Algorithms 2500

Lecture 1B

Project Two

Second Report

Name 1: Brady Snedden

Student I.D. 1: 12422881

Name 2: Ryon Owings

Student I.D. 2: 12433574

Name 3: Alex Shaw

Student I.D. 3: 12423794

Project Goal: Analyze and characterize follower-followee Social Networks with respect to topological metrics

Abstract: There are many social media websites which use the follower-followee structure. By using graph structures, we can see how to improve this follower-followee structure and lead it to better optimization. We will be finding ways to optimize these structures by finding their shortest paths between users and their communities. We implemented this by creating various classes to help us keep track of objects and outputs in the experiments. These classes include Edge, Vertex, Graph, and Output. The first three were used to keep data stored throughout the experiment, and the Output function to make collecting data easier. The main algorithm we used was the Floyd-Warshall algorithm, the results of which were used in nearly every other function to find the results needed. Other functions that we used to find results were functions that we created specifically for the given experiment, such as the Degree Distribution, Graph Diameter, and Shortest Path functions. The data that we found was interesting. In the Directed Degree Distribution, we found that the In-Degree graphs between Unweighted and Weighted were very similar, as were the Out-Degree graphs. The graph diameter had many vertex pairs for the undirected graph, but only one for the directed graph. The Closeness Centrality Directed and Undirected graphs were very different from one another, where in directed the highest closeness centrality was at the beginning and the highest closeness centrality for undirected was in the middle. The betweenness centralities followed a similar pattern of the highest number of vertices/edges to a betweenness centrality was at 0, except in one or two graphs. The community detection experiment did not show much, due to how small of a dataset we used. Team member roles were similar to what was stated in the previous report, where each of us took on one aspect of the code, worked together to check each other's work, and write the report. The work was well divided between us three and we were happy with how we improved our code as we went along and found better ways to get our data. This project helped us to see a new way social media websites work.

Implementation:

Classes Used:

Edge: For the edge class, the member variables include the index of its source vertex in the set of vertices, the index of the target vertex, the edge's weight, the edge's value for use in unweighted graphs, and the number of times the edge is used in a shortest path. This will be used in the Graph class and used throughout the various experiments.

Vertex: The member variables of the vertex class will include a variable for the name of the vertex, the number of outgoing edges, the number of incoming edges, along with a variant of these two for weighted graphs, and the number of times the vertex is used in a shortest path. Within the class is a default constructor, made to initialize all the variables listed. This will be used in the Graph class and used throughout the various experiments.

Graph: The member variables for the graph class will include vector variables for vertices, edges, and unweighted edges. There are four 2-D arrays for an edge matrix and a version for undirected version, and two 2D arrays for the Floyd-Warshall algorithm. There are three int variables used to count the number of vertices, directed edges, and undirected edges. There is a constructor to initialize the values, and a destructor to avoid memory leaks. This will be used in all the experiments and hold the data that is given throughout the experiments.

Output: The member variables for the output class will include two ints for the given value (depending on the experiment) and a tally for a total of the value. There is a function that will create a table that lets you name the x-axis, y-axis, and what you want the values to be in the table. and generates a table that can be easily converted into a graph for report purposes. This will be used mainly for making the data for the report easier to obtain.

Functions/Algorithms:

Floyd-Warshall: The Floyd-Warshall algorithm is the cornerstone of our project. Nearly every other function in our project depends on this algorithm to do their own jobs. The Floyd-Warshall algorithm is one of the algorithms that we learned in class to find the length of the shortest path between all the vertices in a given data set. It does this with a double nested for loop to create a 2D matrix to compare the lengths between each vertice.

Degree Distribution Function: This function was created specifically to find the Degree Distribution in Unweighted Directed/Undirected graphs and Weighted Directed/Undirected graphs. In general it does this by finding the max degree distribution of what you're trying to find. First, it checks if your current degree is greater than than the current max degree. If it is, then it will clear the current list of vertices with the max degree, replace the max degree with the current degree, and place the vertex's name into the list of vertices with the max degree. If the current degree is equal to the current max degree, then it will add the vertex's name into the list

of vertices with the max degree. After checking this for every vertex, the function will output the largest degree distribution, and a list of the names of all the vertices with the highest degree.

Shortest Path Function: The Shortest Path Function will use the d and pred matrices from Floyd-Warshall to find the number of times an edge and vertex has been used in a shortest path.

Graph Diameter Function: Similar the the Degree Distribution Function, this algorithm was created just to solve this problem, and has inspiration from both Floyd-Warshall and the Degree Distribution Function. The point of this function is to find the Longest Shortest Path, or the shortest path found that takes the longest to get between two points, thereby finding the 'diameter' of the graph. It creates a vector for vertex pairs, or the pair of vertices that create the diameter. It initializes both to zero, and goes into a loop where the first pair is always 0 and the next pair loops through 0 to the last vertex, and repeats through the double for loop until every combination of vertices has been tested. If it finds a Longest Shortest path that is longer than the currently stored one, then it will clear the current list of vertices with the longest shortest path, replace the longest short path with the current length between the two vertices, and put the names of the two vertices into the list, similar to how the Degree Distribution Function operates. Also similarly, if the length is equal, the the two vertex names are added to the list. At the end of the function, it will output the Longest Shortest Path (or the Graph Diameter) and the list of all the vertices that make endpoints of the path.

Closeness Centrality Function: The Closeness Centrality Function calculates the closeness centrality value (aka ccv - the average path length from that vertex to all others) for each vertex. It then displays the max ccv and corresponding vertex (vertices) to the screen. It also displays number of occurrences of ccv's in each range of values. The Closeness Centrality Function requires the shortest path length matrix (d) generated by FloydWarshall(), and requires a vector containing all of the graph's vertices. How this function works is by creating a 2-D matrix using 2 four loops, and fills out the matrix by finding the shortest path from vertex i to the other vertices. Then it finds the Longest Shortest Path for vertex V and divides it by the number of paths. This finds the average Shortest Path length for each vertex, or the Closeness Centrality.

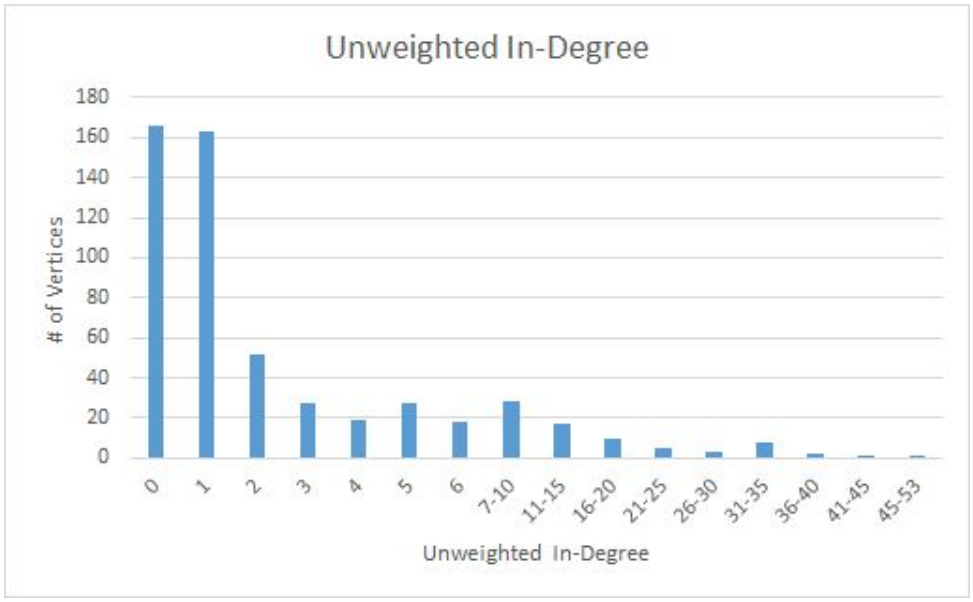
Betweenness Centrality Function: Very similar to the Degree Distribution Function, tweaked for the specifications given. When working on the directed and undirected vertex V, the code was altered to allow the difference between directed and undirected, by changing variables and using the Vertex classes DirectedBetweenness and UndirectedBetweenness variables. For the unweighted edges, we used the code from the vertex closeness centrality, but changed every instance of 'vertices' to 'edges', in variable names and using the Edge class. For weighted edge, we made alterations to unweighted to that into account the weight needed.

Community Detection Function: The Community Detection Function will use the Graph copy constructor to create two unweighted graphs to work with for this problem: one for undirected and one for directed. For each, we will sort the unweighted betweenness centrality of edge e in descending order, and repeatedly remove edges with the highest unweighted betweenness centrality value 5 times, recalculating the unweighted graph diameter each time. This will show us the communities of the vertices, or where the vertices bunch up at.

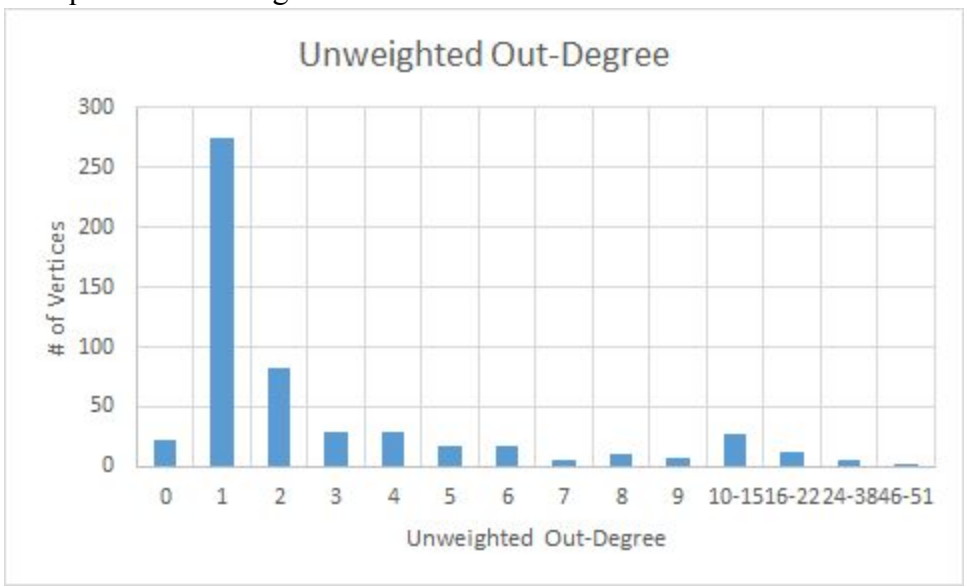
Experiments:

Directed Degree Distribution

Unweighted Degree Distribution

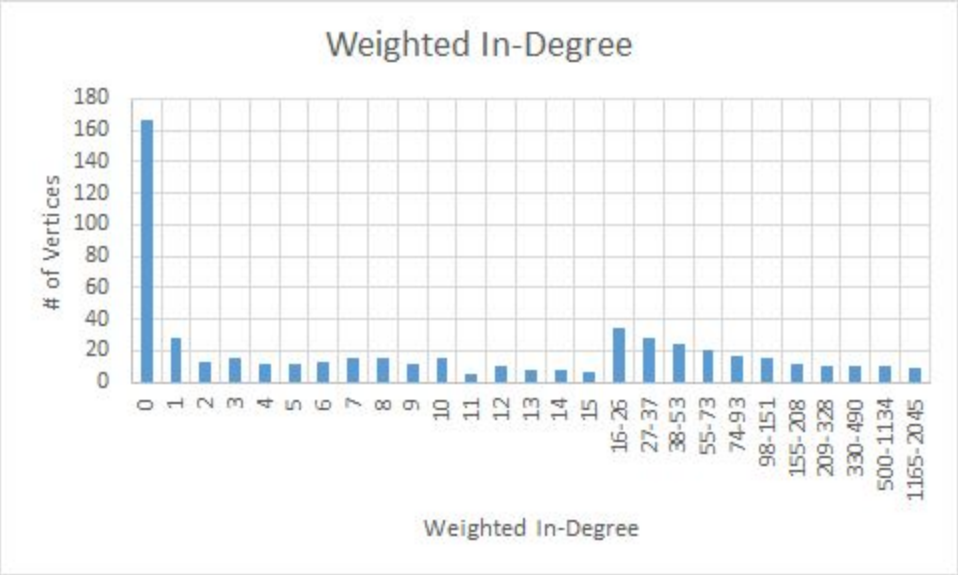


In-degree: The number of edges from a vertex to other vertices (the number of incoming edges) This data shows that starting at 0 and going further down the list, the unweighted In-Degree goes down. By the time the data reaches 7, the data needs to be grouped together to make the graph readable. By the time you reach the last degree, 53, there is only one occurrence, and it is so infrequent that the degree before it was 45.

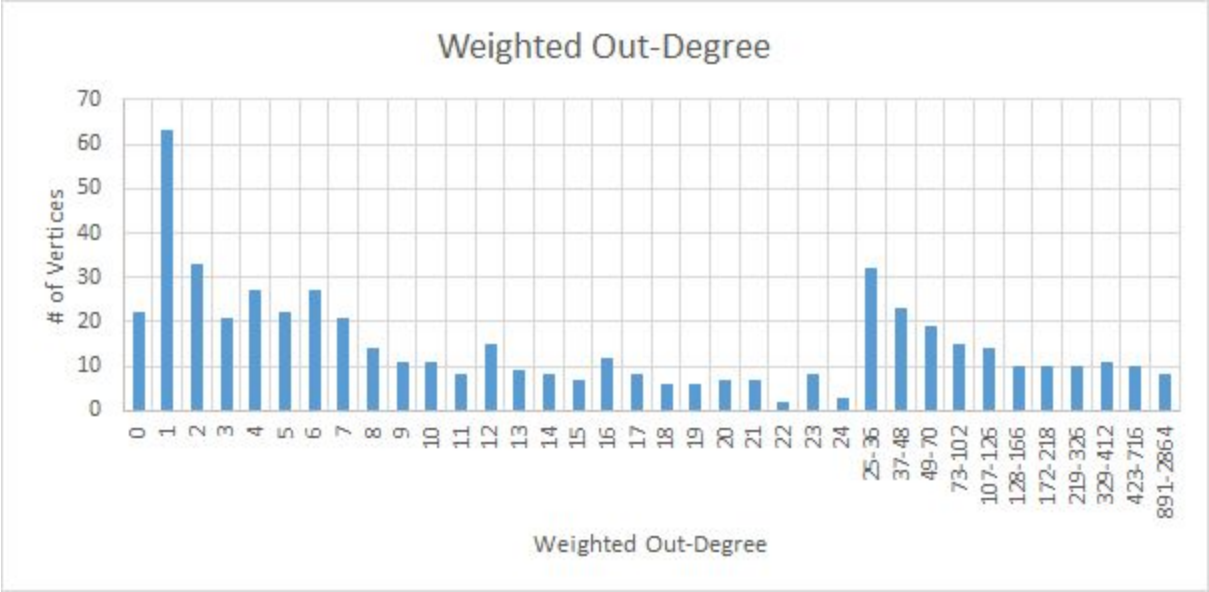


Out-degree: the number of edges (the number of out-going edges)
This out-degree graph is slightly different than the In-Degree. In this graph, the number of vertices are centered more at the 1 degree, and 0 has lost much of its own number of vertices. 1 was the highest Unweighted Out-Degree, with about 283 vertices to it.

Weighted Degree Distribution



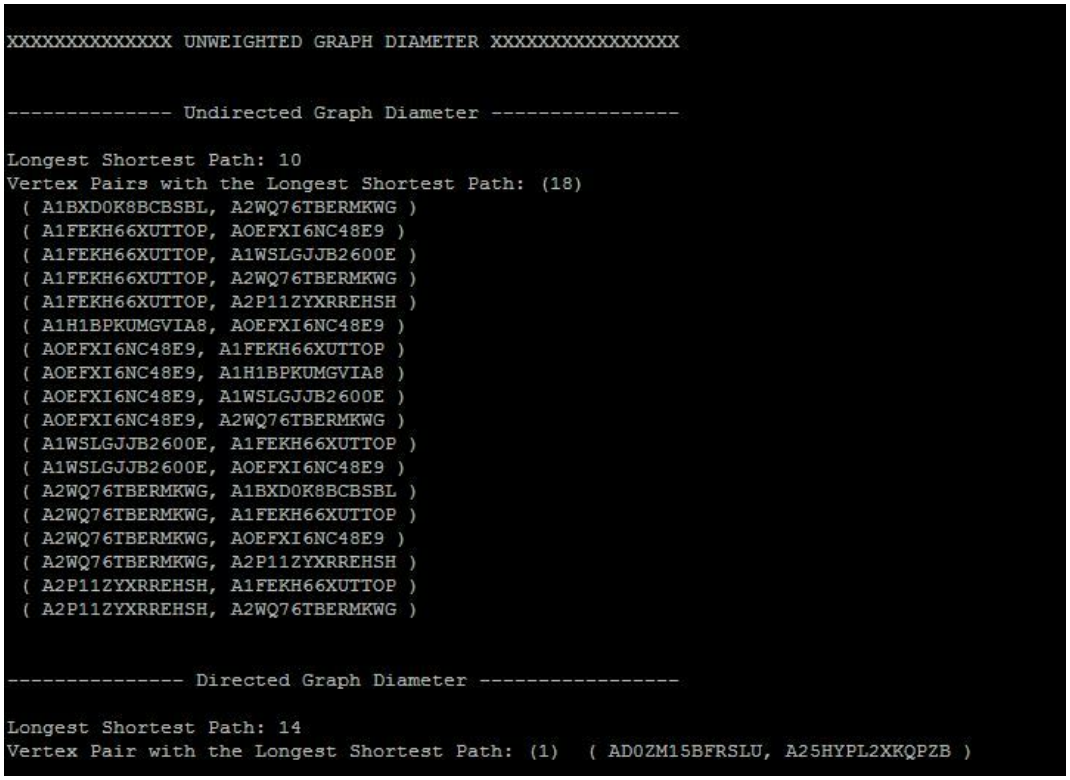
In-degree: the weighted sum of incoming edges
Compared to Unweighted In-Degree, this graph has many more outlying degrees, reaching out to the thousands, whereas in the Unweighted In-Degree it reached only to 53. However, once you passed 53, the degrees were more spread out, as each group is in a group of the next 10 degrees (the degrees in between 330 and 490, there were 10 degrees found with at least 1 vertice). As with its Unweighted counterpart, as the degrees became larger, the number of vertices became smaller, with the last 40 having only one vertex attached to it. the In-Degree 0 had the highest number of vertices, with about 168.



Out-degree: the weighted sum of out-going edges

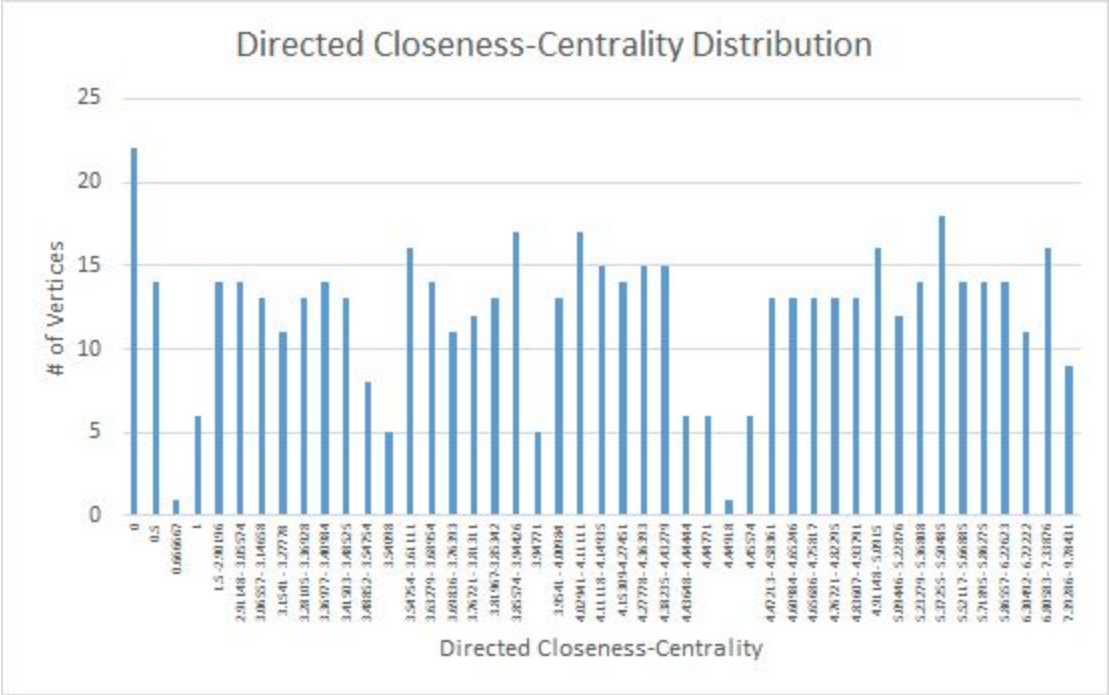
With the Weighted Out-Degree, the number of vertices are more spread out when compared to the Unweighted Out-Degree. Another difference is that the Out-Degree goes from having a smaller number of vertices, and the next Out-Degree has a larger one. This is apparent at the beginning, where the 0 Out-Degree has less than 1. This trend continues until we reach the 25 Out-Degree, where the numbers of vertices become less than 5 on any given degree. By the time we reach the 126 degree, they is only 1 vertex on each degree, except for degree 404. The Out-Degree 1 had the highest number of vertices, with about 65.

Unweighted Graph Diameter

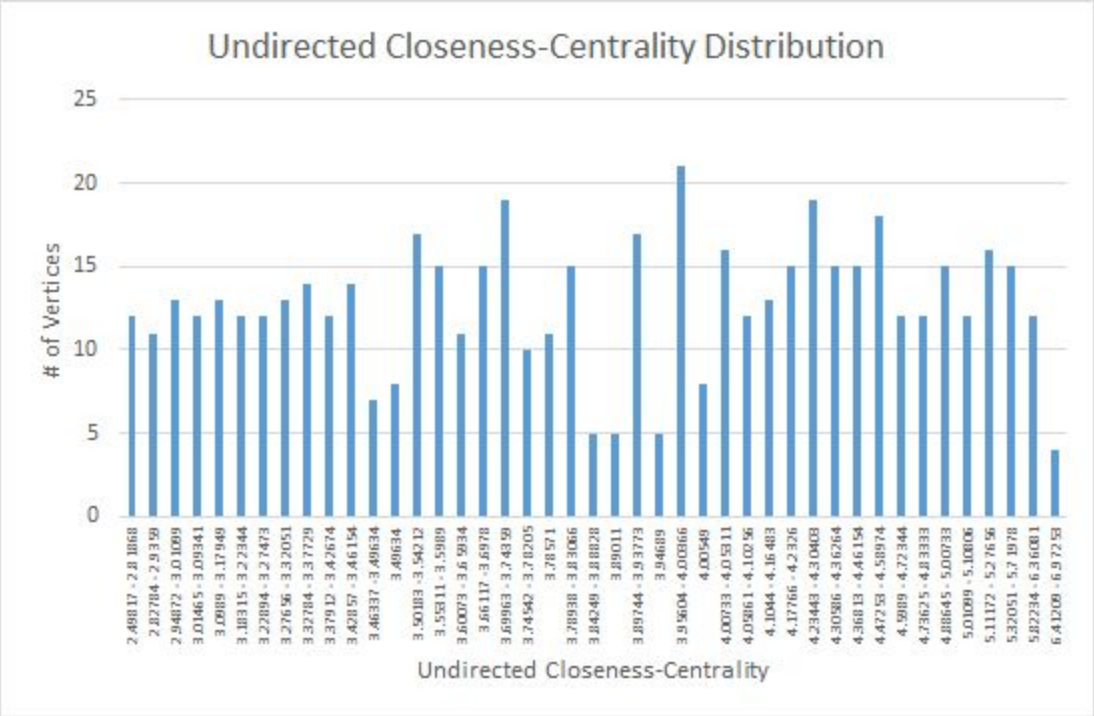


The image above is the data results for the unweighted graph diameter. The majority of the data lists the vertex pairs that have the longest shortest path for undirected graphs, which there are 18 pairs. This is very different from the directed longest shortest path, which only has one vertex pair.

Unweighted Closeness Centrality



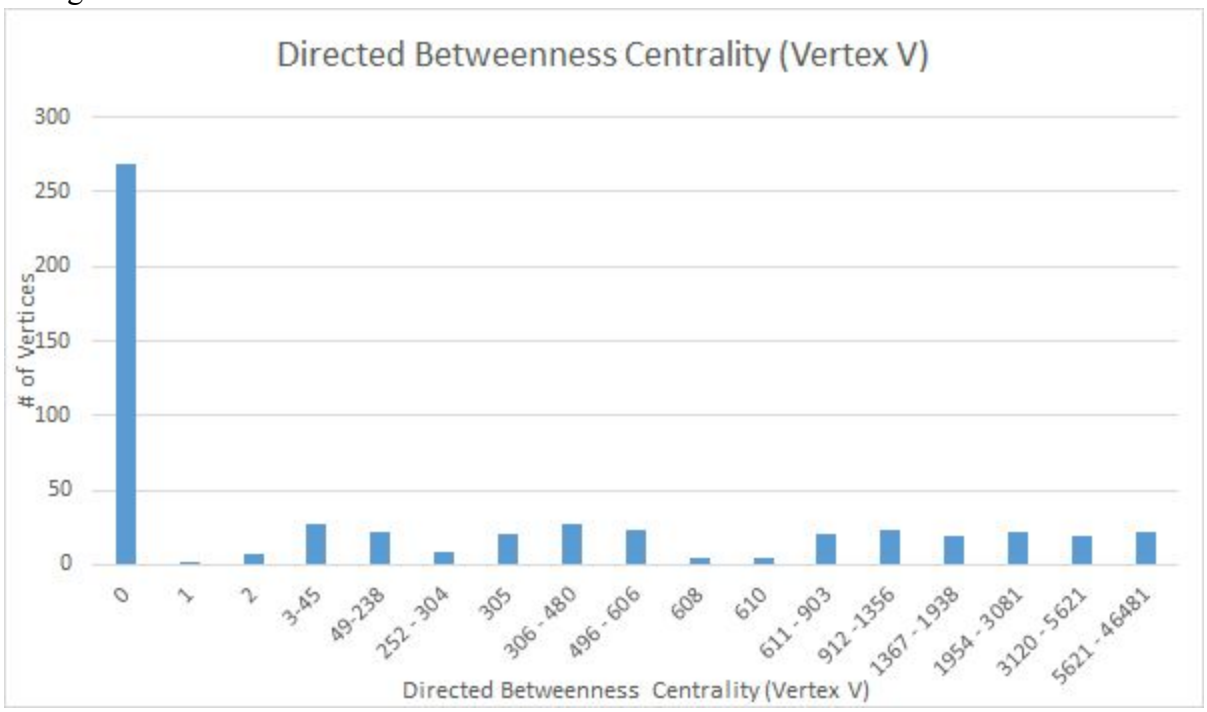
For the directed graph, there were 392 unique directed-closeness centralities, so to make the graph easier to read, the centralities are compressed into groups of 11 centralites per bar, except with significant outliers (for example, 4.45574 has 6 vertices to it where its surroundings only have 1 or 2), To be considered an outlier, the closeness centrality must have 5 or more vertices to it. Zero (0) had the largest closeness centrality, followed by .5 and 1. Generally, the highest closeness centralities occurred at the beginning, as stated previously, then quickly dropped to only 1 or 2 vertices to each closeness centrality. Every so often there was a closeness centrality with 5 or 6 vertices to it, but this only occurred 6 times. Closeness centralities with vertices became rarer towards the end, as between 7.39286 and 9.78431, there were only 9 vertices attached to 9 separate closeness centralities. In comparison, there were nearly 200 vertices attached to 90 separate closeness centralities between 4 and 6.



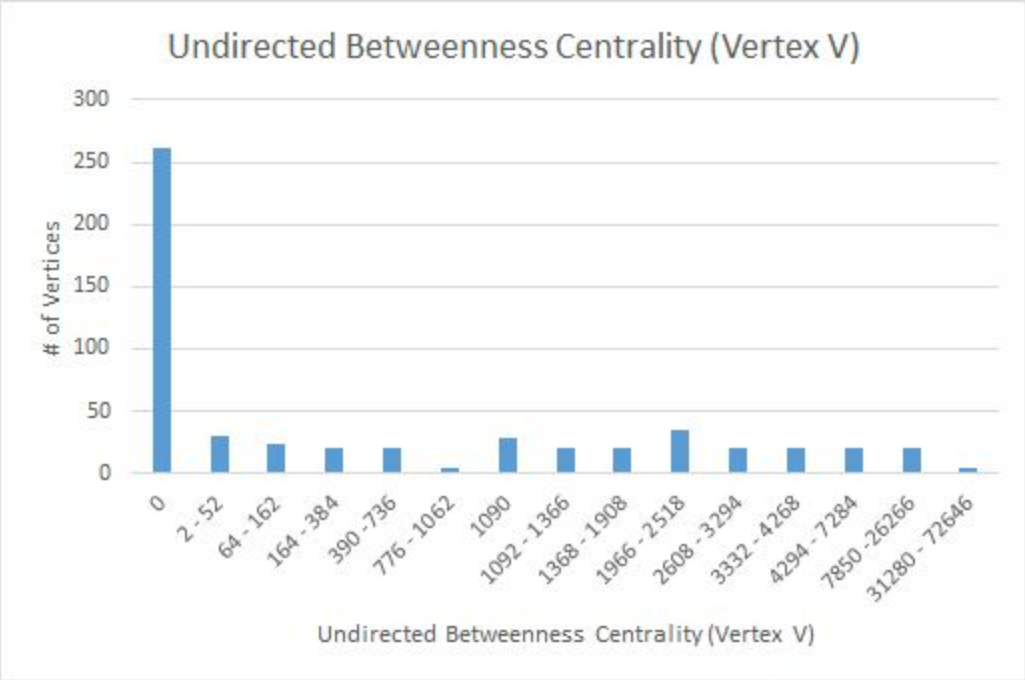
The Undirected Closeness Centrality Distribution is very different from the Directed Closeness Centrality. To start off, there were never more than 11 vertices to a single closeness centrality, as compared to 22 in directed closeness centrality. The greatest single closeness centrality, which was 3.78571, was not at the beginning of the data set, but rather closer to the center of the data set. The data also did not start at 0, but at 2,49817.

Betweenness Centralities

Unweighted betweenness centrality of vertex v means the number of shortest paths that pass through v

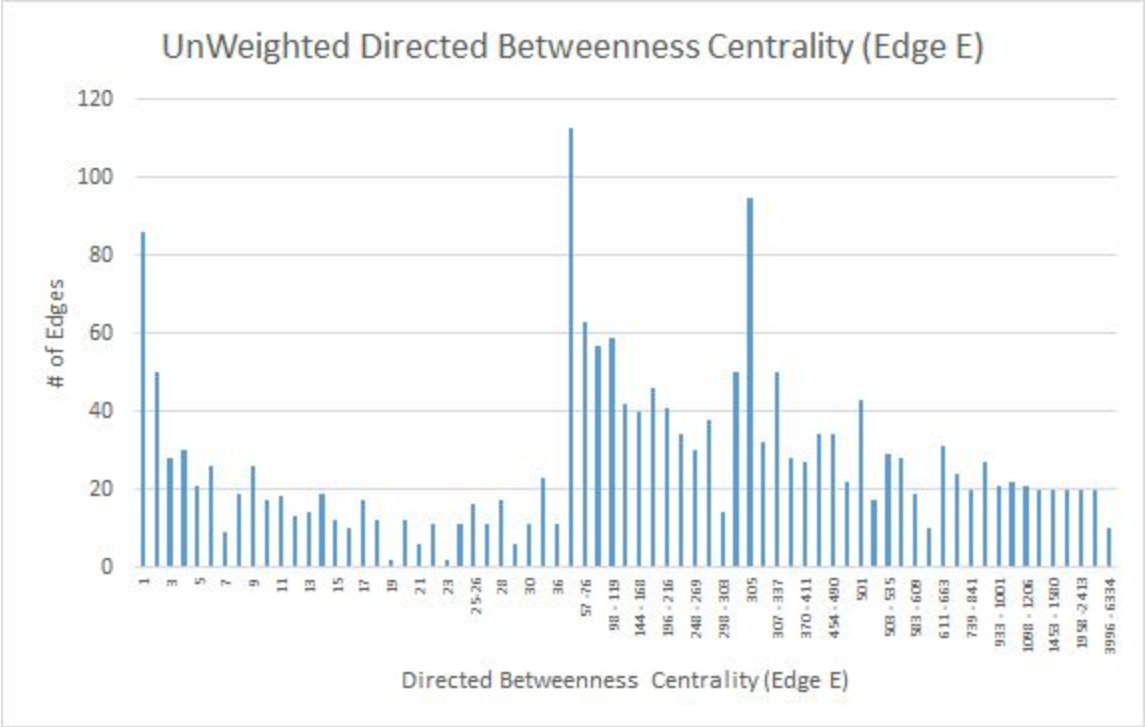


The Unweighted Directed Betweenness Centrality has the majority of the vertices at centrality 0, with 269 vertices. After this, the number of vertices to any given betweenness centrality drops dramatically, with the next highest number of vertices at betweenness centrality 2, with 7 vertices. The majority of the betweenness centralities only have 1 or 2 vertices to each, and they quickly begin to become sporadic, where the next 20 betweenness centralities are separated by 100 numbers, and by the last 20 betweenness centralities, they were separated by 35,000 numbers.

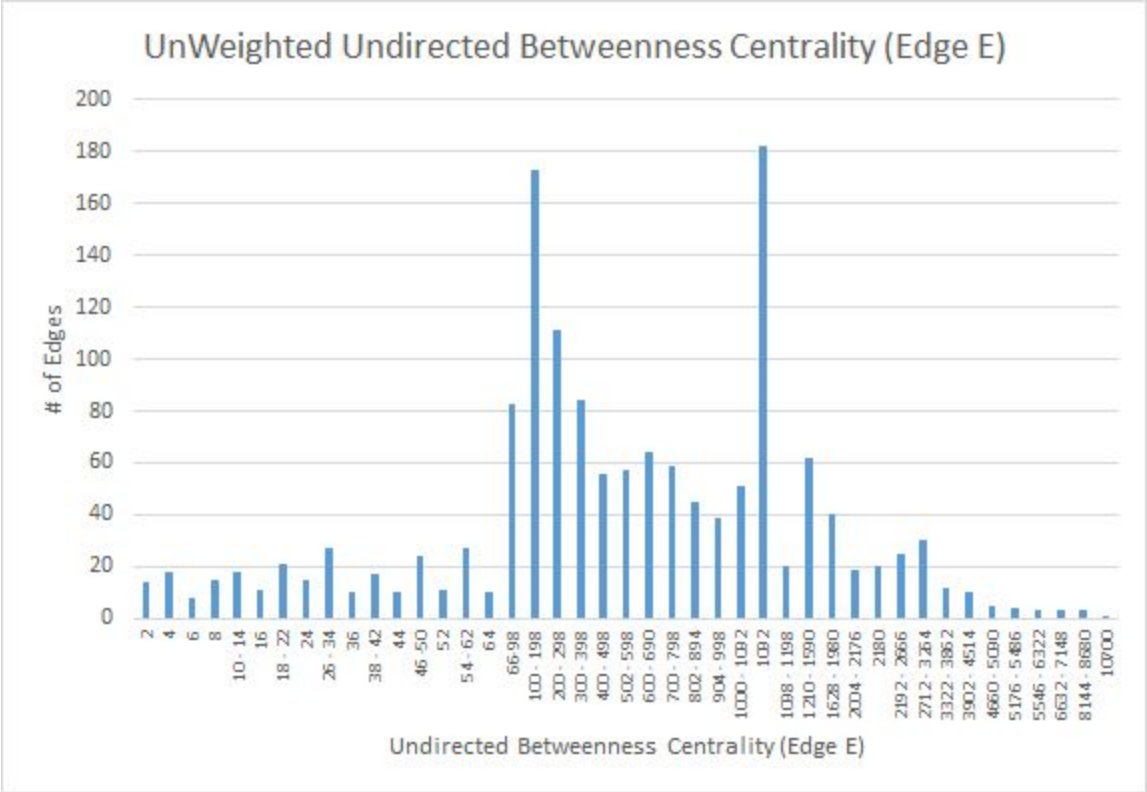


Similar to the directed betweenness centrality graph, the majority of the vertices are connected to the betweenness centrality of 0, with 261 vertices, and drops dramatically from there. The next highest is at 1090, with 29 vertices. The other bars are grouped into groups of 20 betweenness centralities, and by seeing how similar of a height each one is, the majority of the betweenness centralities had 1 or 2 vertices to them. The betweenness centralities are spread out even farther than the directed graph, where by the end the last 4 betweenness centralities are spread over 40,000 values apart.

Unweighted betweenness centrality of edge e means the number of shortest paths that pass through e

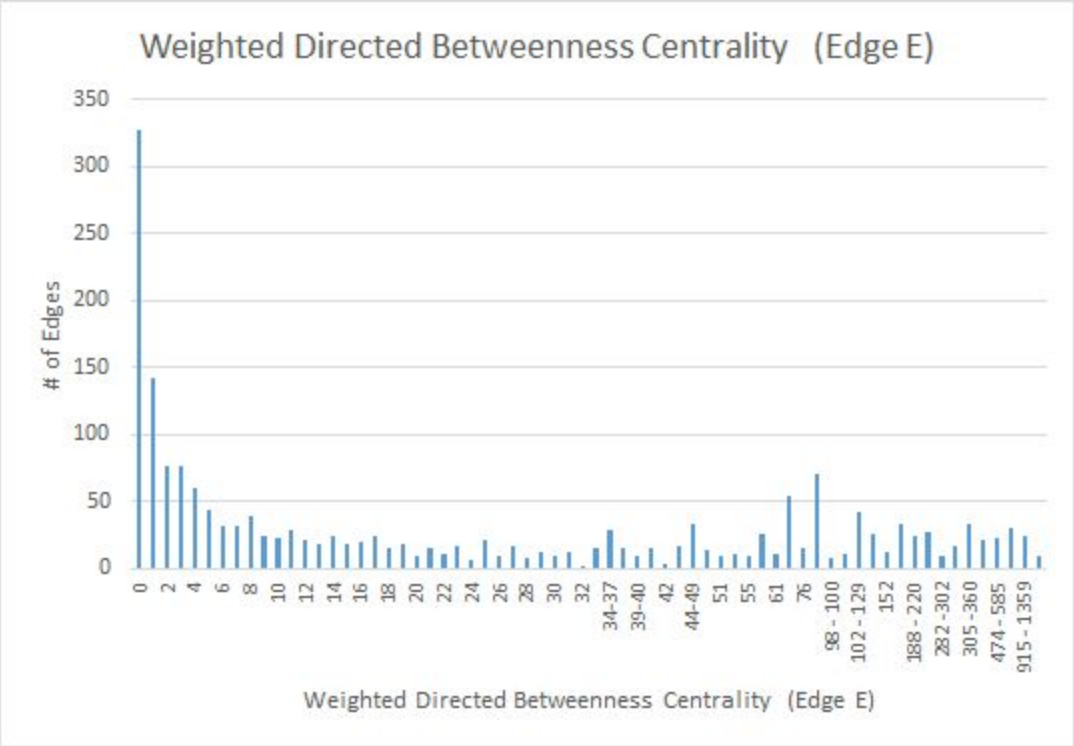


The unweighted edge betweenness centrality has a lot of higher number of vertices to each betweenness centrality, with many of the early betweenness centralities having over 20 edges to it. This is many more than with vertices, where there were only one or two betweenness centralities with more than 20 vertices in the entire graph. The highest number of edges to a betweenness centrality is 0, with 86 edges. So while there were more betweenness centralities with significant values, the individual betweenness centralities never to to as high of a value at the individual betweenness centrality of the vertex graph.

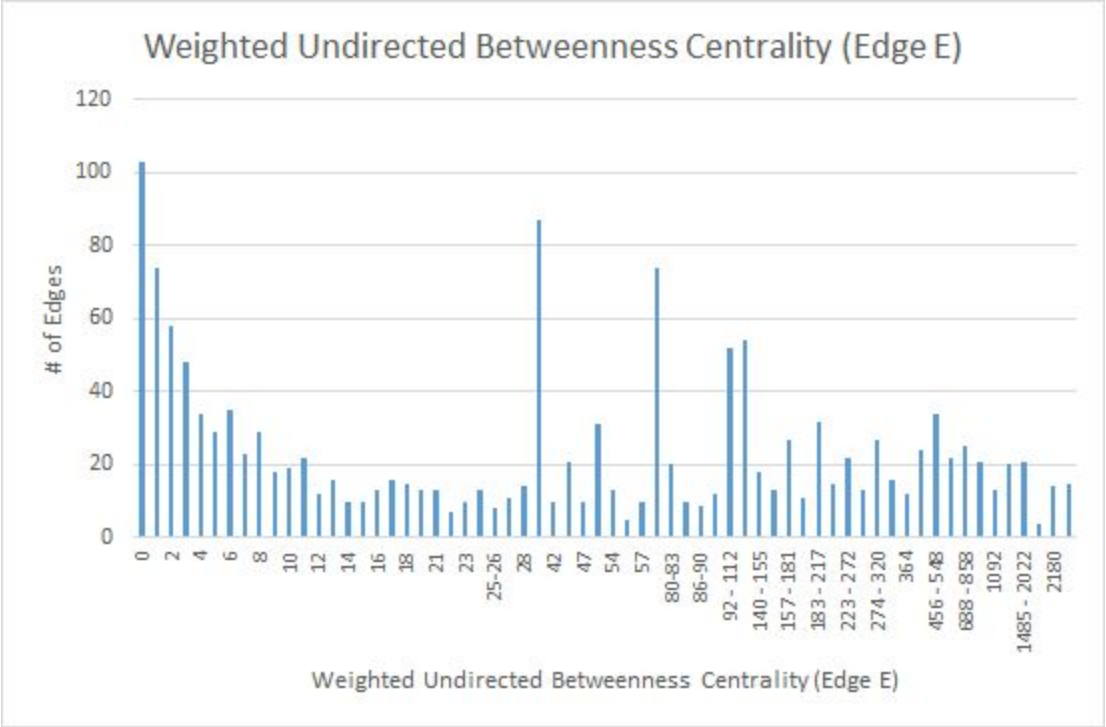


The unweighted undirected betweenness centrality for edge e is very different from its directed counterpart. To start off with, it does not start at 0, and furthermore, the highest number of edges to a betweenness centrality is not at the beginning, but it is towards to middle or end of the graph. Since there were so many betweenness centralities with edges, the bars for the graph were split between the 100 values (100 - 199 , 200 - 299, etc.) The highest betweenness centrality with the most edges is 1092, with 182 edges. The betweenness centralities become spread out towards the end of the graph, where the bars are separated by the 500s (ex. 1000-1499).

Weighted betweenness of edge e_means the unweighted betweenness centrality of edge e divided by the edge weight



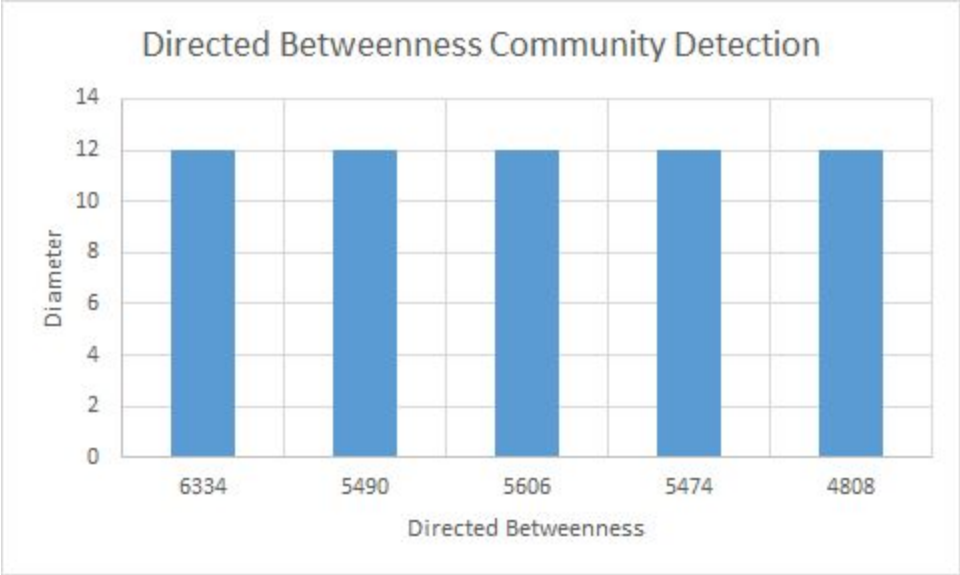
The weighted directed betweenness centrality for edge E is has a graph that is similar to the unweighted version of the graph. The majority of the edges are within the first 30 or so betweenness centralities, with the majority of those being at betweenness centrality 0, with 327 edges. There were less betweenness centralities than in the unweighted graph, and there were an exponential growth in the length between the next betweenness value at the end of the graph.

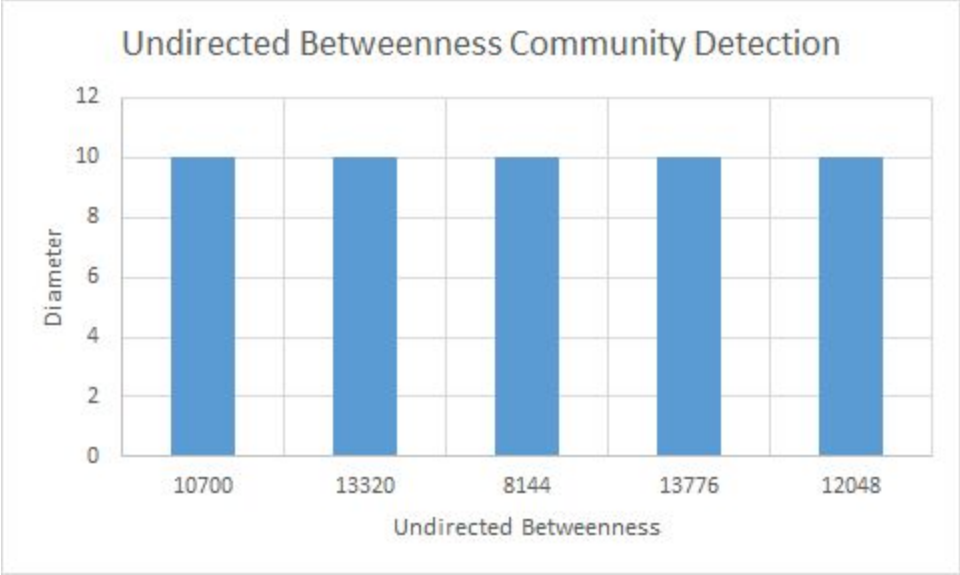


The weighted undirected graph, unlike the unweighted undirected graph, the weighted undirected graph is similar to the weighted directed graph. There are more betweenness centralities than the directed graph, but this graph still has the first 30 or so betweenness centralities with a significant number of edges to it, with the most being at betweenness centrality 0, with 103 edges. After the first 100 betweenness centralities, the values become more sporadic, but not nearly as sporadic as the previous graphs, were the last 10 betweenness centralities are 2000 values apart.

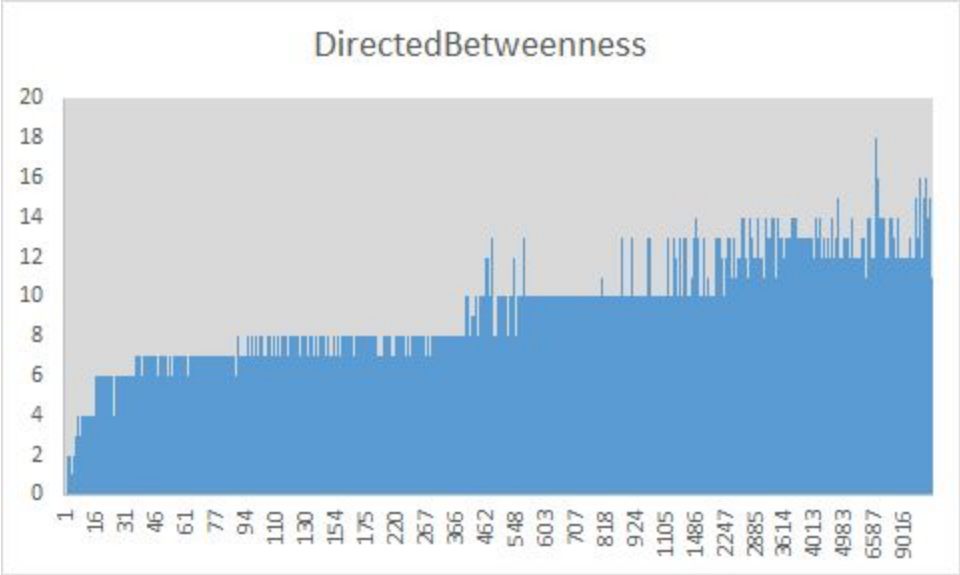
Community Detection

Using unweighted betweenness centrality of edge e for undirected and directed graphs





These two graphs are very similar to one another, and as such, a combined explanation should suffice. Both graphs have 5 unique communities, each with the same diameter length. The diameter for all the directed betweenness communities is 12, where in undirected it is 10. The diameters did not change much because of how small of an example dataset that was specified. But we did do a larger dataset experiment, where we began to see changes (we did an set where the highest unweighted betweenness centrality was removed 1000 times instead of 5). In that dataset, a graph of which are below, show that as the undirected betweenness degree was increased, on average, the more the diameters increased.



Team Member Roles: Brady will create the various functions needed to run various experiments and create the classes for the vertices and edges. Ryon will create the additional functions required for the other experiments and handle the program's input of the data. Alex will create the basis of the Floyd-Warshall algorithm, which will be used throughout the project, as well as the ShortestPath algorithm. We will all check each other's work to make sure that we are all understanding what algorithms are being put into the project, and how they should be working. We will also collaborate on the reports so everyone gets a say on what is being put into them.

Conclusion: Throughout our time working on this project, we continued to find ways to optimize and improve our code and how the experiments would run. One person would create a prototype of code, and when another member would read it over, we would often find ways to improve it, and implement a better version of the code. We also found ways to improve how to output our data and get our information for our report, mostly with the implementation of the Output class. The work was well split up between all of us, and was enjoyable to work together and find solutions together. Some final considerations are that this project was fun to work on, since we got to use algorithms in a new way and gave us a new view on how social media works.