Assignment 2 Q3

March 22, 2021

0.1 Q.3.

Modify the Barabasi-Albert algorithm to accentuate/strengthen the bias of rich getting richer phenomenon such that the probability of a newly added node getting connected to an existing node is now "proportional to the square of its degree". Compute and compare topological features a comparable size of networks created using BA algorithm. Create variants of higher order.

0.2 Libraries Import

[1]:
```
import math
import random
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from collections import Counter
```

[2]:
```
"""
Function         : createInitialGraph
Input Parameters : Initial number of nodes, m0
Purpose          : Create an initial graph with m0 nodes each connected with␣
 ↪atleast one edge
Returns          : A Graph containing m0 nodes each connected to each other␣
 ↪using atleast one edge
"""

def createInitialGraph(m0, total_edges):
    node_list = [item for item in range(0, m0)]          # To define the nodes
    G = nx.Graph()                                       # Defines a new␣
 ↪instance of the graph
    G.add_nodes_from(node_list)                          # The nodes defined␣
 ↪are added to the graph without any edges as of now

    edges_added = 0                                      # A variable to keep␣
 ↪track of the number of edges added
    while edges_added < total_edges:                     # Run the loop until␣
 ↪the defined number of edges is not made
        for node in node_list:                               # For each of the␣
 ↪node present in the graph
```

```
            added_flag = False                              # Initially there␣
↪is no edge from this node, therefore added_flag is False
            while not added_flag:                           # Until the edge␣
↪from current node is added into the graph
                random_vertex = random.randint(0, m0-1)     # Pick a random␣
↪vertex to which the current node will be connected to
                if random_vertex != node:                   # Ensure that the␣
↪random vertex selected is not the same as the current node
                    if not G.has_edge(random_vertex, node): # Also ensure that␣
↪these two nodes are not connected before
                        G.add_edge(random_vertex, node)     # If above␣
↪conditions are satisfied, then add the edge between these nodes
                        added_flag = True                   # Set the flag to␣
↪True
                        edges_added += 1                    # Increment the␣
↪edge count
    return G                                            # Returns the Graph␣
↪thus obtained
```

[3]:

```
"""
Function         : getRandomNode
Input Parameters : Graph, degree list of the Graph and order of variant(i.e.␣
↪power)
Purpose          : Finds the node whose probability is greater than a random␣
↪number
Returns          : Returns the node whose probability is greater than a random␣
↪number
"""

def getRandomNode(G, degree_list, power):
    node_list = G.nodes()                                   # Gets the list␣
↪of nodes
    degree_dict = dict(degree_list)                         # Convert the␣
↪degree list into a dictionary
    denom_sum = 0                                           # Initialize the␣
↪variable denominator sum
    for value in degree_dict.values():                      # For each value␣
↪contained in the degree dictionary
        powered_value = int(math.pow(value, power))         # Power the␣
↪value
        denom_sum += powered_value                          # Add to␣
↪denominator sum

    probability_list = []                                   # Declare␣
↪probability list
```

```
    cum_prob = []                                                        # Declare␣
↪Cumulative probability list
    prev_prob = 0                                                        # Initialize␣
↪cumulative probability value

    for node in G.nodes():                                               # For each node␣
↪in the Graph
        degree_node = G.degree(node)                                     # Get the degree␣
↪of the node in the iteration
        powered_degree = int(math.pow(degree_node, power))               # Calculate the␣
↪numerator value for higher order
        prob = powered_degree/denom_sum                                  # Calculate the␣
↪probability value
        prev_prob += prob                                                # Add to the␣
↪cumulative probability
        probability_list.append(prob)                                    # Append the␣
↪probability value to the probability list
        cum_prob.append(prev_prob)                                       # Store the␣
↪cumulative probability in the cum_prob list

    random_num = random.uniform(0, 1)                                    # Generate a␣
↪random number between 0 and 1
    for i in range(len(cum_prob)):                                       # For each value␣
↪present in the cum_prob list
        if cum_prob[i] >= random_num:                                    # If the random␣
↪number selected is less than the current cumulative probability value
            return i                                                     # Then return␣
↪the current node
```

[4]:

```
"""
Function         : barabasiModel
Input Parameters : Graph(G), m(Number of edges to be drawn), m0(Initial number␣
↪of nodes), n(Total number of nodes), Power(Order of Variant)
Purpose          : Connects the new nodes to m previously added nodes and␣
↪generate a graph in accordance with Barabasi-Albert Model
Returns          : A Graph which satisfies the Barabasi-Albert Model
"""

def barabasiModel(G, m, m0, n, power):
    for i in range(m0, n):                                                   # For each␣
↪of the nodes left after making the initial graph
        G.add_node(i)                                                        # Add the␣
↪node into the graph
        degree_list = nx.degree(G)                                           # Find the␣
↪degree_list of the graph
```

```
        num_edges = 0                                                        # Counter to␣
↪keep track of number of edges
        while num_edges < m:                                                 # Loop until␣
↪the number of edges is not equal to the count of edges to be connected to
            random_vertex = getRandomNode(G, degree_list, power)    # Get the␣
↪node to connect to
            if (i, random_vertex) not in G.edges():                          # If the␣
↪edge is not already present in the graph
                G.add_edge(i, random_vertex)                                 # Add the␣
↪edge into the graph
                num_edges += 1                                               # Increase␣
↪the count of number of edges
            else:
                pass
    return G                                                                 # Return the␣
↪graph which satisfies the Barabasi-Albert model
```

[5]:

```
"""
Function          : getDegreeDist
Input Parameters : Graph(G), degree distribution dictionary
Purpose           : To add the degree distribution values of the current graph␣
↪into the degree distribution dictionary already present
Returns           : The degree distribution dictionary with the degree␣
↪distribution of current graph added to it.
"""
def getDegreeDist(G, degree_distribution_dict):
    degree_list = nx.degree(G)                                              # Get the␣
↪degree of the nodes in the graph(list of tuples)
    degree_dict = dict(degree_list)                                         # Convert the␣
↪list of tuples obtained into a dictionary
    degree_dist = Counter(degree_dict.values())                             # Find the␣
↪frequency of occurence of each of the degree

    for x, y in degree_dist.items():                                        # Loop for␣
↪each of the degree-frequency pair
        if x in degree_distribution_dict.keys():                            # If that␣
↪degree is present already
            degree_distribution_dict[x].append(degree_dist[x])      # Append that␣
↪to already existing list
        else:
            degree_distribution_dict[x] = []                                # Else create␣
↪new list
            degree_distribution_dict[x].append(degree_dist[x])      # Store the␣
↪value into the new list corresponding to the degree
    return degree_distribution_dict                                         # Returns the␣
↪degree distribution dictionary
```

[6]:
```python
"""
Function         : getMeanStd
Input Parameters : degree distribution dictionary and number of nodes in the␣
 ↪graph
Purpose          : To get the mean dictionary, std deviation dictionary and␣
 ↪their corresponding lists
Returns          : The mean dictionary, std deviation dictionary and their␣
 ↪corresponding lists.
"""
def getMeanStd(degree_distribution_dict, num_nodes):
    mean_dict = {}
    std_dev_dict = {}
    for x, y in degree_distribution_dict.items():
        mean = np.mean(y)/num_nodes                                       #␣
 ↪Calculates Mean of the values
        std_dev = np.std(y)/num_nodes                                     #␣
 ↪Calculates standard deviation of the values
        mean_dict[x] = mean
        std_dev_dict[x] = std_dev

    mean_list = []
    std_list = []
    for x in sorted(mean_dict):
        mean_list.append(mean_dict[x])                                    #␣
 ↪Creates mean list
        std_list.append(std_dev_dict[x])                                  #␣
 ↪Creates standard deviation list
    return mean_dict, std_dev_dict, mean_list, std_list
```

[7]:
```python
"""
Function         : plotDegreeDist
Input Parameters : mean dictionary, mean list, std deviation list, scale and␣
 ↪power
Purpose          : To Plot the error bar
Returns          : Shows the plot and returns nothing
"""
def plotDegreeDist(mean_dict, mean_list, std_list, power, scale = 'log'):

    # For degree distribution and plot the degree distribution
    fig = plt.figure(figsize = (15,15))                                       ␣
 ↪ # Sets the figure size

    plt.errorbar(np.array(sorted(mean_dict)), mean_list, std_list, fmt='ok')  ␣
 ↪ # Plots the error bar
    if scale == 'log':                                                        ␣
 ↪ # If scale selected is log scale
```

```
        ax=plt.gca()
        ax.set_xscale('log')
        ax.set_yscale('log')
    plt.title('Degree Distribution for Barabasi Albert Model Graph for order '+␣
 ↪str(power))  # Sets the title of the Plot
    plt.xlabel('Degree(k)')                                                    ␣
 ↪ # Sets the x axis label
    plt.ylabel('Pk')                                                           ␣
 ↪ # Sets the y axis label
    plt.show()                                                                 ␣
 ↪ # Shows the plot
```

[8]:
```
"""
Function         : plotDegreeDistNormal
Input Parameters : mean dictionary, mean list, std deviation list, scale and␣
 ↪power
Purpose          : To Plot the scatter plot
Returns          : Shows the plot and returns nothing
"""
def plotDegreeDistNormal(mean_dict, mean_list, std_list, power, scale = 'log'):

    # For degree distribution and plot the degree distribution
    fig = plt.figure(figsize = (15,15))                                        ␣
 ↪ # Sets the figure size

    plt.scatter(np.array(sorted(mean_dict)), mean_list)                        ␣
 ↪ # Plots the scatter plot
    if scale == 'log':                                                         ␣
 ↪ # If scale selected is log scale
        ax=plt.gca()
        ax.set_xscale('log')
        ax.set_yscale('log')
    plt.title('Degree Distribution for Barabasi Albert Model Graph for order '+␣
 ↪str(power))  # Sets the title of the Plot
    plt.xlabel('Degree(k)')                                                    ␣
 ↪ # Sets the x axis label
    plt.ylabel('Pk')                                                           ␣
 ↪ # Sets the y axis label
    plt.show()                                                                 ␣
 ↪ # Shows the plot
```

[9]:
```
"""
Function         : runHigherOrderBA
Input Parameters : Number of nodes(n) and order(power)
Purpose          : To create the graph satisfying Barabasi Albert Model and␣
 ↪calculate the parameters
```

```
Returns          : Shows the plot and returns nothing
"""

def runHigherOrderBA(n, power):
    char_path_length_list = []                          # Declare charcteristic␣
↪path length list
    clustering_coefficient_list = []                    # Declare clustering␣
↪coefficient list
    degree_distribution_dict = {}                       # Declare degree␣
↪distribution dictionary
    m0 = 10                                             # Setting the initial␣
↪number of nodes
    print("Random nodes m0:", m0)
    edges_to_be_added = 20                              # Setting the number of␣
↪edges in the initial graph
    print("Total edges added to initial random graph:", edges_to_be_added)
    m = 6                                             # Sets the m value, the␣
↪count of number of edges the newly added node will connect to
    print("Number of nodes the newly added node will be connected to:", m)
    for i in range(100):                                # Loop for 100 instances
        print("Running instance:", str(i+1))
        G = createInitialGraph(m0, edges_to_be_added)   # Creates the initial␣
↪graph of m0 nodes by calling createInitialGraph method
        G = barabasiModel(G, m, m0, n, power)           # Generates a Barabasi␣
↪Albert Graph
        if i == 0:
            print("Number of edges in Barabasi Albert Model:",len(G.edges()))
        char_path_length = nx.average_shortest_path_length(G)     # Computes␣
↪Characteristic Path length
        clustering_coefficient = nx.average_clustering(G)         # Computes␣
↪clustering coefficient
        char_path_length_list.append(char_path_length)            # Appends␣
↪characteristic path length to its list
        clustering_coefficient_list.append(clustering_coefficient)# Appends␣
↪clustering coefficient to its list

        degree_distribution_dict = getDegreeDist(G, degree_distribution_dict) ␣
↪# Get degree distribution of current instance into the dictionary

    print("\nAverage characteristic path length over 100 instances:", np.
↪mean(char_path_length_list))
    print("\nAverage clustering coefficient over 100 instances:", np.
↪mean(clustering_coefficient_list))
    mean_dict, std_dev_dict, mean_list, std_list =␣
↪getMeanStd(degree_distribution_dict, n) # Get mean and std dev of degree␣
↪distribution
```

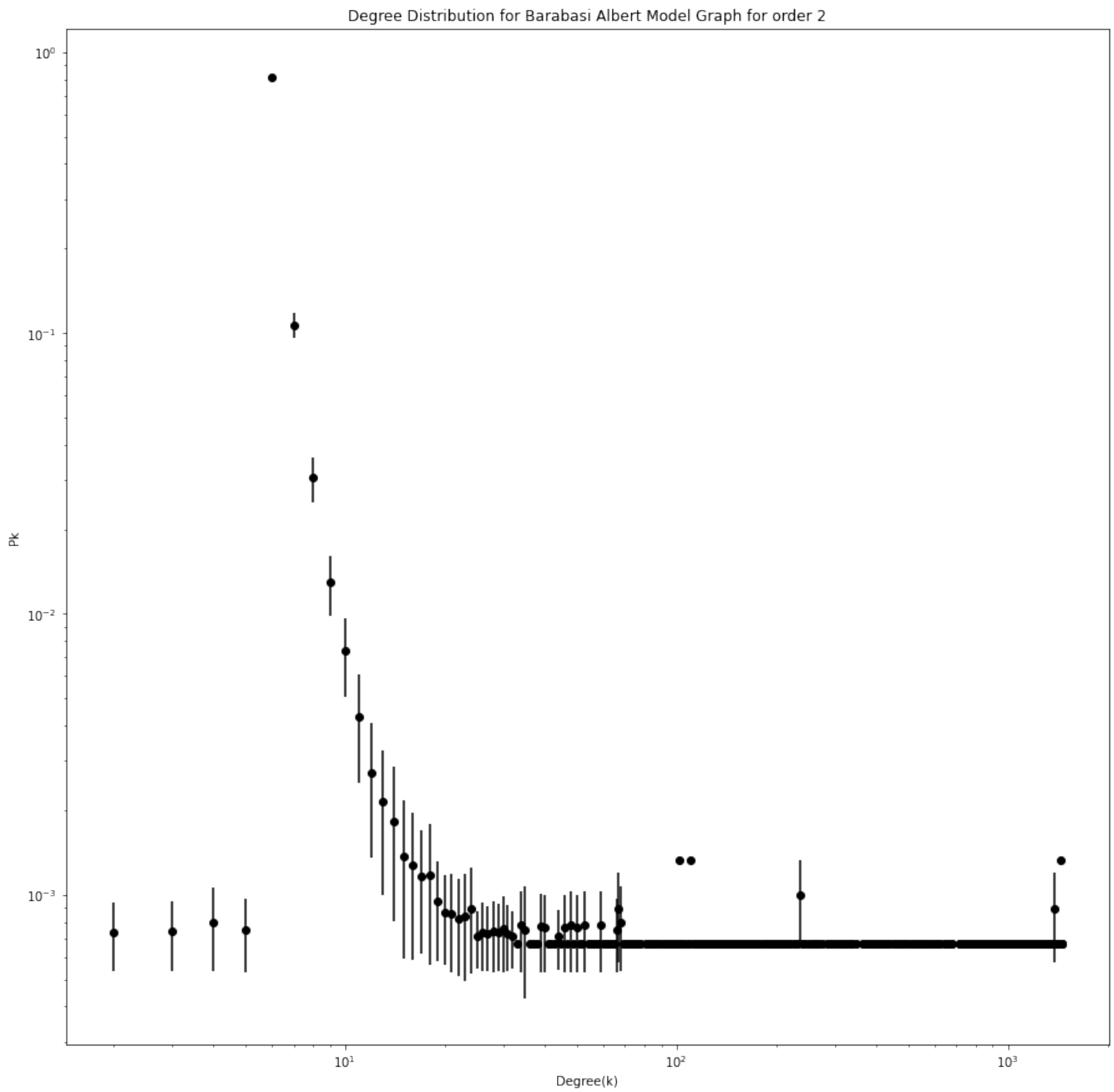
```
    plotDegreeDist(mean_dict, mean_list, std_list, power)              #␣
↪Plots the mean and std dev of degree distribution with log log scale
    plotDegreeDist(mean_dict, mean_list, std_list, power, scale='normal') #␣
↪Plots the mean and std dev of degree distribution with linear scale
    plotDegreeDistNormal(mean_dict, mean_list, std_list, power, scale = 'log')␣
↪# Plots the degree distribution graph with log log scale
```

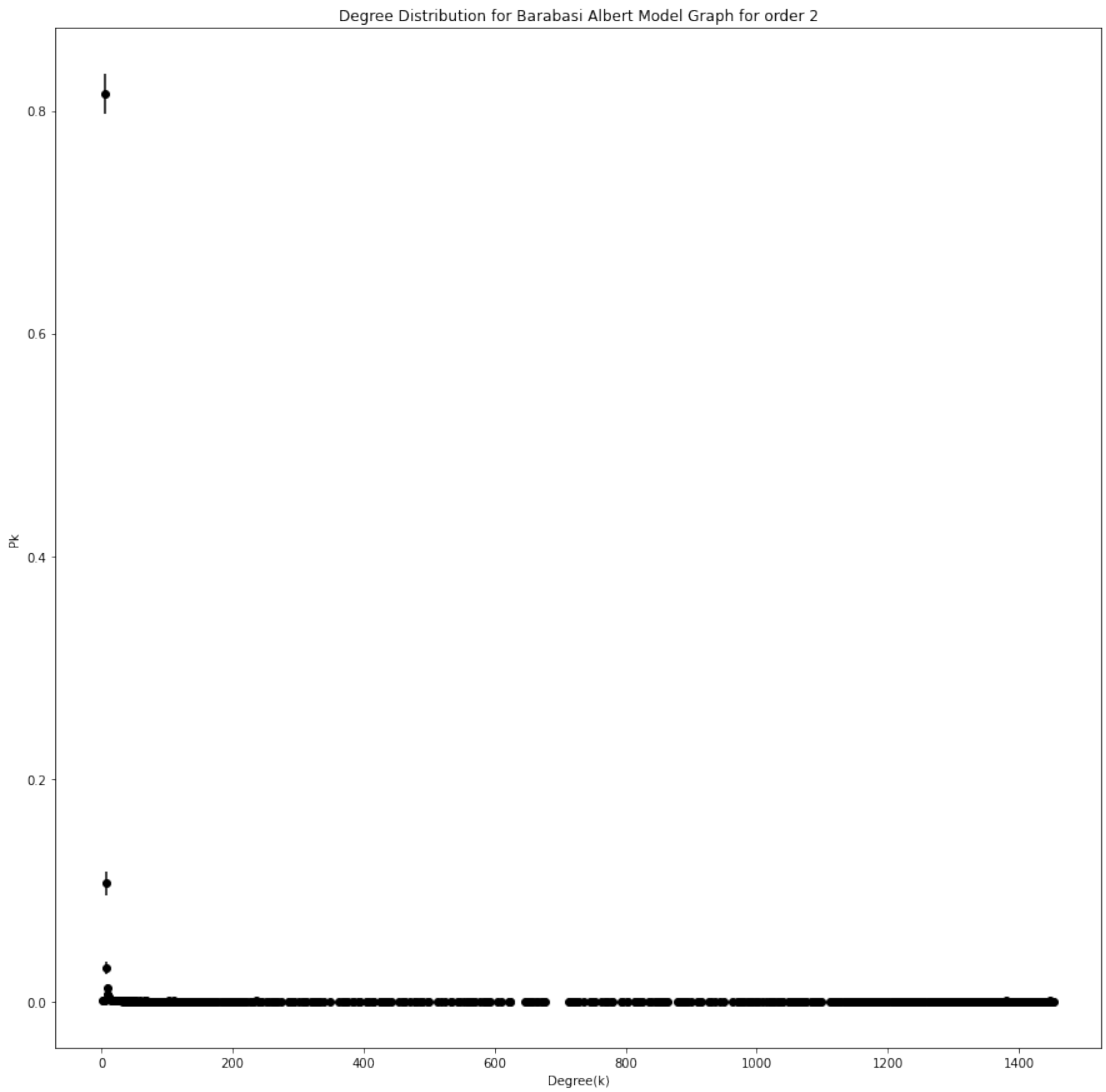
[10]:
```
# Run Second Order Variant for BA Model with 1500 nodes
print("Running BA Model for order = 2")
runHigherOrderBA(1500, 2)
```

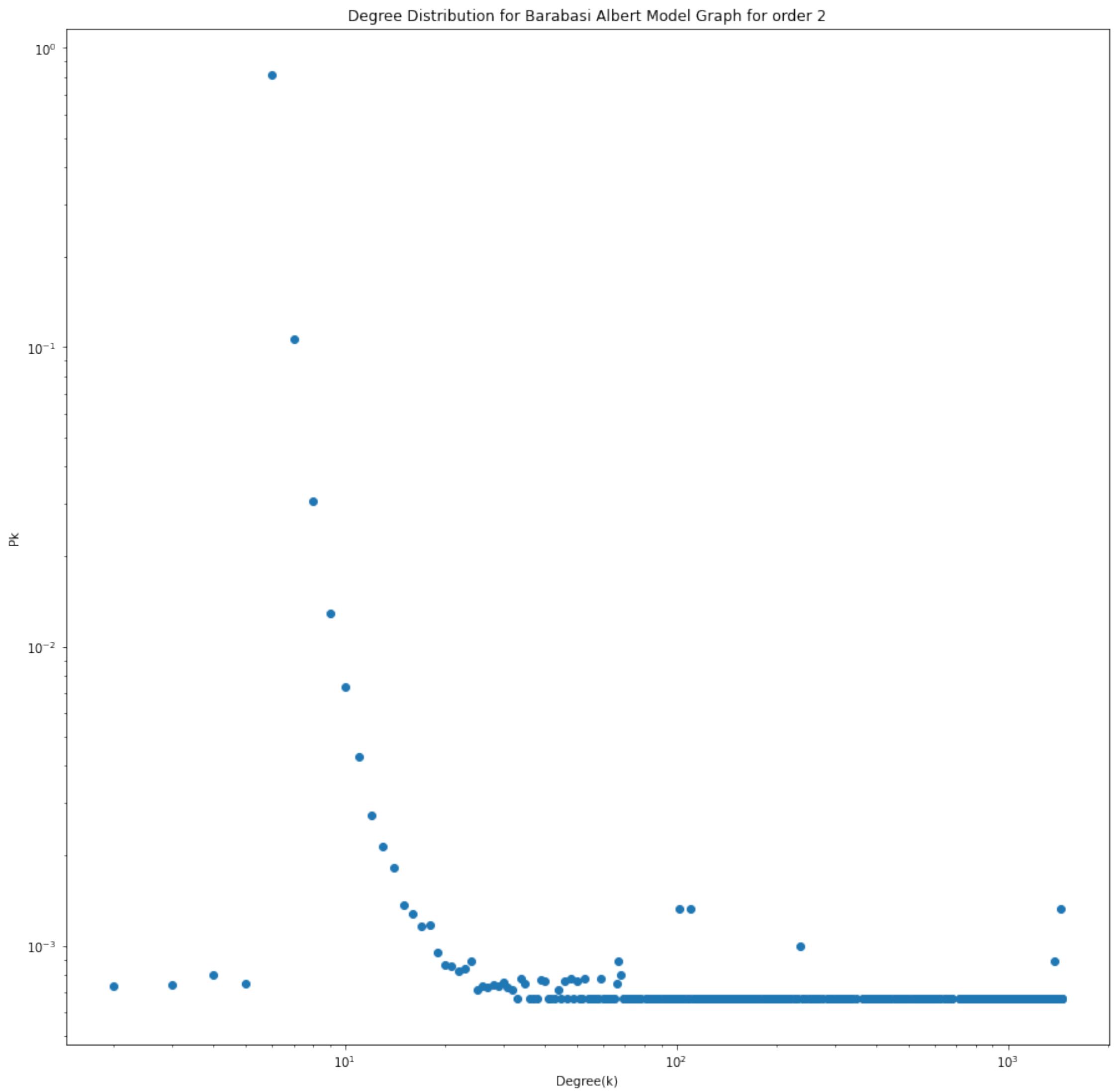
```
Running BA Model for order = 2
Random nodes m0: 10
Total edges added to initial random graph: 20
Number of nodes the newly added node will be connected to: 6
Running instance: 1
Number of edges in Barabasi Albert Model: 8960
Running instance: 2
Running instance: 3
Running instance: 4
Running instance: 5
Running instance: 6
Running instance: 7
Running instance: 8
Running instance: 9
Running instance: 10
Running instance: 11
Running instance: 12
Running instance: 13
Running instance: 14
Running instance: 15
Running instance: 16
Running instance: 17
Running instance: 18
Running instance: 19
Running instance: 20
Running instance: 21
Running instance: 22
Running instance: 23
Running instance: 24
Running instance: 25
Running instance: 26
Running instance: 27
Running instance: 28
Running instance: 29
Running instance: 30
Running instance: 31
```

```
Running instance: 32
Running instance: 33
Running instance: 34
Running instance: 35
Running instance: 36
Running instance: 37
Running instance: 38
Running instance: 39
Running instance: 40
Running instance: 41
Running instance: 42
Running instance: 43
Running instance: 44
Running instance: 45
Running instance: 46
Running instance: 47
Running instance: 48
Running instance: 49
Running instance: 50
Running instance: 51
Running instance: 52
Running instance: 53
Running instance: 54
Running instance: 55
Running instance: 56
Running instance: 57
Running instance: 58
Running instance: 59
Running instance: 60
Running instance: 61
Running instance: 62
Running instance: 63
Running instance: 64
Running instance: 65
Running instance: 66
Running instance: 67
Running instance: 68
Running instance: 69
Running instance: 70
Running instance: 71
Running instance: 72
Running instance: 73
Running instance: 74
Running instance: 75
Running instance: 76
Running instance: 77
Running instance: 78
Running instance: 79
```

```
Running instance: 80
Running instance: 81
Running instance: 82
Running instance: 83
Running instance: 84
Running instance: 85
Running instance: 86
Running instance: 87
Running instance: 88
Running instance: 89
Running instance: 90
Running instance: 91
Running instance: 92
Running instance: 93
Running instance: 94
Running instance: 95
Running instance: 96
Running instance: 97
Running instance: 98
Running instance: 99
Running instance: 100

Average characteristic path length over 100 instances: 1.9934355525906162

Average clustering coefficient over 100 instances: 0.6864270974292676
```


Degree Distribution for Barabasi Albert Model Graph for order 2
Pk
Degree(k)
10^0
10^{-1}
10^{-2}
10^{-3}
10^1
10^2
10^3


Degree Distribution for Barabasi Albert Model Graph for order 2
Pk
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000
1200
1400
Degree(k)



The above plots show the Second order preferential attachment for Barabasi Albert Model. The second order preferential attachment function can be written in terms of the formula as (kj * kj)/Summation of all (kj * kj) where j varies from 1 to n.

The network created contained the number of nodes same as that in Q2. i.e. 1500. The initial random network created consisted of 10 nodes and 20 edges and at each iteration 1 node with 6 edges were added to it.

The following values were obtained in terms of characteristic path length and clustering coefficient.

Average characteristic path length over 100 instances: 1.9934355525906162

Average clustering coefficient over 100 instances: 0.6864270974292676

We can see that the characteristic path length has decreased upon increasing the order to 2 as compared to the original BA Model obtained in Q2. The average characteristic path length obtained was 2.96 in Q2 whereas it dropped to 1.99 in case of second order. Whereas clustering coefficient was 0.031 in Q2 but has now increased to 0.686 in case of second order.

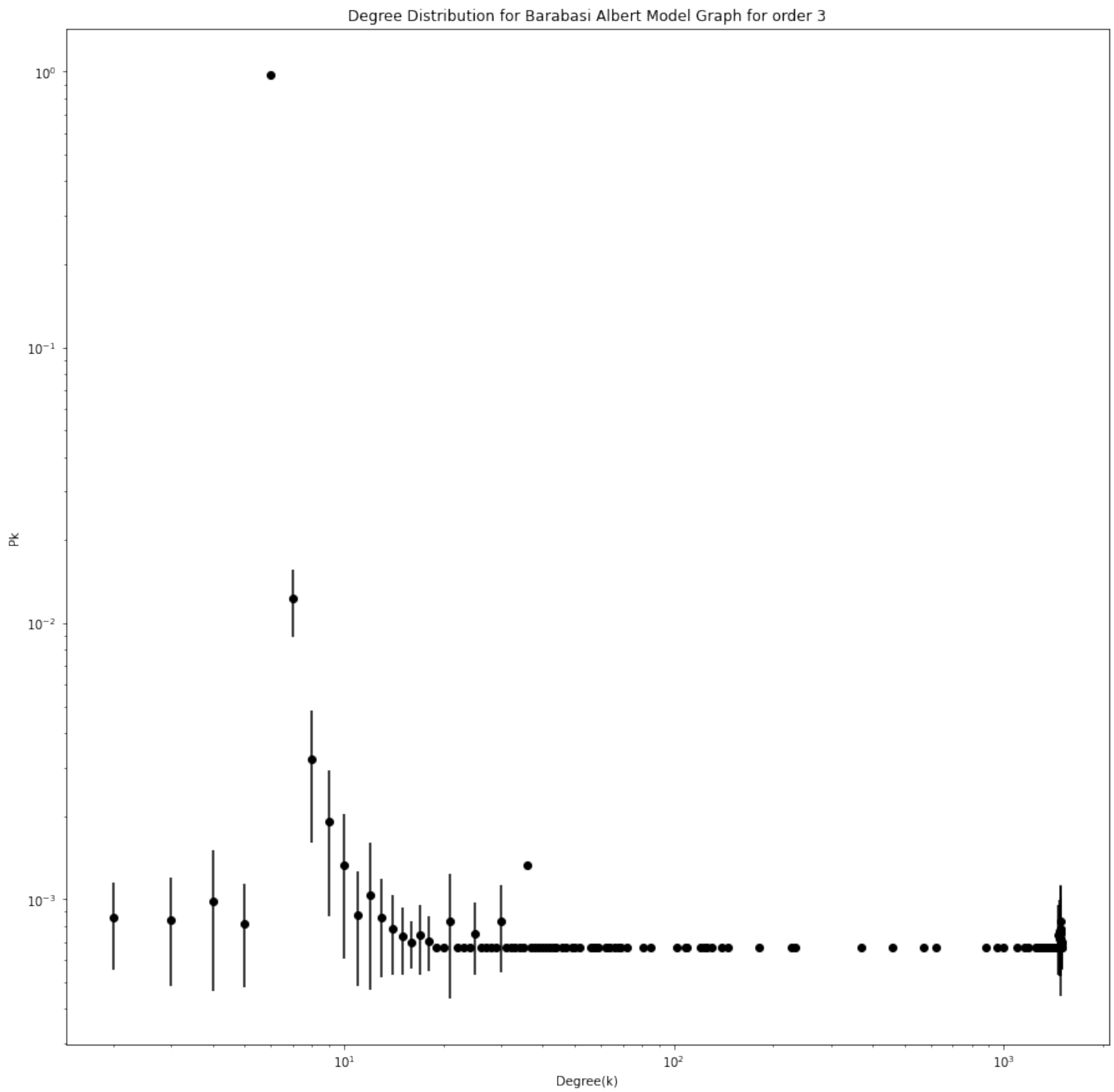
Also, as can be seen from the plots of degree distribution that there are more hubs present in the second order BA model as compared to first order BA model. Maximum degree obtained in first oder was 200, whereas in second order, the maximum degree obtained was more than 1400. The plot shown is less scale free due to the presence of long constant tail(in terms of hubs).

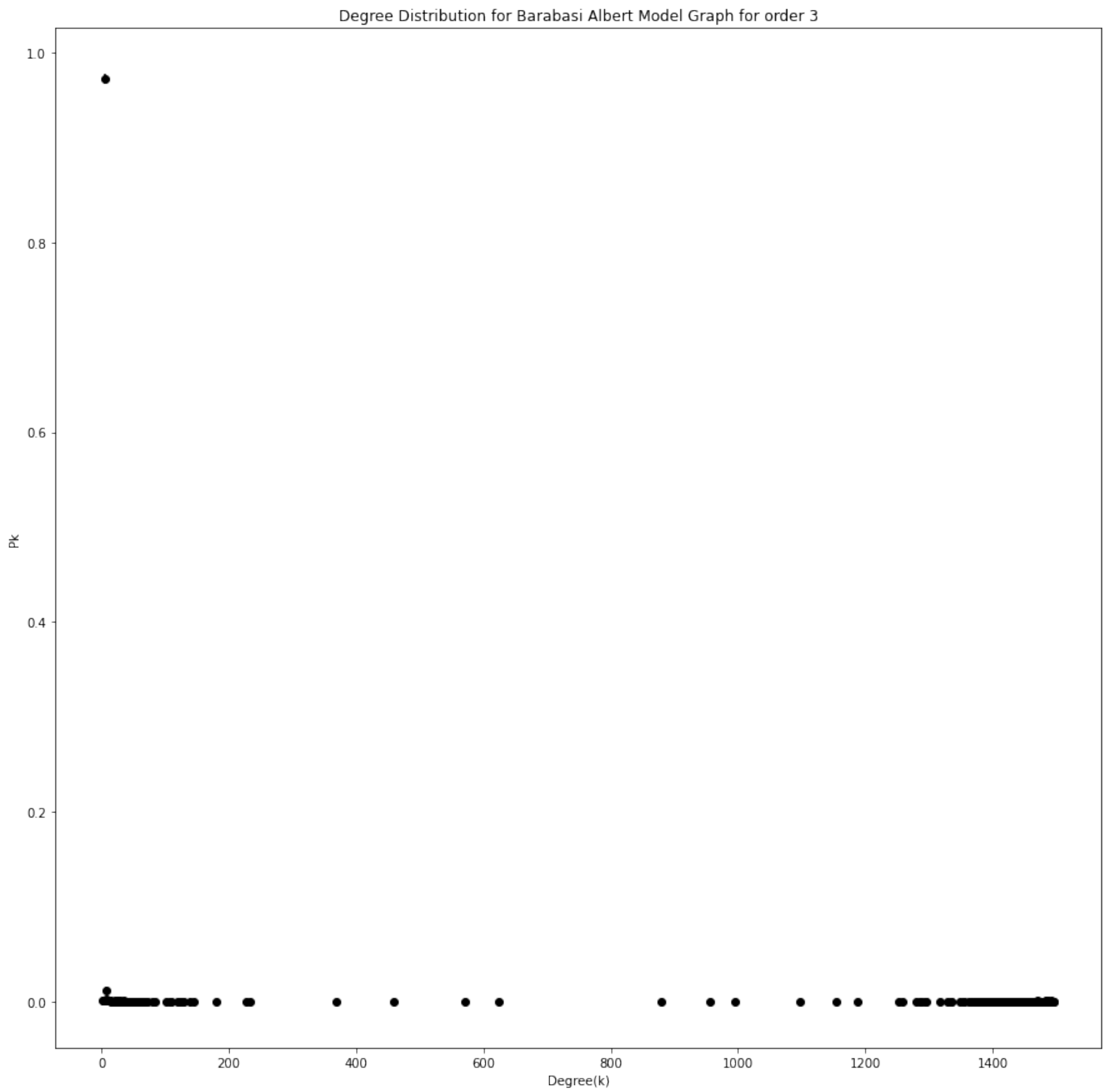
[11]:
```python
# Run Higher Order Variants for BA Model with 1500 nodes
for i in range(3, 6, 1):
    print("Running BA Model for order = ", str(i))
    runHigherOrderBA(1500, i)
```

```
Running BA Model for order =  3
Random nodes m0: 10
Total edges added to initial random graph: 20
Number of nodes the newly added node will be connected to: 6
Running instance: 1
Number of edges in Barabasi Albert Model: 8960
Running instance: 2
Running instance: 3
Running instance: 4
Running instance: 5
Running instance: 6
Running instance: 7
Running instance: 8
Running instance: 9
Running instance: 10
Running instance: 11
Running instance: 12
Running instance: 13
Running instance: 14
Running instance: 15
Running instance: 16
Running instance: 17
Running instance: 18
Running instance: 19
Running instance: 20
Running instance: 21
Running instance: 22
Running instance: 23
Running instance: 24
Running instance: 25
Running instance: 26
Running instance: 27
Running instance: 28
Running instance: 29
Running instance: 30
Running instance: 31
```

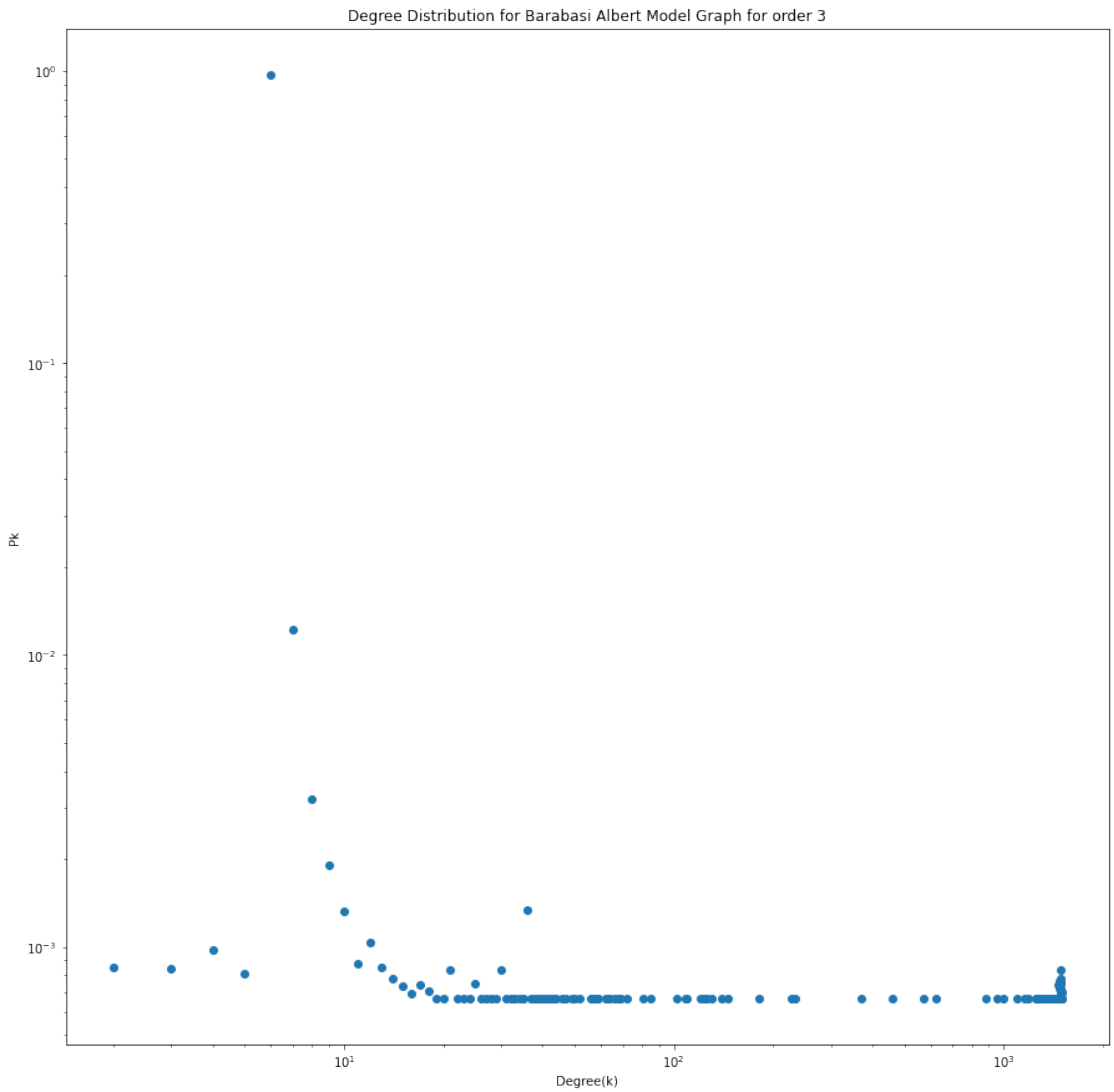
```
Running instance: 32
Running instance: 33
Running instance: 34
Running instance: 35
Running instance: 36
Running instance: 37
Running instance: 38
Running instance: 39
Running instance: 40
Running instance: 41
Running instance: 42
Running instance: 43
Running instance: 44
Running instance: 45
Running instance: 46
Running instance: 47
Running instance: 48
Running instance: 49
Running instance: 50
Running instance: 51
Running instance: 52
Running instance: 53
Running instance: 54
Running instance: 55
Running instance: 56
Running instance: 57
Running instance: 58
Running instance: 59
Running instance: 60
Running instance: 61
Running instance: 62
Running instance: 63
Running instance: 64
Running instance: 65
Running instance: 66
Running instance: 67
Running instance: 68
Running instance: 69
Running instance: 70
Running instance: 71
Running instance: 72
Running instance: 73
Running instance: 74
Running instance: 75
Running instance: 76
Running instance: 77
Running instance: 78
Running instance: 79
```

```
Running instance: 80
Running instance: 81
Running instance: 82
Running instance: 83
Running instance: 84
Running instance: 85
Running instance: 86
Running instance: 87
Running instance: 88
Running instance: 89
Running instance: 90
Running instance: 91
Running instance: 92
Running instance: 93
Running instance: 94
Running instance: 95
Running instance: 96
Running instance: 97
Running instance: 98
Running instance: 99
Running instance: 100

Average characteristic path length over 100 instances: 1.992284385145653

Average clustering coefficient over 100 instances: 0.7921502147340398
```


Degree Distribution for Barabasi Albert Model Graph for order 3
10^0
10^{-1}
10^{-2}
10^{-3}
Pk
10^1
10^2
10^3
Degree(k)



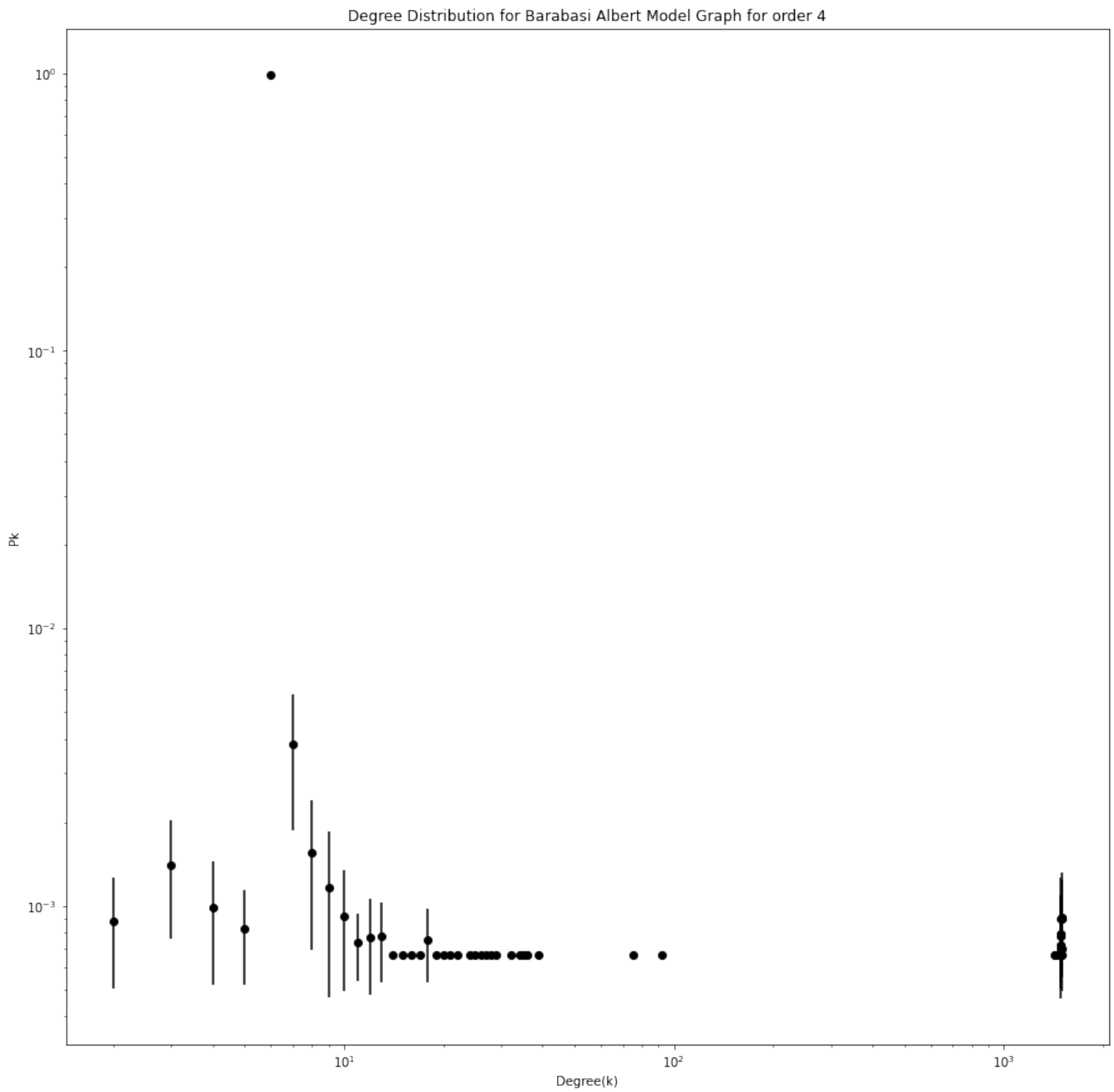
Degree Distribution for Barabasi Albert Model Graph for order 3
1.0
0.8
0.6
0.4
0.2
0.0
Pk
0
200
400
600
800
1000
1200
1400
Degree(k)

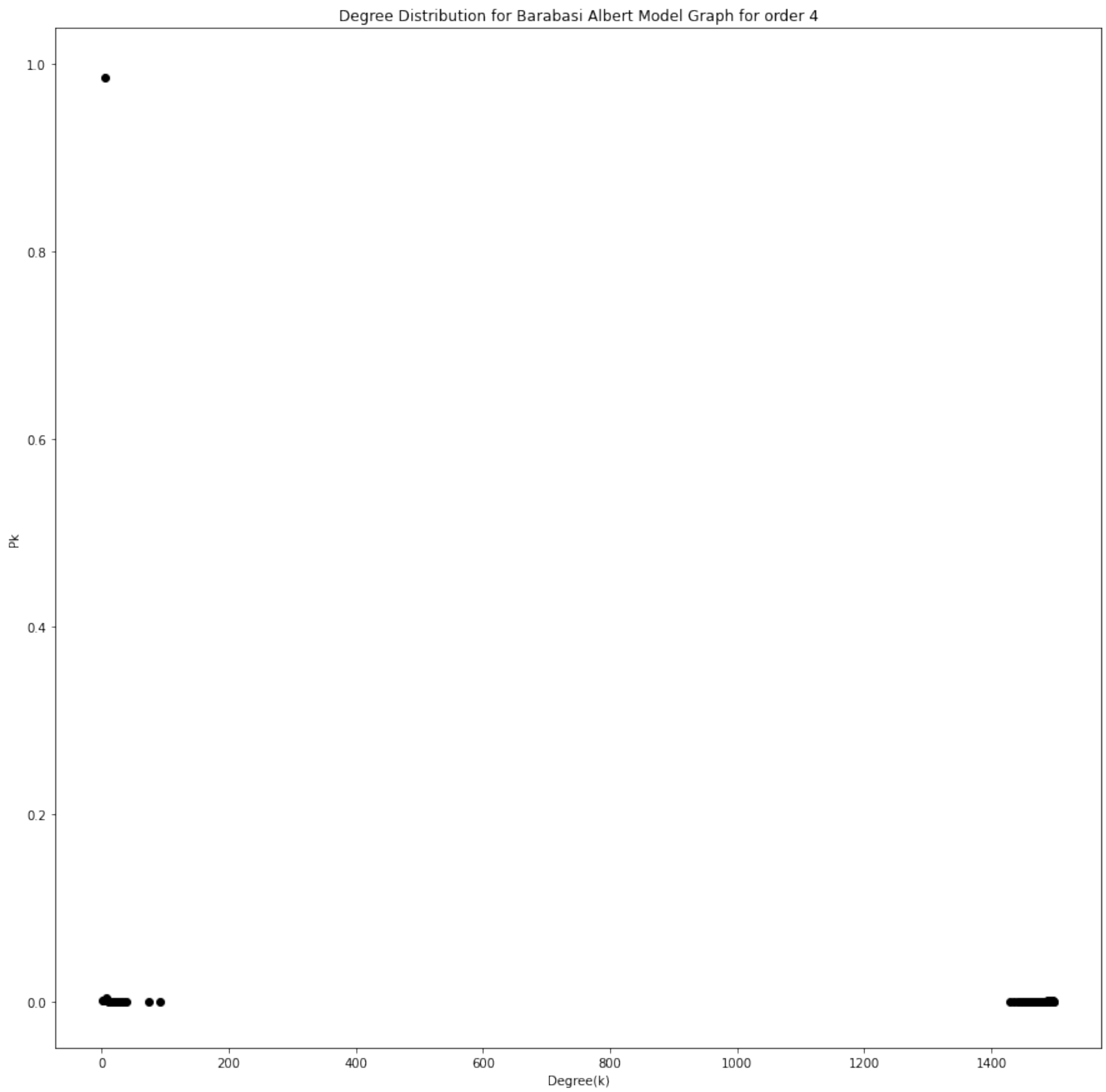


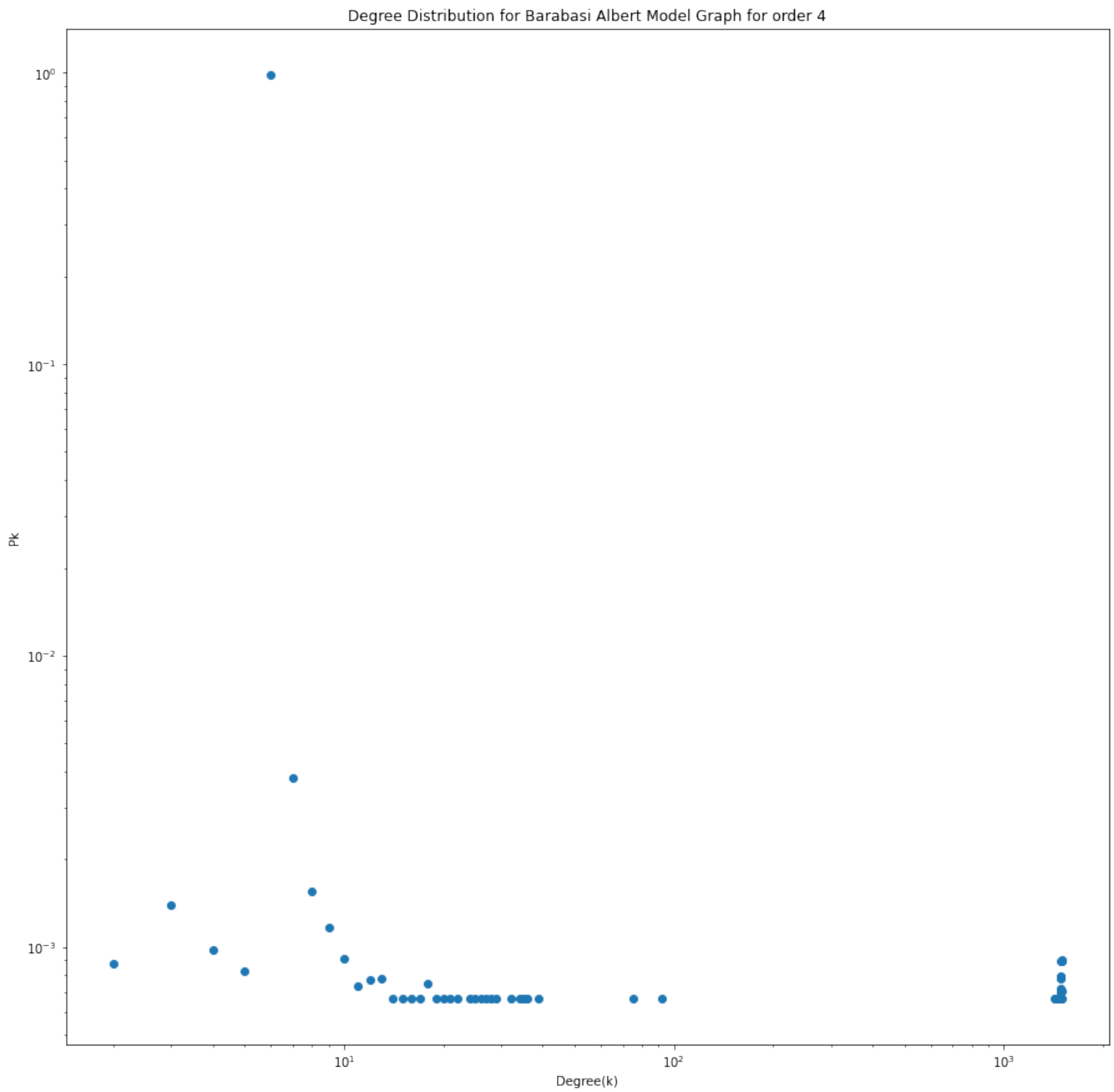

```
Running BA Model for order =  4
Random nodes m0: 10
Total edges added to initial random graph: 20
Number of nodes the newly added node will be connected to: 6
Running instance: 1
Number of edges in Barabasi Albert Model: 8960
Running instance: 2
Running instance: 3
Running instance: 4
Running instance: 5
Running instance: 6
Running instance: 7
Running instance: 8
Running instance: 9
```

```
Running instance: 10
Running instance: 11
Running instance: 12
Running instance: 13
Running instance: 14
Running instance: 15
Running instance: 16
Running instance: 17
Running instance: 18
Running instance: 19
Running instance: 20
Running instance: 21
Running instance: 22
Running instance: 23
Running instance: 24
Running instance: 25
Running instance: 26
Running instance: 27
Running instance: 28
Running instance: 29
Running instance: 30
Running instance: 31
Running instance: 32
Running instance: 33
Running instance: 34
Running instance: 35
Running instance: 36
Running instance: 37
Running instance: 38
Running instance: 39
Running instance: 40
Running instance: 41
Running instance: 42
Running instance: 43
Running instance: 44
Running instance: 45
Running instance: 46
Running instance: 47
Running instance: 48
Running instance: 49
Running instance: 50
Running instance: 51
Running instance: 52
Running instance: 53
Running instance: 54
Running instance: 55
Running instance: 56
Running instance: 57
```

```
Running instance: 58
Running instance: 59
Running instance: 60
Running instance: 61
Running instance: 62
Running instance: 63
Running instance: 64
Running instance: 65
Running instance: 66
Running instance: 67
Running instance: 68
Running instance: 69
Running instance: 70
Running instance: 71
Running instance: 72
Running instance: 73
Running instance: 74
Running instance: 75
Running instance: 76
Running instance: 77
Running instance: 78
Running instance: 79
Running instance: 80
Running instance: 81
Running instance: 82
Running instance: 83
Running instance: 84
Running instance: 85
Running instance: 86
Running instance: 87
Running instance: 88
Running instance: 89
Running instance: 90
Running instance: 91
Running instance: 92
Running instance: 93
Running instance: 94
Running instance: 95
Running instance: 96
Running instance: 97
Running instance: 98
Running instance: 99
Running instance: 100

Average characteristic path length over 100 instances: 1.9922601912386038

Average clustering coefficient over 100 instances: 0.8388467890276671
```


Degree Distribution for Barabasi Albert Model Graph for order 4
10^0
10^{-1}
10^{-2}
10^{-3}
Pk
10^1
10^2
10^3
Degree(k)

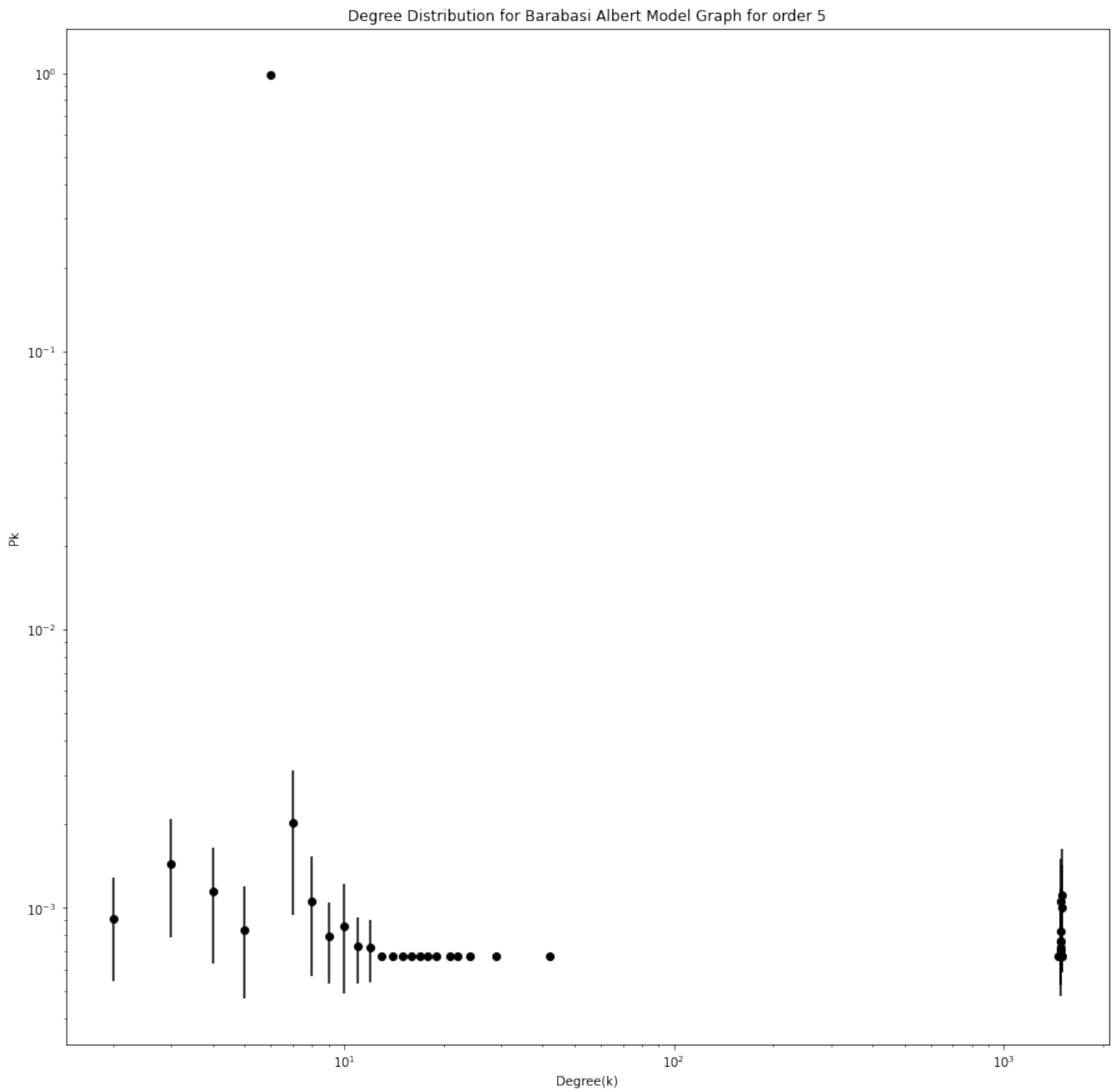

Degree Distribution for Barabasi Albert Model Graph for order 4
Pk
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000
1200
1400
Degree(k)

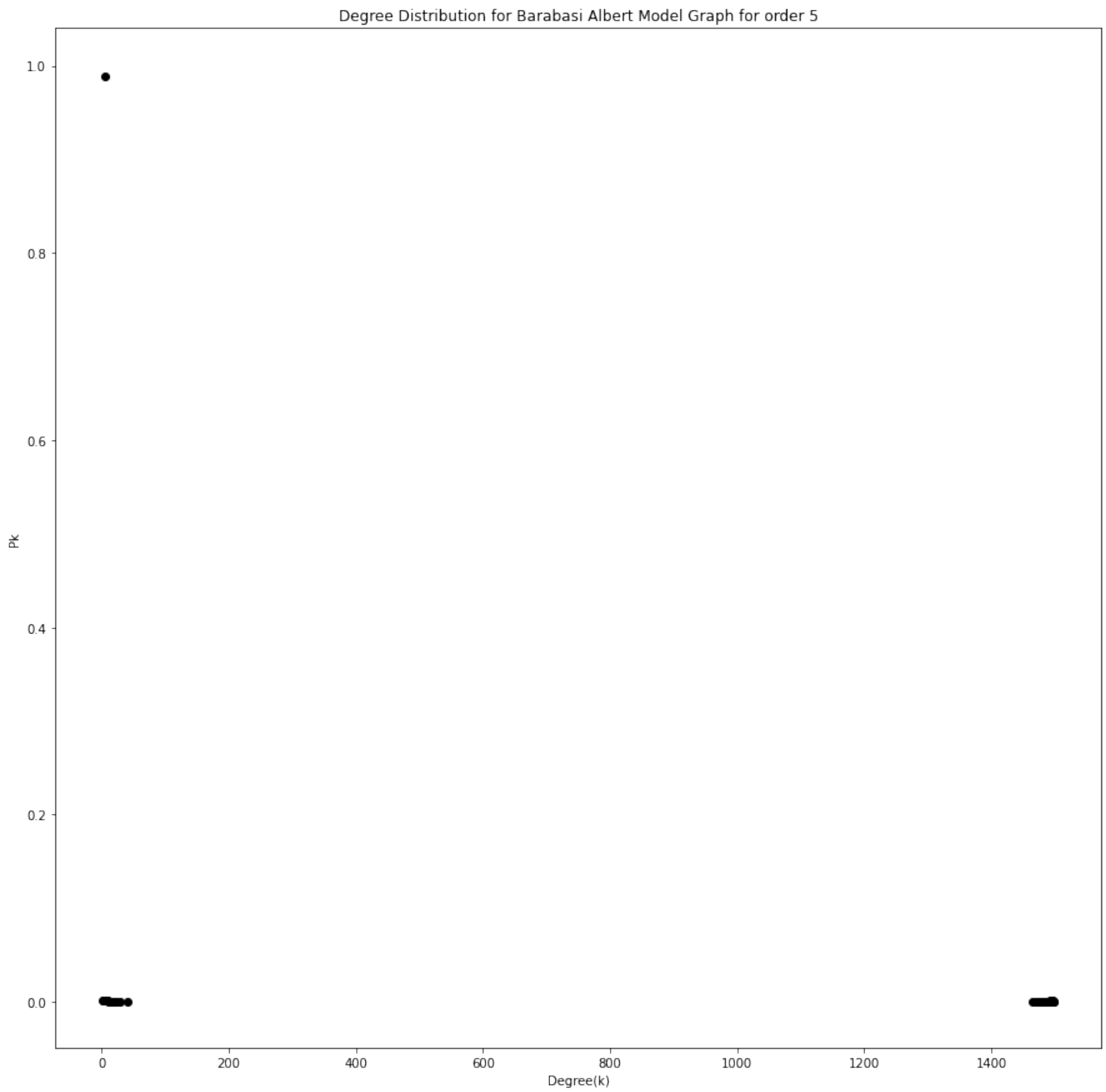


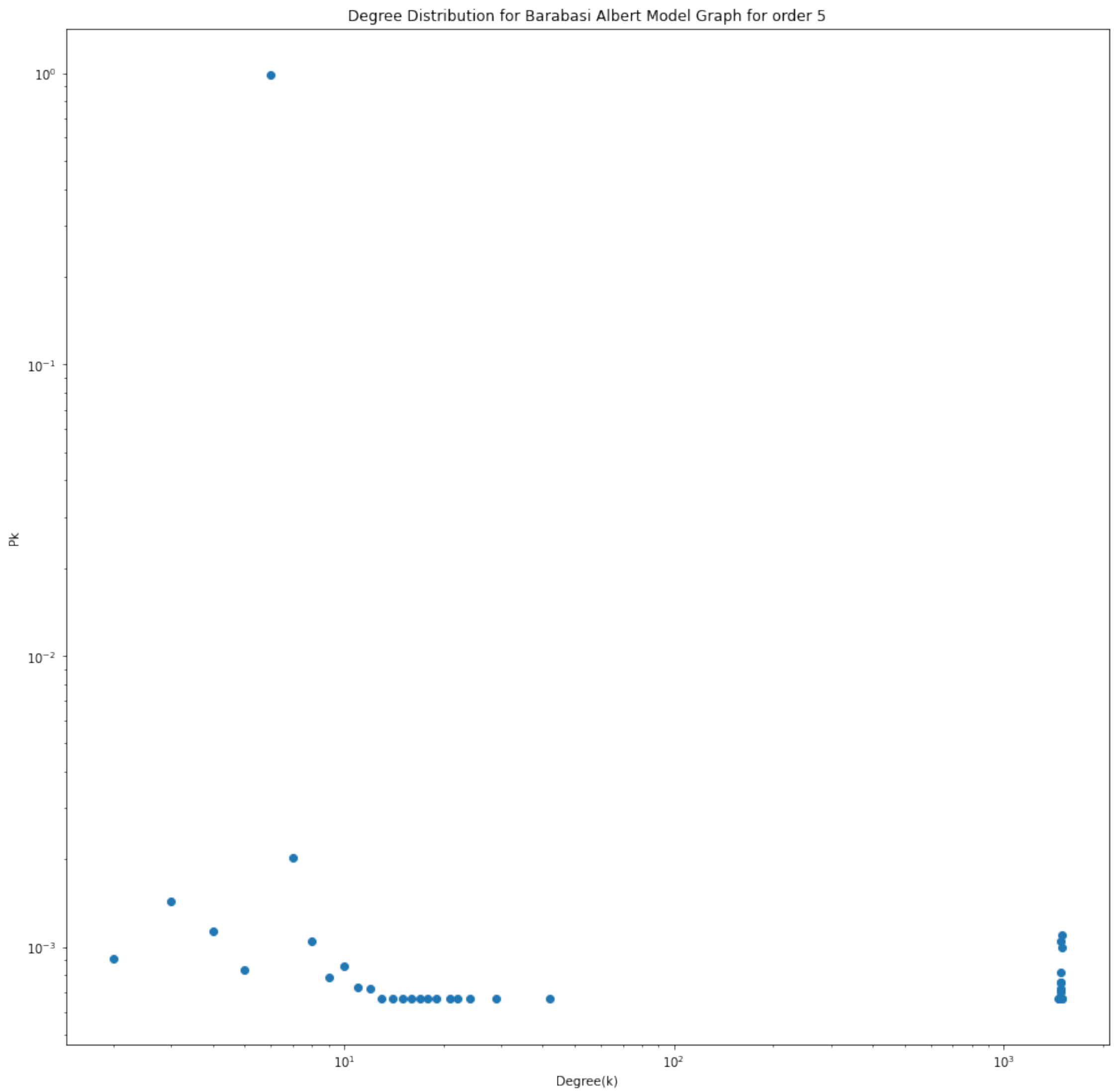

```
Running BA Model for order =  5
Random nodes m0: 10
Total edges added to initial random graph: 20
Number of nodes the newly added node will be connected to: 6
Running instance: 1
Number of edges in Barabasi Albert Model: 8960
Running instance: 2
Running instance: 3
Running instance: 4
Running instance: 5
Running instance: 6
Running instance: 7
Running instance: 8
Running instance: 9
```

```
Running instance: 10
Running instance: 11
Running instance: 12
Running instance: 13
Running instance: 14
Running instance: 15
Running instance: 16
Running instance: 17
Running instance: 18
Running instance: 19
Running instance: 20
Running instance: 21
Running instance: 22
Running instance: 23
Running instance: 24
Running instance: 25
Running instance: 26
Running instance: 27
Running instance: 28
Running instance: 29
Running instance: 30
Running instance: 31
Running instance: 32
Running instance: 33
Running instance: 34
Running instance: 35
Running instance: 36
Running instance: 37
Running instance: 38
Running instance: 39
Running instance: 40
Running instance: 41
Running instance: 42
Running instance: 43
Running instance: 44
Running instance: 45
Running instance: 46
Running instance: 47
Running instance: 48
Running instance: 49
Running instance: 50
Running instance: 51
Running instance: 52
Running instance: 53
Running instance: 54
Running instance: 55
Running instance: 56
Running instance: 57
```

```
Running instance: 58
Running instance: 59
Running instance: 60
Running instance: 61
Running instance: 62
Running instance: 63
Running instance: 64
Running instance: 65
Running instance: 66
Running instance: 67
Running instance: 68
Running instance: 69
Running instance: 70
Running instance: 71
Running instance: 72
Running instance: 73
Running instance: 74
Running instance: 75
Running instance: 76
Running instance: 77
Running instance: 78
Running instance: 79
Running instance: 80
Running instance: 81
Running instance: 82
Running instance: 83
Running instance: 84
Running instance: 85
Running instance: 86
Running instance: 87
Running instance: 88
Running instance: 89
Running instance: 90
Running instance: 91
Running instance: 92
Running instance: 93
Running instance: 94
Running instance: 95
Running instance: 96
Running instance: 97
Running instance: 98
Running instance: 99
Running instance: 100

Average characteristic path length over 100 instances: 1.9921671069601956

Average clustering coefficient over 100 instances: 0.8382023895259105
```


Degree Distribution for Barabasi Albert Model Graph for order 5
10^0
10^{-1}
10^{-2}
10^{-3}
Pk
10^1
10^2
10^3
Degree(k)


Degree Distribution for Barabasi Albert Model Graph for order 5
1.0
0.8
0.6
0.4
0.2
0.0
Pk
0
200
400
600
800
1000
1200
1400
Degree(k)



The above plots show the Third order, Fourth order and Fifth order preferential attachment for Barabasi Albert Model. The higher order preferential attachment function can be written in terms of the formula as (kj^order)/Summation of all (kj^order) where j varies from 1 to n.

The network created contained the number of nodes same as that in Q2. i.e. 1500. The initial random network created consisted of 10 nodes and 20 edges and at each iteration 1 node with 6 edges were added to it.

The following values were obtained in terms of characteristic path length and clustering coefficient.

For Third Order

Average characteristic path length over 100 instances: 1.992284385145653

Average clustering coefficient over 100 instances: 0.7921502147340398

For Fourth Order

Average characteristic path length over 100 instances: 1.9922601912386038

Average clustering coefficient over 100 instances: 0.8388467890276671

For Fifth Order

Average characteristic path length over 100 instances: 1.9921671069601956

Average clustering coefficient over 100 instances: 0.8382023895259105

We can see that the characteristic path length has remained constant upon increasing the order to 3, 4 and 5 as compared to the original BA Model obtained for order 2. The average characteristic path length obtaifor second order but has now increased to 0.792 in case of third order and becoming constant to 0.838 in fourth and fifth order.

Also, as can be seen from the plots of degree distribution that the scale free nature of the network is destroyed upon increasing the order of the BA Model preferential attachment.

[ ]:

```

```