June 7, 2019

ALGORITHMIC EXPERIMENTS OF REAL-WORLD PHENOMENA

1. OVERVIEW

Introduction: This experiments involve testing various graph algorithms experimentally to determine properties of models of real-world networks. The specific graph algorithms include **Diameter algorithm, Clustering-coefficient algorithm, and Degree-distribution algorithm**. These algorithms will be used on two random graph that model the real-world networks. We generate two random graphs called **Erdos-Renyi** random graphs and **Barabasi-Albert** random graphs**.**

Input: For the graph algorithms, their input is Graph, which is an random graph represents model of real-world. For **Erdos-Renyi** graph algorithm, the inputs are number of vertices n and probability of having edge between two vertices p. For **Barabasi-Albert** algorithm, the inputs are number of vertices n and degree d the number of edges per new node.

2. GRAPH ALGORITHM ANALYSIS

Diameter algorithm

The diameter of a network is the maximum distance between a pair of vertices in the network. This algorithm aims to find the diameter given a graph.

In pseudo-code, we have the following algorithm

```
Starting from a random node n
MaxDistance = 0
Perform BFS startig from node n find the eccentricity E and farest node F
While MaxDistance <= E:
    n = F
    MaxDistance = E
    Perform BFS startig from node n find the eccentricity E and farest node F
return MaxDistance
```

Algorithm implementation:

Starting from a random vertex as current vertex and set the max distance zero from current vertex, we perform a Breath First Search from current vertex. In BFS search, always choosing the last node we search as new node and return the distance from the current node to the new node, which is the eccentricity for current node. If the new distance is larger than the distance we have so far. Return the distance. Otherwise set the new node as current node and perform BFS. Repeat above steps until we find the max distance.

Idea behind the algorithm: diameter is equal to the largest value returned by an all pairs shortest path. Unlike the brutal force that compare the distance of shortest path among all the vertices pairs in the graph, using BFS would be faster to find the maximal eccentricity over all nodes.

Clustering-coefficient algorithm

Clustering coefficient is a measure of the degree to which nodes in a graph tend to cluster together.

In pseudo-code, we have the following algorithm

```
//Calculate denominator first

number_of_two_edge = 0
for node in graph
    number_of_two_edge += (degree of node * (degree of node -1)) /2

//Calculate triangle count
D is an array store degeneracy order
L is an array stores the order
Get HL is map that mark the node is in L already
NV is a two-dimensional vector whose index is the node id and value is
the neighbor of node that comes before node in L
Degree is an vector stores the degree of current node

for each node in graph:
    select the fist node in D that has the smallest index
    remove the node from D insert the node in L
    change the HL of current ndoe = True
    for each node w that is the neighbor of node:
        subtract Degree[w] by 1
        move w in D to index Degree[w]

triangle_count = 0
for each node in L:
    get every pair of neighbor u and w in NV[node]:
           if u and w is neighbor:
               triangle_cout += 1

clustering coefficient = 3 * triangle_count / number_of_two_edge
```

Algorithm implementation:

Clustering coefficient equal to 3 * number of triangles / number of 2-edge paths. Compute the 2-edge path is easy. We just need to iterate through all the node in the graph and sum up their deg(v)(deg(v)-1)/2. To find the number of triangles in the graph, we need to get the degeneracy order of the graph, that is, if a graph has degeneracy d, then there exists an ordering of the vertices in G in which each vertex has at most d neighbors that are earlier in the ordering. To get the order, initialize an order list and keep track of all the nodes that's already in L . Also, store a list of neighbors that come before v in L for each node v and the degeneracy for all nodes. Repeat n time, move node from D[i] to L, change the degeneracy accordingly. After we generating the order, iterate the nodes according to the order. For every pair of node that is neighbor of current node and comes before it in order list. If there is edge between them. they form a triangle and count += 1. Finally, we can calculate clustering coefficient according to the formula above.

Degree distribution algorithm

Degree distribution represents the distribution of degrees of nodes for the graph. For each node's degree, we are interesting in the frequency of degree so we can see the distribution.

In pseudo-code, we have the following algorithm

```
Initialize a map # the key is degree and value is frequency
For each vertex in the graph:
    get the size of vertex's neighbor vertex
    map[size] += 1
```

Algorithm implementation:

Initialize the map of size n (number of node in the graph). Set it key from 1 to n and all the value initialized to be zero which is frequency. Iterate through all the nodes in the graph. Update each node's degree to the map. Return the Map.

3. GRAPH GENERATION ANALYSIS

Erdo-Renyi Generation algorithm

Erdo-Renyi algorithm is an algorithm for generating random graph. The inputs of it are number of nodes and probability p. For each pair of node among n nodes, the probability of existing edge between them is p. In my test, the p = 2 * ln/n, in which case, for a graph of n nodes, there would be n(n-1)/2 possible edges. In our algorithm, there would be expected **2*ln n*n** edges in the graph according to the probability given.

In pseudo-code, we have the following algorithm

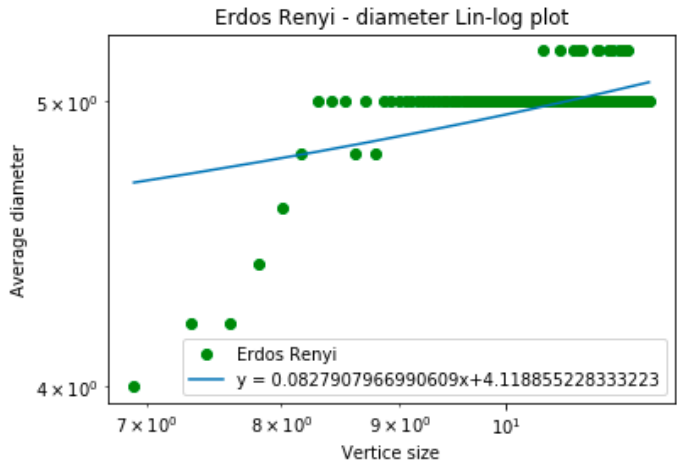
```
Input: number of vertices n, edge probability p = 2(ln n)/n

v = 1 # represents vertices
w = -1 # represents vertices to be connected by v
while v < n do
    draw number r uniformly at random in intervel [0,1)
    w = w + 1 + floor(log(1-r)/log(1-p))
    while w >= v and v < n do
        w= w-v
        v += 1
    if v < n:
        add edge(v,w) to the graph
```

Algorithm analysis:

Begin with n isolated vertices, no edge. The brutal force way of adding edges to the graph is for each pair node in the graph, randomly generate a bit b that is 1 with probability p. if b =1, add it to edge. But this would take o(n^2) time so for our algorithm, we divide the interval according to the waiting time, pick r uniformly at random in the interval [0,1]. The subinterval in which r falls will sample the waiting time so that we can skip over all the runs with b = 0 which resulting in a faster algorithm like pseudo-code above.

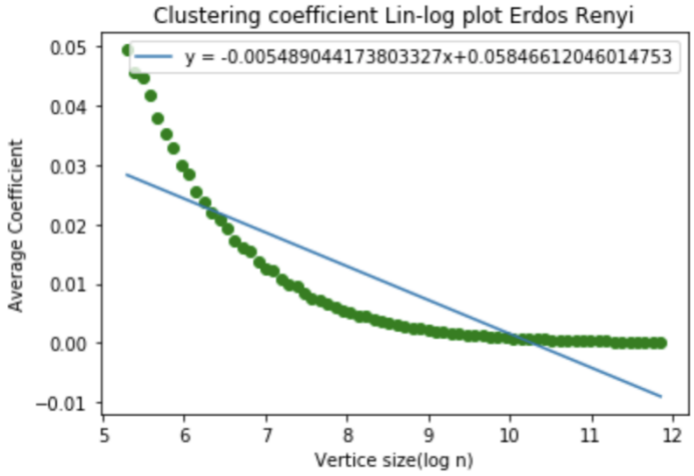
Plot analysis:

Diameter



The graph above shows the average diameter of graphs with node n generated by Erdos-Renyi algorithm. X-axis is node size starting from 100 and increasing by multiplying it with 1.1 with largest size 150000. The Y-axis is the average diameter of five random graph given size x. Also, the diameter is under the threshold 6 as we see from class. I plot the diameter-size function on a lin-log scale. The linear regression line under lin-log plot is : y = 0.0827907x + 4.11885. As we can tell from the plot, **the growth speed is slower than y = log n** (the y of regression line is always below log x).

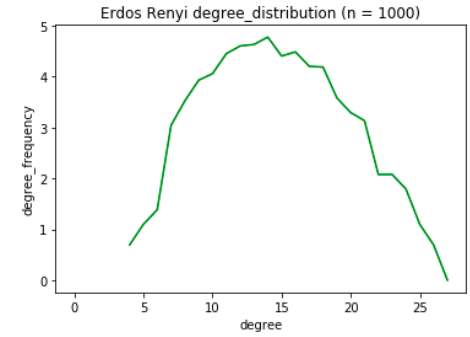
Clustering coefficient

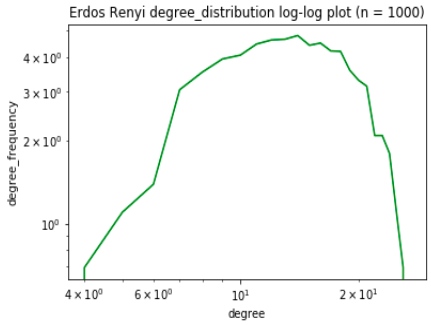


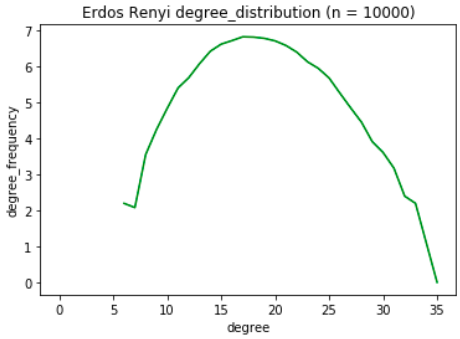
The graph above shows the clustering coefficient of graphs with node n generated by Erdos-Renyi algorithm. X-axis is node size starting from 100 and increasing by multiplying it with 1.1 with largest size 150000. The Y-axis is the average clustering coefficients of five random graph given size x. And I draw the plot on lin-log scale too. From the clustering coefficient graph, we can easily tell that the clustering coefficient is decreasing as node size becoming larger and larger. This makes sense since the number of edges generated by this algorithm is 2*n*ln n while the max number of edges of n nodes is n *(n-1)/2. The growing speed of y = n is much faster than ln n, which means the edges we get from the algorithm is much smaller than the number of possible edges in the graph(max number of edges). So the vertices are less likely to “cluster” toward each other. Since the clustering coefficient is decreasing as n grows, there is no need to compare it to log n, and it’s negative proportional to size n. I draw the regression line which is :

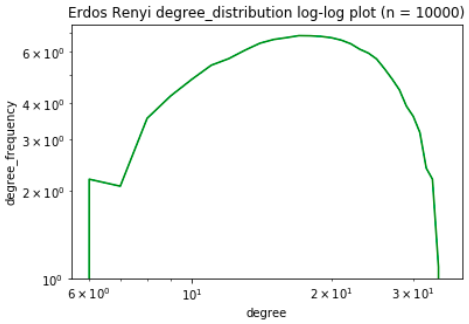
Y = -0.005489 x + 0.05846

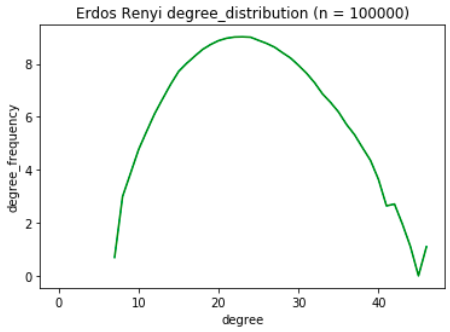
Degree distribution

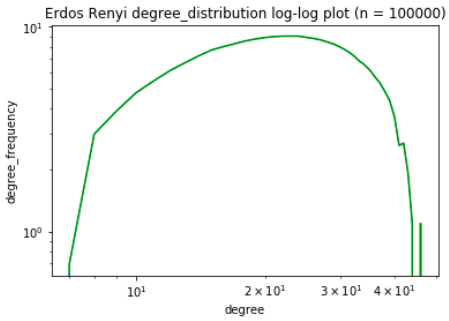












The graph above shows the degree distribution of graphs with node n generated by Erdos-Renyi algorithm. X-axis is degree of vertex . The Y-axis is the frequency of given degree in graph. I draw the plot with nodes size 1000, 10000, and 100,000 on both lin-lin scale and log-log scale. From the 6 graphs above, we can see that as the n grows, the plotting graphs are becoming smoother. In the linear/linear graphs, we can see it looks more like a normal distribution, which makes sense since the algorithm has equal probability for every possible pair of vertices to have an edge in between. **Also, there is no way to plot regression line for plot above, since it doesn't fit the power law.**

Barabasi-Albert Generation algorithm (extra credit)

Barabasi-Albert algorithm is an algorithm for generating random scale-free graph using a preferential attachment mechanism. The input is number of node and degree d, which indicates

new node connects to d other nodes selecting with probability proportional to their degree. N is the number of nodes in the graph.

In pseudo-code, we have the following algorithm

```
Input: number of vertices n, minimum degree d

M = array of length 2d
for v = 0 to n-1:
    for i =0 to d -1:
        M[2*(vd+i)]] = v
        draw number r uniformly at random in intervel [0,1,2,...,2*(vd+i)]
        M[2(vd+i)+1] = M[r]

for i =0 to nd-1:
    if edge is not duplicate or self  edge:
        add edge(M[2*i],M[2*i+1]) to the graph
```

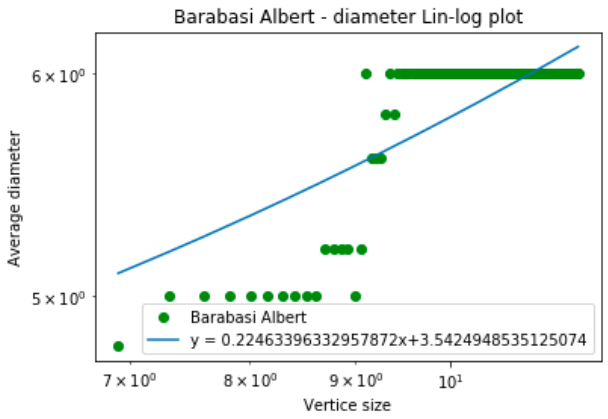
Algorithm analysis:

This algorithm uses preferential attachment technique. That is, at each steps, add new vertex v with one edge back to previous vertices. Probability a previously added vertex u receives the new edge from v is proportional to the degree of u. The implementation of algorithm is shown above in pseudo-code. Initialize an array of length d representing the vertices. Any two adjacent vertices will form an edge. When we try to add a new node, we draw an random number from zero to number of existing vertex in M. And form an edge according to M[r]. the connected vertices that is connected in store in M. the more edges one vertex have, the more likely new vertex will be connected to it. In the final part, check if there is duplicate edge or self-edge and insert other edges into graph.

Plot analysis

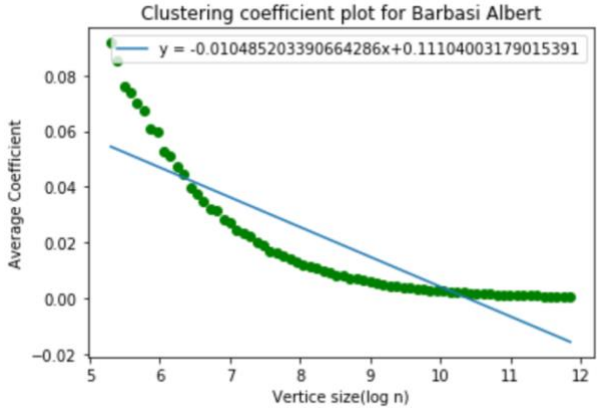
The way I plot below graph is the same as for previous plots, which means they have the same X-axis and Y-axis and correspond scale for Diameter, clustering coefficient and degree distribution respectively.

Diameter



The graph above shows the degree distribution of graphs with node n generated by Barabasi-Albert algorithm. As we can tell from the graph, the diameter grows super low as n size increase, which is obviously grows slower than function log n. the regression line is : y = 0.223633x + 3.5. As we can tell from the plot, **the growth speed is slower than y = log n** (the y of regression line is always below log x).

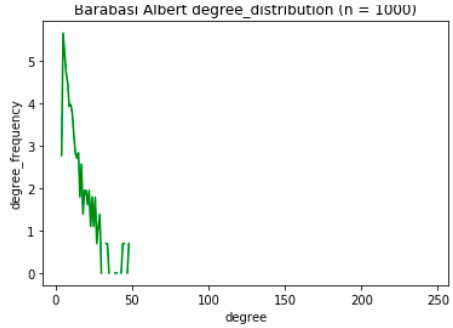
Clustering coefficient

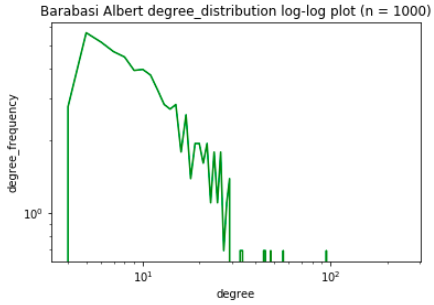


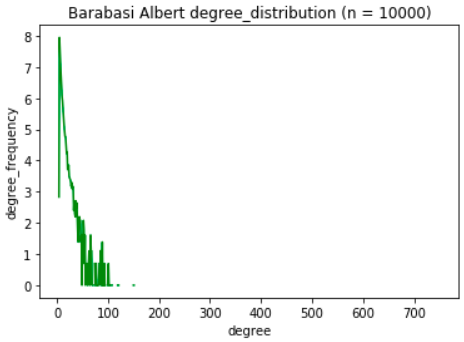
Just like the clustering coefficient graph of Erdo-Renyi algorithm, the clustering coefficient is becoming smaller and smaller as n grows. This is reasonable since number of edges generated by this graph is 5*n while the max number of possible edges is n(n-1)/2. The growing speed possible edges is much faster than ln n, which means the edges we get from the algorithm is much smaller than the number of possible edges in the graph(max number of edges). Since the clustering coefficient is decreasing as n grows, there is no need to compare it to log n. I plot the regression line which is:

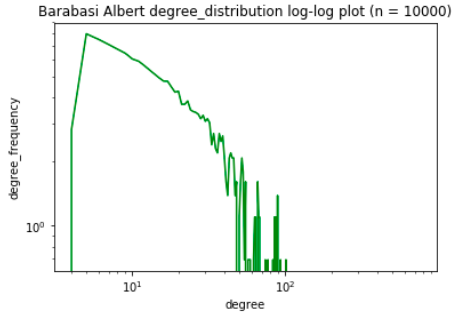
Y= -0.0104852x + 0.11104

Degree distribution

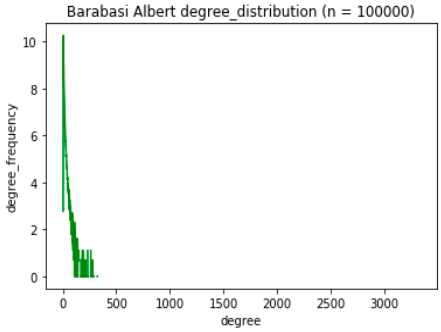


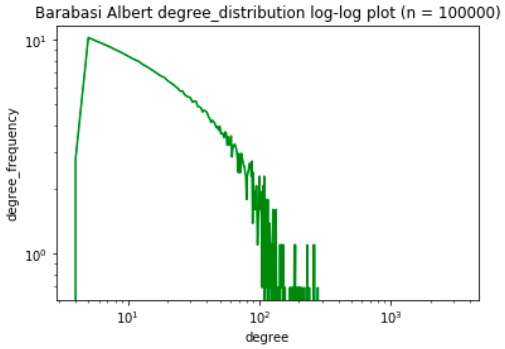






\





The graph above shows the degree distribution of graphs with node n generated by Erdos-Renyi algorithm. X-axis is degree of vertex . The Y-axis is the frequency of given degree in graph. I draw the plot with nodes size 1000, 10000, and 100,000 on both lin-lin scale and log-log scale. From the 6 graphs above, we can see that as the n grows, the plotting graphs are becoming smoother. In the linear/linear graphs, we can see it looks more like a normal distribution, which makes sense since the algorithm has equal probability for every possible pair of vertices to have an edge in between.

Also, there are regression lines for plot above, and as n grows, the slope of log-log regression line is closer to c = -3 as shown in class.