Won Joon Lee
15647075
CS165
Due 6/7/19

Project 3 Report

Introduction:

Inputs: For Diameter and Clustering coefficient, the input sizes used were 10^1 to 10^5, increased by a factor of 10 each time.

For Degree Distribution, I used 1,000, 10,000, and 100,000.

Graph Type: As my student ID is odd, I used **Erdos-Renyi** Graphs.

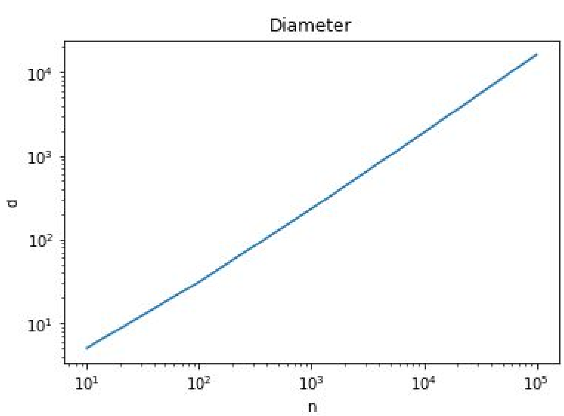
Definitions:

- **Eccentricity:** length of the longest shortest path from v to w.
- **Diameter:** maximum distance between a pair of nodes in the network.
  - A.k.a Maximum **Eccentricity** over all nodes.
- **Clustering Coefficient:** The probability of a connecting to c given that a connects to b and b connects to c.
  - A.k.a "friend of a friend is a friend"
  - C = 3 * number of triangles / number of 2-edge paths
- **Degeneracy:** Smallest value of d for which every subgraph has a node of degree at most d.
- **d-degeneracy Ordering:** The ordering of the nodes such that for each node, the number of its neighbors that come earlier in the ordering is at most d.

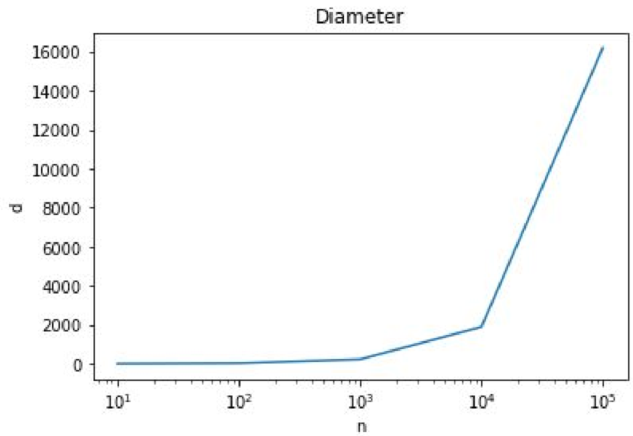
Algorithms:

Diameter:

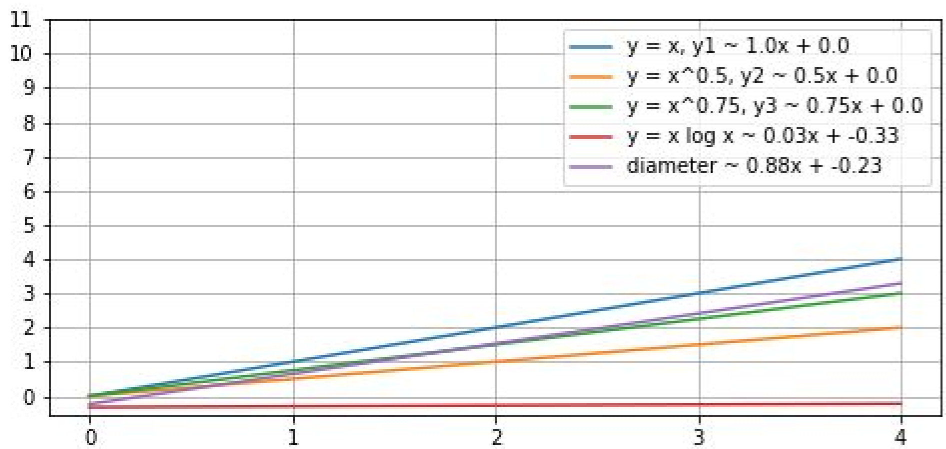
Lin-Lin Graph:



Lin-Log Graph:



Regression Graph:



Data:

Size (n)	Diameter (d)
10	5
100	31
1,000	232
10,000	1,894
100,000	16,192

Pseudocode:

1. Start at node 1. (current node = 1)
2. Set D_{max} = 0
3. Do BFS from r - BFS returns a pair<int, int> representing <Eccentricity of current node, last node explored>.
4. If Eccentricity of current node > D_{max} , do the following:
    a. set D_{max} to Eccentricity of current node.
    b. Set current node as the last node explored.
    c. Repeat Step 3.

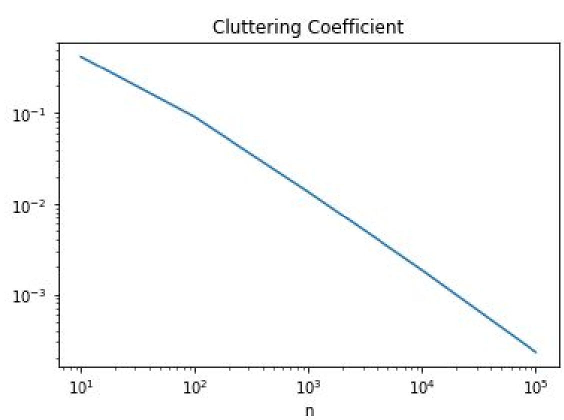
Return D_{max} at the end.

Description:

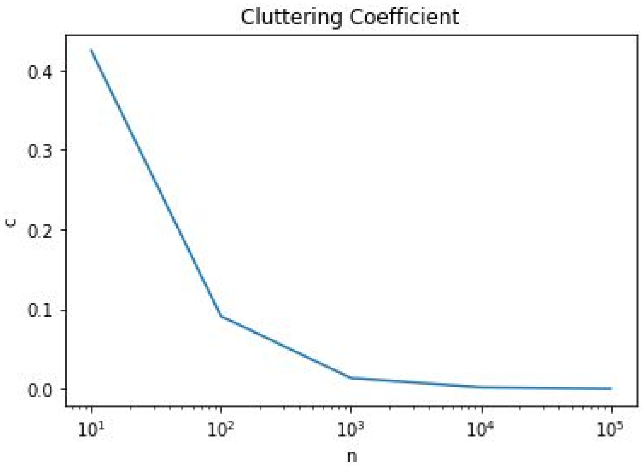
While there are multiple ways to get the diameter of a graph, I used this one because it was very simple yet effective. Compared to the naive algorithm's O(nm) time complexity (find the eccentricity of every vertex (each BFS takes O(m) time)), this way takes away the n part of the time complexity. According to the lin-log graph, it is clear that diameter increases as n increases. And according to the linear regression graph, the diameter definitely increases faster than log n as n increases at **d(n) = 0.88n + C.**

Clustering Coefficient:

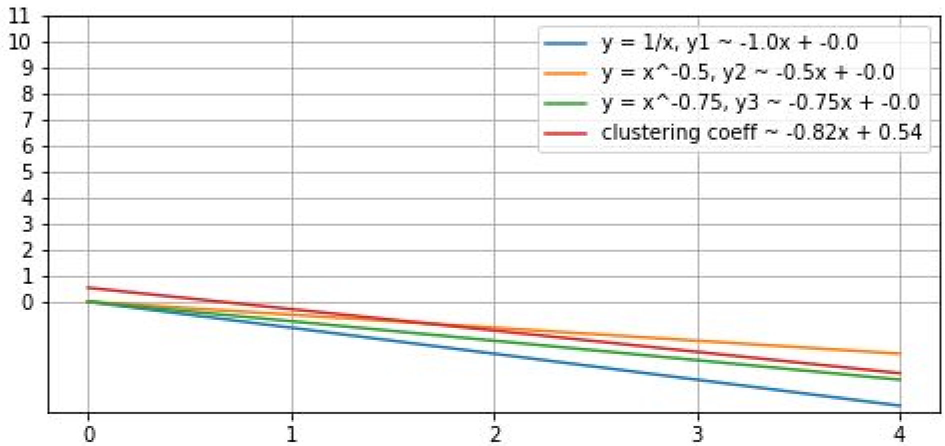
Lin-Lin Graph:



Lin-Log Graph:



Regression Graph:



Data:

Size (n)	Clustering Coeff (c)
10	0.424587
100	0.0911513
1,000	0.0135821
10,000	0.00185361
100,000	0.000231491

Pseudocode:

Prerequisite: Get number of triangles and number of 2-edge paths;

How to get Number of Triangles:

Prerequisite: Get **Degeneracy Ordering.**

- I will skip the pseudocode for this function, because it is directly on the slides in detail.
- https://www.ics.uci.edu/~goodrich/teach/cs165/notes/NetworkAlgs.pdf

1. Set Triangle Count = 0
2. Iterate through the **Degeneracy Ordering:**
3. For each vertex v, do the following:
    a. For each pair of neighbors of v, u and w, that is also earlier in the ordering than v, do the following:
        i. If u and w are neighbors (edge(u,w) exists), then add one to triangle count.

Return Triangle Count at the end.

How to get Number of 2-edge paths:

1. Set Count = 0
2. Iterate through the nodes in the graph.
3. For each node, do the following:
    a. Get its degree
    b. Compute deg choose 2 = deg * (deg - 1) / 2
    c. Add it to Count.

Return Count at the end.

How to get Clustering Coefficient:

Return 3 * number of triangles / number of 2-edge paths

Description:

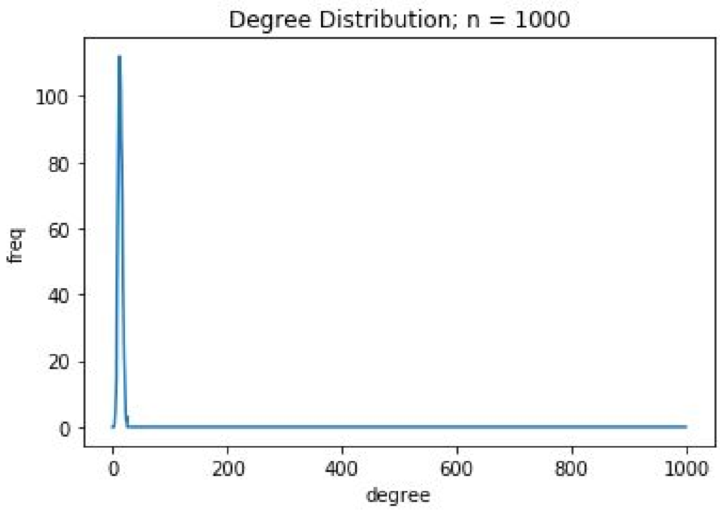
Looking at the lin-log graph, it's evident that the clustering coefficient (C) decreases as n increases. This is natural, as the larger the graph gets, the lower the likelihood of a node forming an edge with its neighbor's neighbor. In addition, according to the linear regression graph, we can see that **C(n) = -0.82n + Constant.**

Degree Distribution:

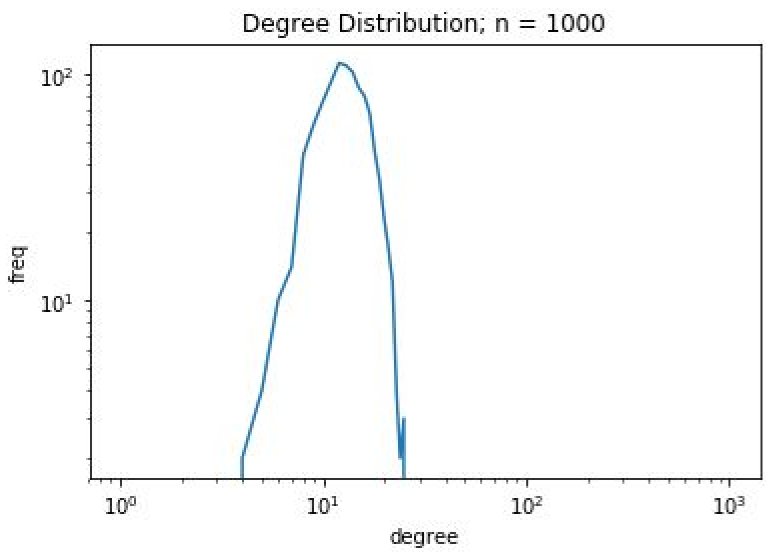
N = 1,000:

Data Graph:

Linear:



Loglog:

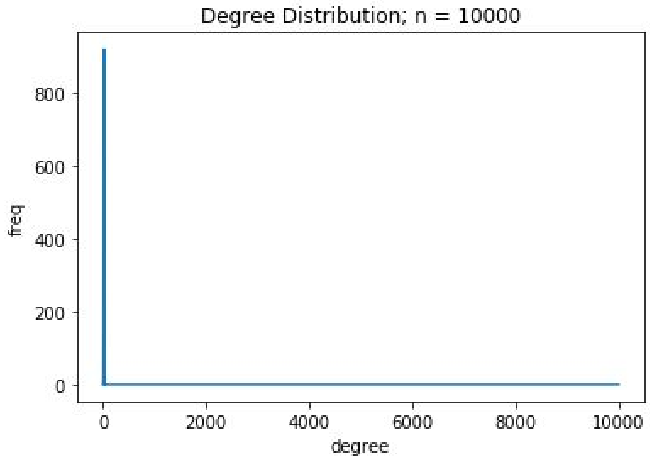


No Power Law.

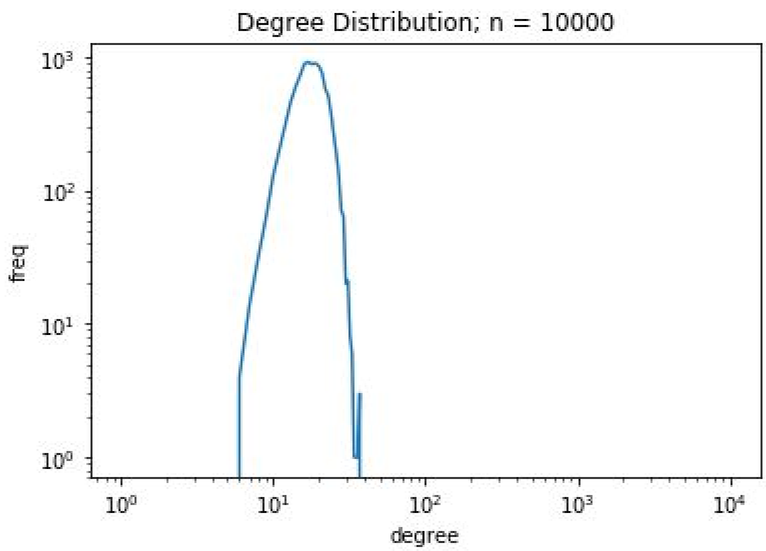
N = 10,000:

Data Graph:

Linear:



Loglog:

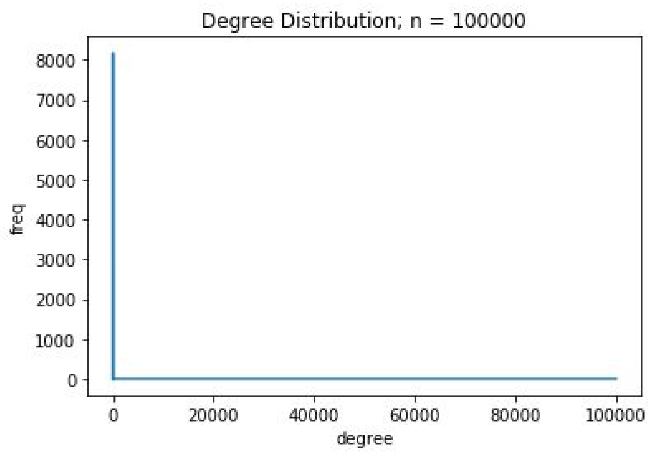


No Power Law.

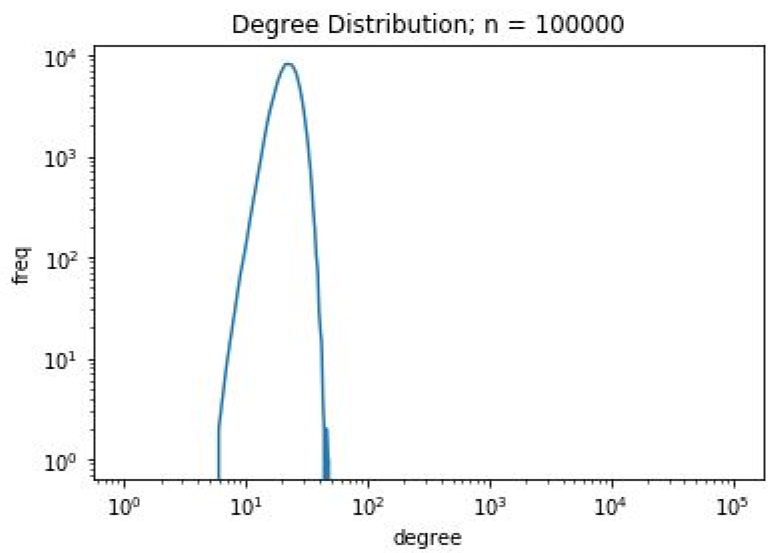
N = 100,000:

Data Graph:

Linear:



Loglog:



No Power Law.

Pseudocode:

1. Initialize a histogram count array of size n, and initialize H[i] = 0, for i in 0 to n - 1.
2. Iterate through the nodes of the graph.
3. For each node: do the following:
    a. Get its degree
    b. increment H[degree]

Return H

Description:

None of the Degree Distribution has **Power Law.**, as the log-log graphs do not display a consistent slope. This is weird, as the lin-lin graph displays a clear tail.