Daniel Vu

CS 165

Project 3

Graph Algorithms

I – Introduction

A graph is a non-linear data structure that contains vertices (or called as nodes) and edges (a path connected between two vertices). Graphs are used to illustrate the real-world networks. In social networks, vertices are every people and edges are our social relationships. In the technology world, vertices are physical machines while edges are physical wires. Understanding a graph can help us easier to comprehend the complexity of real-world networks.

Graph algorithms are one of the most commonly known algorithms nowadays. There are infinite ways that graph algorithms can be involved in real-world networks. Some significant applications for graph algorithms are maps that use to illustrate the building transportation systems and navigate the shortest path from two places, also network flows can be analyzed to find the road's intensivity to improve the traffic situation. In social networks, friends can be recommended that if each person is a vertex and graph theory can be used to find a path, or an edge between people to suggest friends.

In this project, I implemented three types of graph algorithms, which are diameter algorithm, clustering-coefficient algorithm and degree-distribution algorithm. I also test these graph algorithms on two types of random graph generation model which are Erdos-Renyi random graphs and Barabasi-Albert random graph.

II – Implementations

1. Three experimental algorithms in this project

A. Diameter algorithm

Diameter is the maximal distance over all node pairs. In the brute-force algorithm, we have to use All Pairs shortest path algorithm which will take largest time complexity. In this project, I use the heuristic idea 2 to find the diameter. We will use random uniformly distributed algorithm to choose a random vertex in a graph and perform breadth-first search algorithm from that vertex to get to the last vertex. Then, we

perform a distance algorithm from both vertices, and again we will repeat the process. We will keep repeating the process if the distance found later is larger than the former distance, otherwise we will store the distance as the diameter of the graph.

Pseudocode:

source chosen random

for diameter is smaller than distance

destination = last node from BFS from source
distance(source,destination)
diameter = max (diameter,distance)
source = destination

The running time for this algorithm is $O(n^2)$.

B. Clustering-coefficient algorithm

A clustering coefficient is a measure of the degree to which nodes in a graph tend to cluster together, create a triangle. The idea is that if a vertices a and b has an edge, vertices b and c has an edges so there is a probability, which is clustering coefficient that vertices a and c also has an edge. Calculate this clustering coefficient number is using this function:

Clustering coefficient = number of triangles * 3 / number of 2-edge paths

To get the number of triangles, I use a linear-time algorithm to find a list **L** of degeneracy ordering of all vertices and a list of neighbors **Nv** that come before each vertex **v** in **L**. Then we traverse through the degeneracy list L and also Nv to check if every pair of neighbors of v in Nv has an edge or not. If they have an edge which means three vertices create a triangle. To get the number of 2-edge paths, the idea is that each vertex can create a 2-edge path with other two neighbors. Therefore, the number of 2-edge paths of each vertex is the combination of degree of vertex choose 2. Equivalently to this function:

$$\text{Number of 2-edge paths} = \sum_{0}^{n-1} \frac{\deg(v)*(\deg(v)-1)}{2}$$

The running time for this algorithm is $O(n^3)$.

C. Degree-distribution algorithm

Degree distribution is the frequency of each degree, meaning that the number of vertices for each degree in the graph. The simple algorithm is to traverse to each node of the graph to get the degree, and for each degree found, increment the frequency of that degree. A histogram array will contain the degree and frequency in order to plot the value and determine if the degree distribution follows the power law or not.

The running time for this algorithm is linear time.

1. Supporting algorithms in this project

A. Breadth First Search Algorithm

Breadth first search for a graph is to traverse from one vertex to every neighbor of it, and from each neighbor, keeps traverse to their neighbors until no vertex left. This is the pseudocode for Breadth First Search traverse:

```
add initial node to queue
while queue is not empty do
    x = queue.front();
    if x has not been visited
        visited[x] = true;
        for every neighbor y of x
            if y has not been visited
                add y to queue
```

B. Shortest Path – Distance Algorithm

Distance is the shortest path connected between two vertices in the graph. I use Breadth First search from one vertex to get the level of neighbors from that vertex to see how many level of neighbors a vertex has to traverse to get to the other vertex. This is the pseudocode for shortest path algorithm:

```
add initial node to queue
visited[source] = true;
while queue is not empty do
    n = queue.front();
    for every neighbor y of x
        if y has not been visited
            visited[y] = true;
            distance[y] = distance[x] + 1;
            add y to queue
return distance[destination]
```

C. Degeneracy algorithm

A d-degeneracy graph is an undirected graph in which every subgraph has a vertex of degree at most d. In the other word, in a d-degenerate ordering list, each vertex has at most d neighbors to its left. The degeneracy ordering list can be implemented in linear time algorithm. This is the pseudocode for degeneracy algorithm:

1. Initialize an output list, L, to be empty.
2. Compute a number, d_v, for each vertex v in G, which is the number of neighbors of v that are not already in L. Initially, d_v *is* just the degrees of v.
3. Initialize an array D such that $D[i]$ contains a list of the vertices v that are not already in L for which $d_v = i$.
4. Let N_v be a list of the neighbors of v that come before v in L. Initially, N_v is empty for every vertex v.
5. Initialize k to 0.
6. Repeat n times:
   - Let i be the smallest index such that $D[i]$ is nonempty.
   - Set k to $\max(k, i)$.
   - Select a vertex v from $D[i]$. Add v to the beginning of L and remove it from $D[i]$. Mark v as being in L (e.g., using a hash table, H_L).
   - For each neighbor w of v not already in L (you can check this using H_L):
     - Subtract one from d_w
     - Move w to the cell of D corresponding to the new value of d_w, i.e., $D[d_w]$
     - Add w to N_v

III – Description of inputs and outputs

My graph algorithms are tested with two different network models:

1. **The Erdos-Renyi Random Graph Model**

In Erdos-Renyi Model, there are **n** number of vertices, and each pair of vertices can create an edge with a probability of **p**. We use n and p to generate a random graph. The idea is to generate a bit to decide if a pair of vertices can be connected. If that bit is 1 then there is an edge. The probability of getting (k-1) times 0's before getting a 1 is $(1\text{-p})^{k-1}$p. We also choose randomly a subinterval in the waiting time (which is geometrically distributed) to get the 1. Hence,:

$$\mathrm{r} < 1 - (1-\mathrm{p})^{\mathrm{k}} \rightarrow k > \frac{\log(1-r)}{\log(1-p)} \rightarrow k \sim 1 + \left|\frac{\log(1-r)}{\log(1-p)}\right|$$

Since p is a small number, therefore to have a faster generation, we will skip over waiting time of running of 0's. This is the pseudocode for Erdos-Renyi random graph generation according to lecture note:

ALG. 1: $\mathcal{G}(n,p)$
Input: number of vertices n, edge probability $0<p<1$
Output: $G=(\{0,\ldots,n-1\},E) \in \mathcal{G}(n,p)$
$E \leftarrow \emptyset$
$v \leftarrow 1$; $w \leftarrow -1$
while $v<n$ **do**
  draw $r \in [0,1)$ uniformly at random
  $w \leftarrow w+1+\lfloor \log(1-r)/\log(1-p) \rfloor$
  while $w \geq v$ **and** $v<n$ **do**
    $w \leftarrow w-v$; $v \leftarrow v+1$
  if $v<n$ **then** $E \leftarrow E \cup \{v,w\}$

2. **The Barabasi-Albert Random Graph Model**

In Barabasi-Albert model, a random graph is generated using a preferential attachment mechanism. Initially, the graph is set with given **n** number of vertices and a **d** degree to set the number of edges for each vertex. This will guarantee a degeneracy of a for the graph also. The fast Barabasi-Albert algorithm is to compute an array **M** of edges with size is 2nd, using random uniformly distributed algorithm to connect a vertex at random index to **d** vertices at other indices in the array M. This is the pseudocode for Barabasi-Albert random graph generation according to lecture note:

ALG. 5: preferential attachment
Input: number of vertices n
minimum degree $d \geqslant 1$
Output: scale-free multigraph
$G=(\{0,\ldots,n-1\},E)$
M: array of length $2nd$
for $v=0,\ldots,n-1$ **do**
  for $i=0,\ldots,d-1$ **do**
    $M[2(vd+i)] \leftarrow v$
    draw $r \in \{0,\ldots,2(vd+i)\}$ uniformly at random
    $M[2(vd+i)+1] \leftarrow M[r]$
$E \leftarrow \emptyset$
for $i=0,\ldots,nd-1$ **do**
  $E \leftarrow E \cup \{M[2i], M[2i+1]\}$

3. Random uniformly distributed algorithm

To create a random uniformly distributed set of input items given the size, we used Mersenne Twister pseudorandom number generator (mt19937) to get items with weights is int value and float value in the interval (0,1).

4. Input Size

The algorithm is examined with different size n having n grow. The minimum input size is 1 and the maximum input size is 10^6 . In total, there are 13 input sizes: 1,5,10, 50,10^2,5x10^2, 10^3, 5x10^3 ,10^4, 5x10^4 , 10^5, 5x10^5 ,10^6.

The probability **p** in Erdos-Renyi Random Graph Model is given as a function of input size:

$$p = 2 * \ln(n)/n$$

The parameter degree **d** in Barabasi-Albert Random Graph Model is given as 5.

Erdos-Renyi random graph generation creates 10 nodes, ~20 edges; 10^2 nodes, ~500 edges; 10^3 nodes, ~7000 edges; 10^4 nodes, ~ 90000 edges; 10^5 nodes, 1.1 x 10^6 ~ edges; 10^6 nodes, ~ edges.

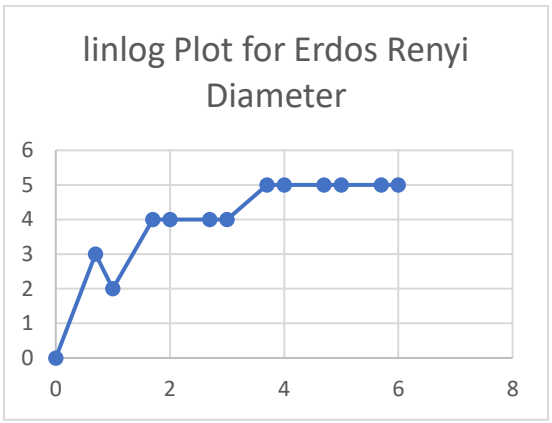
Barabasi-Albert random graph generation creates 10 nodes, ~25 edges; 10^2 nodes, ~500 edges; 10^3 nodes, ~5000 edges; 10^4 nodes, ~ 50000 edges; 10^5 nodes, ~ x 10^6 edges; 10^6 nodes, ~ edges.

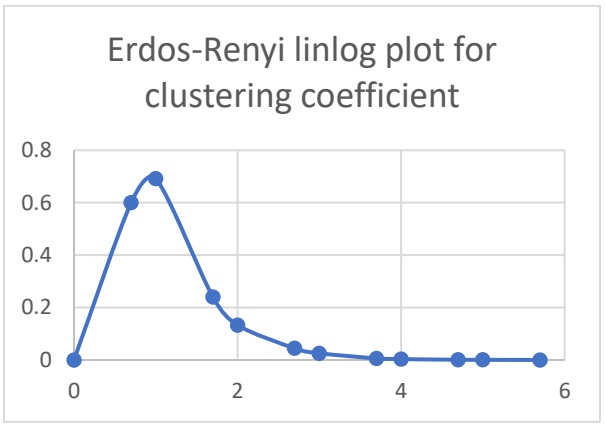
5. Times running for algorithms

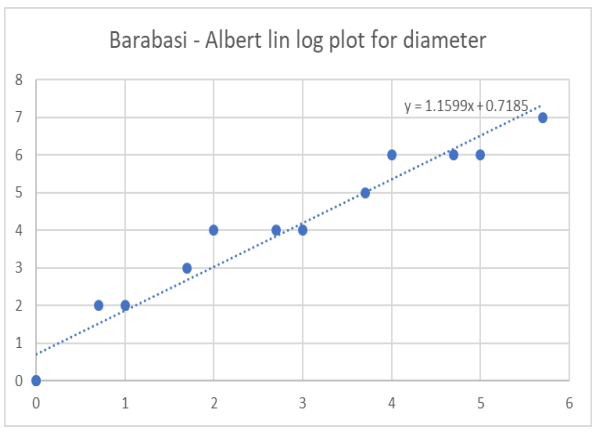
I run total of **ten** times for each input and get the diameter, clustering coefficients and degree distribution of each algorithm for each input sizes. In general, there are 13 input sizes * 10 times/size * 3 algorithms * 2 network models and it takes around 10 hours in total to get final result for all 10 tests.

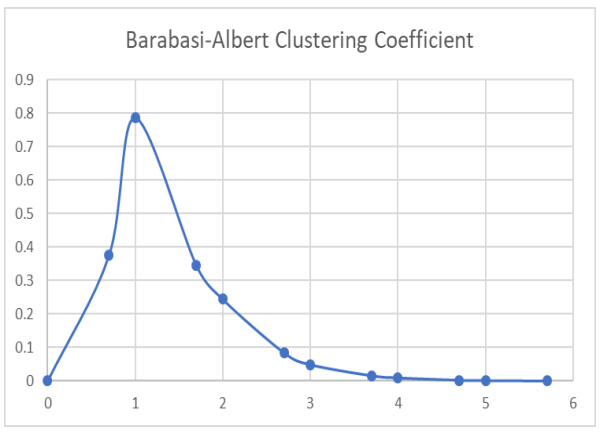
6. Output

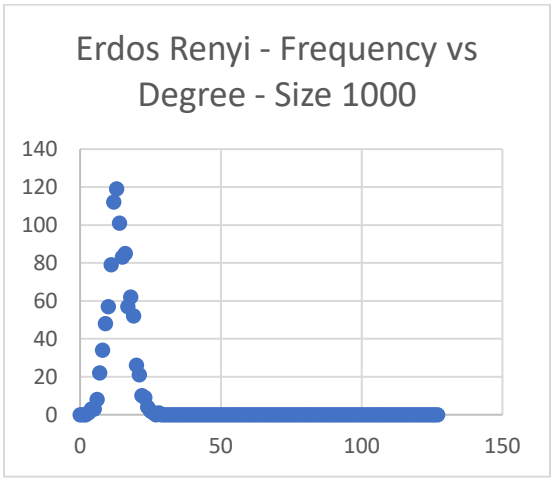
Information of diameter, clustering coefficients and degree distribution of every size is outputted to comma-separated values (csv) files and plot by excel.

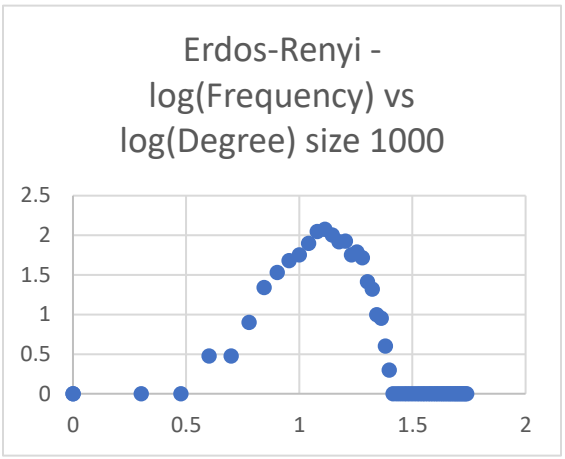


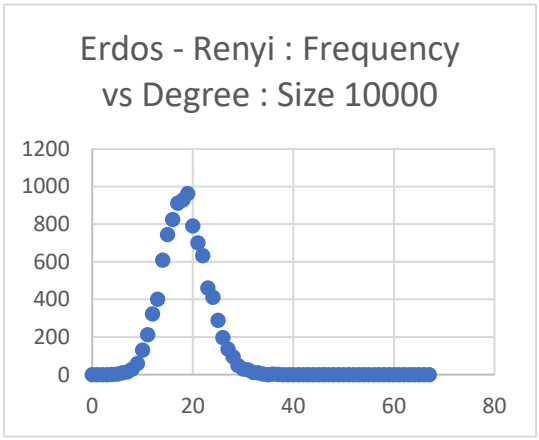


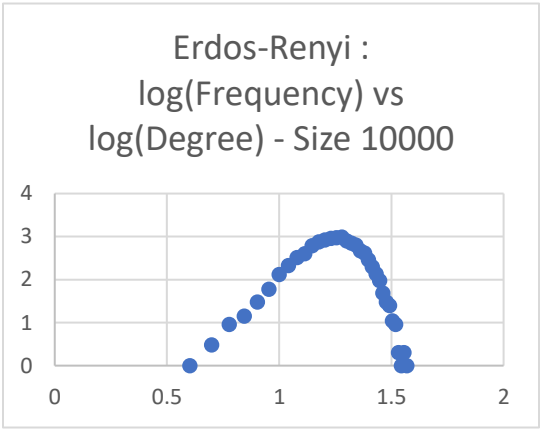


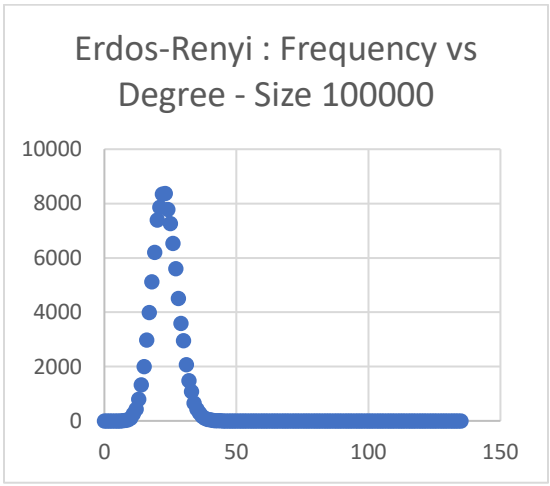


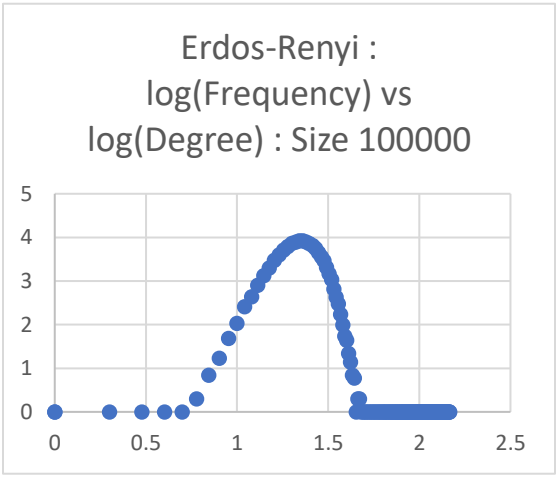

Erdos Renyi - Frequency vs Degree - Size 1000
140
120
100
80
60
40
20
0
0
50
100
150

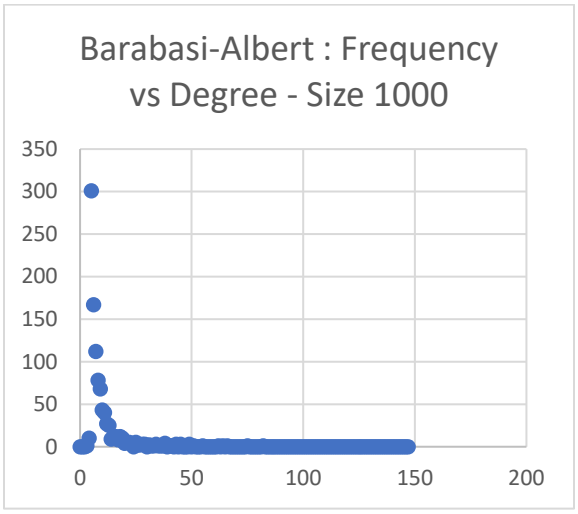

Erdos-Renyi - log(Frequency) vs log(Degree) size 1000
2.5
2
1.5
1
0.5
0
0
0.5
1
1.5
2

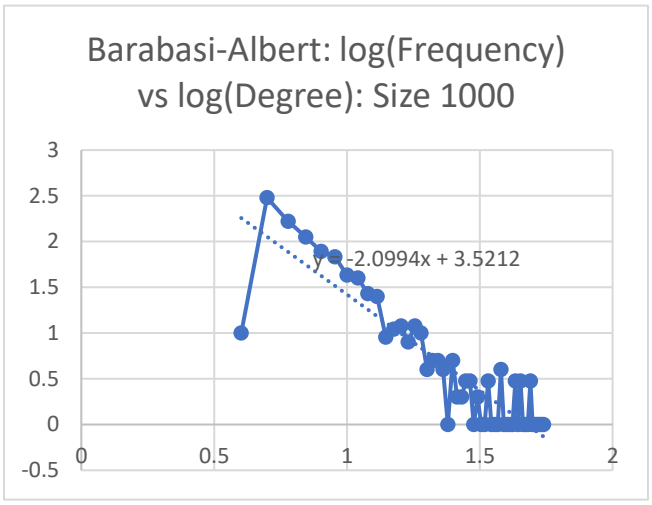

Erdos - Renyi : Frequency vs Degree : Size 10000
1200
1000
800
600
400
200
0
0
20
40
60
80

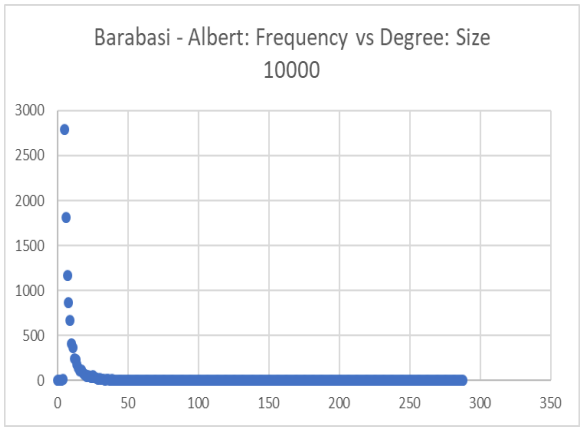

Erdos-Renyi : log(Frequency) vs log(Degree) - Size 10000
4
3
2
1
0
0
0.5
1
1.5
2

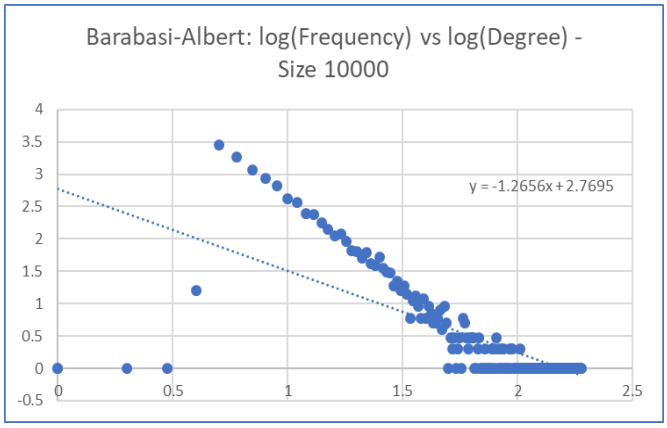

Erdos-Renyi : Frequency vs Degree - Size 100000
10000
8000
6000
4000
2000
0
0
50
100
150

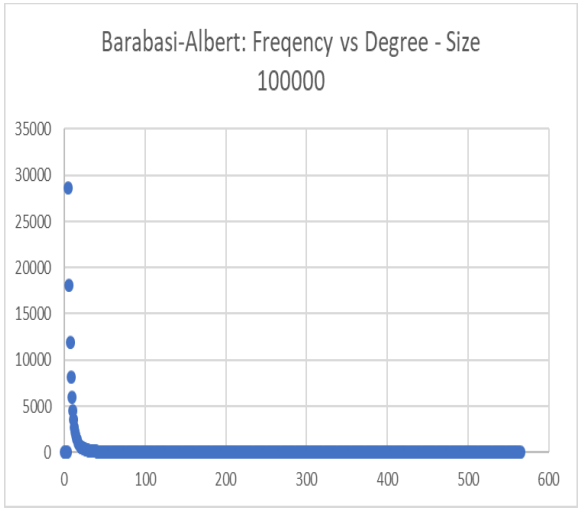

Erdos-Renyi : log(Frequency) vs log(Degree) : Size 100000
5
4
3
2
1
0
0
0.5
1
1.5
2
2.5

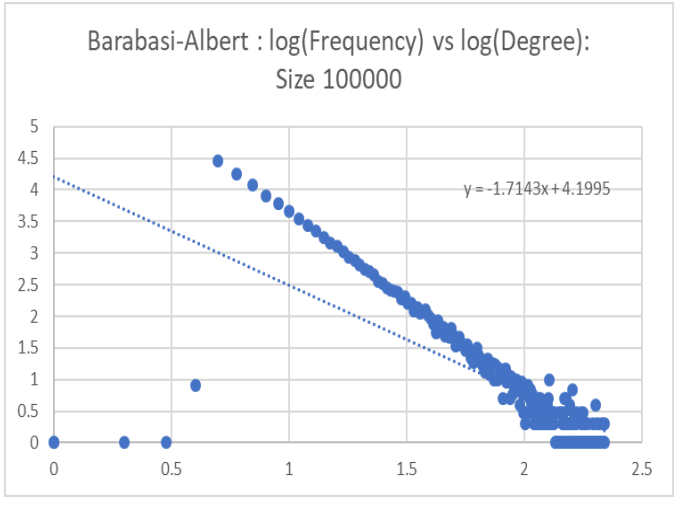

Barabasi-Albert : Frequency vs Degree - Size 1000
350
300
250
200
150
100
50
0
0
50
100
150
200


Barabasi-Albert: log(Frequency) vs log(Degree): Size 1000
y = -2.0994x + 3.5212
3
2.5
2
1.5
1
0.5
0
-0.5
0
0.5
1
1.5
2


Barabasi - Albert: Frequency vs Degree: Size 10000
3000
2500
2000
1500
1000
500
0
0
50
100
150
200
250
300
350


Barabasi-Albert: log(Frequency) vs log(Degree) - Size 10000
y = -1.2656x + 2.7695
4
3.5
3
2.5
2
1.5
1
0.5
0
-0.5
0
0.5
1
1.5
2
2.5


Barabasi-Albert: Freqency vs Degree - Size 100000
35000
30000
25000
20000
15000
10000
5000
0
0
100
200
300
400
500
600


Barabasi-Albert : log(Frequency) vs log(Degree): Size 100000
y = -1.7143x + 4.1995
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0
0.5
1
1.5
2
2.5

IV – Analysis

1. Diameter analysis

For Erdos-Renyi random graphs, it seems to be that the as n grow, the average diameters become more constant as a function of n where around 1000-1000000 the diameter remains the same as 6. According to the linlog plot, it shows that they grow slower but not proportional to log(n). That plot telling that the Erdos-Renyi random graphs will create a graph with larger connections, larger number of edges between nodes so that the path from one vertex to another will keep a small distance when n grows.

With Barabasi-Albert random graphs, as n grow the diameter increase, from 1 to 1000000 in size, the diameter keeps growing from 2 to 7. It seems that the also grow as a function of logn:

$$\text{diameter} = \log(n) * 1.16 + 0.7$$

This plot shows that with the Barabasi-Albert random graphs, as n grow the diameter will grow so that the distance from one vertex to another vertex can become larger.

2. Clustering coefficient analysis

For Erdos-Renyi random graphs, the clustering coefficient shows that when n grows, the clustering coeffient will decrease in terms of log(n).

For Barabasi-Albert random graphs, the clustering coefficient shows that when n grows, the clustering coeffient will decrease in terms of log(n).

3. Degree distribution analysis

For Erdos-Renyi random graphs, it seems to be that the degree distribution is not heavy-tailed and exponential decay from mean since the lin-lin plot is a curve graph and the log-log plot is also a curve graph. Therefore, it does not have a power law.

For Barabasi-Albert random graphs, it seems to be that the degree distribution is a power law since the loglog plot creates a regression line with a slope around -2. Therefore, the function of degree-distribution is Frequency = Degree^{-2}.