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.
   - Because our data was given in the form of a `csv`, we decided to just pipe that input directly to a `vector<string>`.
   - We then pass said `vector<string>` to a function that parses using C++ string functions.
   - 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.
   - Initialize a `map` of `<int, int>` for unweighted and `<int, double>` for unweighted.
   - For both weighted and unweighted simply use the source as the `key`.
   - For unweighted, use the `size()` method of the vector class to determine the out degree[1].
   - 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.
   - This is done almost the same way, except there is a weight `map`.
   - 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.
   - Initialize an adjacency matrix — a `vector<vector<int>>`
     - Default all values to infinity — in our case, 999999.
   - Copy over data from our `map` to said adjacency matrix.
     - If unweighted and an edge exists, default to 1.
   - 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.
   - Run the Floyd-Warshall algorithm, with triple C-Style `for` loops.

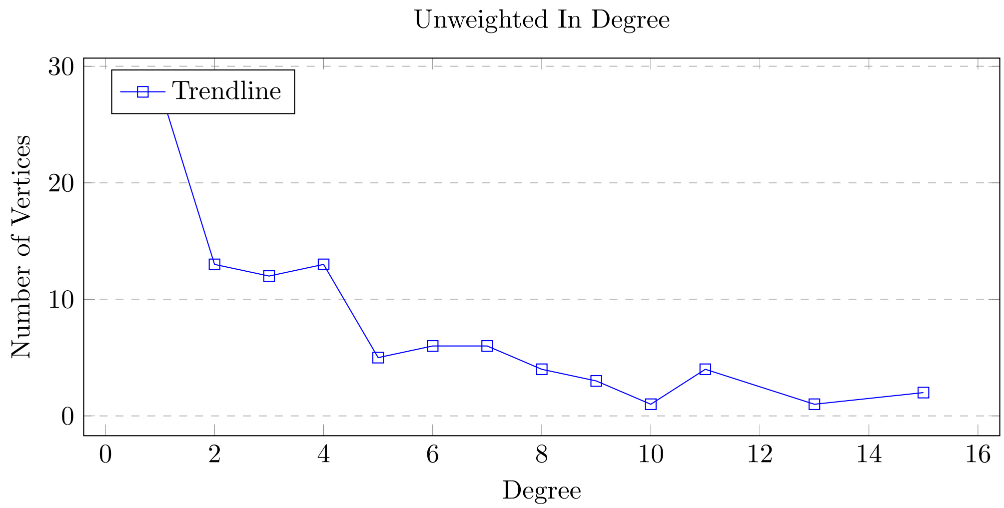
[1] Remember, the key return the a `vector` of pairs. The number of pairs are directly proportional the out degree.

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

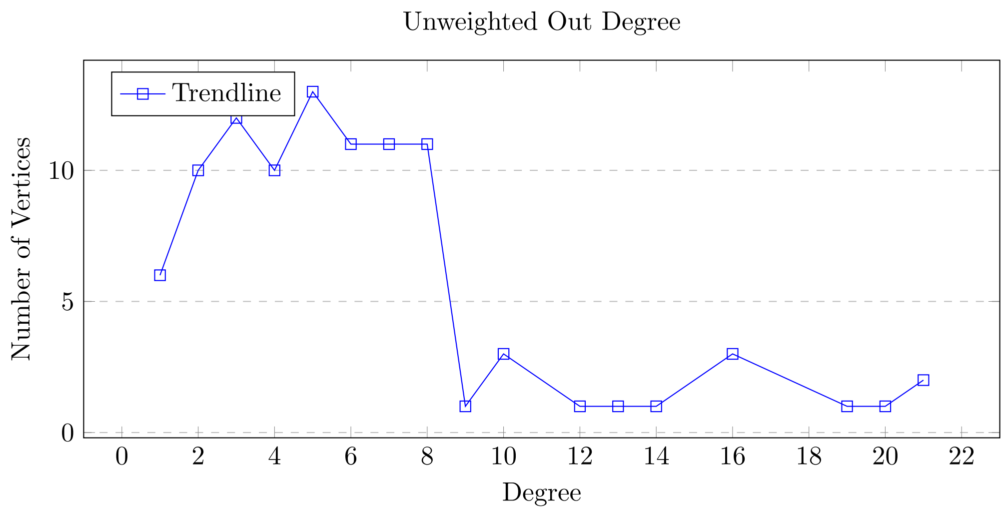
3 Experiments

3.1 Degree Distribution

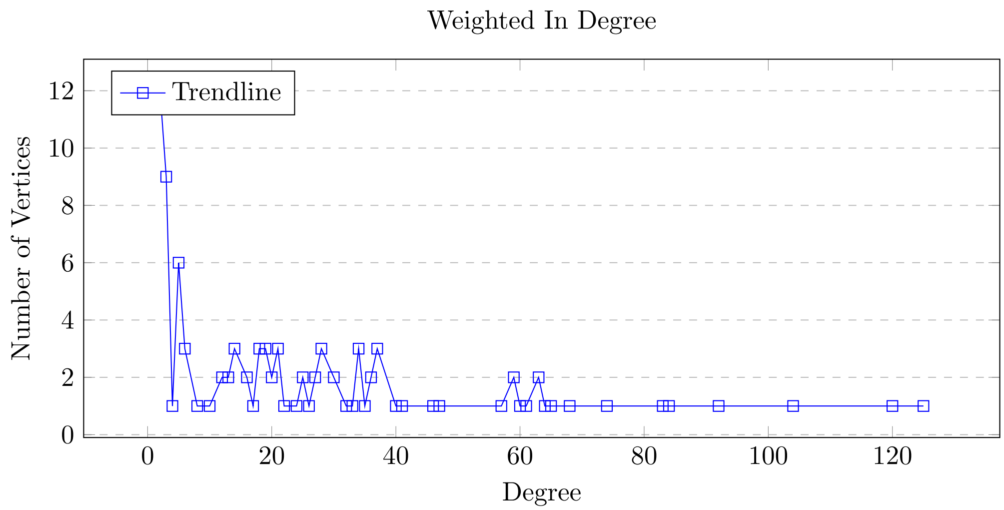
3.1.1 Unweighted In Degree



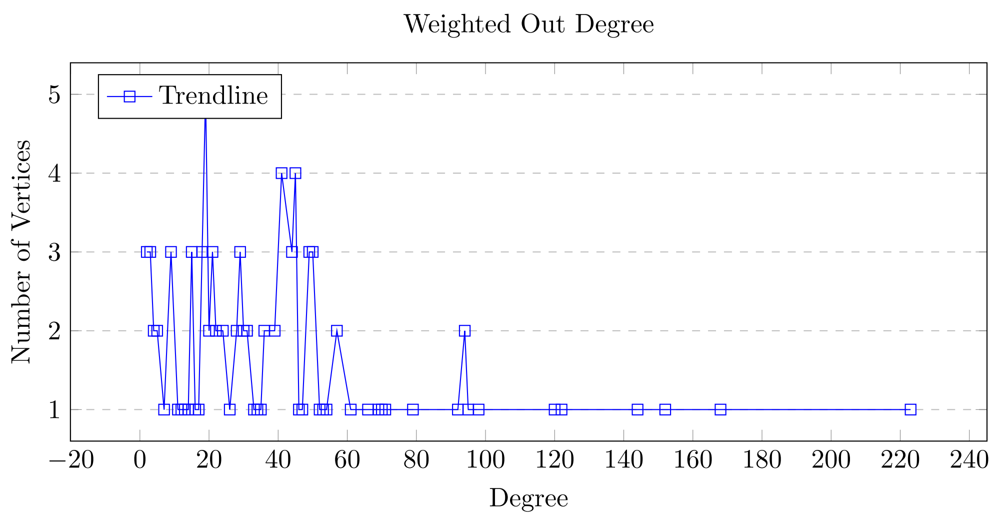
3.1.2 Unweighted Out Degree



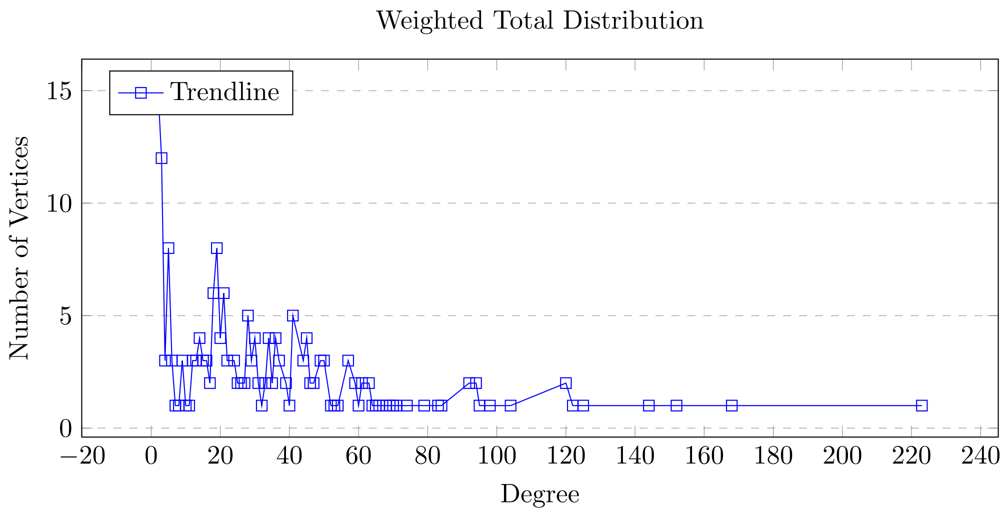
3.1.3 Weighted In Degree



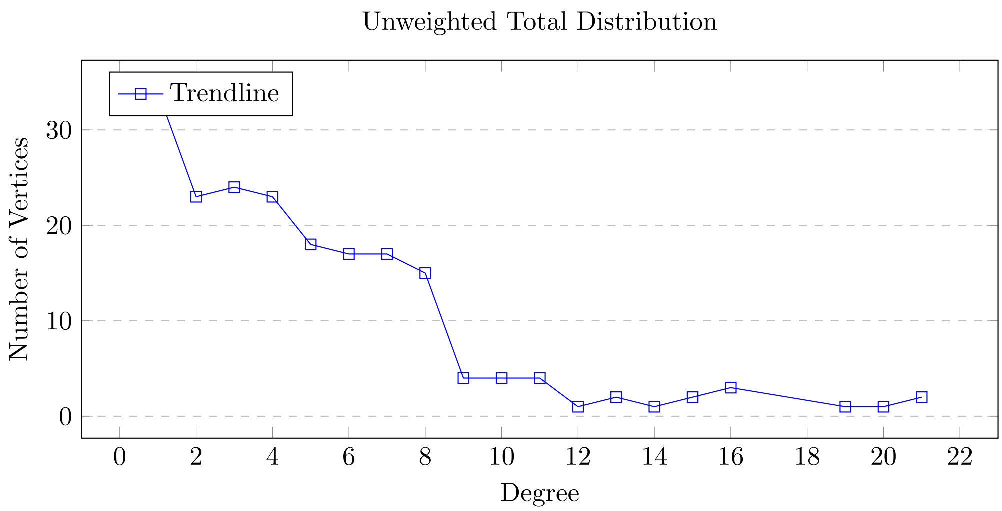
3.1.4 Weighted Out Degree



3.1.5 Weighted Total Distribution



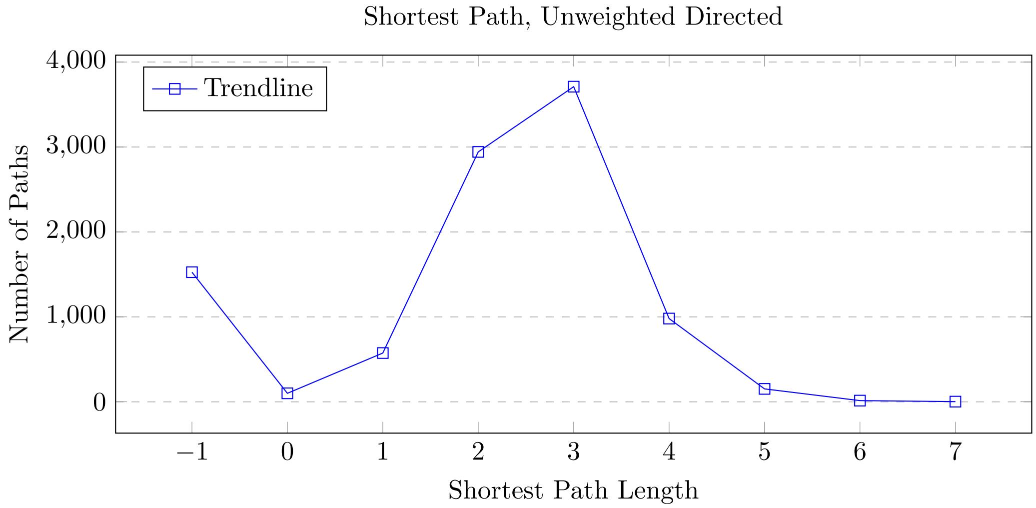
3.1.6 Unweighted Total Distribution



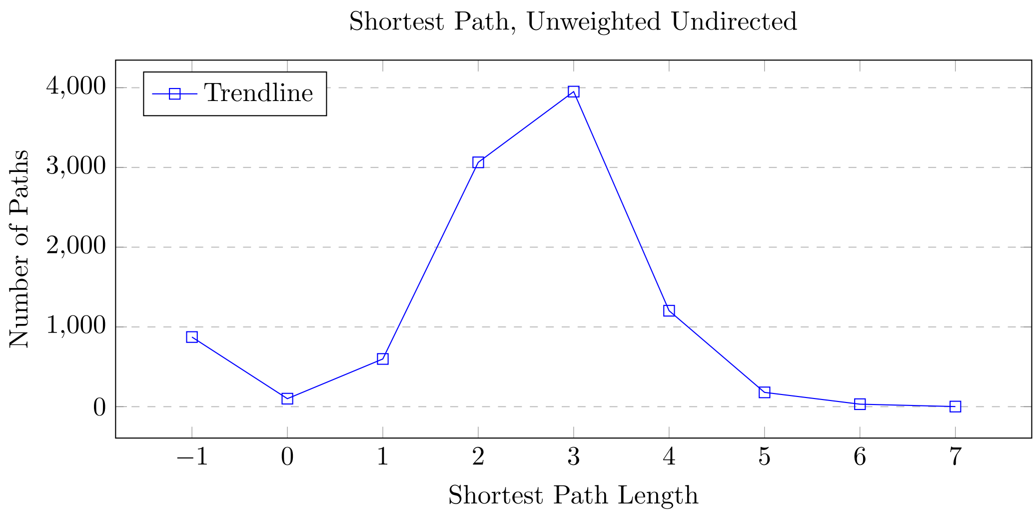
3.2 Shortest Path

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

3.3 Shortest Path, Unweighted Directed

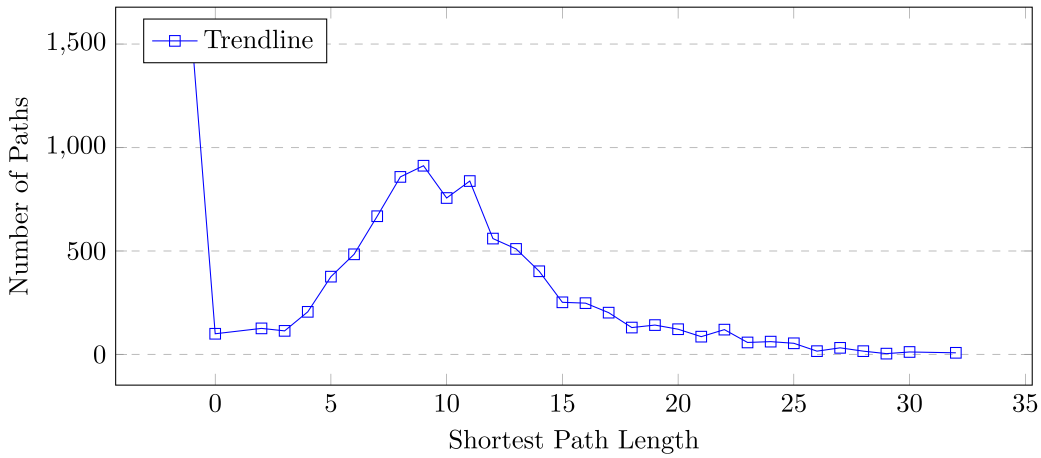


3.4 Shortest Path, Unweighted Undirected



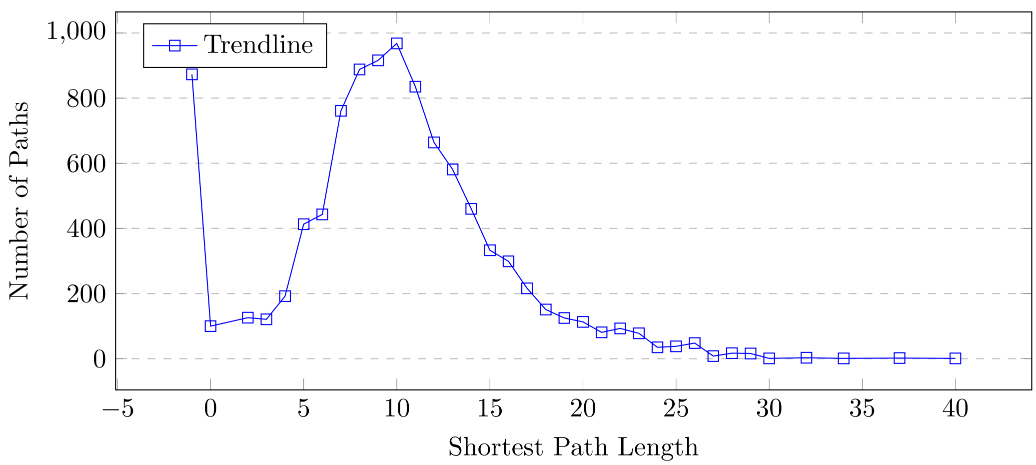
3.5 Shortest Path, Weighted Directed

Shortest Path, Weighted Directed



3.6 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