Project Three: Algorithmic Experiments of Real World Phenomena

Jean-Paul Nguyen

June 7, 2019

1 Introduction

1.1 Purpose

The purpose of this project is to implement and examine the properties of connected unweighted undirected graphs, which are simulated through network models. In particular, the properties of these graphs are diameters, clustering coefficients, and degree distributions, and the network model to generate the graphs is the Erdos-Renyi Random Graph Model. From the network model and properties, I can observe and find trends in these properties and use those trends to inform what the Erdos-Renyi Model tries (and fails) to simulate in the real world.

1.2 Experimental Setup

From the required API of the project, a Node class and a Graph class were created, and four main functions were developed to run on the Graph class. All these were developed in C++ and used a variety of supplied standard library data structures such as queues, maps, and vectors – note that in the pseudocode, such structures are referred to abstractly, but they are used and analyzed with respect to the actual C++ implementations.

The Graph class is of note since the given API had no constraint on what the actual implementation of the graph structure would be. With the problem in hand being dealt on graphs simulating real life networks, I decided it was reasonable to see such networks as sparse since every node really has a finite number of edges and relationships with other nodes in practice (e.g. social networks). As such, to best represent a sparse graph, the Graph class represents the graph structure as an adjacency list, with an array of C++ vectors that hold nodes. While it made checking for the existence of edges a bit slower, the space used is smaller than $O(n^2)$ from an adjacency matrix, so the trade-off allows for larger graphs to be tested.

With the graphs generated by Erdos-Renyi model and the average diameters, average clustering coefficients, and degree distributions collected, the data was plotted according to the specifications of the project. More detail on the implementation, the testing, and the observations of these data will be discussed later in their respective sections for the describe network model and properties. One thing to note: I stuck to 3 reps for data collection since I wanted to avoid potentially using too much memory when making these graphs and running algorithms with lots of data structures on them multiple times.

1.3 Outline

The paper will delve into the following topics:

1. The Erdos-Renyi Random Graph Model
2. Diameters

3. Clustering Coefficients

4. Degree Distributions

The background and implementations are discussed for all these topics, and the three properties are later measured and analyzed in relation to the random graphs from the Erdos-Renyi Random Graph Model. Each property has been observed to maintain some relation in the real world, and this paper will delve into how well the graphs of the Erdos-Renyi model compare.

2 Graph Generation: Erdos-Renyi Random Graph Model

There are many instances of real-world complex systems that can be modeled as network graphs, such as social networks, biological networks, or Internet links. In an effort to model these systems for the project, I used the Erdos-Renyi Random Graph Model since my ID number was odd. The Erdos-Renyi Model uses the idea that every pair of nodes is equally likely to be connected in a graph with some probability, i.e. the existence of an edge in a random graph generated from the model is independent from that of other edges. This model works on the idea that every relationship in the world was made independently of each other (though we will see that this is unrealistic in our results).

2.1 Implementation

Here is the pseudocode of the function I used to generate the random graphs: This function deals on the

Algorithm 1: Pseudocode for Erdos-Renyi random graphs

Input: n nodes and probability $0 \leq p < 1$
Output: Random graph G with n nodes

G = graph with n nodes;
$v = 1$, $w = -1$;
while $v < n$ **do**
  r = random float from $[0.0, 1)$;
  $w = w + 1 + \left\lfloor \frac{\log 1-r}{\log 1-p} \right\rfloor$;
  while $w \leq v$ *and* $v < n$ **do**
    $w = w - v{+}{+}$;
  end
  if $v < n$ **then**
    add edge (v, w) to G;
  end
end
return G;

idea that the "waiting time" to add an edge normally is $1 - (1-p)^k$ for some k that indicates $k-1$ edges are "skipped" before the k^{th} edge is added to the graph. Splitting the interval $[0, 1)$ into waitings times p, $p(1-p)$, ..., we can have r be some random float from that interval. Indeed,

$$r < 1 - (1-p)^k \longleftrightarrow k > \frac{\log(1-r)}{\log(1-p)}$$

From this, we choose k to be $1 + \left\lfloor \frac{\log(1-r)}{\log(1-p)} \right\rfloor$ to choose a particular node. In the function, k is set to w, and after adjusting v and w to cross such that $w > v$, the edge is added to G up until v is greater than n.

For this project, I was told to set p to be $\frac{2 \ln n}{n}$ for the number of nodes n. This $p > \frac{1}{n}$ for $n \geq 10$, i.e. the range of numbers of nodes I tested on, which means that the graph likely has a giant component where all the nodes are connected. This is interesting to note since it affects the observed properties of diameter, clustering coefficient, and degree distribution for the graphs. In addition, with this property, I had tailored in particular the diameter and clustering coefficient algorithms to assume that all the nodes in the given graphs were connected.

3 Diameter Algorithm

The *diameter* refers to the maximum distance between two nodes in a graph or network. Also known as the longest shortest path, the diameter can be used to describe the reach of the nodes; for a given node v, the number of steps it takes to reach another node w would be the *eccentricity*, with the largest being the diameter. This is important for other properties of graphs such as degrees of separation (e.g. Bacon and Erdos numbers) and the "small-world" property, which describe real-world social networks as graphs with small average distances and high clustering coefficients. With the diameter, I would be able to tell if a given graph has a small average distance since the diameter is the maximum.

Normally, to get the diameter of a given unweighted undirected graph G with n nodes, the common graph algorithm Breadth-First Search (BFS) is used on every node, with the eccentricity of each node retrieved by the largest distance travelled after the search – from here, the maximum eccentricity is returned as the diameter. However, if the graph has m edges, BFS would run with a time complexity of $O(n + m)$. This is a problem; random graphs from Erdos-Renyi would normally have way more edges than nodes, and running BFS on every node would take $O(mn)$ total time, which is very long. As a result, I decided to make an algorithm that uses a heuristic to get the diameter in a significantly shorter time, though it unfortunately cannot guarantee the exact diameter.

3.1 Implementation

Here is the pseudocode of the function I used to approximate the diameter of a graph:

Algorithm 2: Pseudocode for diameters

Input: Graph G with n nodes and m edges
Output: Approximation (lower-bound) of diameter d

```
S = set of nodes explored;
d = 0;
repeat
    s = random node from G not in S;
    add s to S;
    repeat
        b = farthest node with distance d_b in BFS on G from s;
        add b to S;
        if d ≥ d_b then
            break;
        end
        s = b, d = d_b;
    until;
until 5 times unless size of S ≥ n then end;
return d;
```

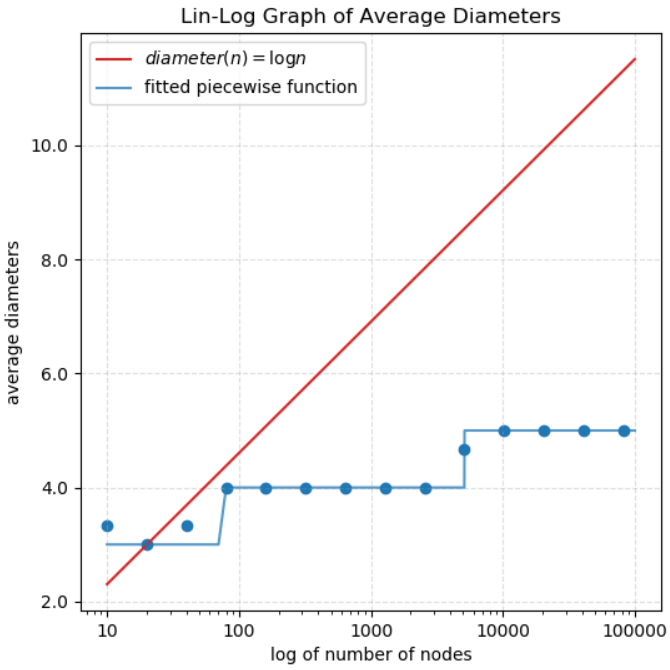
I used the following heuristic (which was the second one given in the class per Dr. Goodrich):

1. Choose a random node s in the graph.
2. Run BFS on s, and retrieve the farthest node b reached by BFS.
3. Repeat the previous step with b until the maximum distance reached by BFS does not change.

Here, this heuristic decreases the number of times the $O(n+m)$ is called by a significant amount, i.e. way less than n. For insurance, I continue to run this heuristic 5 times to continue checking the maximum distance, keeping track of which nodes that BFS has been ran on already with a set. Note that this set has a time complexity of $O(\log n)$ for finding, though the nature of the heuristic cuts down the number of nodes that BFS is ran on by skipping the nodes in the middle of the graph that have guaranteed smaller eccentricities. Overall, this time complexity would be around $O(nm)$ worst-case still, though in practice the actual time would be way less since BFS is run way less than n times.

However, there is a problem with this algorithm: There is no guarantee that the found max is actually the diameter. At best, the found max is a lower bound on what the diameter is in reality; the algorithm could be very unlucky and never run BFS on the node with the highest eccentricity. There has been much research into this topic of finding the diameter in a short period of time, but unfortunately there is no sure-fire way to find the true diameter in faster than $O(nm)$ time. As such, there is a chance the data I have collected on diameters is undershooting, though I believe the data are very close approximations to the actual diameters. During analysis, this caveat on my algorithm will be kept in mind.

3.2 Performance



n	diameter
10	3.33333
20	3
40	3.33333
80	4
160	4
320	4
640	4
1280	4
2560	4
5120	4.66667
10240	5
20480	5
40960	5
81920	5

Figure 1: Graph and Data of Diameters

Here is a graph of the average diameters run on random graphs from 10 to 81,920, doubling the size each increment. Notice that the trend of the average diameters was growing, so $diameter(n) = \log n$ was also drawn in the graph; due to the lin-log axes, the function is a straight line.

3.3 Analysis

From the graph, it is clear that the average diameters are growing as a function of n. Based on my data, the diameters seem to be in a piecewise function interestingly. I tried my best to fit a piecewise function of n here:

$$diameter(n) = \begin{cases} 3 & n < 80 \\ 4 & 80 \leq n < 5120 \\ 5 & n \geq 5120 \end{cases}$$

With values of $n > 81920$, we would most likely expect the average diameter to get to 6 only with extremely large n's. Clearly, this means that the average diameter would grow at a **much slower rate** compared to $\log n$. The graph confirms it, with $\log n$ reaching above 10 while the fitted piecewise function takes a while to get to 5. Since the average diameters seem to be very small even for large n's, it would make sense to say that the average length of short paths in these Erdos-Renyi graphs are also short, which fulfills one part of the "small-world" property from real world networks.

Of course, there is a caveat. Because my algorithm does not find the exact diameter, it is possible that there are some incorrect measurements in the data when calculating the averages. This may mean that the function may look different usually, for instance if the diameters actually varied more frequently for the node sizes. However, since the diameters are lower-bound approximations that I believe to be pretty close, the analysis of diameters growing slower than $\log n$ can still be valid. Even if a given graph of 81920 had a diameter of 6 instead of 5, not much in the trend would really change. Overall, the diameters of random graphs from the Erdos-Renyi model tend to grow slow as n increases, reaching at most 5 (or possibly 6) once n approaches 100,000.

4 Clustering-Coefficient Algorithm

The *clustering coefficient* refers to a measurement of how often nodes of a graph tend to cluster or group towards each other in a graph. As the "small-world" property states, real-world networks tend to have high clustering coefficients – this means that two nodes are more likely to be connected if they have the same neighbors. These can be seen in social networks like Facebook and Instagram where friends tend to have many mutual friends. Thus, I should want the random graphs from the Erdos-Renyi model to also have high clustering coefficients.

The clustering coefficient of a graph is just the average of all the clustering coefficients of the nodes. For a node v, the clustering coefficient is the actual number of edges between neighbors of v divided by the maximum number of edges between neighbors, which is the same as choosing two from the degree of v. Another way to think of the clustering coefficient of a graph is the following:

$$CC(G) = \frac{3 \times \text{number of triangles}}{\text{number of two edge paths}}$$

This is the interpretation I will use for my implementation of the function, though with it comes the obstacle of getting the count of triangles in the graph. Checking every combination of three nodes in a graph would normally be $O(n^4)$ normally in an adjacency list, which is not suitable to get the coefficient quickly. Instead, what could be used is a *degeneracy ordering* which orders the nodes by repeatedly removing the node of smallest degree from the graph. This ordering is such that a degeneracy of d implies that a node has at most

d neighbors that are earlier in the ordering. That cuts down a lot of the work since I can then process the nodes in the degeneracy ordering, using earlier neighbors of each node to check for triangles.

4.1 Implementation

Here is the pseudocode of the function I used to get the degeneracy ordering of a graph:

Algorithm 3: Pseudocode for degeneracy ordering

```
Input: Graph G with n nodes and m edges
Output: Ordered list of nodes and early neighbors L
L = empty list of nodes;
degrees = map with node as key and int as value;
added = map with node as key and bool as value;
D = list of lists of nodes where D[i] holds all nodes with i degree in the ordering;
for node i in G do
    degree[i] = degree of node i;
    add node i to D[degree[i]];
end
repeat
    v = node from smallest degree i where D[i] is not empty;
    remove v from D[i];
    added[v] = true;
    N_v = list of neighbors earlier in the ordering;
    for neighbor w of v do
        if not added[w] then
            remove w from D[degree[w]];
            add w to D[degree[w]];
            degree[w] = degree[w] − 1;
            add w to N_v;
        end
    end
    add v, N_v to back of L
until n times;
reverse L;
return L;
```

Here, I use two maps, *degree* and *added*, and a list of lists D to keep track of the node with the smallest degree during the ordering. With a constant time lookup for maps and a linear search to remove from D, the algorithm would seem to take $O(n^2)$ time, but in actuality the algorithm is $O(n+m)$ since every node is found proportional to the degree that they have, and the sum of all degrees is $2m$. Note that to keep the time complexity, I added v to the back of L, then reversed the list, making the time complexity $O(n+m)$ instead of just $O(m)$. Due to the nature of the algorithm constructing in reverse order, reversing L is necessary (since adding v to the front with the standard vector would be more costly).

I also wanted to keep track of the neighbors that were earlier in the ordering, so along with a node v, a list of neighbors N_v is also added in a pair to the ordering L. This will allow the clustering coefficient algorithm to merely check for one edge instead of three for every triangle, which is clearly faster as the algorithm goes through each node and edge. Following is the pseudocode of the function I used to get the clustering coefficient of a graph:

Algorithm 4: Pseudocode for clustering coefficients

Input: Graph G with n nodes and m edges
Output: Clustering coefficient C

$num = 0$, $dem = 0$;
L = degeneracy ordering of G;
for *node* i *in* G **do**
  d_i = degree of i;
  $dem = dem + \frac{d_i(d_i-1)}{2}$;
end
for *node* v *in* L **do**
  let N_v be neighbors of v that are earlier in L;
  for *each pair* (u, w) *of neighbors in* N_v **do**
    if (u, w) *is an edge in* G **then**
      num++;
    end
  end
end
$C = \frac{3 \times num}{dem}$;
return C;

In this function, *num* and *dem* are constructed, adeptly named for the numerator and denominator of the returned coefficient. *dem* is the number of two-edge paths, which can easily be found by going through each node and choosing two from the number of neighbors the node has. In other words, for a degree d, $\binom{d}{2}$ is added to *dem*. For *num*, the count of triangles needs to be done, so by using the degeneracy ordering L, the count is done faster than by choosing three nodes and checking. In particular, as the nodes are processed in the order of L, the neighbors of a node v that are earlier in the degeneracy ordering is N_v. Each pair in N_v is checked to be in G, and if so, the count of triangles (and thus *num*) is incremented. At the end, 3 is multiplied to *num* and divided by *dem* to get the returned clustering coefficient C.

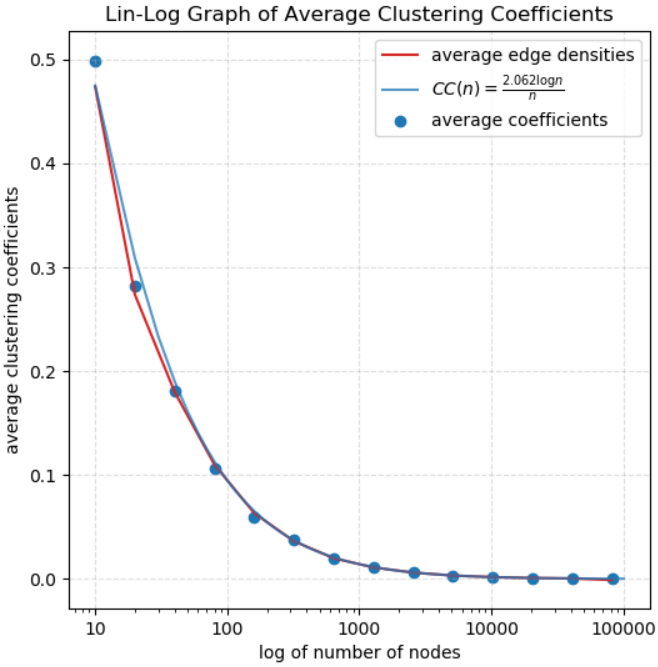
Compared to the $O(n^4)$ method of checking every combination of 3 nodes in G for n nodes and m edges, this algorithm performs significantly faster. Getting *dem* is $O(m)$, getting the degeneracy ordering is $O(m+n)$, and checking the pairs of N_v would also be $O(m)$ with a hashmap of edges. In actual implementation, I did check with the is-neighbor function instead of the hashmap, which may have made my time complexity more like $O(mn)$ worse case, so technically my real time complexity would be $O(mn)$. That being said, it is possible to make this algorithm just $O(m+n)$ using more data structures.

4.2 Performance

The following figure shows the graph and the data of average clustering coefficients run on random graphs from 10 to 81,920, doubling the size of each increment. I also include data of average edge densities for each number of nodes; this was to check the idea that the clustering coefficient should be compared to the edge density, a measure of how likely random pairs of vertices are to be neighbors. Calculating the edge density for a graph of n nodes and m edges is the following:

$$density(n) = \frac{m}{\frac{n(n-1)}{2}}$$

If there is a huge difference, i.e. the clustering coefficient is higher than the edge density on a graph, then the graph is considered to be highly clustered. In addition, an estimated curve is displayed for the coefficients.



n	coeff	density
10	0.498818	0.474074
20	0.28151	0.273684
40	0.180463	0.179915
80	0.106548	0.110021
160	0.0596039	0.0634696
320	0.0378293	0.0363375
640	0.0199743	0.0200248
1280	0.01145	0.0111313
2560	0.00603592	0.00613368
5120	0.00337372	0.00333287
10240	0.00178622	0.00179878
20480	0.000958604	0.000969799
40960	0.000507385	0.000517829
81920	0.000269571	-0.000985961

Figure 2: Graph and Data of Clustering Coefficients

4.3 Analysis

As we can see, the values of the average clustering coefficients are decreasing as the number of nodes increase. Using $\frac{\log n}{n}$ as the function to parameterize, I fitted to the data the following function for the clustering coefficient CC:

$$CC(n) = \frac{2.062 \log n}{n}$$

Since $\log n$ is growing extremely slowly compared to n, the function ends up decreasing and approaching zero. In fact, the clustering coefficient is extremely close to the calculated p used for Erdos-Renyi generation, $\frac{2 \ln n}{n}$. Thus, as the random graphs of the Erdos-Renyi model gets bigger, the clustering in the graphs seem to lessen. Visually, it even looks like the edge densities and average coefficients follow close to the same trend, with the differences between the data for each n being very small as well. This implies that the graphs are not considered to be highly clustered, which violates the second part of the "small-world" property.

Because of how Erdos-Renyi generate the random graphs, this makes sense; since the generation is done by choosing edges to add at random with independence, clusters are rarely going to happen because there is no preference for certain edges done due to previously chosen edges. As the result, coefficients are low since not as many triangles are created via the model. Interestingly, the model also influences why the edge densities seem to have a very similar trend to clustering coefficients with the data. The edge density describes how likely a pair of randomly chosen nodes are neighbors, and since the graph is generated with a certain probability for an edge to be added, the edge density seems to reflect the p of the model too. Overall, the nature of Erdos-Renyi random graph generation keeps the produced graphs from having larger clustering coefficients, meaning there are not many clusters that are generated as n grows. This keeps such graphs from modeling real life networks well since they do not follow the "small-world" property.

5 Degree-Distribution Algorithm

As heavily discussed, the *degree* is the number of neighbors that each node has in a given graph, and the *degree distribution* visualizes how many nodes have certain degrees. There are two typical ways that distributions look: "bell-shaped" and "heavy-tailed". Graphs with bell-shaped degree distributions can be fitted with a normal or Gaussian distribution, i.e. most of the nodes tend to have a degree close to a mean, and the frequency of nodes with a smaller or larger degree is smaller. On the other hand, graphs with with heavy-tailed degree distributions tend to have a lot of nodes with a small degree and not as much nodes with large degrees.

These heavy-tailed degree distributions exhibit a property called the *power law* where if graphed on a log-log scale, the resulting graph would look linear. In particular, there would be a slope α such that

$$freq(d) = cd^{-\alpha} \longleftrightarrow \log freq(d) = -\alpha \cdot \log d + \log c$$

With regards to degree distributions of real life networks, heavy-tailed distributions are more accurate to what they look like since networks tend to have this signature of a power law. For instance, the distribution of degrees in banking networks have this power law. As such, for the random graphs of Erdos-Renyi to model real networks, I would expect a power law to exist i.e. an α can be estimated from the data.

5.1 Implementation

Here is the pseudocode of the function I used to get the degree distribution of the graph:

Algorithm 5: Pseudocode for degree distribution

Input: Graph G with n nodes and m edges
Output: Map M with degree as key and count as value

```
M = map with int as key and value;
for i = 1 to n do
    M[degree of i]++;
end
return M;
```

Unlike the other algorithms, the algorithm for degree distributions is straightforward. Running in linear time, the algorithm keeps the frequencies of every degree in the graph using a map and returns the map, with the degree as the key and frequency of degree as the value.

5.2 Performance

The next page shows three pairs of graph where the degree distribution is collected and displayed for random graphs of $1,000$, $10,000$, and $100,000$ nodes. Each pair consists of the degree distribution represented as a lin-lin scale bar graph on the left and a log-log scatterplot on the right of the same data.

5.3 Analysis

Clearly from the graphs, there is a noticeable curvature for both the lin-lin and log-log scales. Recall that a heavy-tailed distribution would exhibit the power law by having a negatively-sloped linear relationship for the log-log scaled distribution. However, it is obvious that no such linear relationship exists for Erdos-Renyi random graphs, being more akin to a bell-shaped distribution; there is no slope α such that the line $\log freq(d) = -\alpha \log d + \log c$ can be fitted for degree d and some constant $c > 0$. That means the power law

does not apply to the random graphs of Erdos-Renyi, furthering the idea that the model fails to represent networks accurately.

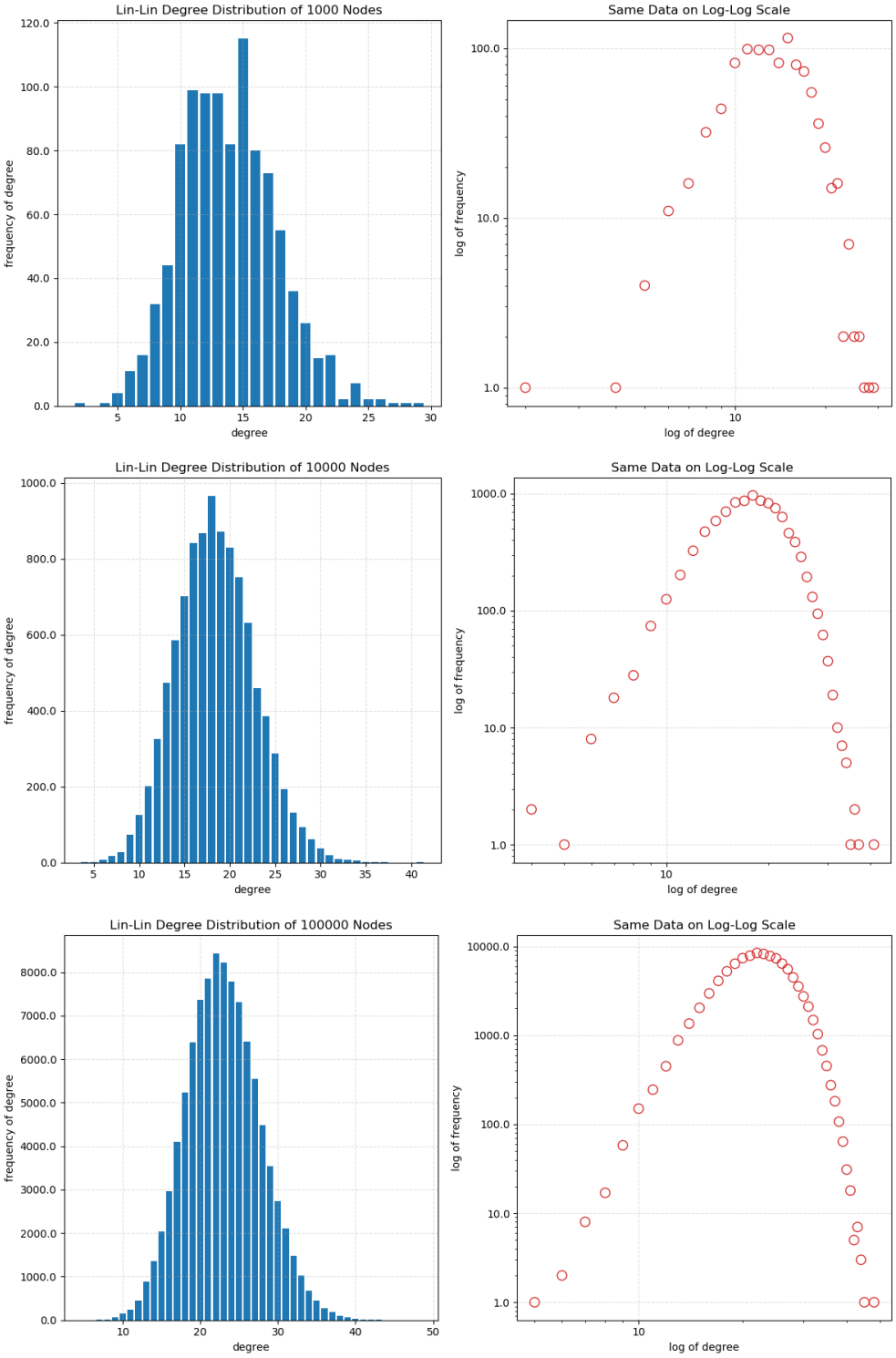


Figure 3: Degree distributions at 1000, 10000, and 100000 nodes

6 Conclusion

The goal of this project was to observe the properties of unweighted undirected random graphs generated by the Erdos-Renyi model, in particular the diameters, clustering coefficients, and degree distributions. In each section for the properties, I discussed that networks in real life tend to have the following trends:

- Diameters of large graphs are normally small, with most nodes in a graph being reachable within 6 jumps in the shortest paths.
- Clustering coefficients are high, representing the idea that clusters are more likely to form since nodes with mutual neighbors tend to also be neighbors. This property and the small diameters make up the "small-world" property.
- Degree distributions display the power law, forming a heavy-tailed distribution since many nodes have small degrees while few have large degrees.

With these trends in mind, I tested on graphs from the Erdos-Renyi model to see if this way of generating graphs through choosing edges at random independently was able to accurately represent networks in real life via these properties.

However, from the data collected, that does not seem to be the case. The diameters of these graphs grow at a rate slower than $\log n$. While these graphs do have approximately small diameters less than 6, the clustering coefficients are also small (and seem to be near equal to the chosen p for generation), which violates the small-world property. Finally, degree distributions are bell-shaped for these graphs instead of heavy-tailed, meaning the power law does not apply. As such, the Erdos-Renyi model seems to be a poor choice for representing real world networks.

The reason why this model seems to not perform well is because the model works on the idea that a pair of nodes in the graph are able to connect independently from other pairs of nodes. However, this independence means clusters are less likely to happen and degrees are more distributed in a normal fashion. As other models like Barabasi-Albert assume, real life networks tend to have a preferential attachment or "rich-get-richer" property: The more connections a node has, the more likely the node is to get more connections. This property would easily work into frequent clusters and heavy-tailed degree distributions. In the end, while the Erdos-Renyi model can generate random graphs will small diameters, the independence of edge probability keeps the model from making high cluster coefficients and heavy-tailed degree distributions for graphs.