Programming Project II, Second Report

Illya Starikov, Claire Trebing, Timothy Ott

Due Date: May 1st, 2016

1 Abstract

As stated in the prior report, social networks have revolutionized the way we communicate. Seeing as social *networks* are a real-life representation of a graph, we would like to more closely examine them. Particularly, we would like to test

- Degree Distribution
- Shortest Path Distribution
- Graph Diameter
- Closeness Centrality
- Betweenness Centrality Distribution
- Community Detection

This report will showcase our results — more specifically, we would like to discuss, in detail, our implementation and experiments. Also, we will list any relevant, interesting results we obtain.

2 Implementation

Our implementation is as follows, from a higher level:

1. Get and parse graph input.
    (a) Because our data was given in the form of a `csv`, we decided to just pipe that input directly to a `vector<string>`.
    (b) We then pass said `vector<string>` to a function that parses using C++ string functions.
    (c) We pipe the parsed data to a data structure of `map<int, vector<pair<int, double>>>`
        - The `int` is the key for retrieval of the `vector`

- The `vector<pair<int, double>>` stores a `vector` of the edges, in pairs — where the pair `<int, double>` are proportional to the target vertex and weight.

2. Move on to calculating the out degree of the vertexes.
   (a) Initialize a `map` of `<int, int>` for unweighted and `<int, double>` for unweighted.
   (b) For both weighted and unweighted simply use the source as the `key`.
   (c) For unweighted, use the `size()` method of the vector class to determine the out degree[1].
   (d) For weighted, sum the `second` property of the `pair`s in the `vector` — note that the second property of `pair` is the weight. This gives a total weight.

3. Move to calculating the in degree of the vertexes.
   (a) This is done almost the same way, except there is a weight `map`.
   (b) Iterate over the entirety of our data structure[2], and store where the target vertex points to in all the edges in the weight `map`.

4. Calculate shortest path via the Floyd-Warshall algorithm.
   (a) Initialize an adjacency matrix — a `vector<vector<int>>`
      - Default all values to infinity — in our case, 999999.
   (b) Copy over data from our `map` to said adjacency matrix.
      - If unweighted and an edge exists, default to 1.
   (c) If requested an undirected shortest path, make the graph indirected.
      - This is done by making a mirror image of the adjacency matrix A, by setting $\forall i, j \in A, A_{i,j} = A_{j,i}$. Just copying over the diagonal.
   (d) Run the Floyd-Warshall algorithm, with triple C-Style `for` loops.

5. Implemented Graph Diameter
   (a) Took the map and using the Floyd Walsh algorithm, found the shortest paths between each vertex
   (b) Begin by assuming that the path from 0 to 1 is the longest path
   (c) Compare each other path to the longest path. If a longer path is found, replace the longest path with that path. If a path of equal length is found, add the path to the list of paths.

[1] Remember, the key return the a `vector` of pairs. The number of pairs are directly proportional the out degree.

[2] An adjacency map of sorts, `map<int, vector<pair<int, double>>>`.

   (d) Output the list of all paths of the maximum length.

6. Next we endeavor to find the various closeness centrality of the vertices.
   (a) Initialize one vector to hold the solutions set and four separate vectors to contain the vertices that correspond to the four vertices with the highest closeness centrality values.
   (b) Add each edge weight for each vertex together (iterating over each vertex and each edge) and pushing the result into the solution set vector.
   (c) After this has been pushed we check the value against the recorded highest four values to see if this vertex either equals or exceeds those four values and change our recorded values/vertices accordingly.
   (d) Finally, we output both our results for highest closeness centrality values and the data needed to construct the graph.

7. Calculate Betweenness Centrality
   - Betweenness Vertex
     (a) Take the map and use the Floyd Walsh algorithm to determine the shortest paths between each vertex
     (b) While constructing the paths, if a vertex is used in a path, increase the betweenness value for that vertex.
     (c) Go through the array of betweenness values and find the one with the maximum value. If two are equal betweenness, record them both.
     (d) Output the results.
   - Betweenness Edge
     (a) Take the map and use the Floyd Walsh algorithm to determine the shortest paths between each vertex
     (b) While constructing the paths, if an edge is used in a path, increase the betweenness value for that edge.
     (c) Go through the array of betweenness values and find the one with the maximum value. If two are equal betweenness, record them both.

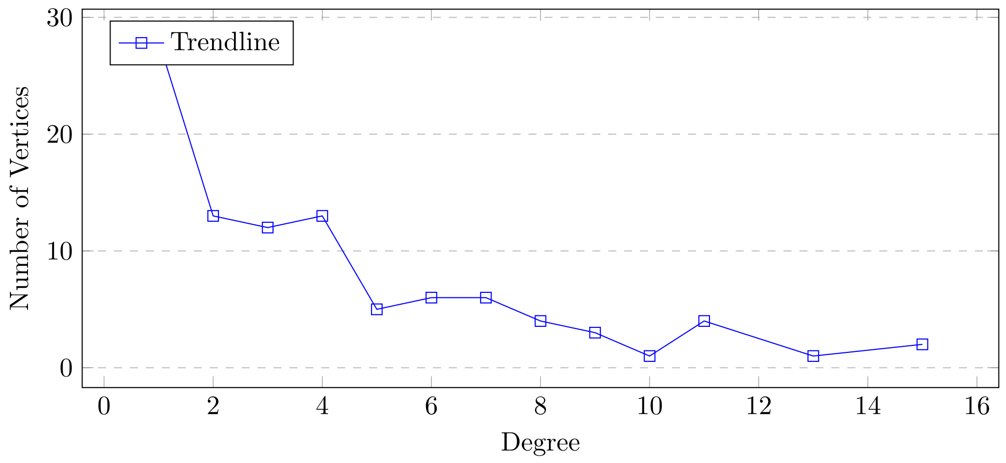
8. Lastly we must run an algorithm to detect the communities present within the graph.
   (a) Take the results from the Unweighted Edge Betweeness algorithm and take the maximum value.
   (b) Remove the edge that corresponds to this result from the original network.
   (c) Re-run the Shortest Path and Graph Diameter Algorithms on the new network.
   (d) Repeat these steps until a total of five deletions from the original network have been completed.

3 Experiments

3.1 Degree Distribution

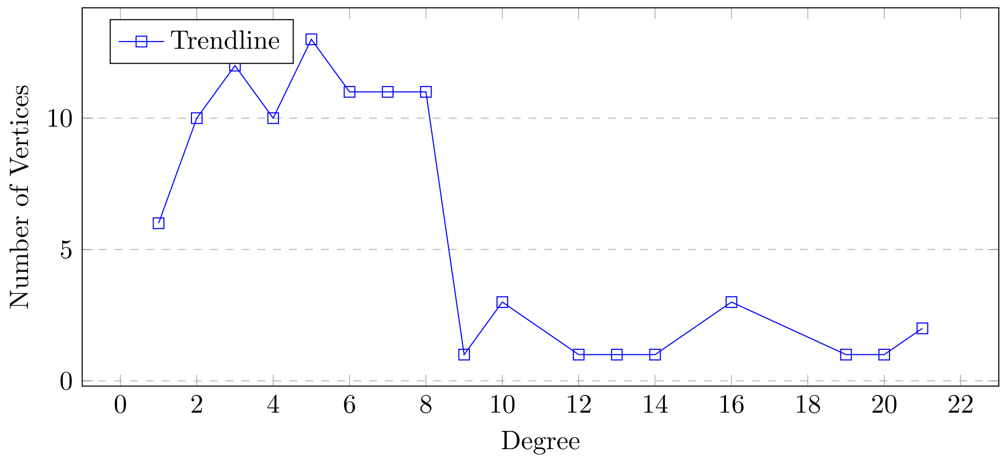
3.1.1 Unweighted In Degree

Unweighted In Degree

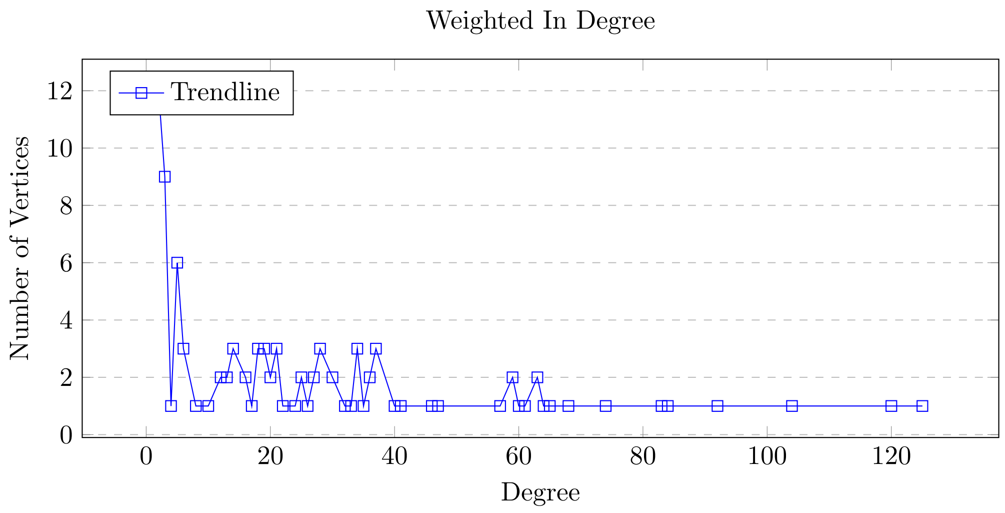


3.1.2 Unweighted Out Degree

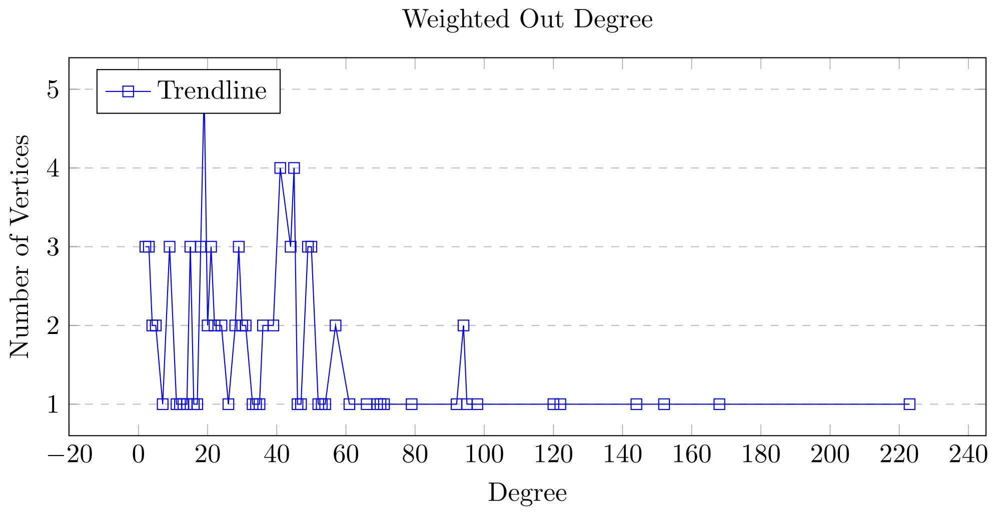
Unweighted Out Degree



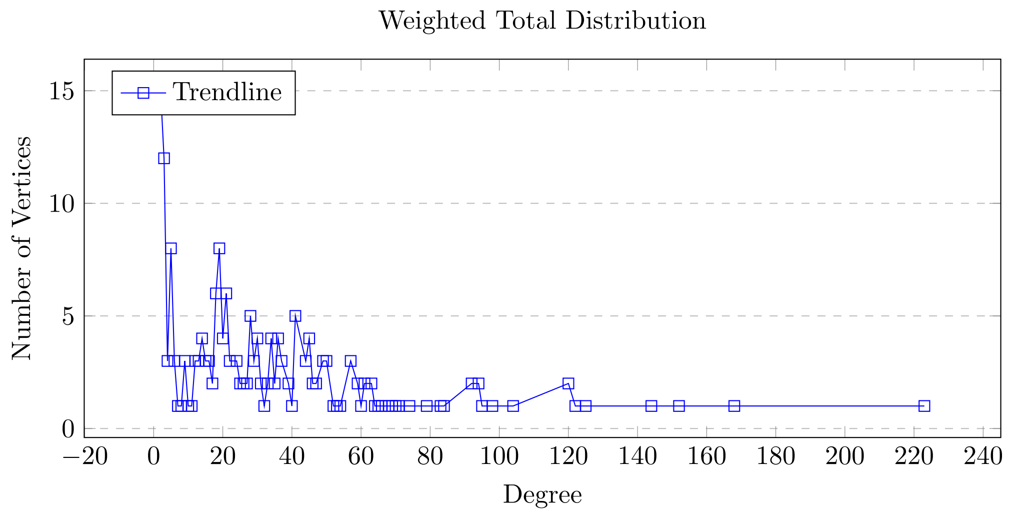
3.1.3 Weighted In Degree



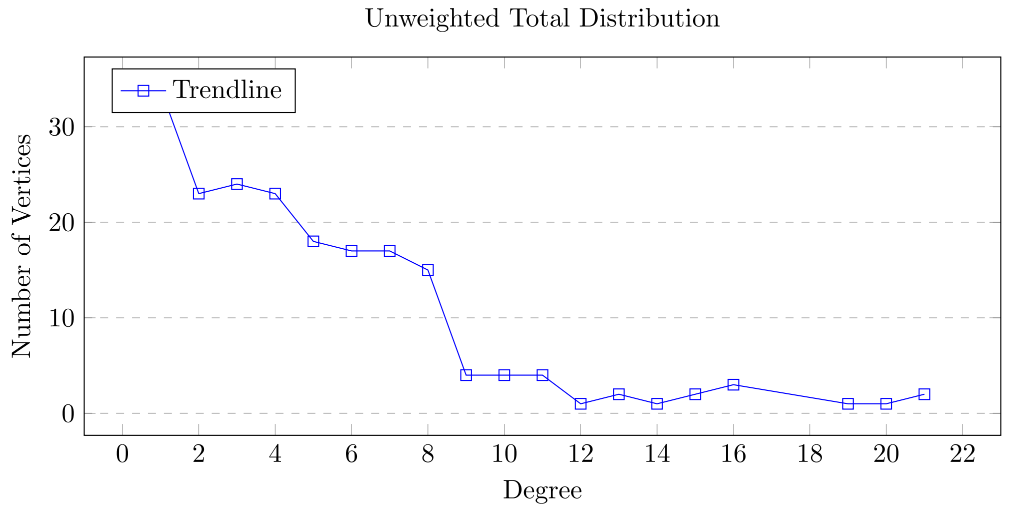
3.1.4 Weighted Out Degree



3.1.5 Weighted Total Distribution



3.1.6 Unweighted Total Distribution



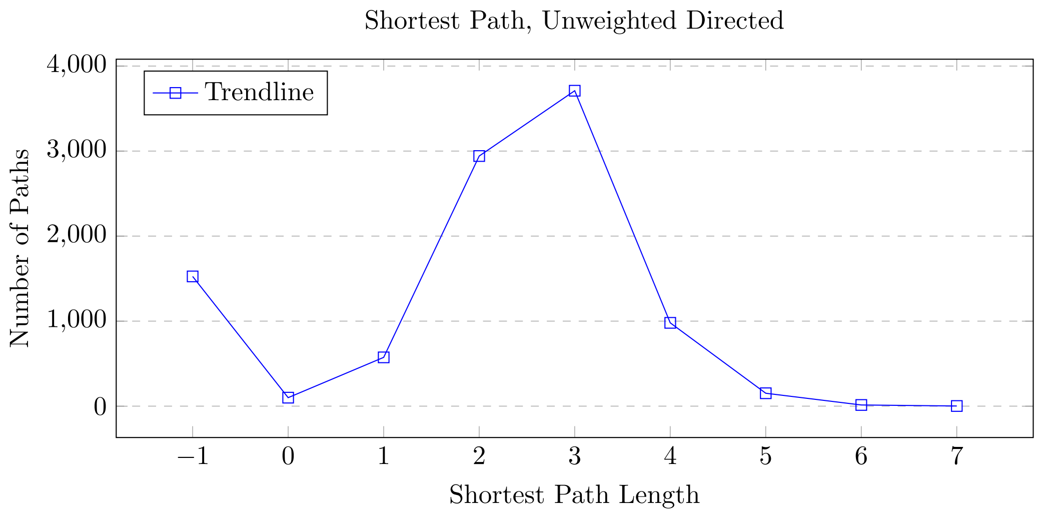
3.1.7 Highest In Degree

Out Degree, Unweighted: **21**	• 5 • 7
Out Degree, Weighted: **223**	• 3
In Degree, Unweighted: **15**	• 70 • 78
In Degree, Weighted: **125**	• 70
Total Degree, Unweighted: **22**	• 5 • 7
Total Degree, Weighted: **226**	• 3

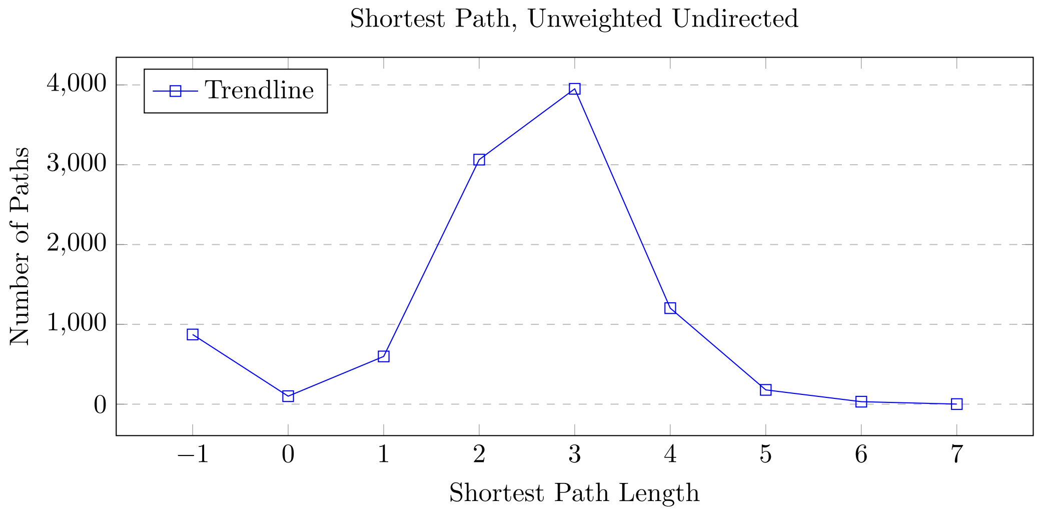
3.2 Shortest Path

Please not that -1 corresponds to a path between two vertices not existing.

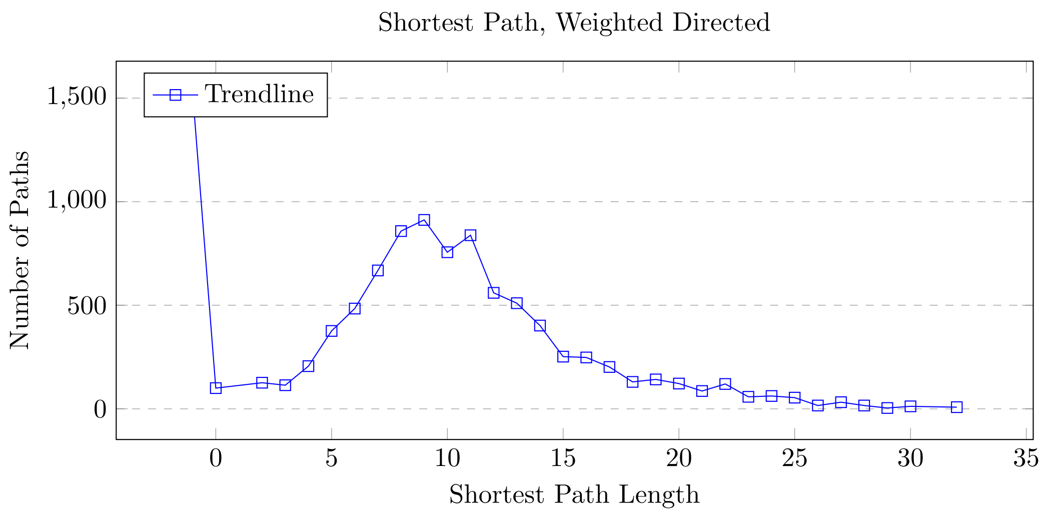
3.2.1 Shortest Path, Unweighted Directed



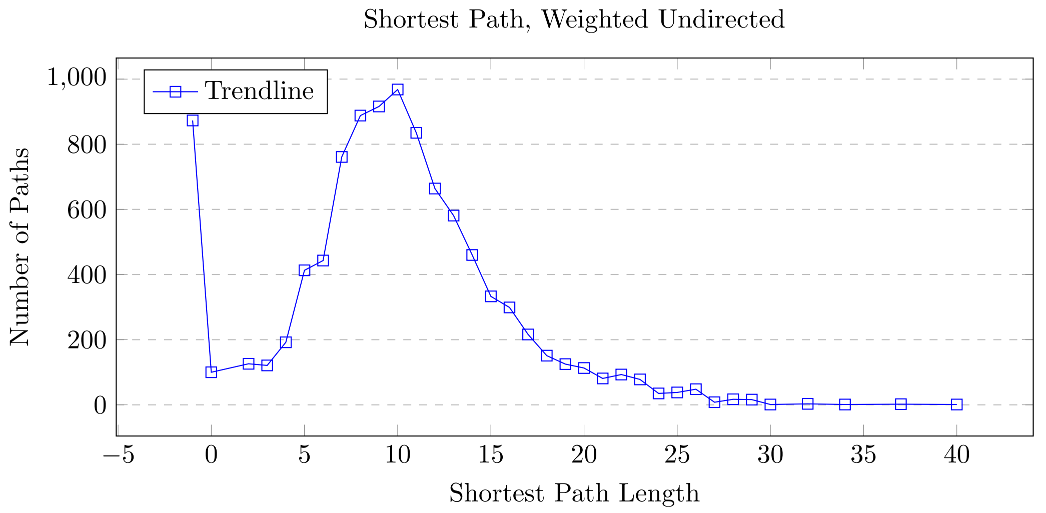
3.2.2 Shortest Path, Unweighted Undirected



3.2.3 Shortest Path, Weighted Directed

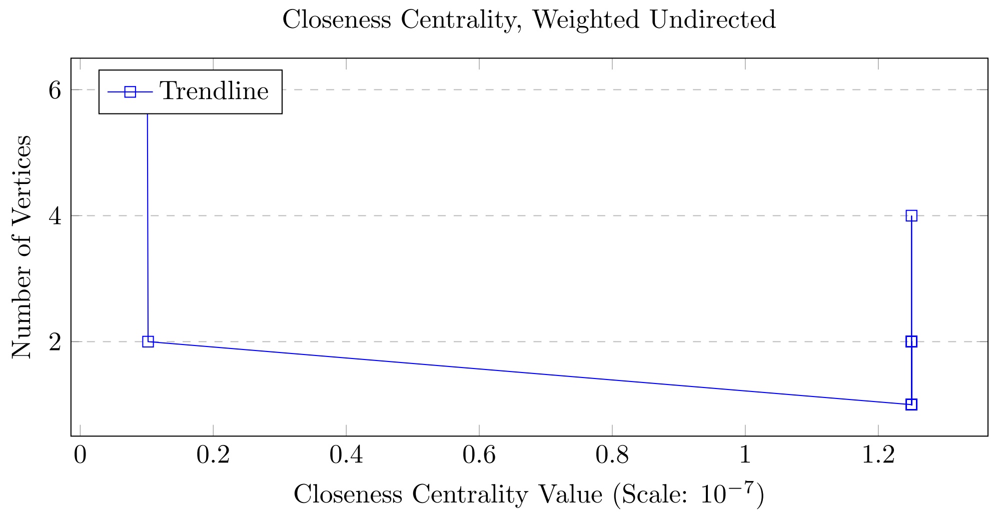


3.2.4 Shortest Path, Weighted Undirected

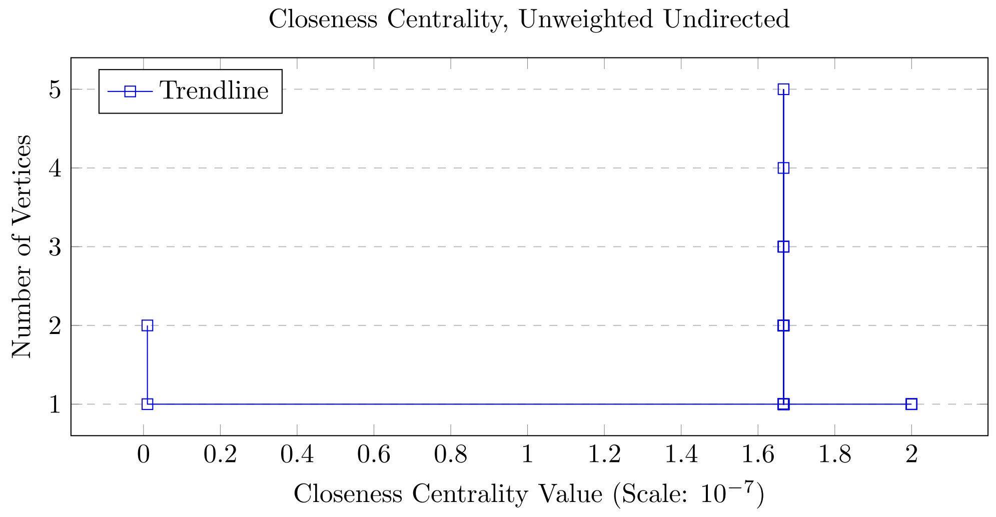


3.3 Closeness Centrality

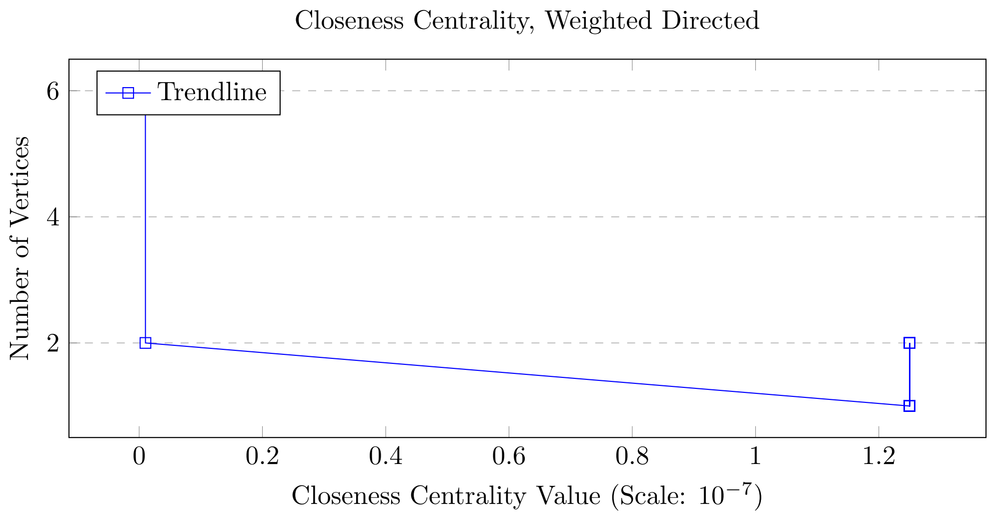
3.3.1 Closeness Centrality, Unweighted Directed



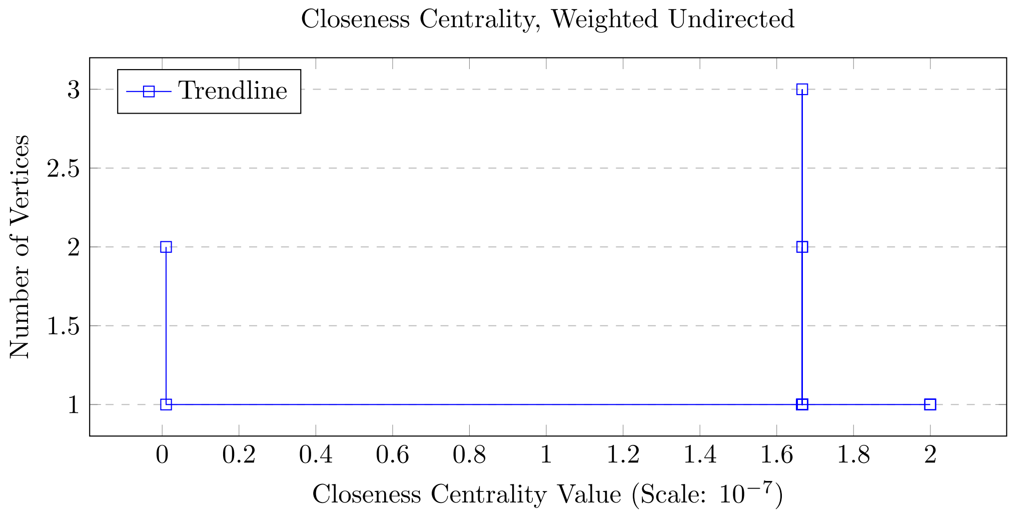
3.3.2 Closeness Centrality, Unweighted Undirected



3.3.3 Closeness Centrality, Weighted Directed



3.3.4 Closeness Centrality, Weighted Undirected

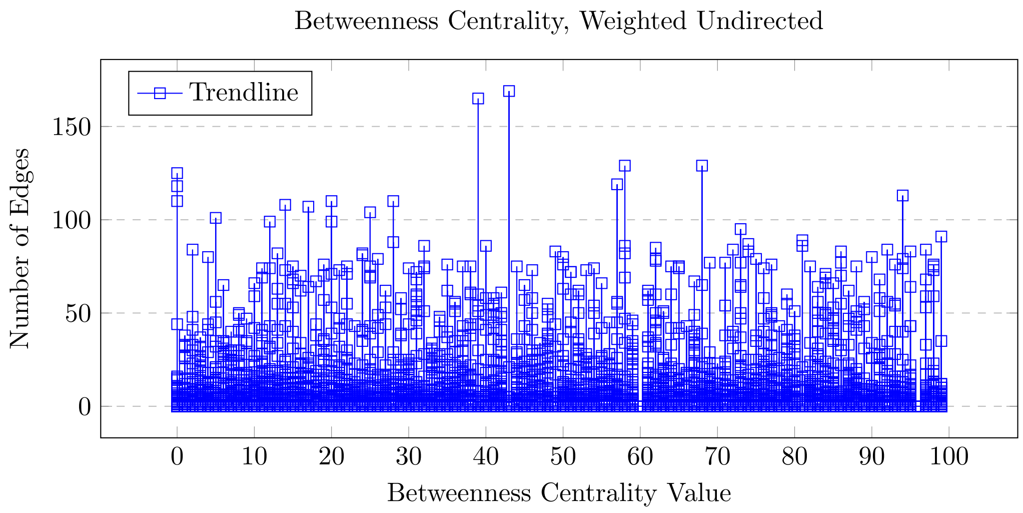


3.3.5 Highest Closeness Centrality

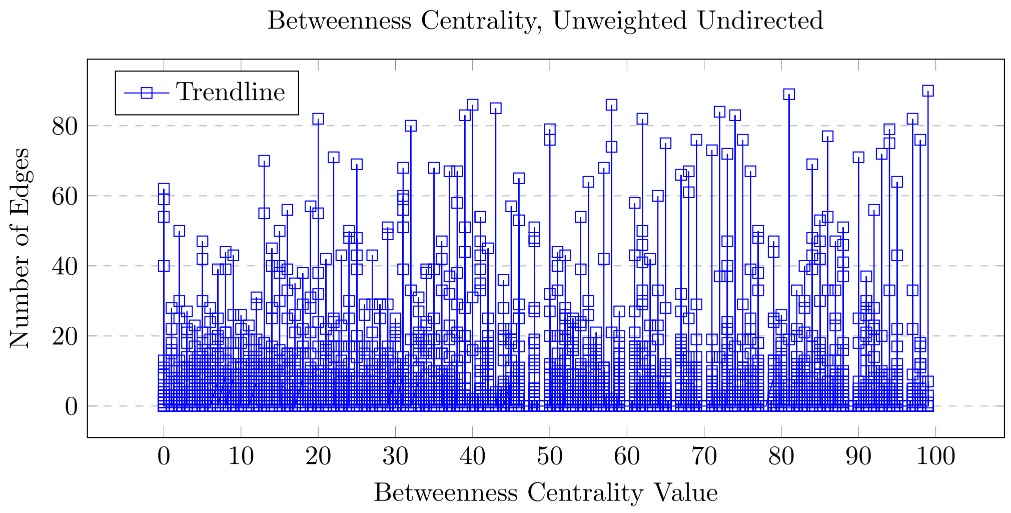
Type	**Value**	**Vertex**
Unweighted, Directed	0.000000124997e	5
	0.000000124997	7
	0.000000124997	1
	0.000000124997	3
Weighted, Directed	0.00000012498937590	5
	0.00000012498914157	64
	0.00000012498907908	52
	0.00000012498875101	11
Weighted Undirected	0.00000019996612574	47
	0.00000019996412644	0
	0.00000019996168734	70
	0.00000019995872852	89
Unweighted Undirected	0.00000019999132038	0
	0.00000019999036046	89
	0.00000019998952055	70
	0.00000019998928057	47

3.4 Betweenness Centrality

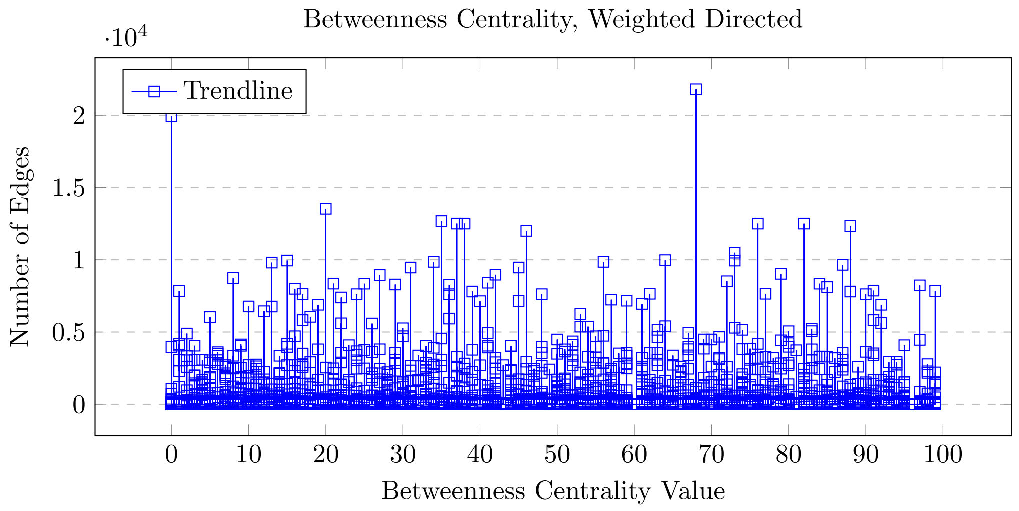
3.4.1 Betweenness Edge, Unweighted Directed



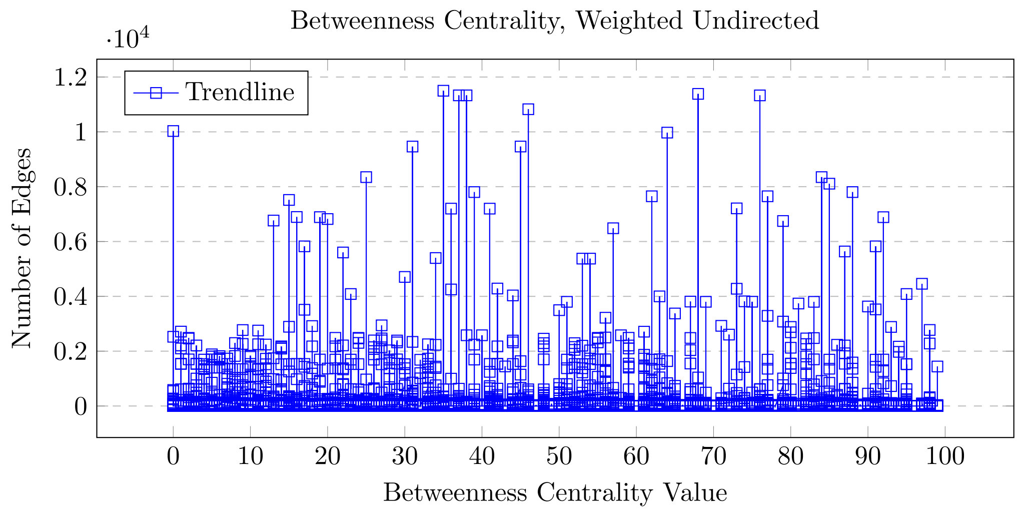
3.4.2 Betweenness Edge, Unweighted Undirected



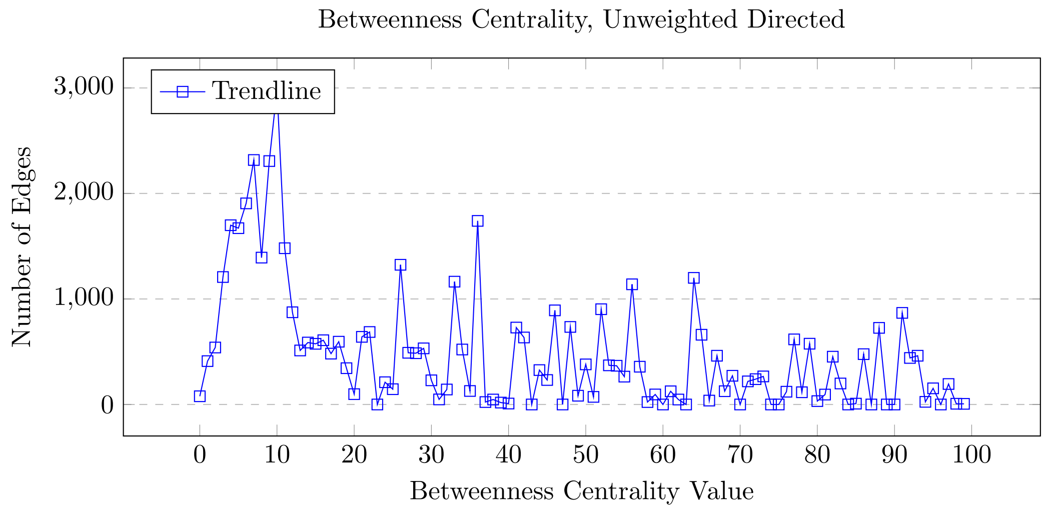
3.4.3 Betweenness Edge, Weighted Directed



3.4.4 Betweenness Edge, Weighted Undirected

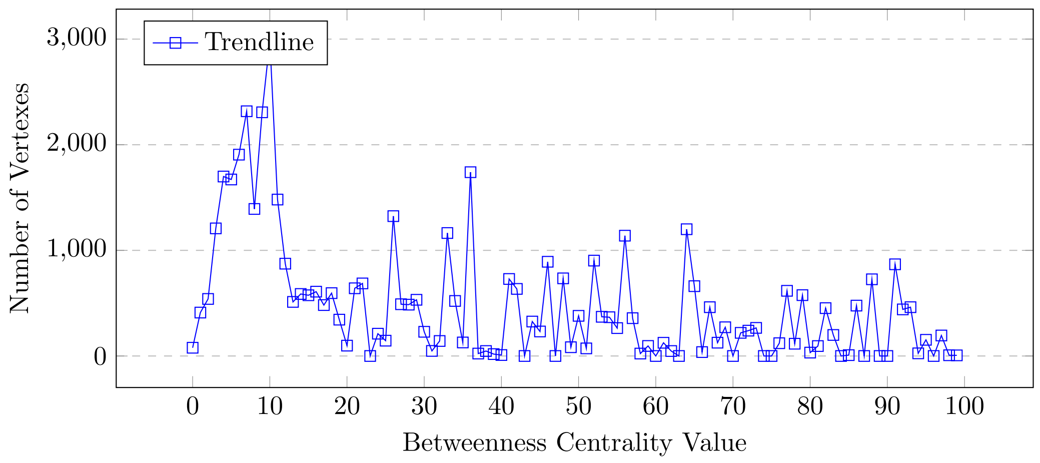


3.4.5 Betweenness Vertex, Unweighted Directed



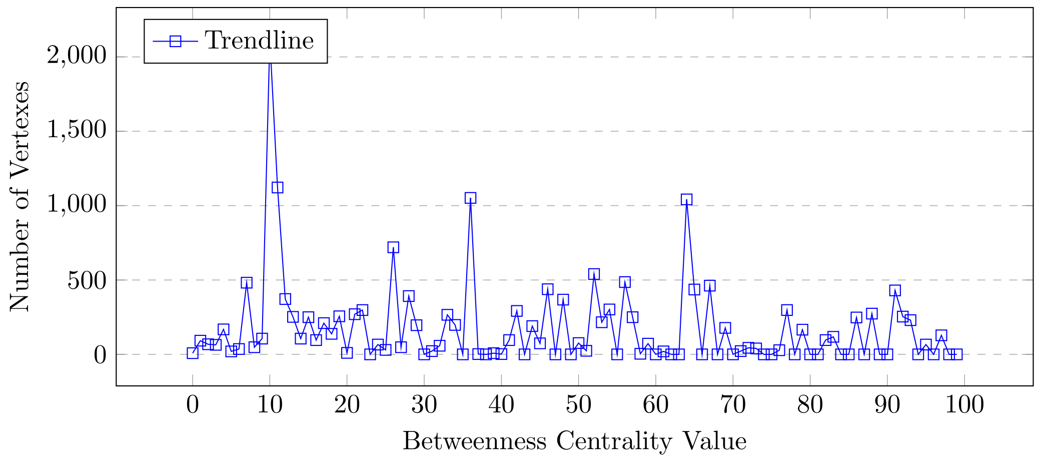
3.4.6 Betweenness Vertex, Weighted Undirected

Betweenness Centrality, Unweighted Directed

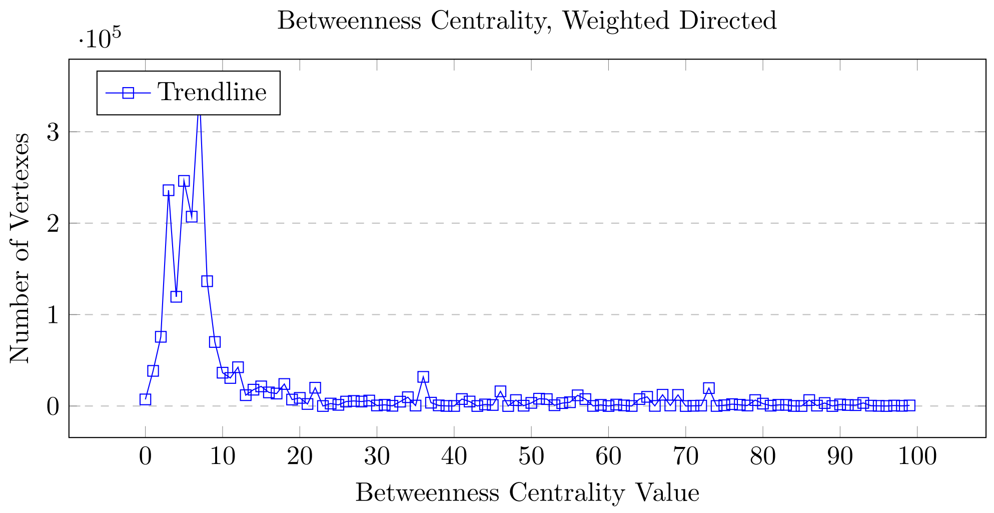


3.4.7 Betweenness Vertex, Unweighted Undirected

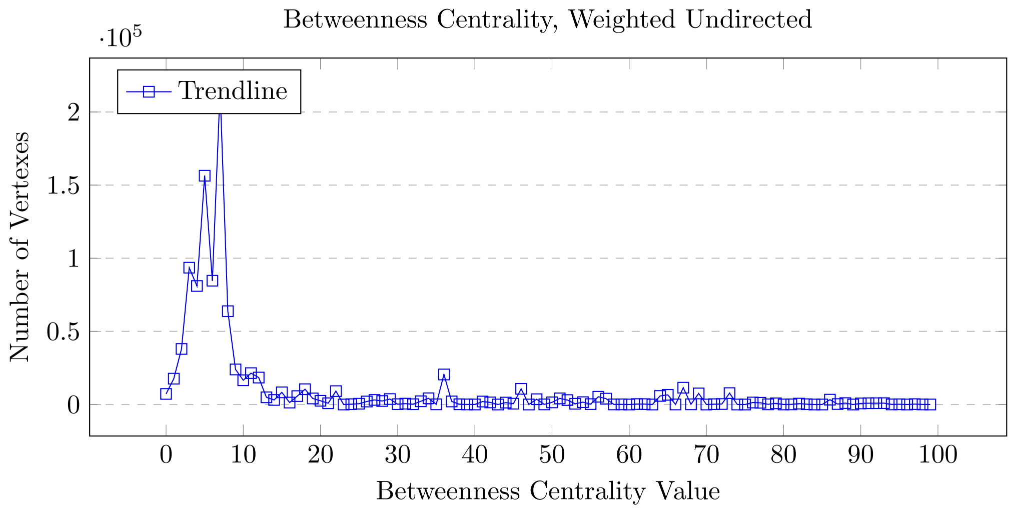
Betweenness Centrality, Unweighted Undirected



3.4.8 Betweenness Vertex, Weighted Directed



3.4.9 Betweenness Vertex, Weighted Undirected



4 Team Roles

- Illya Starikov
  - Project Manager

  - Implementation
    - Weight Distribution
    - Shortest Path
- Timothy Ott
  - Report Writeup
  - Implementation
    - Closeness Centrality
    - Community Detection
- Claire Trebing
  - Report Writeup
  - Implementation
    - Unweighted/Weighted Graph Diameter
    - Betweenness Centrality Distribution

5 Interesting Results

Below are interesting results we have found in regards to the project so far.

- Our pick of data structure (essential a Binary Search Tree) made the project easily testable. Because of Binary Search Tree's $\mathcal{O}(\lg n)$ retrieval complexity, this made our project run under 100 macro-seconds every compilation.
- For degree distribution, there seemed to be roughly 10 vertexes that were very dense. Although the graph seems fairly homogeneous many vertexes had a total degree of 15 or more.
  - When observing total weighted distribution (which appears almost to be $f(x) = \frac{1}{x}$), there is a tail caused by three nodes.
- There are always more paths between an arbitrary two vertexes than there are not. Meaning from any two nodes, it is more likely that there is a path.
  - This furthers our claim that this graph is dense.
- Closeness Centraility is so miniscule it cannot be computed by a standard `double` — the accuracy was not good enough. `long double` was used instead.

6 Conclusions

The experiments detailed above seek to find meaning in what would ordinarily be only raw data, the results of which, while being open to some interpretation, describe the small subsection of a social community that is given by the data set we were provided. For instance, the distribution of degrees is a measure of how interconnected individual nodes or profiles are to the rest of the graph. Because this graph is directed, it closely resembles a social network such as Tumblr where individuals can follow another profile but that profile is not required to follow back. From our results, particularly those of the unweighted distribution, we can tell that most of our nodes are followed by under 5 profiles and in turn follow between 2 and 8 profiles. The algorithms for Shortest Path are also a test of interconnectedness, though this time measuring the degrees of separation between nodes. From our results we can see that on average the nodes of our network are no more than two or three connections away from each other. The Graph Diameter measurements are meant to give an idea of the outliers in this community or to define the size of the outward edges of this network. Centrality metrics seek to identify the most important nodes within the network, using a variety of qualifications to define what is most important. A measure of Closeness centrality is effectively the inverse of a nodes farness or distance from other nodes. The next centrality metric that we tested for was the Betweenness Centrality which attempts to determine those nodes that most often act as a bridge between two other nodes. This measurement has applications within social networks to determine which individual (or in our social network examples, profile) has the most influence on communication between other individuals. Lastly we attempted to locate and determine the smaller communities within our social network. One way to accomplish this is to locate those edges that have the highest betweenness centrality and eliminate them from the network. These edges are most likely to be used as bridges between communities so by eliminating them we are able to separate and identify those communities. Once we have eliminated those edges we simply recalculate the shortest paths and diameters of the network to see how the network has changed. As we can see, these experiments and algorithms are of great use to us to sift through a large quantity of raw data and attribute meaning to them where there originally was little to none. After all, after applying these algorithms to a simple adjacency list we are now able to infer a large amount of information about the underlying situation that is being represented by this graph.