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. **Graph Diamater**

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.

---

[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>>>`.

(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.
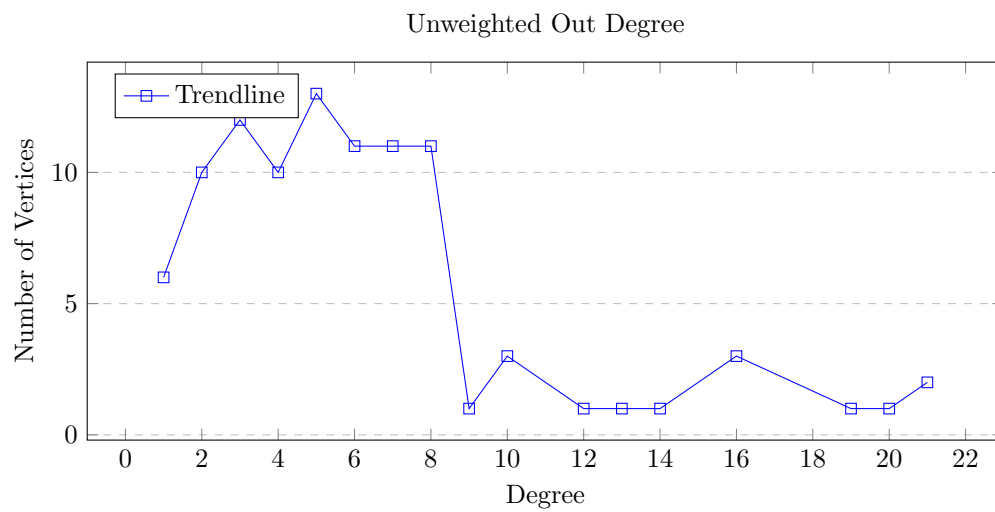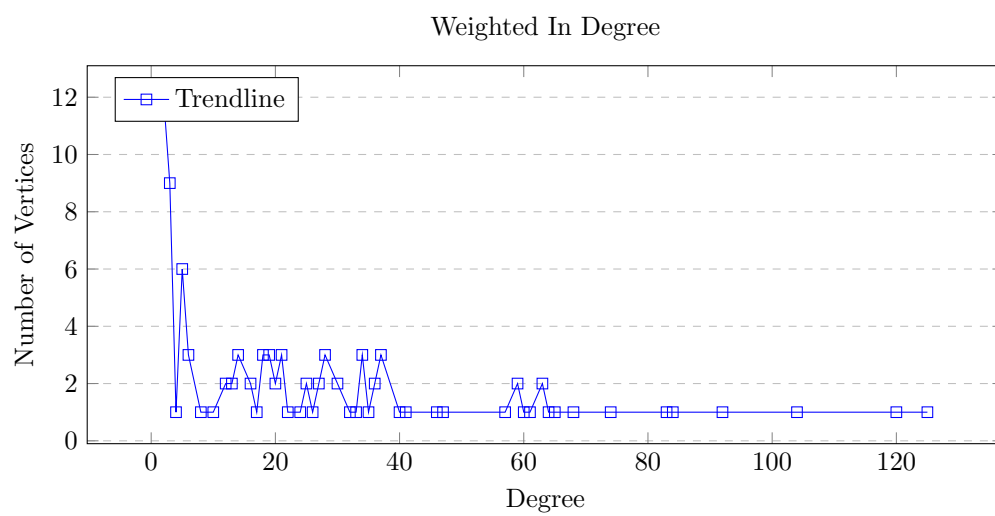
3 Experiments

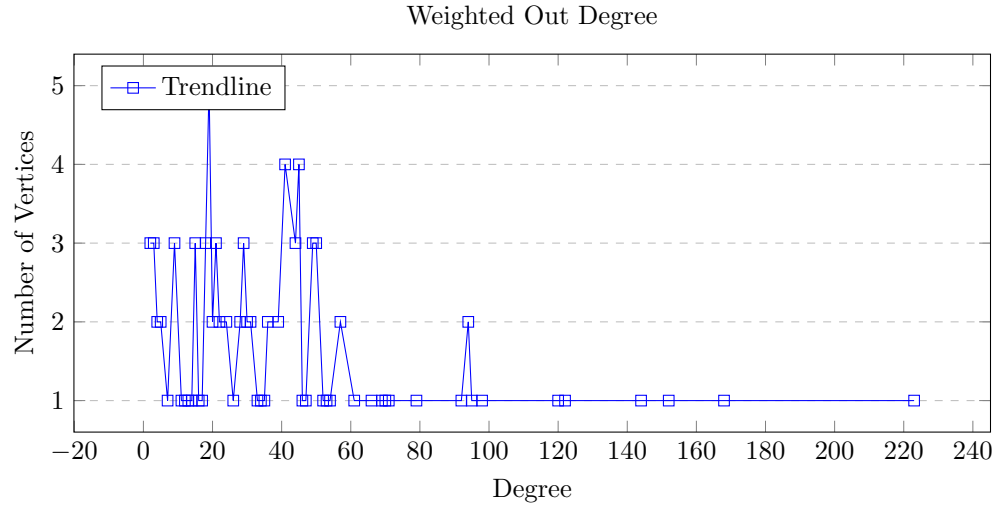## 3.1 Degree Distribution

3.1.1 Unweighted In Degree

Unweighted In Degree
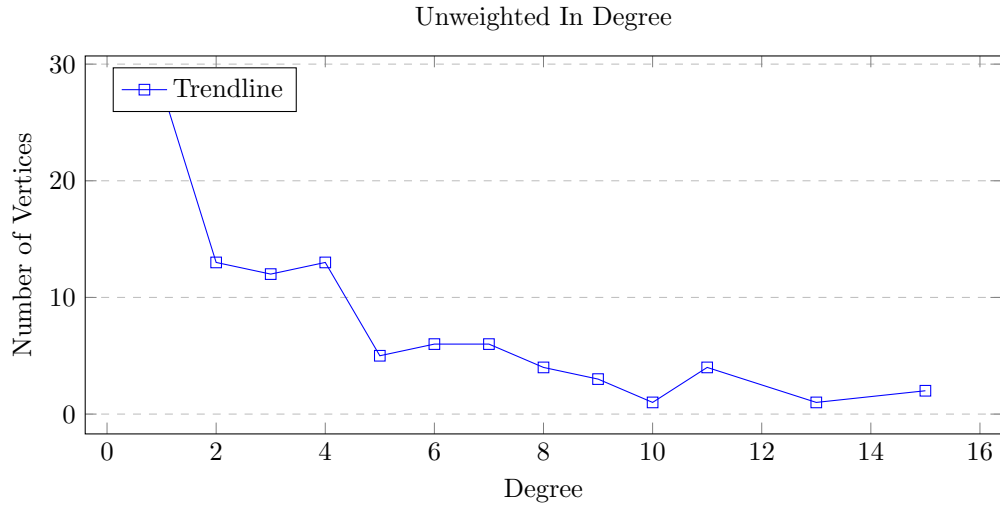
3.1.2 Unweighted Out Degree

Unweighted Out Degree

3.1.3 Weighted In Degree

Weighted In Degree

3.1.4 Weighted Out Degree

Weighted Out Degree
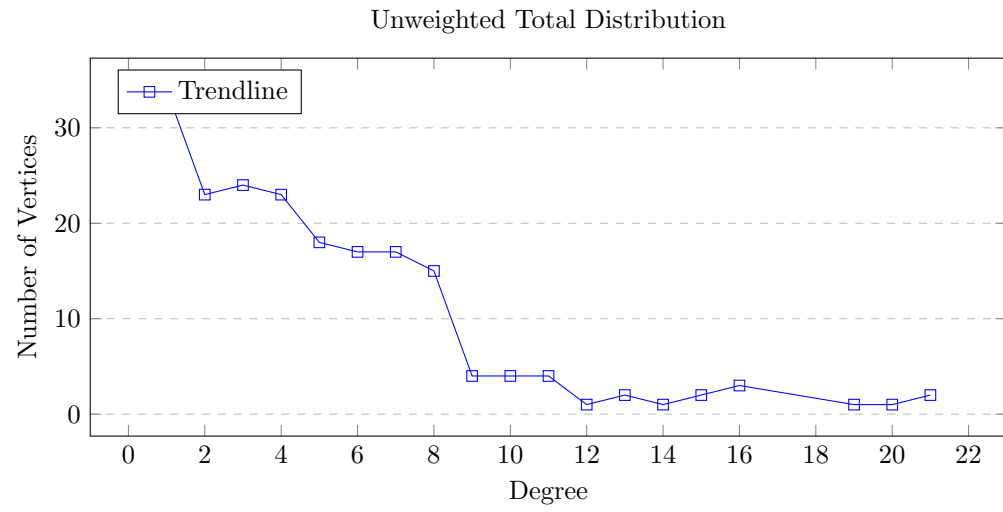

3.1.5 Weighted Total Distribution

Weighted Total Distribution

5

3.1.6 Unweighted Total Distribution
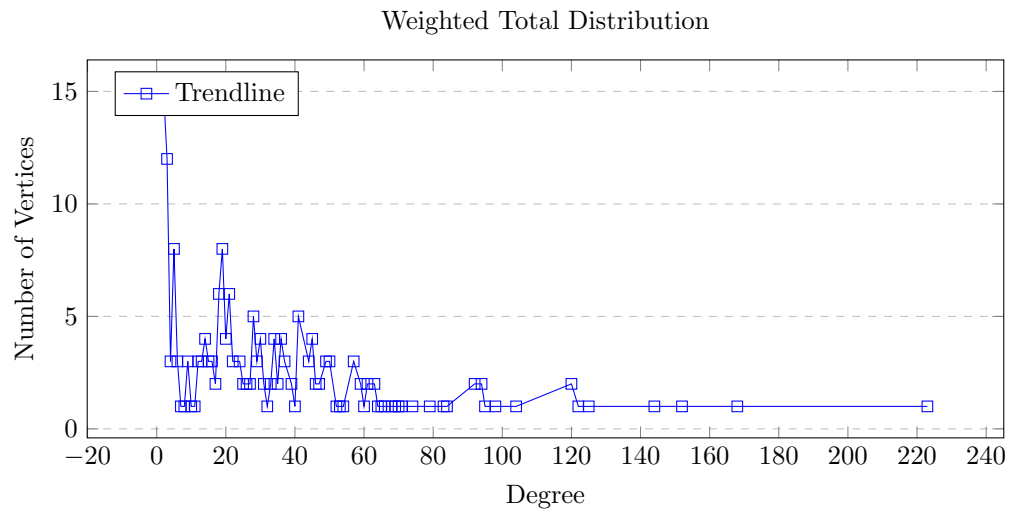
Unweighted Total Distribution

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**
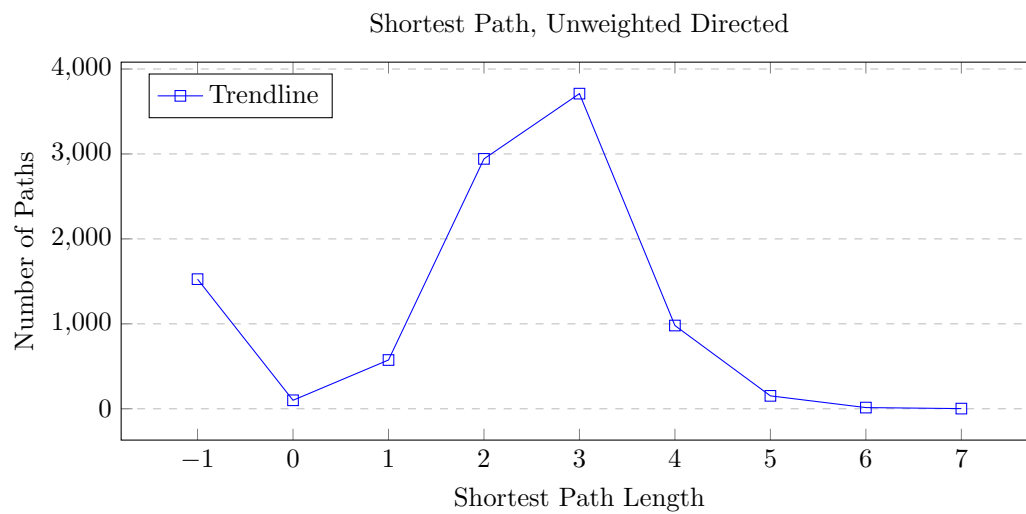
- 70

TOTAL DEGREE, UNWEIGHTED: **22**
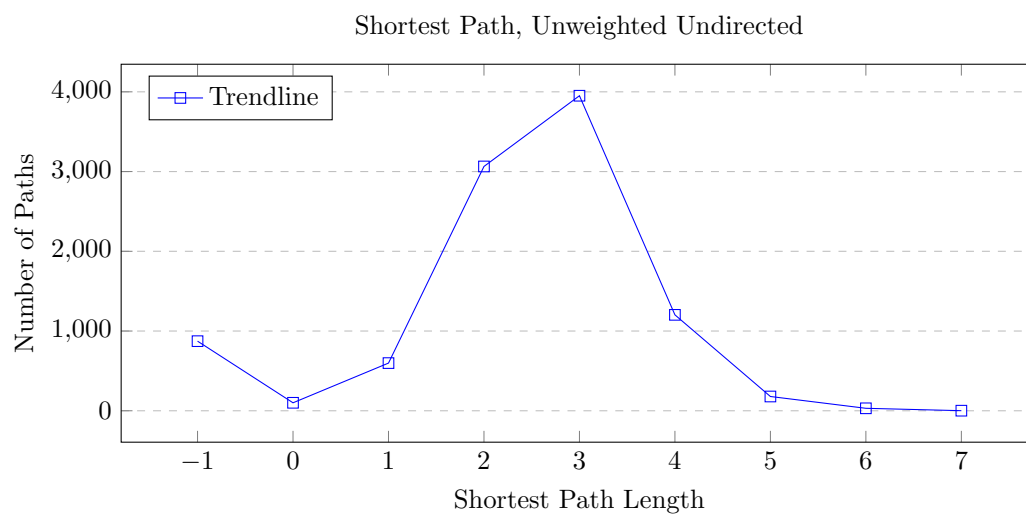
- 5
- 7

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
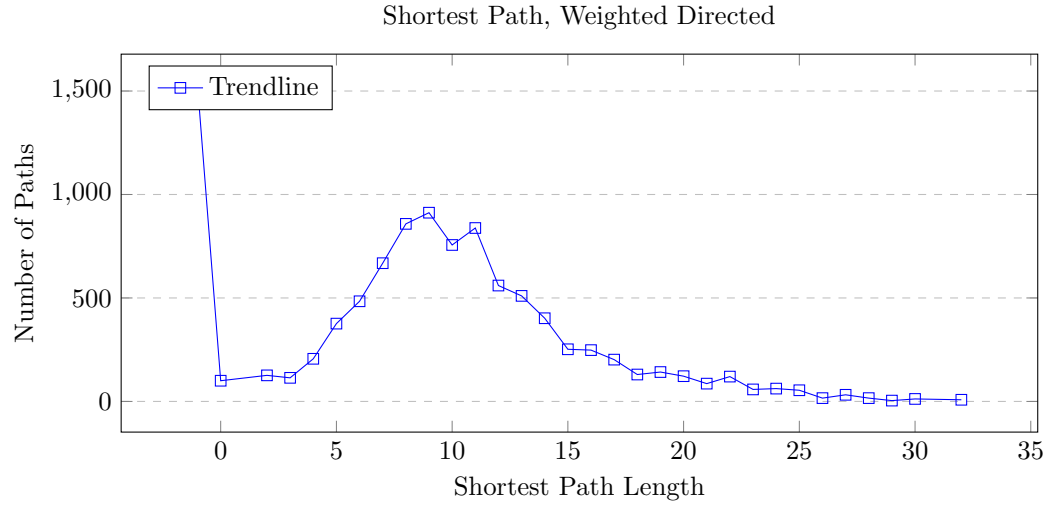
Shortest Path, Unweighted Directed

3.2.2 Shortest Path, Unweighted Undirected

Shortest Path, Unweighted Undirected
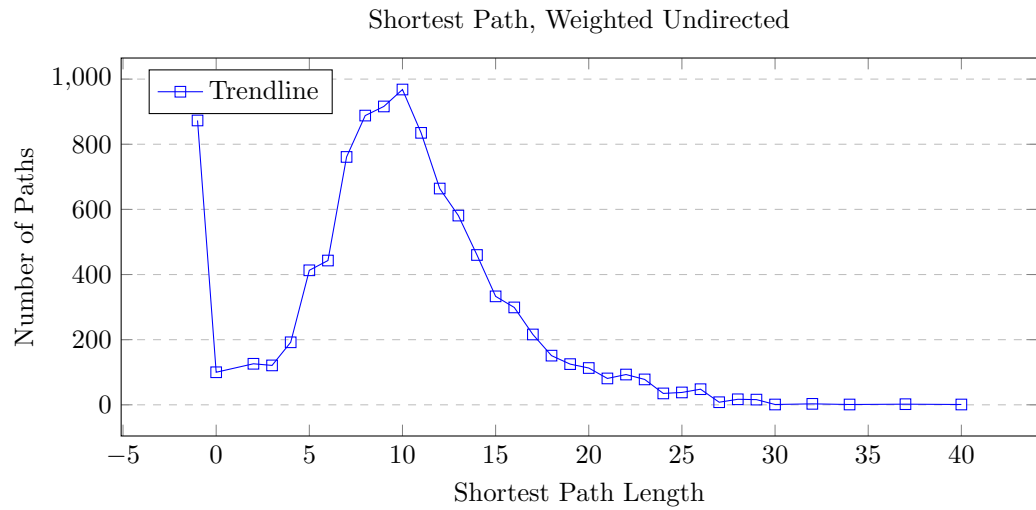
3.2.3   Shortest Path, Weighted Directed

Shortest Path, Weighted Directed

3.2.4   Shortest Path, Weighted Undirected

Shortest Path, 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 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. **INSERT SOME CRAP ABOUT DIAMETER RESULTS**. 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. As we can see from the results nodes **INSERT NODES** are the four nodes with the highest Closeness Centrality and in general **MORE CRAP ABOUT CENTRALITY**. 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. **BETWEENNESS CRAP** . 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. **COMMUNITY CRAP** . 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.