Adarsh B Shankar
ID: 11972805
CS 165 – Spring 2021

Project 3: Graph Algorithms

Overview:

Project 3 involves testing different graph algorithms experimentally to determine the properties of models of real-world networks. In this project we are going to test 3 different algorithms:

- Diameter Algorithm – Determine the length of longest path between any two vertices in the graph.
- Clustering-Coefficient – Determine the ratio of three times the number of triangles over the number of 2-edge paths.
- Degree-Distribution – For each possible degree, determine the number of vertices with that degree.

And we are going to test these algorithms on different of random graphs based on two different models, one is the Erdős–Rényi model and the other is the Babarási-Albert model.

Erdős – Rényi Model:

Description:

- G(n,p): - Graph G with n nodes, with every pair connected individually with probability p and average degree d = (n-1)*p~np

Implementation:

Pseudocode:

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 $w\geqslant v$ **and** $v<n$ **do**
    $w\leftarrow w-v;\ v\leftarrow v+1$
  if $v<n$ **then** $E\leftarrow E\cup\{v,w\}$

C++ implementation:

```
Graph create_ER_graph(int n) {

  Graph erdos_renyi = make_graph(n, vector < int > {}, vector < int > {});
  double p = (2 * (log(n))) / n;
  int v = 1;
  int w = -1;
  double r;

  while (v < n) {
    mt19937 seed2 = get_seed2();
    r = uniform_int_distribution < int > (0, 1)(seed2);
    int x = (floor((log(1 - r) / log(1 - p))));
    w = w + 1 + x;
    while (w >= v and v < n) {
      w = w - v;
      v = v + 1;
    }
    if (v <= n) {
      erdos_renyi.add_edge(v, w);
    }
  }

  return erdos_renyi;
}
```

Adarsh B Shankar
ID: 11972805
CS 165 – Spring 2021

Diameter:

Implementation (using Heuristic Idea 2):

- Use a random generator to get the start node

- Using BFS from the start, traverse the graph

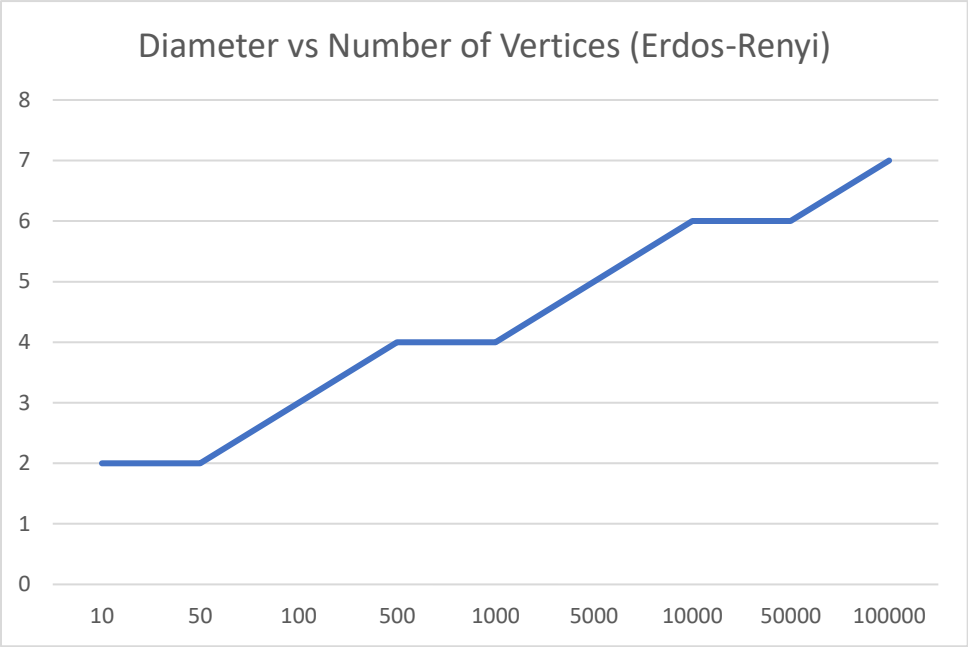
- BFS has been modified to return the node in the search for the maximum distance between that node and every other node in the graph

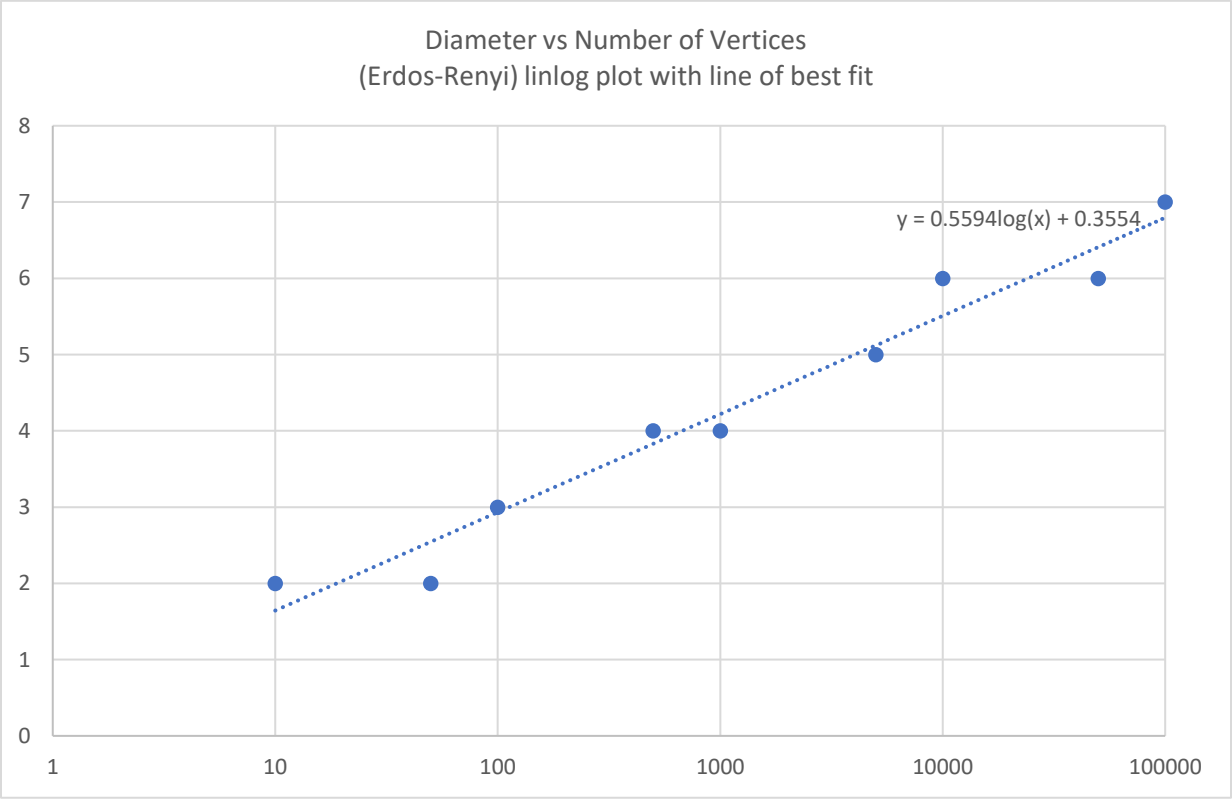
- Repeat the BFS till the node with the maximum distance has been found

C++ Code:

```
int get_diameter(Graph graph) {
  int size = graph.get_num_nodes();
  int diameter = 0;

  if (size > 1) {
    int max = 0;
    mt19937 seed = get_seed();
    int ind_start = get_generator(1, size)(seed);
    Node start(ind_start);
    tuple < Node, int > furthest = make_tuple(start, max);

    do {
      diameter = get < 1 > (furthest);
      furthest = breadth_first_search(graph, get < 0 > (furthest));
    } while (get < 1 > (furthest) > diameter);
  }

  return diameter;
}
```

Adarsh B Shankar
ID: 11972805
CS 165 – Spring 2021





From the above plots we can see that the diameter increases linearly as the number of vertices increased from 10 to 10^5. And from the lin-log plot we can say that the diameter: $f(d) = O(n)$ which is proportional to $\log n$.

Adarsh B Shankar
ID: 11972805
CS 165 – Spring 2021

Clustering Coefficient:

Implementation: - Return 3*(Number of triangles) / Number of 2 edge paths

Counting Triangles implementation:

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
3. Initialize an array D such that $D[i]$ contains a list of the vertices v that ar 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 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

- Compute a d-degeneracy ordering of the vertices, e.g., using the algorithm of the previous slide.
- Process the vertices according to this ordering, L:

  For each vertex, v:

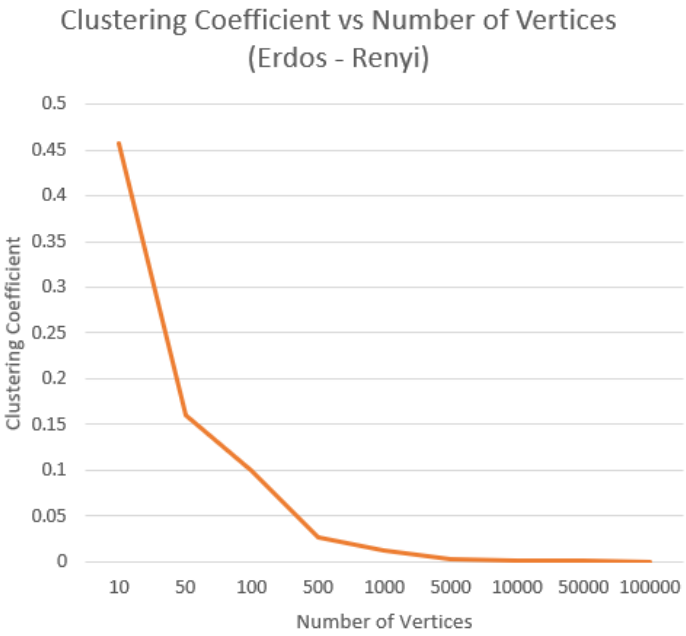
  For each pair of vertices, u and w, adjacent to v and earlier in the ordering, i.e., u and v are in the list N_v from the degeneracy algorithm:

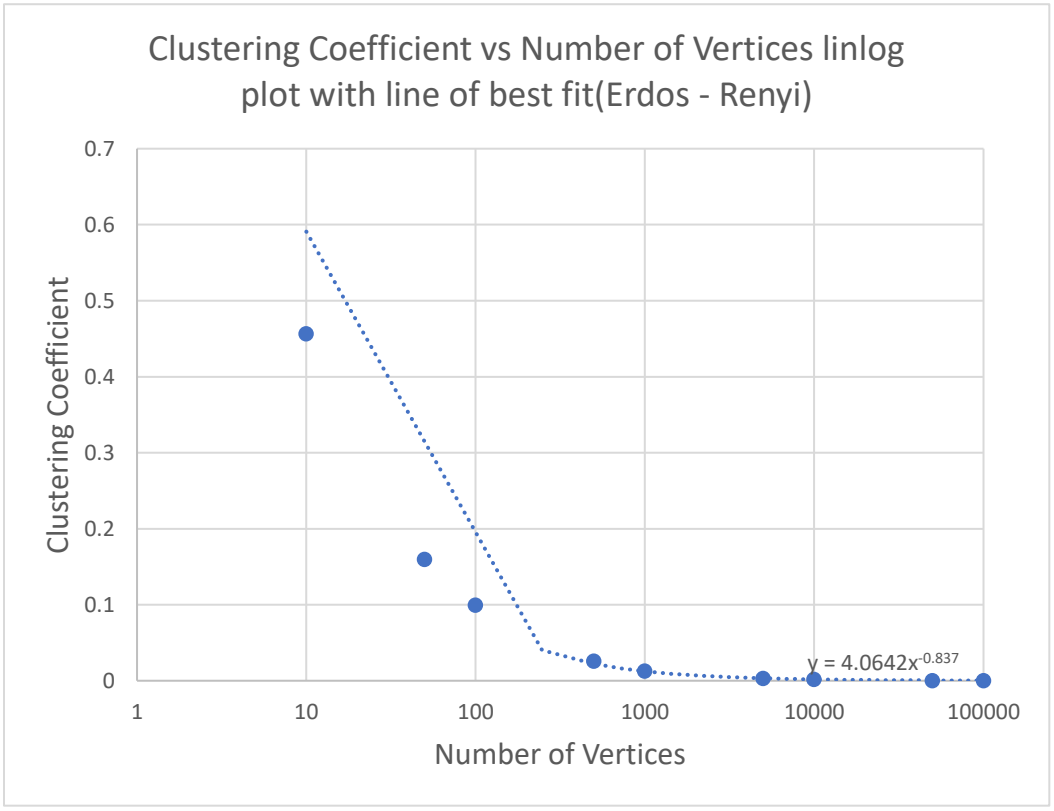
  If (u,w) is an edge in the graph, then add one to the triangle count.

C++ Code:

```
int count_triangles(Graph g) {
  int triangles = 0;
  int degeneracy = 0;
  int num_nodes = g.get_num_nodes();

  int deg[num_nodes];
  vector < int > N[num_nodes];

  stack < int > L; //Initialize an output list, L, to be empty.
  map < int, unordered_set < int >> D;

  //Initialize an array D such that D[i] contains a list of the vertices v that are not already in L for which dv
  for (int i = 1; i <= num_nodes; ++i) {
    Node n(i);
    int degree = g.get_neighbors(n).size();
    D[degree].insert(i);
    deg[i - 1] = degree;
  }
  //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.,
  unordered_set < int > H;
  for (int ind = 0; ind < num_nodes; ind++) {
    int min = -1;
    for (auto j = D.begin(); j != D.end(); j++) {
      if (!j -> second.empty()) {
        min = j -> first;
        break;
      }
    }
    unordered_set < int > nlist = D[min];
    degeneracy = max(degeneracy, min);

    int nid = * nlist.begin();
    L.push(nid);
    H.insert(nid);

    D[min].erase(nid);
    vector < Node > nblist = g.get_neighbors(Node(nid));
    /*
    - For each neighbor w of v not already in L (you can check this using HL):
    * Subtract one from dw
    * Move w to the cell of D corresponding to the new value of dw , i.e., D[dw]
    * Add w to Nv
    */
    for (int x = 0; x < nblist.size(); x++) {
      Node nxt = nblist[x];
      int nodeID = nxt.id;
      if (H.find(nodeID) == H.end()) {
        int cur = deg[nodeID - 1]--;
        D[cur].erase(nodeID);
        D[cur - 1].insert(nodeID);
        N[nid - 1].push_back(nodeID);
      }
    }
  }

  //Count the traingles
  while (!L.empty()) {
    int top = L.top();
    vector < int > nodes = N[top - 1];
    int num_nodes2 = nodes.size();
    for (int a = 0; a < num_nodes2 - 1; a++) {
      for (int b = a + 1; b < num_nodes2; b++) {
        if (g.is_adjacent(Node(nodes[a]), Node(nodes[b]))) {
          triangles++;
        }
      }
    }
    L.pop();
  }

  return triangles;
```

Adarsh B Shankar
ID: 11972805
CS 165 – Spring 2021





From the above plots we can see with the equation $y = 4.0642x^{-0.837}$ that the clustering coefficient decreases proportionally as the number of vertices increased from 10 to 10^5 so we can say that the clustering coefficient: $f(C) = O(1/n)$ which is proportional to $\log n$.

Adarsh B Shankar
ID: 11972805
CS 165 – Spring 2021

Degree Distribution:

Implementation:

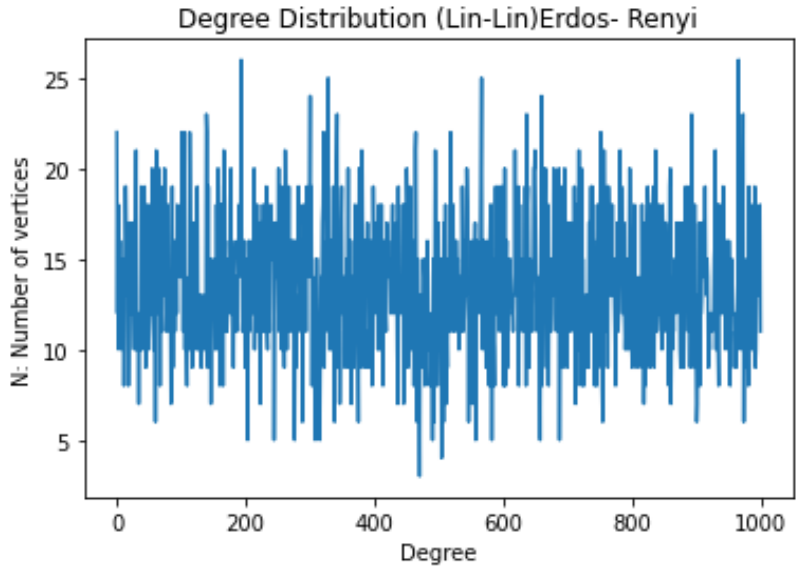
```
def degree_distribution(G):
        map = new HashTable
        for each vertex v in G:
                map[v.degree]++
return map
```

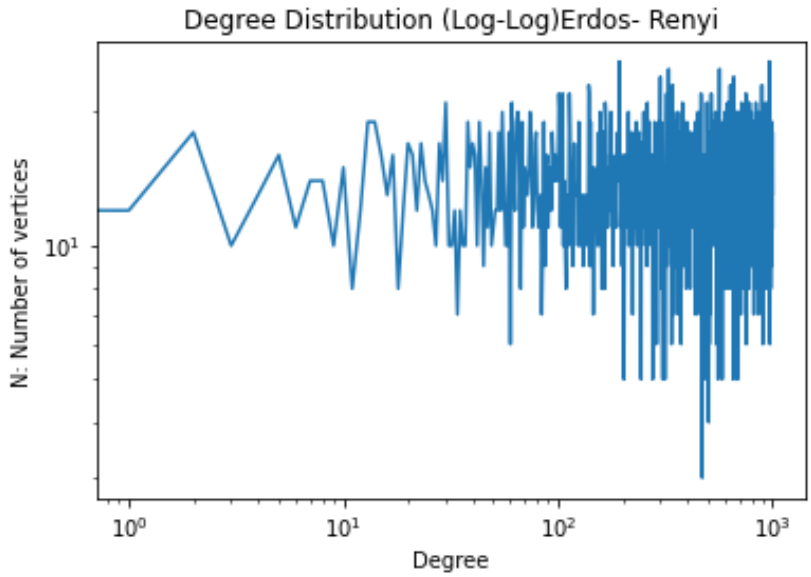
C++ code:

```
std::map < int, int > get_degree_distribution(Graph graph) {
  map < int, int > deg_dist;
  int size = graph.get_num_nodes();
  int degree = 0;

  for (int i = 1; i <= size; ++i) {
    Node n(i);
    degree = graph.get_neighbors(n).size();
    deg_dist[degree] = deg_dist[degree] + 1;
  }
  return deg_dist;
}
```

Plotting distribution for different sizes ($N: number\ of\ vertices$):

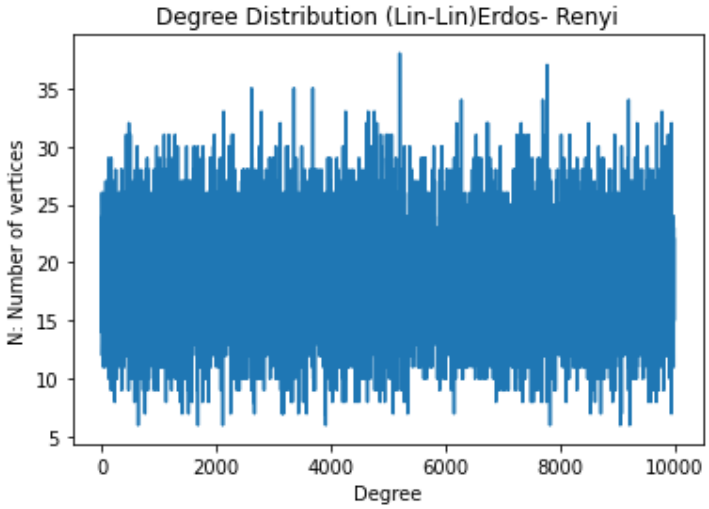
- $N = 1000$

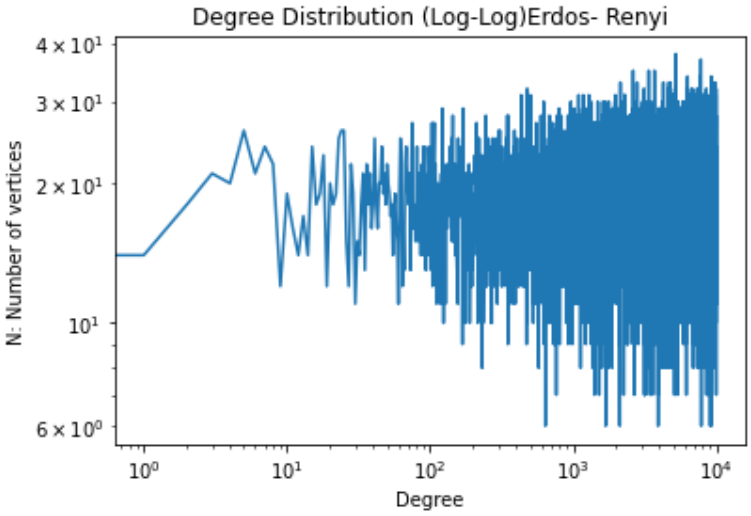




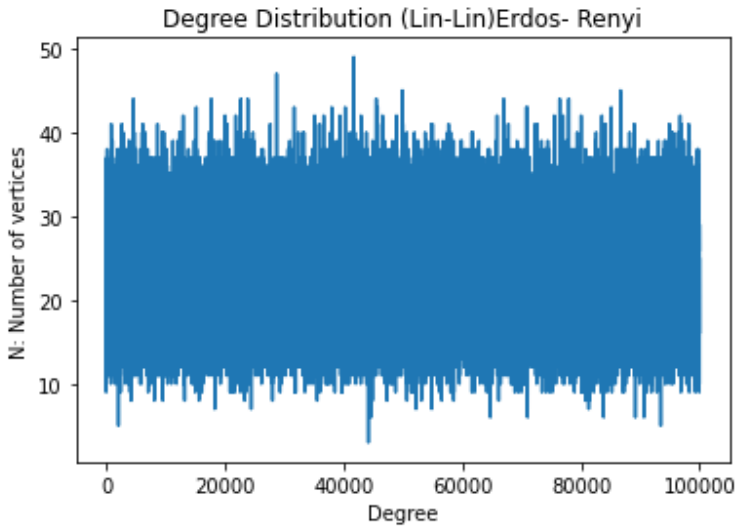
Adarsh B Shankar
ID: 11972805
CS 165 – Spring 2021

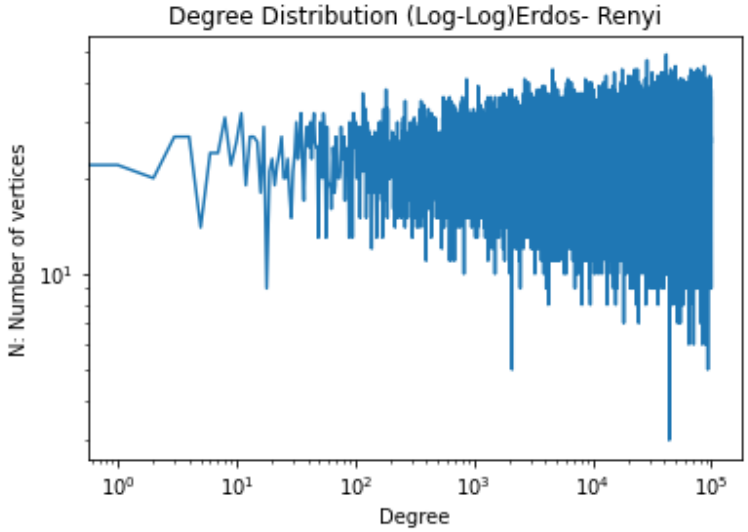
- $N = 10{,}000$





- $N = 100{,}000$





Degree Distribution – Analysis:

As we can see the above graphs do not have a heavy/long tail, hence we can conclude that these distributions <u>do not exhibit the power law</u>.

Adarsh B Shankar
ID: 11972805
CS 165 – Spring 2021

Barabási – Albert Model:

Description:

- G(n,d): - Graph G with n nodes, with minimum degree d ≥ 1 with each new node being connected with a probability = $^{d_i}/_{\sum_i d_i}$ $(here \rightarrow d = 5\ (\text{const}))$

Implementation:

Pseudocode:

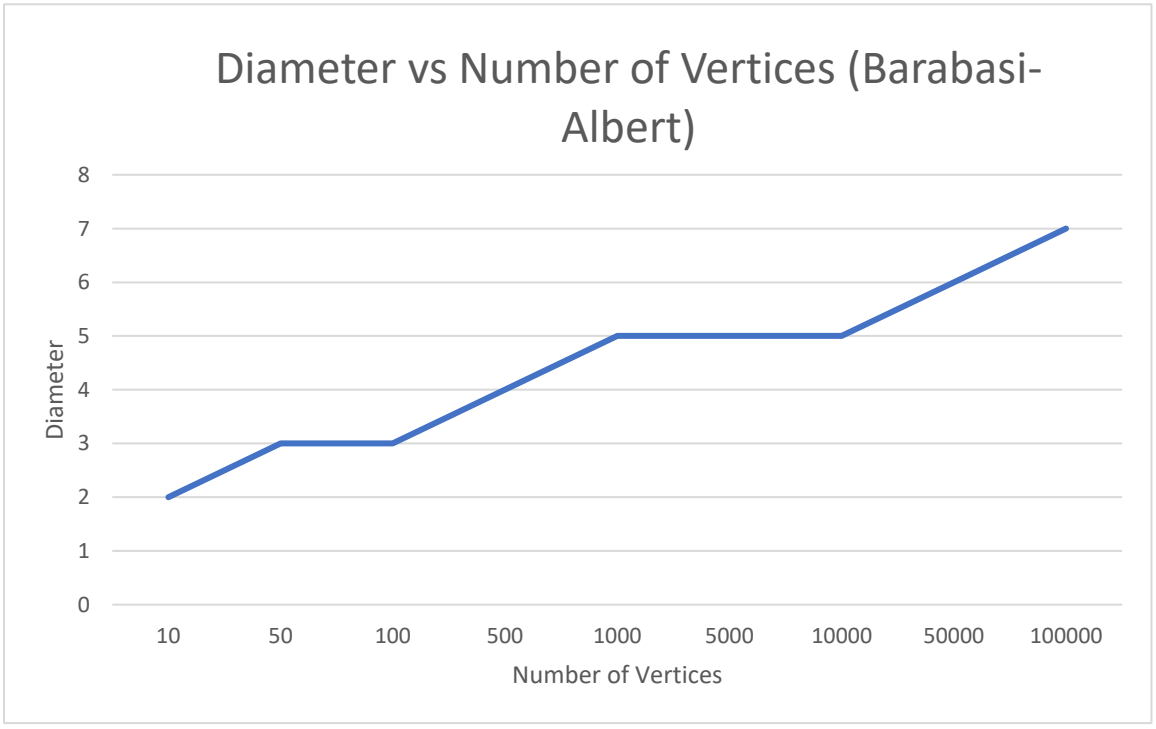
ALG. 5: preferential attachment
Input: number of vertices n
minimum degree $d \geqslant 1$
Output: scale-free multigraph
$G=(\{0,\dots,n-1\},E)$
M: array of length $2nd$
for $v=0,\dots,n-1$ **do**
  for $i=0,\dots,d-1$ **do**
    $M[2(vd+i)] \leftarrow v$
    draw $r \in \{0,\dots,2(vd+i)\}$ uniformly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

C++ Code:

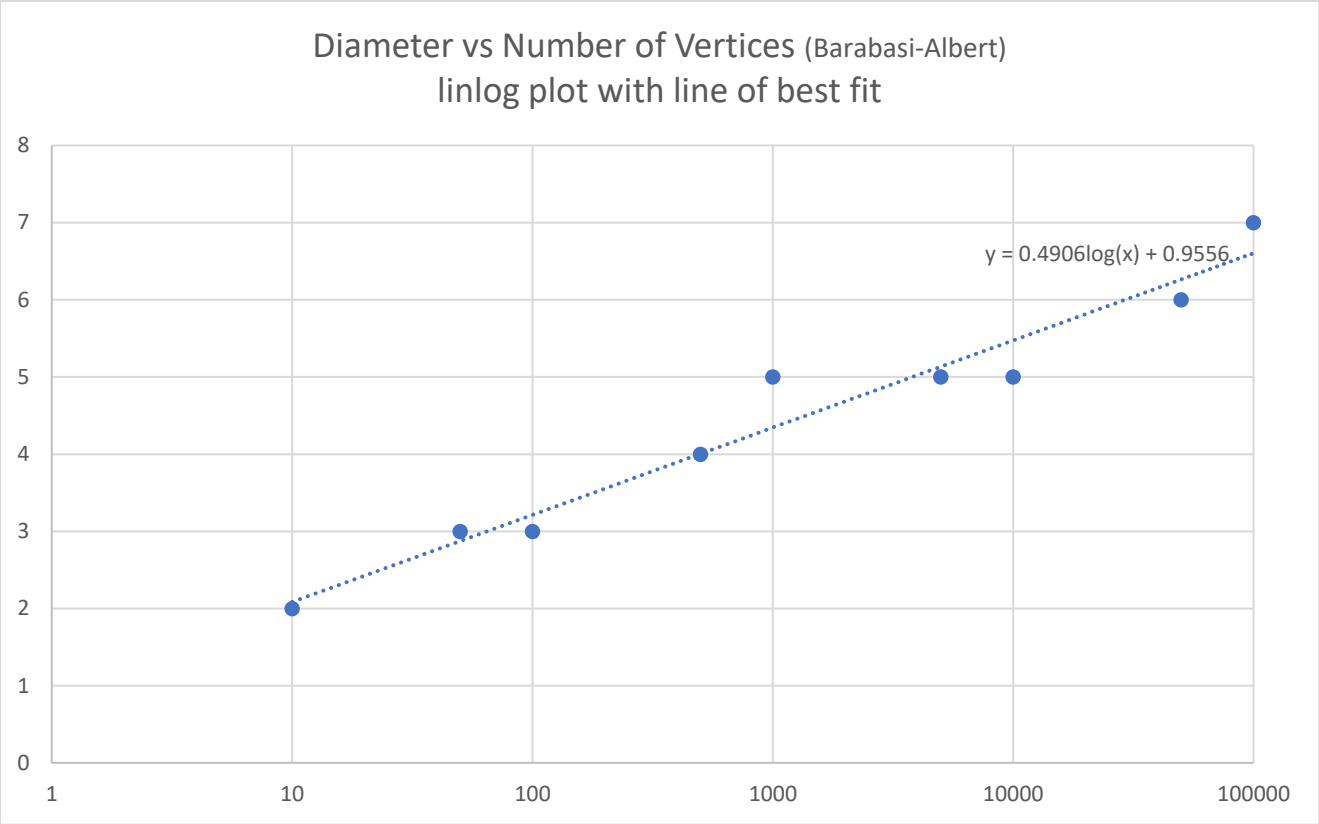
```
Graph create_BA_graph(int nodes, int degree) {

  Graph barabasi_albert = make_graph(nodes, vector < int > {}, vector < int > {});
  int size = 2 * nodes * degree;
  int L[size];
  set < tuple < int, int >> edges;

  for (int i = 0; i < nodes; i++) {
    for (int j = 0; j < degree; j++) {
      mt19937 seed = mt19937(chrono::system_clock::now().time_since_epoch().count());
      L[2 * (i * degree + j)] = i;
      int p = uniform_int_distribution < int > (0, 2 * (i * degree + j))(seed);
      L[2 * (i * degree + j) + 1] = L[p];
    }
  }

  for (int x = 0; x < nodes * degree; x++) {
    int nid1 = L[2 * x] + 1;
    int nid2 = L[2 * x + 1] + 1;
    tuple < int, int > edge = make_tuple(nid1, nid2);
    tuple < int, int > edge2 = make_tuple(nid2, nid1);
    if (nid1 != nid2 && edges.find(edge) == edges.end()) {
      barabasi_albert.add_edge(nid1, nid2);
      edges.insert(edge);
      edges.insert(edge2);
    }
  }

  return barabasi_albert;
}
```

Diameter:

(Implementation and Code on Page 2)



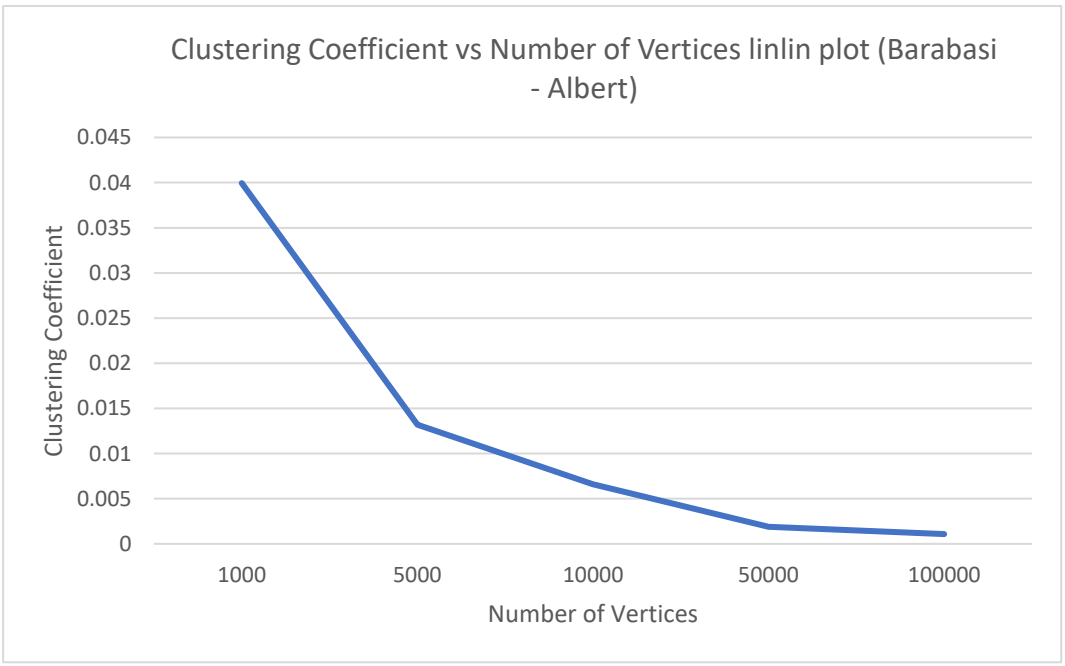
Adarsh B Shankar
ID: 11972805
CS 165 – Spring 2021



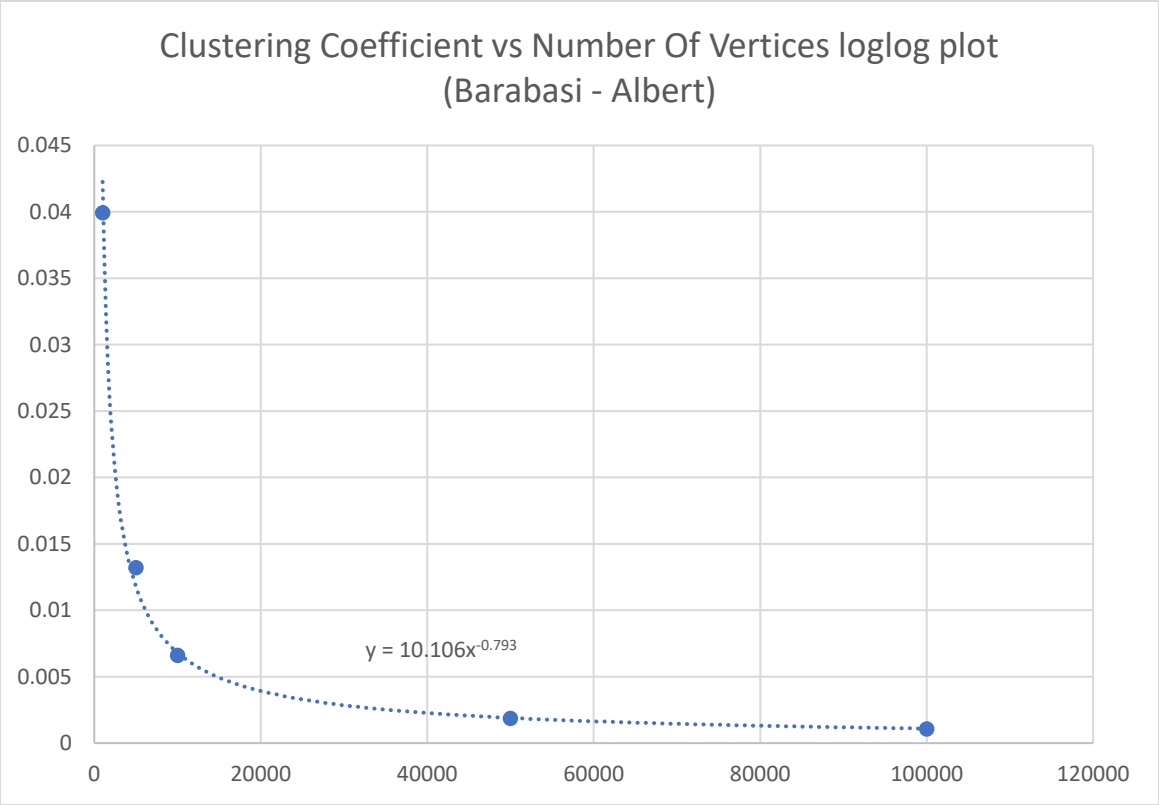
From the above plots we can see that the diameter increases linearly as the number of vertices increased from 10 to 10^5. And from the lin-log plot we can say that the diameter: $f(d) = O(n)$ which is proportional to $\log n$.

Clustering Coefficient:

(Implementation and Code on Page 4)



Adarsh B Shankar
ID: 11972805
CS 165 – Spring 2021



From the above plots we can see with the equation $y = 10.1062x^{-0.793}$ that the clustering coefficient decreases proportionally as the number of vertices increased from 10 to 10^5 so we can say that the clustering coefficient: $f(C) = O(1/n)$ which is proportional to $\log 1/n$.

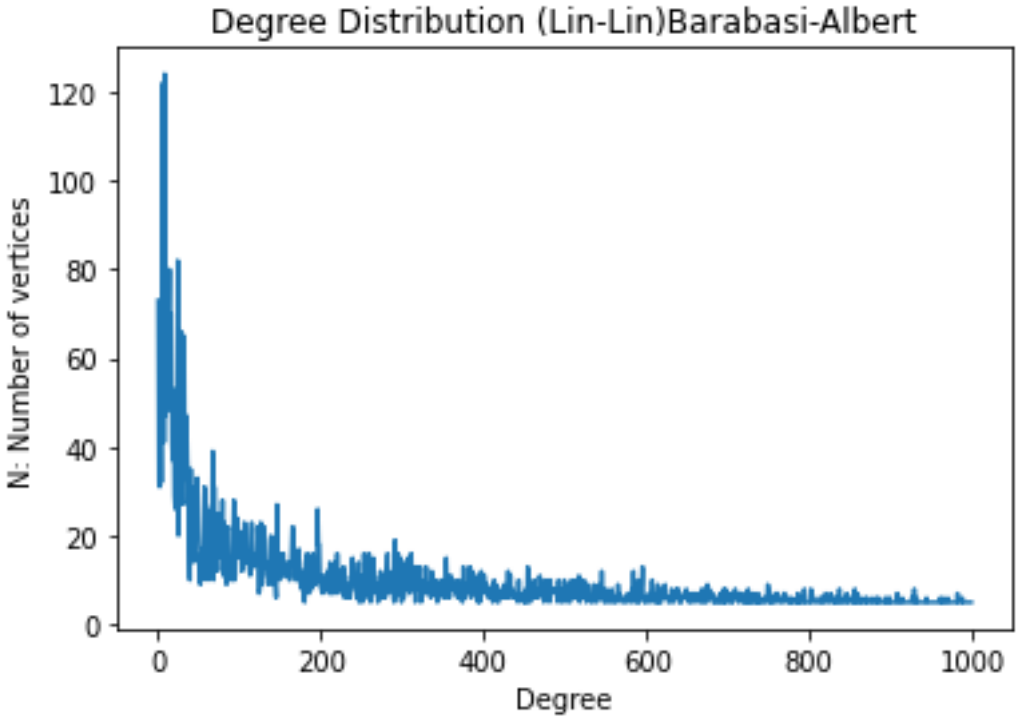
Adarsh B Shankar
ID: 11972805
CS 165 – Spring 2021

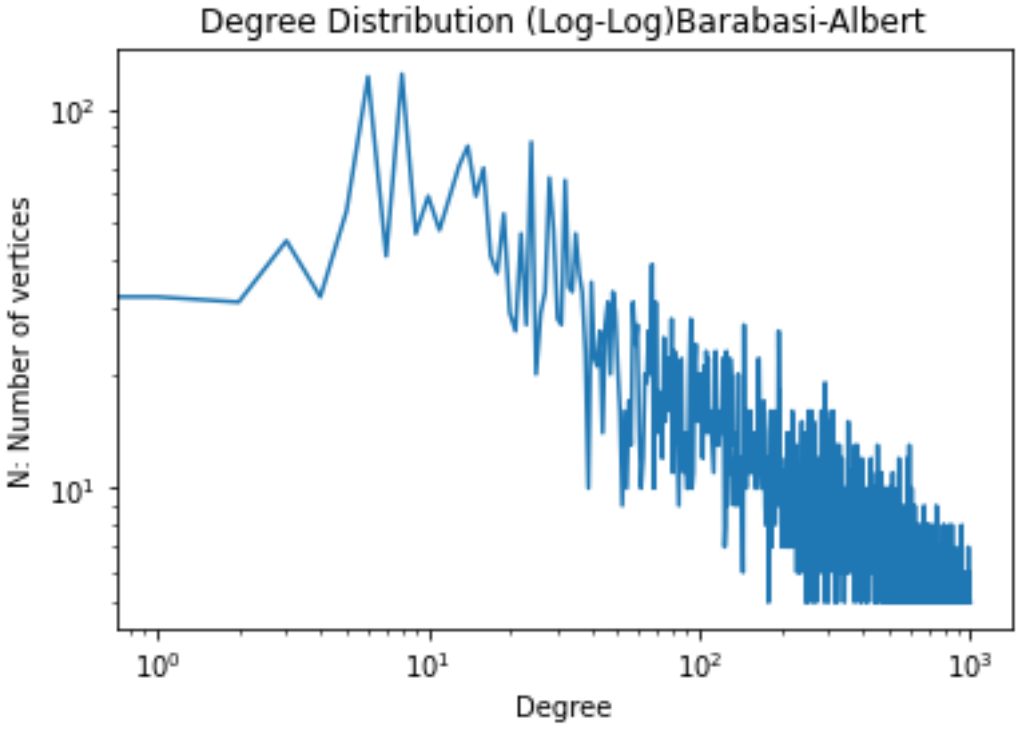
Degree Distribution:

(Implementation and Code on Page 6)

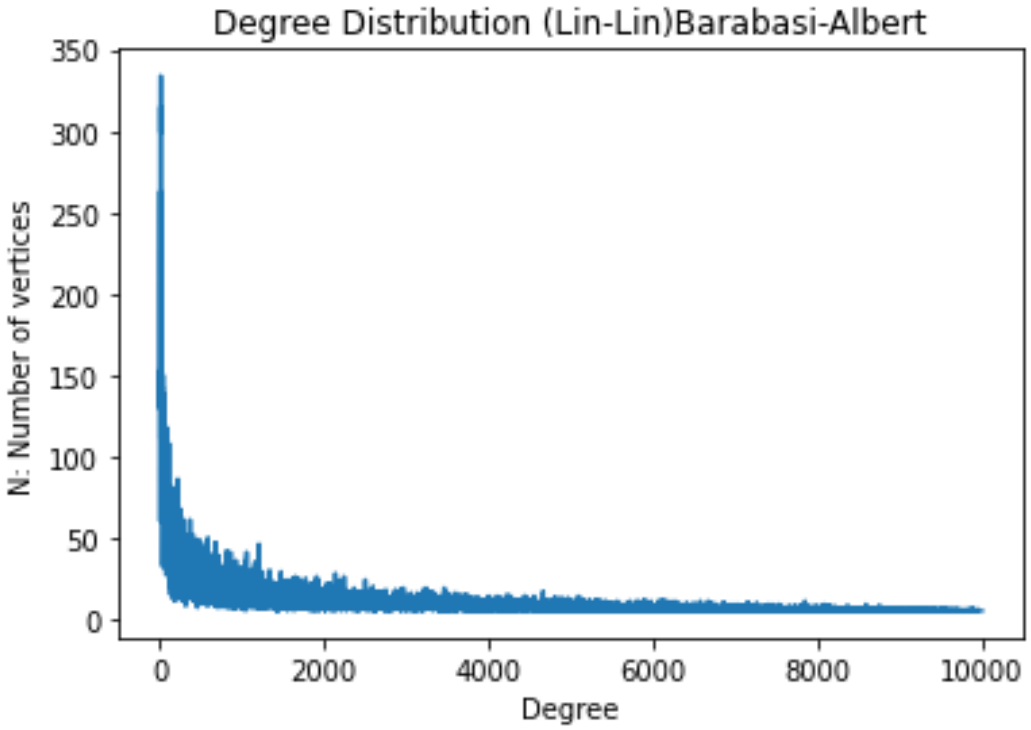
Plotting distribution for different sizes ($N: number\ of\ vertices$):

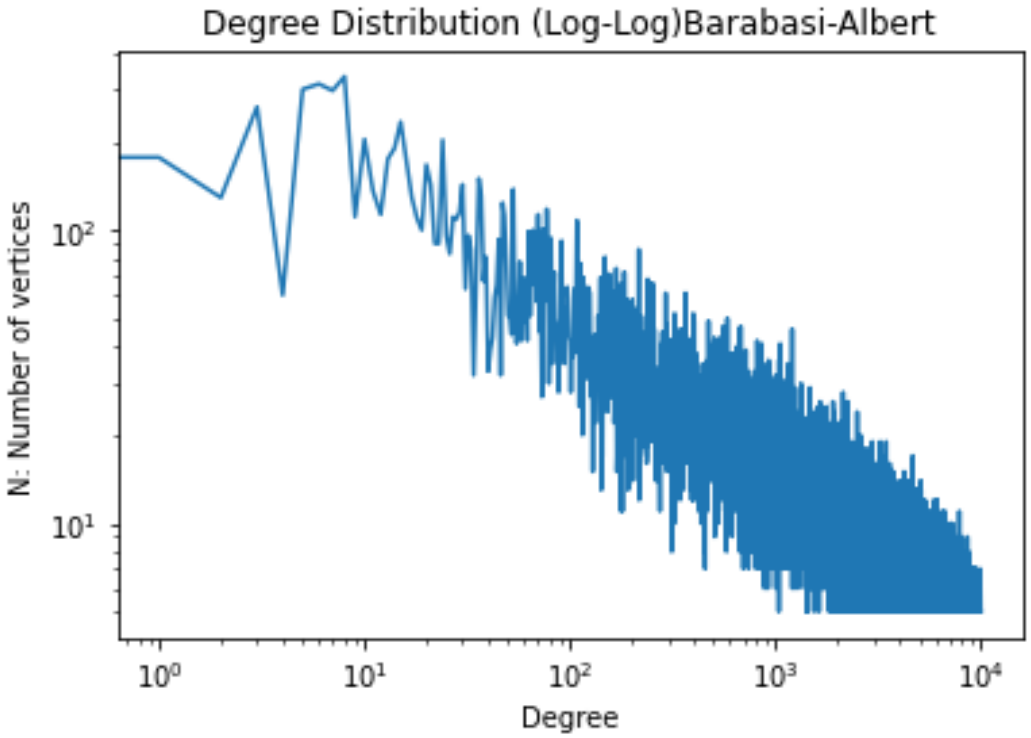
- $N = 1000$



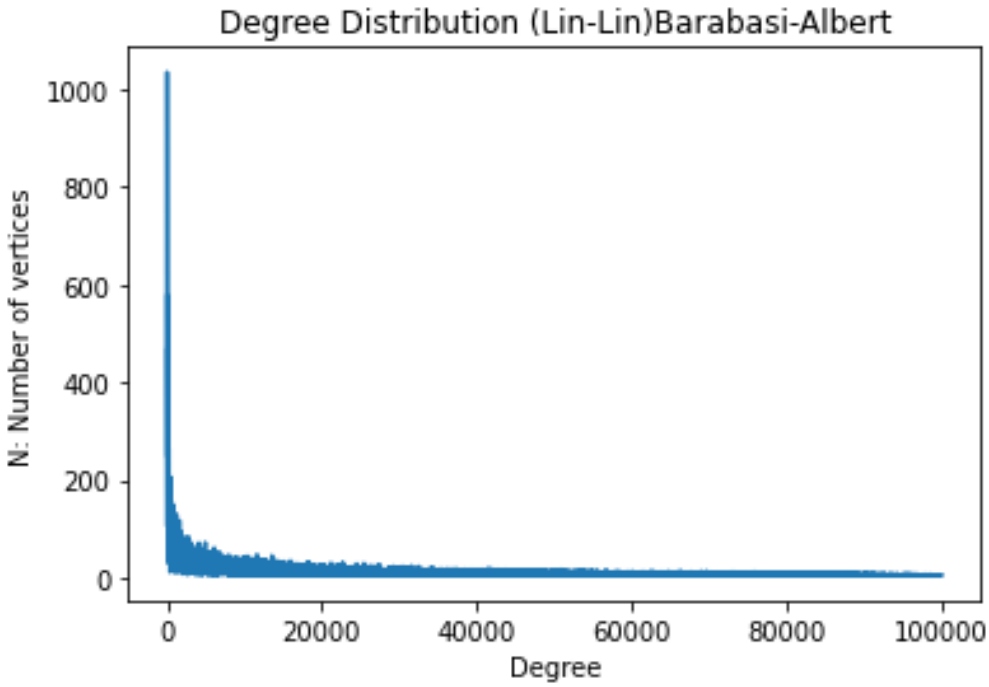


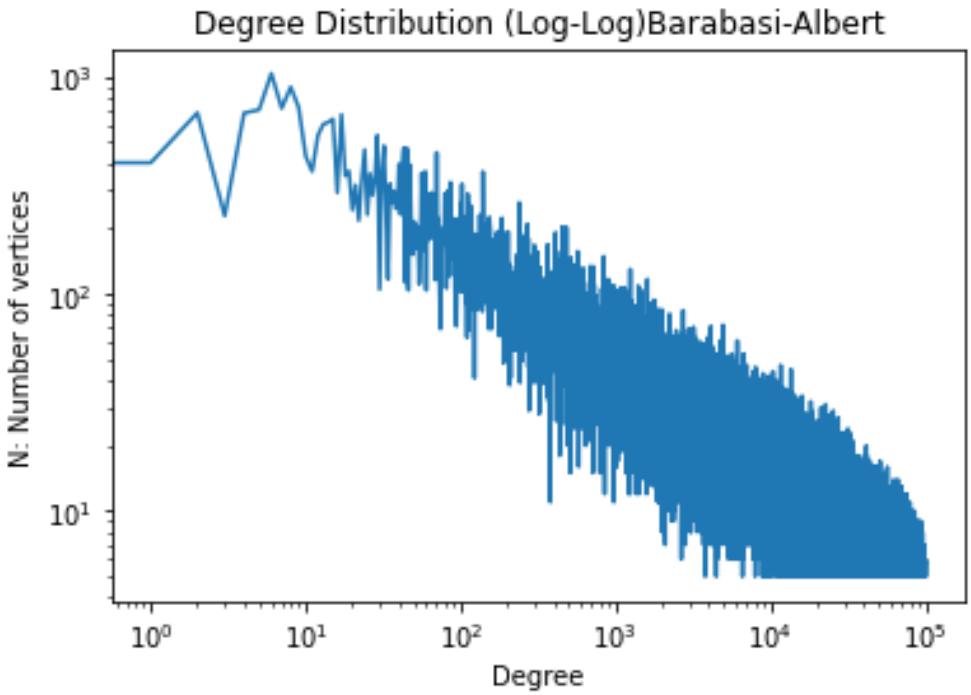
- $N = 10{,}000$


Degree Distribution (Lin-Lin)Barabasi-Albert
N: Number of vertices
Degree
0
50
100
150
200
250
300
350
0
2000
4000
6000
8000
10000


Degree Distribution (Log-Log)Barabasi-Albert
N: Number of vertices
Degree
10^1
10^2
10^0
10^1
10^2
10^3
10^4

- $N = 100{,}000$



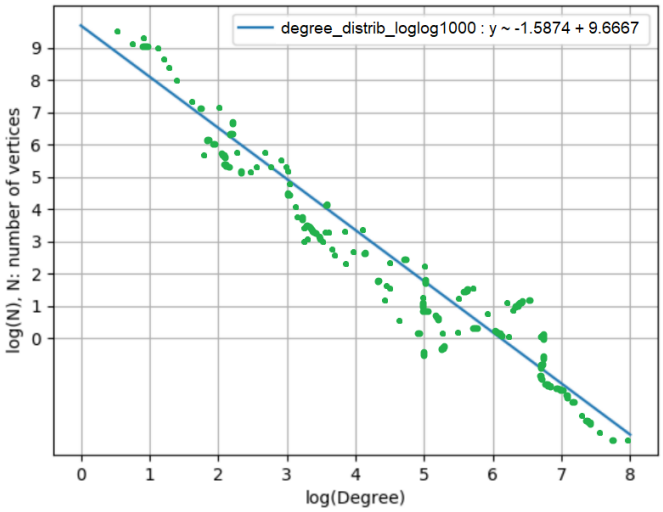


The above graphs show that the degree distribution for Barabasi-Albert models do exhibit a power law unlike the degree distributions for Erdos-Renyi models so we can perform a regression analysis on this model and find the line of best fit.

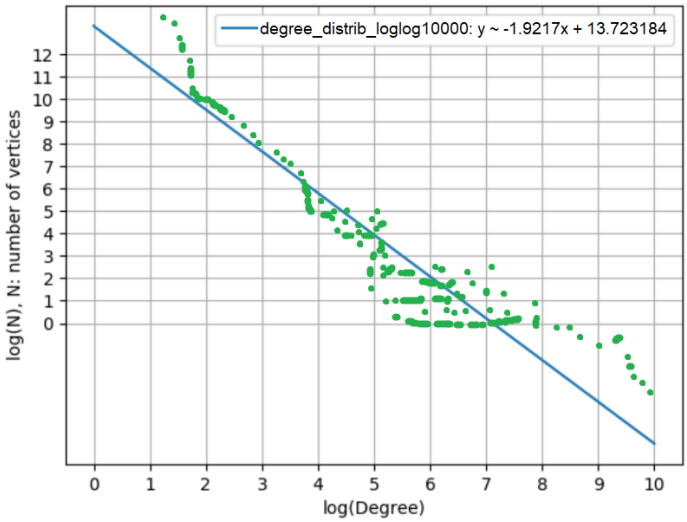
Adarsh B Shankar
ID: 11972805
CS 165 – Spring 2021

Regression Analysis for Barabasi Albert models only:

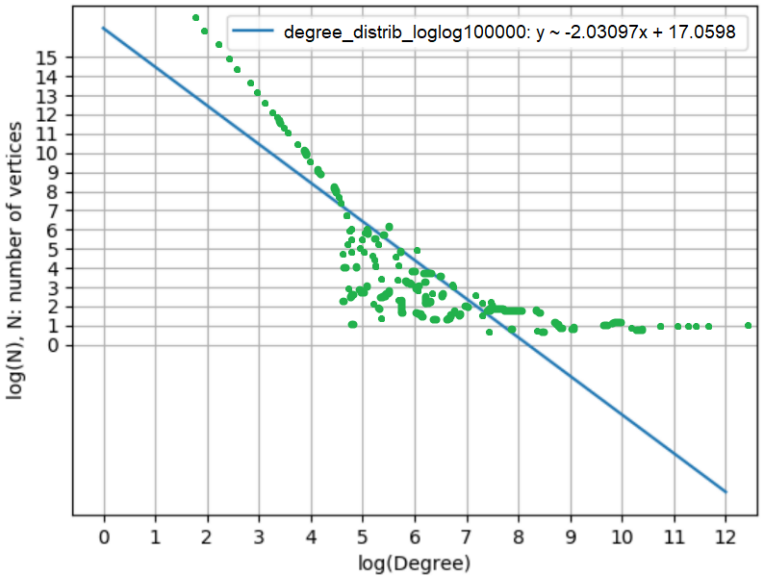
- $N = 1000$



- $N = 10{,}000$



- $N = 100{,}000$



Results:

The above graphs (for the number of vertices = 1000,10000,100000) clearly exhibit a power law since the log-log plots exhibit a heavy/long tail. And after doing a regression analysis we find that the exponents of the power law (slopes) of the best fitting lines are: $-1.59, -1.92, -2.03$ respectively.

Conclusion:

After experimenting the three algorithms on both models, we can conclude the following:

- Diameter $\propto \log(n)$
- Clustering Coefficient $\propto \log(1/n)$
- Degree Distribution $\propto n$

Therefore, we can say that all three properties of the graph are in some way proportional to the number of nodes in the graph when using either graph model.

Erdős – Rényi Model:

When experimenting the degree distribution on Erdős – Rényi models, we see that none of the graphs exhibit a heavy/long tail, so they do not have a power law as they follow what looks like Poisson distribution. This is due to the probability for each node being added to the edge. This probability does not model to any real-world network. Thus, if we were to do a study on real-world networks the Erdős – Rényi model would not be preferred for the experiments. But because the model is so simple and efficient, it is a good model to test the functionality of graph algorithms when implementing them.

Barabási – Albert Model:

The Barabási – Albert models, on the other hand, do exhibit a long/heavy tail in the degree distribution graphs so we can conclude that they do have a power law. This is due to the preferential attachment algorithm, vertices with higher degrees are likely to get more edges connected. This models closer to real-world networks such as social or communication networks where the more connections/friends you have (degree) the likelier you are to make more which will generally lead to more heavy-tailed distributions. Thus, if we were to do a study on real-world networks the Barabási – Albert model would be preferred for the experiments. Although both model algorithms run in $O(n+m)$ time, this model is more complex than the Erdős – Rényi model because we're calculating the probability by summing over the degrees of the remaining nodes for ever vertex in the graph.