Email	1,005	25,571	42	E-mail network
DBLP	317,080	1,049,866	13,477	DBLP collaboration network
Amazon	334,863	925,872	75,149	Amazon product network
YouTube	1,134,890	2,987,624	8,385	YouTube online social network
Wikipedia	1,791,489	28,511,807	17,364	Wikipedia hyperlinks

The details of each dataset are as follows:

1. **Email:** This is a dataset of emails from a European research institute. Each node is a person from the institute. There is an edge between institute members if one member has sent the other at least one email [30].
2. **DBLP:** The DBLP computer science bibliography is home to a list of computer science research papers and proceedings. This dataset is an author collaboration network where the nodes are authors and two authors are connected if they have published at least one paper together [31].

3. **Amazon:** The Amazon dataset is a network of products on Amazon which are frequently bought with each other. Every node is a product, and there exists an edge between two products if they are frequently co-purchased [32].
4. **YouTube:** While primarily known as a video sharing platform, YouTube used to have a social network aspect as well. YouTube had a contacts feature which allowed YouTube members to connect with one another. This dataset shows these connections between YouTube members. Users are represented by nodes and an edge is added between nodes if the users are connected [33]. YouTube has since discontinued the contacts feature.
5. **Wikipedia:** This dataset is a directed network of Wikipedia hyperlinks. Each node is a Wikipedia page, and there is an edge between nodes if one page links to the other [34].

These datasets have been preprocessed by SNAP and are packaged in a text file. Each line of the file consists of two numbers. Every number represents a unique node, and a line containing two numbers represents an edge between those two nodes.

4.2 Technologies

This project is implemented with Python, HTML, CSS, and JavaScript and utilizes the following libraries:

- NetworkX: Python graph and networks library
- Python-Louvain: Python Louvain community detection library
- Matplotlib: Python plotting and visualization library
- NumPy: Python multidimensional array and math library

- D3.js: JavaScript interactive data visualization library

4.3 System Requirements

Given the large size of some datasets, a MacBook Pro with 16GB memory proved insufficient for effectively processing large amounts of data. Experiments were subsequently moved to the cloud in order to increase the system memory capacity and computing power.

Experiments were performed on a Google Cloud Platform Compute Engine virtual machine (VM) instance with machine type e2-highmem-4. The e2-highmem-4 machine type is a general purpose machine with 4 vCPUs and 32GB memory [35]. This helped improve the program execution speed and resolve out of memory errors.

However, data files over about 450MB could not be uploaded to the VM due to session timeout in the middle of upload. This prevented testing on some of the extremely large datasets available on SNAP [29].

The interactive visualizations were hosted on localhost with XAMPP 7.3.1 for OS X. XAMPP is an open source web server solution [36].

4.4 User Survey

This project conducts a user survey to measure the effectiveness of different visualization layouts. Users are presented with two questions for every visualization:

1. How visually pleasing is the visualization?
2. How effective is the visualization in displaying inter-community relationships?

The users can answer the questions on a scale of one to three where:

- 1 = good
- 2 = average
- 3 = bad

Seven different layouts are judged in the survey:

- Static Layouts
 1. Random
 2. Kamada-Kawai
 3. Spring
 4. NetworkX
- Interactive Layouts
 1. Default
 2. Small
 3. Large

A graph is created for every iteration of community detection for each of the five SNAP datasets. If a resulting graph has more than three nodes, it is drawn in all seven different layouts and included in the survey. This survey presents eight different graphs drawn in seven layouts each. There are thus a total of 56 visualizations in this survey.

A web server to publicly host interactive visualizations could not be attained for this project. As a result, users were unable to view interactive visualizations in an

interactive web environment as intended. Instead, users were shown screenshots of the interactive visualizations. Due to this limitation, the full advantages of interactive layouts could not be accurately captured by this survey.

The user survey was created using Qualtrics’s online survey software tool [37]. The survey results data and analysis were also gathered from Qualtrics. The survey was distributed to Dr. Katerina Potika’s San Jose State University (SJSU) students and Xinyuan Fan’s family. A total of 21 responses were collected. The full survey is available in Appendix A.

CHAPTER 5

Results

5.1 Recursive Community Detection

Recursive community detection is performed on an input graph until the minimum number of communities is reached. The goal is to condense a large network into graphs of its communities. These community graphs have fewer nodes and are thus more manageable to visualize. The maximum number of nodes in a graph drawing is configurable so the user can skip visualizing the graph if it is still too big after community detection. This project is interested in seeing how many community detection iterations are required before a network converges to the minimum number of communities for each dataset. Table 2 shows a summary of the datasets and how many community detection iterations were required for each. Table 3 goes over the detailed community detection results for each dataset.

Table 2: Number of community detection iterations performed on each dataset given the original network properties.

Dataset	Nodes	Edges	Communities	Community Detection Iterations
Email	1,005	25,571	42	2
DBLP	317,080	1,049,866	13,477	3
Amazon	334,863	925,872	75,149	3
YouTube	1,134,890	2,987,624	8,385	3
Wikipedia	1,791,489	28,511,807	17,364	3

All datasets except for the Email dataset converged to a single node in three iterations of community detection. The Email dataset converged to a graph of twenty disconnected nodes in the second community detection iteration. This can be explained by the fact that the Email dataset is significantly smaller than the other

Table 3: Detailed community detection results for all datasets.

Dataset	Ground Truth Communities	Iteration	Nodes	Edges
Email	44	0	1,005	25,571
		1	28	36
		2	20	0
DBLP	13,477	0	317,080	1,049,866
		1	203	6,067
		2	6	15
		3	1	0
Amazon	75,149	0	334,863	925,872
		1	238	4,370
		2	3	3
		3	1	0
YouTube	8,385	0	1,134,890	2,987,624
		1	5,739	10,496
		2	7	21
		3	1	0
Wikipedia	17,364	0	1,134,890	2,987,624
		1	37	503
		2	2	1
		3	1	0

datasets. Due to the immense quantity of source material for the larger datasets, SNAP has already preprocessed the data to exclude nodes with no connections or else the networks would become extremely large [29]. But since the Email dataset only has one thousand nodes compared to hundreds of thousands or millions of nodes of other datasets, this level of data clean up is not necessary. If the Email dataset did not include any disconnected nodes, it is reasonable to assume that the first round of community detection would create a graph of nine nodes and 36 edges, and the second round of community detection would converge to a graph with one node and zero edges.

The communities end up converging much faster than originally anticipated. The initial communities generated by the Louvain method for community detection are always smaller than the ground truth number of communities, sometimes significantly so. For instance, the Wikipedia dataset has 17,364 ground truth communities, but the first iteration of community detection only produces 37 communities from the 1,134,890 node input graph. In this case, 37 nodes is a fair number for visualization and produces a graph that is neither too large nor small. However, sometimes the large jumps in iteration sizes skip over the ideal graph size range for visualization. The YouTube dataset, for instance, has 5,739 nodes in the first iteration and seven nodes in the second iteration. Graphs with over a few hundred nodes can look overwhelming in a visualization and have performance issues in an interactive environment. But graphs with less than ten nodes sometimes do not convey enough information to present a clear image of community relationships.

The fast convergence rate is likely due to the fact that the Louvain algorithm itself is already iterative in nature. The Louvain method partitions nodes into small communities and then combines those nodes into one node. The process is repeated until the modularity cannot be further optimized [38]. This could explain why the number of nodes drops so quickly between each iteration of Louvain community detection in this project.

Despite the large differences in graph sizes between community detection iterations, the recursive community detection method is very effective at shrinking networks for visualization. It would take impractically long to draw every node and edge for large networks. Furthermore, trying to fit too much visual information into a limited visualization space conversely makes the graph drawing overcrowded and detracts from the visualization effectiveness. Community detection condenses the graph

size while retaining the community structure of the network. It is a powerful tool for visualizing networks too large to be graphed in their entirety.

5.2 Node Size Scaling

Experiment results showed that capped node scaling produced the best visualization results followed by logarithmic node size scaling, then minimum node size scaling. Most of the minimum node size scaling results included nodes far too large for effective visualizations. Both logarithmic and capped node size scaling produce aesthetically pleasing visualizations, but the capped node size scaling results are more proportionally accurate to how many nodes from the original network are contained in a community. A comparison of the three different node size scaling results is shown in Figure 6. Since capped node size scaling consistently produced the best results across all datasets, all of the experiment visualizations presented outside of Section 5.2 are graphed using capped node size scaling.

5.2.1 Minimum Node Size Scaling

Minimum node size scaling was the first of the three node size scaling methods formulated for this project. As an initial attempt, it failed to take into account the case where the difference between the largest and smallest raw node sizes is very large. In such cases, dividing all raw node sizes by the minimum raw node size would not do much to control the size of the largest nodes. Thus, minimum node size scaling did not scale well for large datasets, which tend to have more variation in raw node sizes due to the sheer quantity of nodes. There are various instances, like that shown in Figure 7, where one node can fill up the entire visualization. These visualizations are practically useless, making minimum node size scaling the worst scaling method.

Figure 6: A comparison of the results of three different node size scaling methods: minimum (top), logarithmic (bottom left), and capped (bottom right). Visualizations are for the Wikipedia dataset (iteration 1).

Figure 7: YouTube dataset (iteration 1) drawn with minimum node size scaling to demonstrate a case where the node takes up the entire visualization.

5.2.2 Logarithmic Node Size Scaling

Logarithmic node size scaling is a significant improvement from minimum node size scaling. There are no cases where node sizes become so large that they obstruct

the overall graph structure of the visualization. However, logarithmic scaling has the problem of making all nodes appear similar in size. One node could have a raw node size one hundred times larger than that of another node. But in the visualization, the two node sizes would not appear very different due to the nature of logarithmic scales. A comparison of logarithmic versus capped node size scaling is presented in Figure 8. It reveals how uniform the logarithmic node sizes appear compared the capped node sizes which are linearly scaled.

Figure 8: Logarithmic node size scaling (left) versus capped node size scaling (right) on the Amazon dataset (iteration 1).

5.2.3 Capped Node Size Scaling

The capped node size scaling method was conceived last to fix the problems revealed by the minimum and logarithmic node size scaling algorithms. Capped node size scaling sets a maximum node size so that nodes do not become too big like in minimum node size scaling. It also uses linear, rather than logarithmic, scaling so that node sizes remain proportional to the raw node sizes. By evolving from the failures of previous node scaling methods, it makes sense for capped node size scaling to produce the best visualization results. An area for improvement for capped node size scaling would be to set a minimum node size in addition to the maximum node

size for graph drawing. This is because the current implementation can result in cases where some nodes are too small to be viewed clearly. An example of this is shown in Figure 9.

Figure 9: Email dataset (iteration 2) drawn with capped node size scaling to demonstrate a case where the node sizes are too small.

5.3 Layouts

A user survey was conducted to measure how aesthetically pleasing graph drawings for the five SNAP datasets were in different layouts. The survey also asked users to rate how well each of the layouts conveyed inter-community relationships. All survey visualizations are available in Appendix A. The visualizations were rated on a scale of one to three, where "1" is good, "2" is average, and "3" is bad. The results were determined based on which layout was rated "1" the most times. If there were multiple layouts with the same number of votes for "1," then the layout with the average rating closest to "1" was chosen. Table 4 shows a summary of the survey results. Tables 5 and 6 break down the results by static versus interactive layouts and show the corresponding average ratings of the top layouts.

Table 4: Data visualization survey results summary.

Visualization	Nodes	Edges	Most Visually Appealing Layout	Layout with Best Community Structure
Email Iteration 1	27	28	Large	Large
Email Iteration 2	20	0	NetworkX	Large
DBLP Iteration 1	181	6,443	Large	Spring
DBLP Iteration 2	5	10	Large	Large
Amazon Iteration 1	244	4,392	Large	Large
YouTube Iteration 1	5,798	10,432	Small	Spring
YouTube Iteration 2	7	21	Large	Small
Wikipedia Iteration 1	33	445	Kamada-Kawai	Kamada-Kawai

Table 5: Data visualization survey results for the best layout based on visualization aesthetics split by static versus interactive layouts.

Visualization	Best Static Layout	Best Interactive Layout	Best Overall Layout
Email 1	Kamada-Kawai (1.57)	Large (1.48)	Large (1.48)
Email 2	NetworkX (1.67)	Large (1.71)	NetworkX (1.67)
DBLP 1	Kamada-Kawai (1.71)	Large (1.62)	Large (1.62)
DBLP 2	Spring (1.57)	Large (1.33)	Large (1.33)
Amazon 1	Kamada-Kawai (1.71)	Large (1.57)	Large (1.57)
YouTube 1	Spring (1.67)	Small (1.57)	Small (1.57)
YouTube 2	Spring (1.52)	Large (1.38)	Large (1.38)
Wikipedia 1	Kamada-Kawai (1.57)	Large (1.67)	Kamada-Kawai (1.57)

The survey results show that the Large interactive layout tends to produce the most aesthetically pleasing visualizations and also has the best community structure. Unexpectedly, the Large layout, which was created for interactive visualizations of large graphs, was often rated as the best layout for both small and large graphs. The Small layout, on the other hand, was voted the best layout for the largest graph. Multiple of the interactive layout force parameters, such as gravity and friction, were chosen with the intention of helping the interactive layouts settle quickly within the

Table 6: Data visualization survey results for the best layout based on representation of community relationships split by static versus interactive layouts.

Visualization	Best Static Layout	Best Interactive Layout	Best Overall Layout
Email 1	Kamada-Kawai (1.57)	Large (1.52)	Large (1.52)
Email 2	Spring (1.90)	Large (1.71)	Large (1.71)
DBLP 1	Spring (1.67)	Large (1.71)	Spring (1.67)
DBLP 2	NetworkX (1.52)	Large (1.43)	Large (1.43)
Amazon 1	Spring (1.81)	Large (1.71)	Large (1.71)
YouTube 1	Spring (1.57)	Large (1.76)	Spring (1.57)
YouTube 2	Kamada-Kawai/ NetworkX (1.57)	Small (1.33)	Small (1.33)
Wikipedia 1	Kamada-Kawai (1.62)	Small/Large (1.67)	Kamada-Kawai (1.62)

visualization frame. Since larger graphs are more difficult to contain in the visualization window due to their size, the Large layout attempts to make the graph visualization more compact. This way, the visualization can stabilize faster, and less zooming out is required to view the full visualization. This may be useful in an interactive environment, but since the survey is limited to static screenshot representations of interactive visualizations, users would only see the final visualization pre-zoomed out to best fit the entire graph. Disregarding interactive elements such as load time and framing, the Large layout received better ratings than the Small layout for almost all graphs.

In many cases, the layout with the best looking visualization matched with the layout with the best community structure for a given network. Interactive layouts generally outperformed static layouts in both aesthetics and representation of inter-community relationships. Out of the static layouts, the Kamada-Kawai layout produced the most visually appealing visualizations, and the Spring layout was the best at communicating inter-community relationships. Overall, the Large layout seemed

to perform the best with consistently good ratings for visualization aesthetics and community structure.

5.4 Labeling

Labeling in interactive visualizations is successful in conveying community size information about the number of nodes from the original graph and the number of nodes from the previous iteration graph contained in each community. It is useful to see the true community size of a node beyond what can be inferred from the node size. Node size is proportional to the number of nodes from the original graph contained in a community, but it is unable to convey exactly how large the original dataset is. Nodes in two different dataset visualizations could be drawn with the same node size but contain significantly different numbers of nodes from their respective dataset graphs. Community size labeling solves this issue by explicitly labeling nodes with the number of communities contained. This reveals information about the scale of the original network while keeping the visualization graph size small.

5.5 Edge Coloring

Drawing edges in different colors is an effective method for differentiating edges in a visualization. The experimental results of this project found edge coloring to be the most effective on small graphs. This is in line with the findings of Hu, Shi, and Liu’s edge coloring work [5]. When a graph contains thousands of edges, edge coloring alone is insufficient to trace node connections. Highlighting node connections in interactive visualizations is a powerful capability that allows the user to single out a node and its connections from a graph. This feature is especially useful in medium to large size graphs when edge coloring is insufficient for identifying node connections.

5.6 Interactivity

Experiments showed that interactivity is most effective on smaller graphs. As graph size increases, visualizations take longer to stabilize and become less responsive. Stabilization time in interactive visualizations refers to how long it takes for the D3 force simulation to stop. The visualization is settled when all nodes and edges in the graph have stopped moving. For small graphs with less than fifty nodes, the force layout settles within a few seconds, and the graph fits neatly on screen. Performance is good, and there is no lag when highlighting node connections. Interactive visualizations of medium sized graphs, graphs with a few hundred nodes, take longer to settle into position and show mild performance issues. Medium sized graphs take about 30 seconds for the force simulation to stop. They also have a few seconds of lag when trying to highlight node connections. While less effective than on small graphs, interactive visualizations are still useful on medium sized networks, and the performance impacts are relatively minor.

Large graphs with over a thousand nodes, on the other hand, do not perform very well in interactive environments. Visualizations of large graphs can take over two minutes to stabilize, and the full visualizations do not fit in the window screen. Figure 10 shows the initial view of the large YouTube Iteration 1 graph after it has stabilized. It can only fit a portion of the visualization on screen, so the user must manually zoom out to view the full visualization. There are also about four seconds of lag when trying to highlight node connections in large graphs. But even with the performance drawbacks of interactivity on large graphs, interactive visualizations are still more useful than their static counterparts. Zooming and panning are especially useful for large graphs to see details which may otherwise be lost in static visualizations. The slow layout stabilization time is inconvenient but not detrimental to user experience

as the user can manually stop the force simulation by pressing the space bar.

For node highlighting, the majority of the lag time comes from fading out the unconnected nodes and edges to the background. As a workaround, the lag time can be greatly reduced by highlighting node connections without fading out the rest of the graph. This can be achieved by hovering over a node instead of clicking on it. Figure 11 shows an example of node highlighting with and without fading out unrelated nodes and edges on the YouTube Iteration 1 visualization. It is clear from the example that node highlighting without fading out the background can be just as effective as highlighting with fading out the background for large communities. This alternative method of node highlighting does not work as well for small communities since the connecting edges may be hidden by other unrelated edges. In this case, the unrelated nodes and edges must be greyed out in order to reveal the highlighted edge connections. However, as a means to quickly get a big picture view of large community connections, highlighting without fading out the rest of the graph is a good solution.

Table 7 lists the layout stabilization and node highlighting times for all visualizations.

Figure 10: The initial view of the YouTube Iteration 1 interactive graph drawing does not fit the entire visualization in screen.

Figure 11: Node highlighting with (left) and without (right) fading out unconnected nodes and edges example on the YouTube dataset (iteration 1).

Table 7: Force layout stabilization times and node highlighting lag times for interactive visualizations.

Visualization	Graph Size	Force Layout Stabilization Time (s)	Node Highlighting Lag Time (s)
Email 1	Small	5	0
Email 2	Small	5	0
DBLP 1	Medium	30	2
DBLP 2	Small	3	0
DBLP 3	Small	1	0
Amazon 1	Medium	23	1.5
Amazon 2	Small	1	0
Amazon 3	Small	1	0
YouTube 1	Large	127	4
YouTube 2	Small	3	0
YouTube 3	Small	1	0
Wikipedia 1	Small	6	0
Wikipedia 2	Small	1	0
Wikipedia 3	Small	1	0

CHAPTER 6

Conclusion and Future Work

6.1 Conclusion

Data visualization is a powerful tool for quickly and effectively communicating large amounts of information. Network visualization in particular is useful for conveying information about relationships between network actors. Networks are represented with graphs, but creating visually appealing graph drawings can be difficult, especially when networks are large. This project aims to optimize visualization aesthetics and effectively convey inter-community relationships in a network. Graph drawings are created over five known datasets using a combination of techniques including recursive community detection, node size scaling, layout formation, labeling, edge coloring, and interactivity.

By applying recursive community detection, large networks with up to millions of nodes can be succinctly expressed in smaller graph sizes better suited for visualization. Graphs usually converge to the minimum number of communities in three community detection iterations. This is an effect of the Louvain community detection algorithm, which experiments showed vastly decreased the number of nodes in each iteration. The steep jumps in graph size led the recursion to finish within three iterations even for large datasets.

Capped node size scaling proved to be the most effective node size scaling method as it helped create aesthetically pleasing visualizations while keeping the node sizes proportionally accurate to the number of nodes from the original graph contained in each community node. Based on a survey of visualizations in seven different layouts,

the Large interactive layout produced the most visually appealing visualizations and was the most effective in showing the networks' community structure. Labeling in interactive visualizations also helped convey key community size details by showing the number of nodes from the original graph and the number of nodes from the previous community detection iteration graph contained in each community node. Edge coloring helps differentiate edge crossings but loses effectiveness as graph size increases. Interactive visualizations introduce node highlighting, which can be used to highlight node connections when edge coloring is insufficient for distinguishing edges. Interactivity also adds zooming, panning, and dragging capabilities. With these additional features, interactive visualizations are significantly more powerful than static graph drawings. But interactivity is best suited for smaller graphs since interactive visualizations become less responsive as graph size grows.

6.2 Future Work

This project can be further extended by graphing the nodes within each community. For instance, a user may be interested in a particular community and want to examine the relationships within the community in more detail. In an interactive environment, it would be useful if the user could click on a community node and be brought to a new visualization of the nodes within the community. However, a single community could contain hundreds of thousands of nodes, which would be difficult to visualize all at once. Recursive community detection can be used in this case to create a smaller graph of from the community network. This way a large network can be broken down into multiple sub-graphs of more manageable size where any specific community can still be viewed in more detail.

Different community detection algorithms other than Louvain can also be tested to see the number of communities produced in each iteration of recursive community detection. The goal would be to find a community detection algorithm that does not shrink the graph size too significantly between iterations. Another area for improvement would be to add a minimum node size constant in addition to the existing maximum node size constant for capped node size scaling. This will create a linear mapping from the raw node sizes to the range between the constant maximum and minimum visualization node size values.

Finally, it would be useful to label community nodes based on their contents instead of their community sizes. For instance, the Amazon products dataset could have labels such as "Kitchen Appliances" or "Electronics." Content based labels are useful for conveying what each community actually represents. However, finding datasets with such metadata available can be challenging. The datasets used in this project come in a format where each node is represented by a number. But information about what each number represents is not given. This makes it impossible to label nodes according to their contents for this project. If appropriate datasets with content metadata can be found, content based labels would be very helpful for communicating what each community represents in a graph visualization.

LIST OF REFERENCES

- [1] F. Amin, A. Ahmad, and G. S. Choi, “Community detection and mining using complex networks tools in social internet of things,” in *TENCON 2018-2018 IEEE Region 10 Conference*, pp. 2086–2091, IEEE, 2018.
- [2] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, “Fast unfolding of communities in large networks,” *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, p. P10008, oct 2008.
- [3] NetworkX, “Drawing - NetworkX 2.5 documentation.” <https://networkx.org/documentation/stable/reference/drawing.html>, August 2020. Accessed 2020-12-05.
- [4] D3, “Force layout - D3 wiki.” https://d3-wiki.readthedocs.io/zh_CN/master/Force-Layout/. Accessed 2020-11-29.
- [5] Y. Hu, L. Shi, and Q. Liu, “A coloring algorithm for disambiguating graph and map drawings,” *IEEE transactions on visualization and computer graphics*, vol. 25, no. 2, pp. 1321–1335, 2018.
- [6] L. Gutiérrez-Gómez and J.-C. Delvenne, “Unsupervised network embedding for graph visualization, clustering and classification,” *arXiv preprint arXiv:1903.05980*, 2019.
- [7] B. Dumba and Z.-L. Zhang, “Uncovering the nucleus of social networks,” in *Proceedings of the 10th ACM Conference on Web Science*, pp. 37–46, ACM, 2018.
- [8] O.-H. Kwon, T. Crnovrsanin, and K.-L. Ma, “What would a graph look like in this layout? a machine learning approach to large graph visualization,” *IEEE transactions on visualization and computer graphics*, vol. 24, no. 1, pp. 478–488, 2017.
- [9] Y. Wang, Z. Jin, Q. Wang, W. Cui, T. Ma, and H. Qu, “Deepdrawing: A deep learning approach to graph drawing,” *IEEE transactions on visualization and computer graphics*, 2019.
- [10] M. E. J. Newman, “From the Cover: Modularity and community structure in networks,” *Proceedings of the National Academy of Science*, vol. 103, pp. 8577–8582, June 2006.

- [11] Wikipedia, “Modularity (networks).” [https://en.wikipedia.org/wiki/Modularity_\(networks\)](https://en.wikipedia.org/wiki/Modularity_(networks)), October 2020. Accessed 2020-11-26.
- [12] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner, “On modularity clustering,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 20, no. 2, pp. 172–188, 2008.
- [13] K. Potika, “Strength of weak ties in networks.” San Jose State University CS286 Lecture, 2020.
- [14] M. Girvan and M. E. J. Newman, “Community structure in social and biological networks,” *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.
- [15] A. Lancichinetti and S. Fortunato, “Community detection algorithms: A comparative analysis,” *Phys. Rev. E*, vol. 80, p. 056117, Nov 2009.
- [16] M. E. J. Newman, “Fast algorithm for detecting community structure in networks,” *Phys. Rev. E*, vol. 69, p. 066133, Jun 2004.
- [17] R. Tamassia, ed., *Handbook of Graph Drawing and Visualization*. CRC Press, June 2013.
- [18] T. M. J. Fruchterman and E. M. Reingold, “Graph drawing by force-directed placement,” *Softw. Pract. Exper.*, vol. 21, p. 1129–1164, Nov. 1991.
- [19] Wikipedia, “Force-directed graph drawing.” https://en.wikipedia.org/wiki/Force-directed_graph_drawing, April 2020. Accessed 2020-11-28.
- [20] NetworkX, “networkx.drawing.layout.random_layout.” https://networkx.org/documentation/stable/reference/generated/networkx.drawing.layout.random_layout.html, August 2020. Accessed 2020-11-28.
- [21] NetworkX, “networkx.drawing.layout.kamada_kawai_layout.” https://networkx.org/documentation/stable/reference/generated/networkx.drawing.layout.kamada_kawai_layout.html, August 2020. Accessed 2020-11-28.
- [22] T. Kamada and S. Kawai, “An algorithm for drawing general undirected graphs,” *Information Processing Letters*, vol. 31, no. 1, pp. 7 – 15, 1989.
- [23] NetworkX, “networkx.drawing.layout.spring_layout.” https://networkx.org/documentation/stable/reference/generated/networkx.drawing.layout.spring_layout.html, August 2020. Accessed 2020-11-28.

- [24] NetworkX, “networkx.drawing.nx_pylab.draw_networkx.” https://networkx.org/documentation/stable/reference/generated/networkx.drawing.nx_pylab.draw_networkx.html, August 2020. Accessed 2020-11-28.
- [25] L. Verlet, “Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules,” *Phys. Rev.*, vol. 159, pp. 98–103, Jul 1967.
- [26] M. Bostock, “Ordinal scales.” <https://github.com/d3/d3-3.x-api-reference/blob/master/Ordinal-Scales.md>. Accessed 2020-11-29.
- [27] Kite, “colormaps.” <https://www.kite.com/python/docs/matplotlib.pyplot.colormaps>. Accessed 2020-11-29.
- [28] Eyaler, “Force-Directed Graph with Drag/Zoom/Pan/Center/Resize/Labels/Shapes/Filter/Highlight.” <https://gist.github.com/eyaler/10586116>. Accessed 2020-12-07.
- [29] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection.” <http://snap.stanford.edu/data>, June 2014. Accessed 2020-11-17.
- [30] J. Leskovec and A. Krevl, “SNAP: Network datasets: email-eu-core network.” <https://snap.stanford.edu/data/email-Eu-core.html>, June 2014. Accessed 2020-11-27.
- [31] J. Leskovec and A. Krevl, “SNAP: Network datasets: DBLP collaboration network.” <https://snap.stanford.edu/data/com-DBLP.html>, June 2014. Accessed 2020-11-27.
- [32] J. Leskovec and A. Krevl, “SNAP: Network datasets: Amazon co-purchasing network.” <https://snap.stanford.edu/data/com-Amazon.html>, June 2014. Accessed 2020-11-27.
- [33] J. Leskovec and A. Krevl, “SNAP: Network datasets: Youtube social network.” <https://snap.stanford.edu/data/com-Youtube.html>, June 2014. Accessed 2020-11-27.
- [34] J. Leskovec and A. Krevl, “SNAP: Network datasets: wiki-topcats network.” <https://snap.stanford.edu/data/wiki-topcats.html>, June 2014. Accessed 2020-11-27.
- [35] Google Cloud, “Machine types | Compute Engine Documentation | Google Cloud.” <https://cloud.google.com/compute/docs/machine-types>. Accessed 2020-11-17.
- [36] Apache Friends, “XAMPP Installers and Downloads for Apache Friends.” <https://www.apachefriends.org/index.html>. Accessed 2020-11-26.

- [37] Qualtrics, “Qualtrics XM // The Leading Experience Management Software.” <https://www.qualtrics.com/>. Accessed 2020-11-30.
- [38] V. A. Traag, L. Waltman, and N. J. van Eck, “From louvain to leiden: guaranteeing well-connected communities,” *CoRR*, vol. abs/1810.08473, 2018.

APPENDIX

Data Visualization User Survey Questions

Please answer the following two questions for every visualization.

1. How visually pleasing is the visualization?
2. How effective is the visualization in displaying inter-community relationships?

Please rate the visualizations on a scale of one to three where:

- 1 = good
- 2 = average
- 3 = bad

A.1 Email Iteration 1

Figure A.12: Email Iteration 1 draw in seven layouts.

A.2 Email Iteration 2

Figure A.13: Email Iteration 2 drawn in seven layouts.

A.3 DBLP Iteration 1

(a) Random

(b) Kamada-Kawai

(c) Spring

(d) NetworkX

(e) Default

(f) Small

(g) Large

Figure A.14: DBLP Iteration 1 drawn in seven layouts.

A.4 DBLP Iteration 2

Figure A.15: DBLP Iteration 2 drawn in seven layouts.

A.5 Amazon Iteration 1

(a) Random

(b) Kamada-Kawai

(c) Spring

(d) NetworkX

(e) Default

(f) Small

(g) Large

Figure A.16: Amazon Iteration 1 drawn in seven layouts.

A.6 YouTube Iteration 1

(a) Random

(b) Kamada-Kawai

(c) Spring

(d) NetworkX

(e) Default

(f) Small

(g) Large

Figure A.17: YouTube Iteration 1 drawn in seven layouts.

A.7 YouTube Iteration 2

Figure A.18: YouTube Iteration 2 drawn in seven layouts.

A.8 Wikipedia Iteration 1

Figure A.19: Wikipedia Iteration 1 drawn in seven layouts.