

## Question-1

### Graph Parameters

- Firstly we defined the number of nodes and degree for each edge.
- We took Number of Nodes(N) to be 1500
- We took Degree(k) of each Node to be 50

```
1 NUMBER_OF_NODES = 1500
2 INITIAL_AVG_DEGREE = 50
```

### Generating Regular Graph

- We started with making the regular graph with the defined parameters.
- For this, we first initialize an empty ADJ\_MATRIX of size NUM\_NODES X NUM\_NODES.
- Next, we split the Edges into two groups: left and right edges.
- The number of left edges would be Degree/2.
- The number of right edges would be Degree - number of left edges.
- Next, we ran a simple for loop logic to connect two edges, we first check whether taking this edge is allowed or not, by checking if it is less than number of nodes for right edges, if yes then directly mark 1 for this connection in the adj matrix and if no then account for a left edge for this, so as to keep the number of edges equal to the degree.
- Similarly, for left edges we first checked if it is greater than 0, if yes then directly mark 1 for this connection in the adj matrix and if no then account for a right edge for this, so as to keep the number of edges equal to the degree.
- After this we got our initial regular graph.

```
1 # generating the ring link graph
2
3 ADJ_MATRIX = []
4 for i in range(0,NUMBER_OF_NODES):
5     current_row = []
6     for j in range(0,NUMBER_OF_NODES):
7         current_row.append(0)
8     ADJ_MATRIX.append(current_row)
9
10 EDGES_TO_LEFT = INITIAL_AVG_DEGREE // 2
11 EDGES_TO_RIGHT = INITIAL_AVG_DEGREE - EDGES_TO_LEFT
12
13 for choosen_node in range(0,NUMBER_OF_NODES):
14     RIGHT_BOUND = choosen_node + EDGES_TO_RIGHT
15     if(RIGHT_BOUND < NUMBER_OF_NODES):
16         for neighbour in range(choosen_node+1,RIGHT_BOUND):
17             ADJ_MATRIX[choosen_node][neighbour]=1
18             ADJ_MATRIX[neighbour][choosen_node] = 1
19     else:
20         ALLOWED_RIGHT_BOUND = NUMBER_OF_NODES - choosen_node -1
21         for neighbour in range(choosen_node+1,NUMBER_OF_NODES):
22             ADJ_MATRIX[choosen_node][neighbour]=1
23             ADJ_MATRIX[neighbour][choosen_node] = 1
24
25     RIGHT_BOUND_LEFT = EDGES_TO_RIGHT - (ALLOWED_RIGHT_BOUND )
26     for neighbour in range(RIGHT_BOUND_LEFT):
27         ADJ_MATRIX[choosen_node][neighbour]=1
28         ADJ_MATRIX[neighbour][choosen_node] = 1
29
30
31     LEFT_BOUND = choosen_node - EDGES_TO_LEFT
32     if LEFT_BOUND >= 0:
33         for neighbour in range(LEFT_BOUND, choosen_node):
34             ADJ_MATRIX[choosen_node][neighbour] = 1
35             ADJ_MATRIX[neighbour][choosen_node] = 1
36     else:
37         ALLOWED_LEFT_BOUND = choosen_node
38         for neighbour in range(0, choosen_node):
39             ADJ_MATRIX[choosen_node][neighbour] = 1
40             ADJ_MATRIX[neighbour][choosen_node] = 1
41
42     LEFT_BOUND_LEFT = EDGES_TO_LEFT - ALLOWED_LEFT_BOUND -1
```

```

43     for neighbour in range(NUMBER_OF_NODES - LEFT_BOUND_LEFT, NUMBER_OF_NODES):
44         ADJ_MATRIX[choosen_node][neighbour] = 1
45         ADJ_MATRIX[neighbour][choosen_node] = 1
46

```

## Writing Functions for BFS, Calculation of Path Length And Calculation of Clustering Coefficient

- To calculate the avg path length we use the following formula:-
- Average Path Length = Total Distance / Number of Valid Pairs
- Clustering Coefficient (C) = (2 \* Number of Actual Edges Between Neighbors) / (Degree of Node \* (Degree of Node - 1))

Where:

- **Number of Actual Edges Between Neighbors** is the number of edges that actually exist between the neighbors of the node.
- **Degree of Node** is the number of neighbors connected to the node.

The clustering coefficient measures how interconnected the neighbors of a node are to each other. A high clustering coefficient indicates that the node's neighbors are densely connected.

```

1 from collections import deque
2
3 def bfs(source, num_nodes, adj_list):
4     distances={}
5     for i in range(num_nodes):
6         distances[i]=float('inf')
7     distances[source] = 0
8     queue = deque([source])
9
10    while queue:
11        node = queue.popleft()
12        for neighbor in adj_list[node]:
13            if distances[neighbor] == float('inf'):
14                distances[neighbor] = distances[node] + 1
15                queue.append(neighbor)
16
17    return distances
18
19
20 def calculate_avg_path_length(ADJ_MATRIX):
21     num_nodes=len(ADJ_MATRIX)
22     ADJ_LIST = {}
23
24     for i in range(num_nodes):
25         ADJ_LIST[i]=[]
26
27     for i in range(num_nodes):
28         for j in range(num_nodes):
29             if(ADJ_MATRIX[i][j]==1):
30                 ADJ_LIST[i].append(j)
31
32     TOTAL_DISTANCE=0
33     PAIRS=0
34     max_path_length=-1
35
36     for node in range(num_nodes):
37         temp_list=[]
38         distance_array = bfs(node,num_nodes,ADJ_LIST)
39         for dst in range(num_nodes):
40             if((node!=dst) and (distance_array[dst]!=float('inf'))):
41                 TOTAL_DISTANCE+=distance_array[dst]
42                 PAIRS+=1
43         for dist in distance_array.values():
44             if dist != float('inf'):
45                 temp_list.append(dist)
46         max_path_length=max(max_path_length,max(temp_list))
47
48     avg_path_length=TOTAL_DISTANCE/PAIRS
49     return avg_path_length
50
51
52
53 def find_clustering_coefficient(adj_matrix):
54     num_nodes=len(adj_matrix)
55     sum_of_coefficients = 0

```

```

56
57     for src in range(num_nodes):
58         degree=0
59         neighbours=[]
60         common_edge=0
61         for dst in range(num_nodes):
62             if(adj_matrix[src][dst]==1):
63                 degree+=1
64                 neighbours.append(dst)
65
66         if(degree < 2):
67             continue
68
69         for i in range(degree):
70             for j in range(i + 1, degree):
71                 if adj_matrix[neighbours[i]][neighbours[j]] == 1:
72                     common_edge += 1
73
74         coefficient=(2*(common_edge))/(degree*(degree-1))
75         sum_of_coefficients+=coefficient
76
77     avg_clustering_coefficient = sum_of_coefficients / num_nodes
78     return avg_clustering_coefficient
79
80

```

## ✓ Finding the Initial Path Length And Clustering Coefficient

- Next, we stored the path lengths and clustering coefficient for the regular graph, in order to scale the later values.

```

1 ORIGINAL_PATH_LENGTH = calculate_avg_path_length(ADJ_MATRIX)
2 ORIGINAL_CLUSTERING_COEFF = find_clustering_coefficient(ADJ_MATRIX)
3
4 print(ORIGINAL_PATH_LENGTH)
5 print(ORIGINAL_CLUSTERING_COEFF)

```

15.490326884589727  
0.7346938775510015

## ✓ Rewiring the Edges

1. Loop through each node in the adjacency matrix.
2. Identify its neighbors from the original ring lattice.
3. With probability  $p$ , each edge (node, neighbor) is rewired:
  - Remove the existing edge.
  - Find a new random node that is not already connected.
  - Create a new edge to this random node.
4. Repeat for different  $p$  values to observe the transition from a regular lattice to a random graph.

```

1 import copy
2 import random
3 import numpy as np
4 random.seed(34322341)
5 ORIGINAL_ADJ_MATRIX = copy.deepcopy(ADJ_MATRIX)
6
7
8 P_VALUES = np.logspace(-4, 0, num=20)
9 P_VALUES = [round(p, 6) for p in P_VALUES]
10 num_nodes = len(ORIGINAL_ADJ_MATRIX)
11
12 DATA_GATHERED = []
13
14 for p in P_VALUES:
15     #print(p)
16     new_adj = copy.deepcopy(ADJ_MATRIX)
17     for node in range(num_nodes):
18         neighbours=[]
19         for i in range(num_nodes):
20             if(ORIGINAL_ADJ_MATRIX[node][i]==1):
21                 neighbours.append(i)

```

```

22
23     for neighbour in neighbours:
24         if node < neighbour:
25             kya_tujhe_rewire_kru = random.random()
26             if(p > kya_tujhe_rewire_kru):
27                 new_adj[node][neighbour]=0
28                 new_adj[neighbour][node]=0
29
30             not_found = True
31
32             while not_found:
33                 random_edge = random.randint(0,num_nodes-1)
34
35                 if((random_edge !=node) and (new_adj[node][random_edge]==0)):
36                     new_adj[node][random_edge]=1
37                     new_adj[random_edge][node]=1
38                     not_found = False
39
40
41 mera_path_length = calculate_avg_path_length(new_adj)
42 mera_clustering_coeff = find_clustering_coefficient(new_adj)
43 DATA_GATHERED.append({p:[mera_path_length/ORIGINAL_PATH_LENGTH,mera_clustering_coeff/ORIGINAL_CLUSTERING_COEFF]})
44
45

```

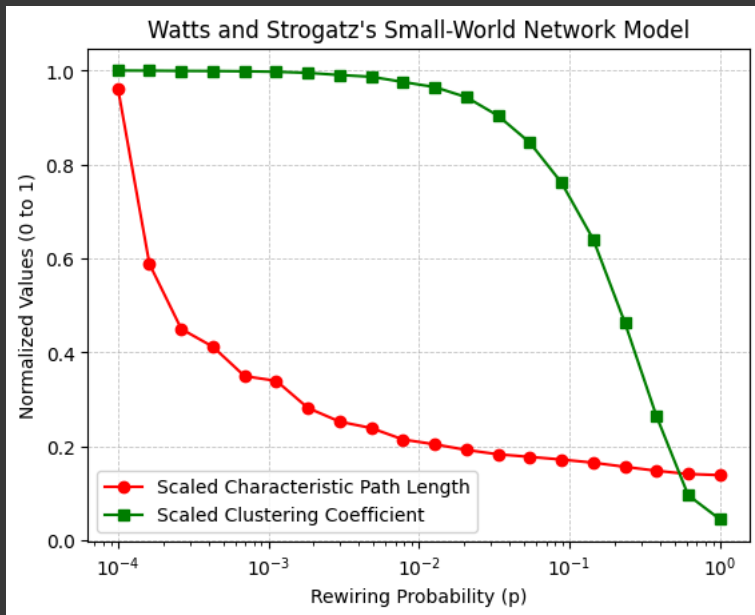
## Plotting The Data

- Next we simply plot the clustering coefficient obtained and the average path lengths using the matplotlib.

```

1 import matplotlib.pyplot as plt
2
3 all_path_lengths = []
4 all_clustering_coeff = []
5
6 for all_data in DATA_GATHERED:
7     dict = list(all_data.values())
8     path_length = dict[0][0]
9     clustering_coeff = dict[0][1]
10    all_path_lengths.append(path_length)
11    all_clustering_coeff.append(clustering_coeff)
12
13 plt.plot(P_VALUES, all_path_lengths, 'r-o', label='Scaled Characteristic Path Length')
14 plt.plot(P_VALUES, all_clustering_coeff, 'g-s', label='Scaled Clustering Coefficient')
15
16 plt.xscale('log')
17
18 plt.xlabel("Rewiring Probability (p)")
19 plt.ylabel("Normalized Values (0 to 1)")
20 plt.title("Watts and Strogatz's Small-World Network Model")
21
22 plt.grid(True, linestyle='--', linewidth=0.6, alpha=0.7)
23 plt.legend()
24 plt.show()

```



## Question 2

```

1 import random
2 import numpy as np
3 import math
4 from collections import deque
5 import matplotlib.pyplot as plt
6
7 # function to create random graph in starting
8 def create_random_graph(n, p=0.5):
9     graph = {i: [] for i in range(n)}
10
11     for i in range(n):
12         for j in range(i + 1, n):
13             if random.random() < p:
14                 graph[i].append(j)
15                 graph[j].append(i)
16
17     return graph
18
19
20 # BFS function to compute shortest path distances
21 def bfs(graph, starting, n):
22     distance_list = [math.inf] * n
23     visited = [0] * n
24     distance_list[starting] = 0
25     visited[starting] = 1
26     q = deque([starting])
27     while len(q) > 0:
28         currentnode = q.popleft()
29         for neigh in graph[currentnode]:
30             if visited[neigh] == 0:
31                 visited[neigh] = 1
32                 distance_list[neigh] = distance_list[currentnode] + 1
33                 q.append(neigh)
34     return distance_list
35
36 num_of_nodes = 500
37 edges = 8
38 iteration = 100
39
40 degree_distributions = []
41 clustering_coeffs = []
42 char_path_lengths = []
43
44
45 global_min_d = math.inf
46 global_max_d = 0
47
48 for iter in range(iteration):
49
50     graph = create_random_graph(10)
51
52     listofnodes = []
53     for i in range(10):
54         degree = len(graph[i])
55         listofnodes.extend([i] * degree)
56
57     # constructing the scale-free graph
58     for i in range(10, num_of_nodes):
59         neigh_of_newnode = []
60         while len(neigh_of_newnode) < edges:
61             n_choice = random.choice(listofnodes)
62             if n_choice not in neigh_of_newnode:
63                 neigh_of_newnode.append(n_choice)
64         graph[i] = neigh_of_newnode
65         for j in neigh_of_newnode:
66             graph[j].append(i)
67         listofnodes.extend([i] * edges)
68         for k in neigh_of_newnode:
69             listofnodes.append(k)
70
71     # Compute clustering coefficient
72     clusteringcoeff = []

```

```

73     for i in range(num_of_nodes):
74         neigh = graph[i]
75         degree = len(neigh)
76         if degree < 2:
77             clusteringcoff.append(0)
78             continue
79         link = 0
80         for j in neigh:
81             for k in neigh:
82                 if j != k and j in graph[k]:
83                     link += 1
84         link = link // 2
85         coff = (2 * link) / (degree * (degree - 1))
86         clusteringcoff.append(coff)
87     avg_clustering_coff = np.mean(clusteringcoff)
88     clustering_coeffs.append(avg_clustering_coff)
89
90     # Compute average path length using BFS
91     tot_dist = 0
92     pairs = 0
93     for i in range(num_of_nodes):
94         shortdist = bfs(graph, i, num_of_nodes)
95         for j in range(num_of_nodes):
96             if i != j and shortdist[j] != math.inf:
97                 tot_dist += shortdist[j]
98                 pairs += 1
99     avg_pathlen = tot_dist / pairs if pairs > 0 else math.inf
100     char_path_lengths.append(avg_pathlen)
101
102     # Compute degree distribution
103     degree_of_nodes = []
104     for node in graph:
105         degree_of_nodes.append(len(graph[node]))
106
107     # Update global min and max degrees
108     current_min = min(degree_of_nodes)
109     current_max = max(degree_of_nodes)
110     if current_min < global_min_d:
111         global_min_d = current_min
112     if current_max > global_max_d:
113         global_max_d = current_max
114
115     num_bins = current_max - current_min + 1
116     total_d = [0] * num_bins
117     for d in degree_of_nodes:
118         total_d[d - current_min] += 1
119     total_n = len(degree_of_nodes)
120     degree_dist = []
121     for c in total_d:
122         prob = c / total_n
123         degree_dist.append(prob)
124     degree_distributions.append(degree_dist)
125
126 # Finding the maximum length for padding
127 max_length = 0
128 for d in degree_distributions:
129     if len(d) > max_length:
130         max_length = len(d)
131
132 # Paddin to make the length same
133 degree_distributions_padded = []
134 for d in degree_distributions:
135     padding_length = max_length - len(d)
136     padded_d = d + [0] * padding_length
137     degree_distributions_padded.append(padded_d)
138
139 degree_distributions = np.array(degree_distributions_padded)
140
141 # Compute mean and standard deviation
142 degree_mean = np.mean(degree_distributions, axis=0)
143 degree_std = np.std(degree_distributions, axis=0)
144 degree_values = np.arange(global_min_d, global_min_d + len(degree_mean))
145
146
147 plt.figure(figsize=(12, 5))
148 # Plot degree distribution with standard dev
149 plt.subplot(1, 3, 1)
150 plt.scatter(degree_values, degree_mean, color='green', label='Mean', alpha=0.6)

```

```

150 plt.scatter(degree_values, degree_mean, color='green', label='Mean', alpha=0.6)
151 plt.scatter(degree_values, degree_std, color='red', label='Std Dev', alpha=0.6)
152 plt.xscale('log')
153 plt.yscale('log')
154 plt.xlabel('Degree (log scale)')
155 plt.ylabel('Probability (log scale)')
156 plt.title('Degree distribution (mean & std dev)')
157 plt.legend()
158
159 # Plot clustering coefficient for 100 runs
160 plt.subplot(1, 3, 2)
161 plt.plot(range(iteration), clustering_coeffs, marker='o', linestyle='', color='blue')
162 plt.xlabel('Graph instance')
163 plt.ylabel('Clustering coefficient')
164 plt.title('Clustering coefficient for 100 iterations')
165
166 # Plot characteristic path length for 100 runs
167 plt.subplot(1, 3, 3)
168 plt.plot(range(iteration), char_path_lengths, marker='o', linestyle='', color='red')
169 plt.xlabel('Graph instance')
170 plt.ylabel('Characteristic path length')
171 plt.title('Path length for 100 iterations')
172
173 plt.tight_layout()
174 plt.show()
175
176 print("Mean clustering coeff for 100 graph:", np.mean(clustering_coeffs))
177 print("Mean characteristic path length for 100 graph:", np.mean(char_path_lengths))
178

```



Mean clustering coeff for 100 graph: 0.08529248937016465

Mean characteristic path length for 100 graph: 2.4826926891534074

In this question i have implemented the BA algorithm for 500 nodes and each node will have 8 edges in the graph I took 8 because it is less then 10 as my initial seed random graph has only 10 nodes and the edge of 11th node should be less then or equal to 10. And all the topology here I have shown I have explained it in q3 I have made 100 graphs here and plotted the graphs for them. Kindly see the explanation present in the Q3.





### Question 3

```

1 import random
2 import numpy as np
3 import math
4 from collections import deque
5 import matplotlib.pyplot as plt
6
7 # function to create random graph in starting
8 def create_random_graph(n, p=0.5):
9     graph = {i: [] for i in range(n)}
10    for i in range(n):
11        for j in range(i + 1, n):
12            if random.random() < p:
13                graph[i].append(j)
14                graph[j].append(i)
15    return graph
16
17 # BFS function to compute shortest path distances
18 def bfs(graph, starting, n):
19     distance_list = [math.inf] * n
20     visited = [0] * n
21     distance_list[starting] = 0
22     visited[starting] = 1
23     q = deque([starting])
24     while len(q) > 0:
25         currentnode = q.popleft()
26         for neigh in graph[currentnode]:
27             if visited[neigh] == 0:
28                 visited[neigh] = 1
29                 distance_list[neigh] = distance_list[currentnode] + 1
30                 q.append(neigh)
31     return distance_list
32
33 num_of_nodes = 500
34 edges = 8
35 iteration = 100
36
37 degree_distributions = []
38 clustering_coeffs = []
39 char_path_lengths = []
40
41 global_min_d = math.inf
42 global_max_d = 0
43
44 for iter in range(iteration):
45
46     graph = create_random_graph(10)
47     listofnodes = []
48     for i in range(10):
49         degree = len(graph[i])
50         listofnodes.extend([i] * (degree ** 2))
51
52     # Constructing the scale-free graph using squared weighting
53     for i in range(10, num_of_nodes):
54         new_neighbors = []
55         while len(new_neighbors) < edges:
56             n_choice = random.choice(listofnodes)
57             if n_choice not in new_neighbors:
58                 new_neighbors.append(n_choice)
59         graph[i] = new_neighbors
60         for j in new_neighbors:
61             graph[j].append(i)
62         listofnodes.extend([i] * (edges ** 2))
63
64         for k in new_neighbors:
65             d_old = len(graph[k]) - 1
66             listofnodes.extend([k] * (2 * d_old + 1))
67
68     # Compute clustering coefficient
69     clusteringcoeff = []
70     for i in range(num_of_nodes):
71         neigh = graph[i]
72         degree = len(neigh)

```

```

73     if degree < 2:
74         clusteringcoff.append(0)
75         continue
76     link = 0
77     for j in neigh:
78         for k in neigh:
79             if j != k and j in graph[k]:
80                 link += 1
81     link = link // 2
82     coff = (2 * link) / (degree * (degree - 1))
83     clusteringcoff.append(coff)
84     avg_clustering_coff = np.mean(clusteringcoff)
85     clustering_coeffs.append(avg_clustering_coff)
86
87     # Compute average path length using BFS
88     tot_dist = 0
89     pairs = 0
90     for i in range(num_of_nodes):
91         shortdist = bfs(graph, i, num_of_nodes)
92         for j in range(num_of_nodes):
93             if i != j and shortdist[j] != math.inf:
94                 tot_dist += shortdist[j]
95                 pairs += 1
96     avg_pathlen = tot_dist / pairs if pairs > 0 else math.inf
97     char_path_lengths.append(avg_pathlen)
98
99     # Compute degree distribution
100    degree_of_nodes = []
101    for node in graph:
102        degree_of_nodes.append(len(graph[node]))
103
104    # Update global min and max degrees
105    current_min = min(degree_of_nodes)
106    current_max = max(degree_of_nodes)
107    if current_min < global_min_d:
108        global_min_d = current_min
109    if current_max > global_max_d:
110        global_max_d = current_max
111
112    num_bins = current_max - current_min + 1
113    total_d = [0] * num_bins
114    for d in degree_of_nodes:
115        total_d[d - current_min] += 1
116    total_n = len(degree_of_nodes)
117    degree_dist = []
118    for c in total_d:
119        prob = c / total_n
120        degree_dist.append(prob)
121    degree_distributions.append(degree_dist)
122
123    # Padding the degree distributions so that all have the same length
124    max_length = 0
125    for d in degree_distributions:
126        if len(d) > max_length:
127            max_length = len(d)
128
129    degree_distributions_padded = []
130    for d in degree_distributions:
131        padding_length = max_length - len(d)
132        padded_d = d + [0] * padding_length
133        degree_distributions_padded.append(padded_d)
134
135    degree_distributions = np.array(degree_distributions_padded)
136
137    # Compute mean and standard deviation of the degree distributions
138    degree_mean = np.mean(degree_distributions, axis=0)
139    degree_std = np.std(degree_distributions, axis=0)
140    degree_values = np.arange(global_min_d, global_min_d + len(degree_mean))
141
142    plt.figure(figsize=(12, 5))
143    # Plot degree distribution with standard deviation
144    plt.subplot(1, 3, 1)
145    plt.scatter(degree_values, degree_mean, color='green', label='Mean', alpha=0.6)
146    plt.scatter(degree_values, degree_std, color='red', label='Std Dev', alpha=0.6)
147    plt.xscale('log')
148    plt.yscale('log')
149    plt.xlabel('Degree (log scale)')
150    plt.ylabel('Probability (log scale)')

```

```

150 plt.plot(range(iteration), probability, (log scale))
151 plt.title('Degree distribution (mean and std dev)')
152 plt.legend()
153
154 # Plot clustering coefficient for 100 runs
155 plt.subplot(1, 3, 2)
156 plt.plot(range(iteration), clustering_coeffs, marker='o', linestyle='', color='blue')
157 plt.xlabel('Graph instance')
158 plt.ylabel('Clustering coefficient')
159 plt.title('Clustering coefficient for 100 iteration')
160
161 # Plot characteristic path length for 100 runs
162 plt.subplot(1, 3, 3)
163 plt.plot(range(iteration), char_path_lengths, marker='o', linestyle='', color='red')
164 plt.xlabel('Graph Instance')
165 plt.ylabel('Characteristic path length')
166 plt.title('Path Length for 100 iteration')
167
168 plt.tight_layout()
169 plt.show()
170
171 print("Mean clustering coeff for 100 graphs:", np.mean(clustering_coeffs))
172 print("Mean characteristic path length for 100 graphs:", np.mean(char_path_lengths))
173

```



Mean clustering coeff for 100 graphs: 0.6341816309581119

Mean characteristic path length for 100 graphs: 1.9707692918366855

Comparison between the BA algorithm and the modified BA algorithm 1) The clustering coefficient in the BA algorithm is not so high meaning they don't have larger values indicating a little less connectivity between the nodes it is not like real world scale free because in real world we have more biases than just the probability we are taking into account here but in the modified BA algorithm, where the probability of a new node getting connected to another node is proportional to the square of its degree, the values of the clustering coefficient are too large suggesting that the network is highly connected which I think is expected as in the modified algorithm we increased the bias even more so the hubs will be even denser, the new nodes will tend to connect to the hubs even more than the BA algorithm forming an even denser network increasing the clustering coefficient somehow it reassembles the real world scale free network. 2) Path length in the BA algorithm is like any real world scale free network it is less than 3 for my graph as the number of nodes is less but I think if we take a large number of nodes it will go up to 5 or 6, that is six degrees of separation. But in the Modified BA algorithm the path length is much less it is less than 2 for the same number of nodes as the real BA algorithm that I used in q2, which is again as we should expect as in this question the network will be so dense so the average path length will obviously decrease, as most of the nodes will be connected to hubs. 3) The degree distribution in the BA algorithm is just like a real world network, we have some hubs having large degree with some possible probability, but we have outliers also with less degree but in the modified BA most of the nodes are hubs or part of hubs so most of them have a large degree which is not like real world, and nodes having small degree are pretty much less, the standard deviation is also more in this case because the modified BA can create a more extreme distribution.

```

1 import random
2 import numpy as np

```

```

3 import math
4 from collections import deque
5 import matplotlib.pyplot as plt
6
7 # function to create random graph in starting
8 def create_random_graph(n, p=0.5):
9     graph = {i: [] for i in range(n)}
10    for i in range(n):
11        for j in range(i + 1, n):
12            if random.random() < p:
13                graph[i].append(j)
14                graph[j].append(i)
15    return graph
16
17 # BFS function to compute shortest path distances
18 def bfs(graph, starting, n):
19     distance_list = [math.inf] * n
20     visited = [0] * n
21     distance_list[starting] = 0
22     visited[starting] = 1
23     q = deque([starting])
24     while len(q) > 0:
25         currentnode = q.popleft()
26         for neigh in graph[currentnode]:
27             if visited[neigh] == 0:
28                 visited[neigh] = 1
29                 distance_list[neigh] = distance_list[currentnode] + 1
30                 q.append(neigh)
31    return distance_list
32
33 num_of_nodes = 100
34 edges = 2
35 iteration = 100
36
37 degree_distributions = []
38 clustering_coeffs = []
39 char_path_lengths = []
40
41 global_min_d = math.inf
42 global_max_d = 0
43
44 for iter in range(iteration):
45
46     graph = create_random_graph(10)
47
48     listofnodes = []
49     for i in range(10):
50         degree = len(graph[i])
51         listofnodes.extend([i] * (degree ** 4))
52
53     # Constructing the scale-free graph for power 4
54     for i in range(10, num_of_nodes):
55         new_neighbors = []
56         while len(new_neighbors) < edges:
57             n_choice = random.choice(listofnodes)
58             if n_choice not in new_neighbors:
59                 new_neighbors.append(n_choice)
60         graph[i] = new_neighbors
61         for j in new_neighbors:
62             graph[j].append(i)
63         listofnodes.extend([i] * (edges ** 4))
64         for k in new_neighbors:
65             d_old = len(graph[k]) - 1
66             increment = 4 * (d_old ** 3) + 6 * (d_old ** 2) + 4 * d_old + 1
67             listofnodes.extend([k] * increment)
68
69     # Compute clustering coefficient
70     clusteringcoeff = []
71     for i in range(num_of_nodes):
72         neigh = graph[i]
73         degree = len(neigh)
74         if degree < 2:
75             clusteringcoeff.append(0)
76             continue
77         link = 0
78         for j in neigh:
79             for k in neigh:

```

```

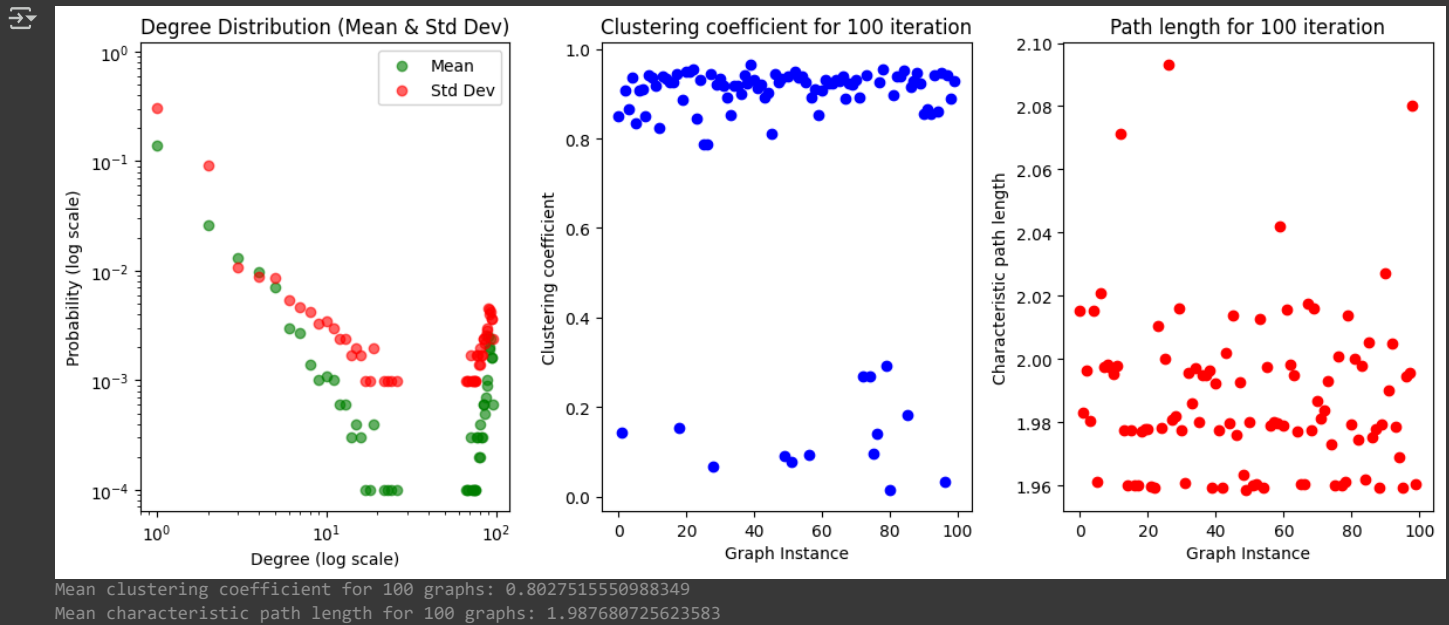
80         if j != k and j in graph[k]:
81             link += 1
82         link = link // 2
83         coff = (2 * link) / (degree * (degree - 1))
84         clusteringcoff.append(coff)
85     avg_clustering_coff = np.mean(clusteringcoff)
86     clustering_coeffs.append(avg_clustering_coff)
87
88     # Compute average path length using BFS
89     tot_dist = 0
90     pairs = 0
91     for i in range(num_of_nodes):
92         dists = bfs(graph, i, num_of_nodes)
93         for j in range(num_of_nodes):
94             if i != j and dists[j] != math.inf:
95                 tot_dist += dists[j]
96                 pairs += 1
97     avg_pathlen = tot_dist / pairs if pairs > 0 else math.inf
98     char_path_lengths.append(avg_pathlen)
99
100    # Compute degree distribution
101    degree_of_nodes = []
102    for node in graph:
103        degree_of_nodes.append(len(graph[node]))
104
105    # Update global min and max degrees
106    current_min = min(degree_of_nodes)
107    current_max = max(degree_of_nodes)
108    if current_min < global_min_d:
109        global_min_d = current_min
110    if current_max > global_max_d:
111        global_max_d = current_max
112
113    num_bins = current_max - current_min + 1
114    total_d = [0] * num_bins
115    for d in degree_of_nodes:
116        total_d[d - current_min] += 1
117    total_n = len(degree_of_nodes)
118    degree_dist = [c / total_n for c in total_d]
119    degree_distributions.append(degree_dist)
120
121    # Padding the degree distributions so that all have the same length
122    max_length = 0
123    for d in degree_distributions:
124        if len(d) > max_length:
125            max_length = len(d)
126
127    degree_distributions_padded = []
128    for d in degree_distributions:
129        padding_length = max_length - len(d)
130        padded_d = d + [0] * padding_length
131        degree_distributions_padded.append(padded_d)
132
133    degree_distributions = np.array(degree_distributions_padded)
134
135    # Compute mean and standard deviation of the degree distributions
136    degree_mean = np.mean(degree_distributions, axis=0)
137    degree_std = np.std(degree_distributions, axis=0)
138    degree_values = np.arange(global_min_d, global_min_d + len(degree_mean))
139
140    plt.figure(figsize=(12, 5))
141    # Plot degree distribution with standard dev
142    plt.subplot(1, 3, 1)
143    plt.scatter(degree_values, degree_mean, color='green', label='Mean', alpha=0.6)
144    plt.scatter(degree_values, degree_std, color='red', label='Std Dev', alpha=0.6)
145    plt.xscale('log')
146    plt.yscale('log')
147    plt.xlabel('Degree (log scale)')
148    plt.ylabel('Probability (log scale)')
149    plt.title('Degree Distribution (Mean & Std Dev)')
150    plt.legend()
151
152    # Plot clustering coefficient for 100 iteration
153    plt.subplot(1, 3, 2)
154    plt.plot(range(iteration), clustering_coeffs, marker='o', linestyle='', color='blue')
155    plt.xlabel('Graph Instance')
156    plt.ylabel('Clustering coefficient')

```

```

157 plt.title('Clustering coefficient for 100 iteration')
158
159 # Plot characteristic path length for 100 iteration
160 plt.subplot(1, 3, 3)
161 plt.plot(range(iteration), char_path_lengths, marker='o', linestyle='', color='red')
162 plt.xlabel('Graph Instance')
163 plt.ylabel('Characteristic path length')
164 plt.title('Path length for 100 iteration')
165
166 plt.tight_layout()
167 plt.show()
168
169 print("Mean clustering coefficient for 100 graphs:", np.mean(clustering_coeffs))
170 print("Mean characteristic path length for 100 graphs:", np.mean(char_path_lengths))
171

```



Here i have modified the BA algorithm for power 4 Here the bias is even more then power 2 , before I was running the code for 500 nodes but here i have ran the code for only 100 nodes and only 2 edges as i was getting memory overflow for large number of nodes in this question because the preferential attachment list grows extremely fast when using power 4 weighting , so we can not compare it directly with q2 or q3 first part but stil we can see that most of the nodes have high degree , most of the nodes have clustering coffecient equal to 1 , path lenght is so small even after having only 2 edges.

## Question 4

### PART-A

#### Importing the Necessary Libraries

```
1 import networkx as nx
2 import numpy as np
3 import random
4 import matplotlib.pyplot as plt
5 np.random.seed(322034345)
6 random.seed(322034345)
```

In this part we are basically asked to implement node deletion strategy in a network. Here is how we implemented it. First we created a function named `deleteRandomNode`: This function randomly removes nodes from a network and measures the characteristic path length and the size of the giant component for different fractions of node removal.

Parameters: `adj_matrix`: `numpy.ndarray` - The adjacency matrix representing the input network.

Flow of code: First we created a graph `G` from the input adjacency matrix. Then defined a range of fractions (0 to 0.5) for node removal.

```
Now For each fraction of node removal (from 0% to 50%):
    Randomly select and remove nodes.
    Identify the largest connected component (Giant Cluster).
    Calculate the characteristic path length.
```

Then it outputs lists of characteristic path lengths and giant cluster sizes for each fraction. In the code it returns the `path_lengths` and `giant_cluster_sizes`

Two graphs were plotted: 1. Characteristic Path Length vs fraction of nodes deleted 2. Giant Cluster Size vs fraction of nodes deleted

Observation: 1. At first the path length remains stable after that there is no significant increase or sharp rise suggesting that network maintains its connectedness despite random deletions. 2. the size of giant cluster decreases linearly as the fraction of node removal increases. Even after removing 50% of the nodes the network remains partially connected.

Conclusion: The network remains robust under random node deletions, with minimal impact on characteristic path length. The giant cluster size decreases gradually, indicating no sudden network collapse.

```
1
2 def deleteRandomNode(adj_matrix, flag):
3     if(flag==0):
4         fractions = np.linspace(0, 0.05, 10)
5     else:
6         fractions = np.linspace(0, 0.5, 60)
7     G = nx.from_numpy_array(adj_matrix)
8     nodes = list(G.nodes())
9
10    path_lengths = []
11    giant_cluster_sizes = []
12    initial_largest_cc_size = len(max(nx.connected_components(G), key=len))
13
14    for f in fractions:
15        G_copy = G.copy()
16        num_remove = int(f * len(nodes))
17        remove_nodes = random.sample(nodes, num_remove)
18        G_copy.remove_nodes_from(remove_nodes)
19
20        b = nx.connected_components(G_copy)
21        components = list(b)
22        if components:
23            largest_cc = max(components, key=len)
24            H = G_copy.subgraph(largest_cc)
25            if(flag==0):
26                if nx.is_connected(H):
27                    a = nx.average_shortest_path_length(H)
28                    path_lengths.append(a)
29                else:
30                    path_lengths.append(np.nan)
31            else:
```



```

32         giant_cluster_sizes.append(len(largest_cc)/initial_largest_cc_size)
33     else:
34         giant_cluster_sizes.append(0)
35     path_lengths.append(np.nan)
36 if(flag==0):
37     return path_lengths, fractions
38 else:
39     return giant_cluster_sizes, fractions
40
41
42 def deleteTargetNode(G,flag):
43     if(flag==0):
44         fractions = np.linspace(0, 0.05, 10)
45     else:
46         fractions = np.linspace(0, 0.5, 60)
47     nodes = sorted(G.nodes(), key=lambda x: G.degree[x], reverse=True) # Sort by degree
48
49     path_lengths = []
50     giant_cluster_sizes = []
51     initial_largest_cc_size = len(max(nx.connected_components(G), key=len))
52
53     for f in fractions:
54         G_copy = G.copy()
55         num_remove = int(f * len(nodes))
56         remove_nodes = nodes[:num_remove] # Remove highest-degree nodes first
57         G_copy.remove_nodes_from(remove_nodes)
58
59         components = list(nx.connected_components(G_copy))
60         if components:
61             largest_cc = max(components, key=len)
62             H = G_copy.subgraph(largest_cc)
63             if(flag==0):
64                 if nx.is_connected(H):
65                     path_lengths.append(nx.average_shortest_path_length(H))
66                 else:
67                     path_lengths.append(np.nan)
68             else:
69                 giant_cluster_sizes.append(len(largest_cc)/initial_largest_cc_size)
70         else:
71             giant_cluster_sizes.append(0)
72             path_lengths.append(np.nan)
73     if(flag==0):
74         return path_lengths, fractions
75     else:
76         return giant_cluster_sizes, fractions
77
78
79

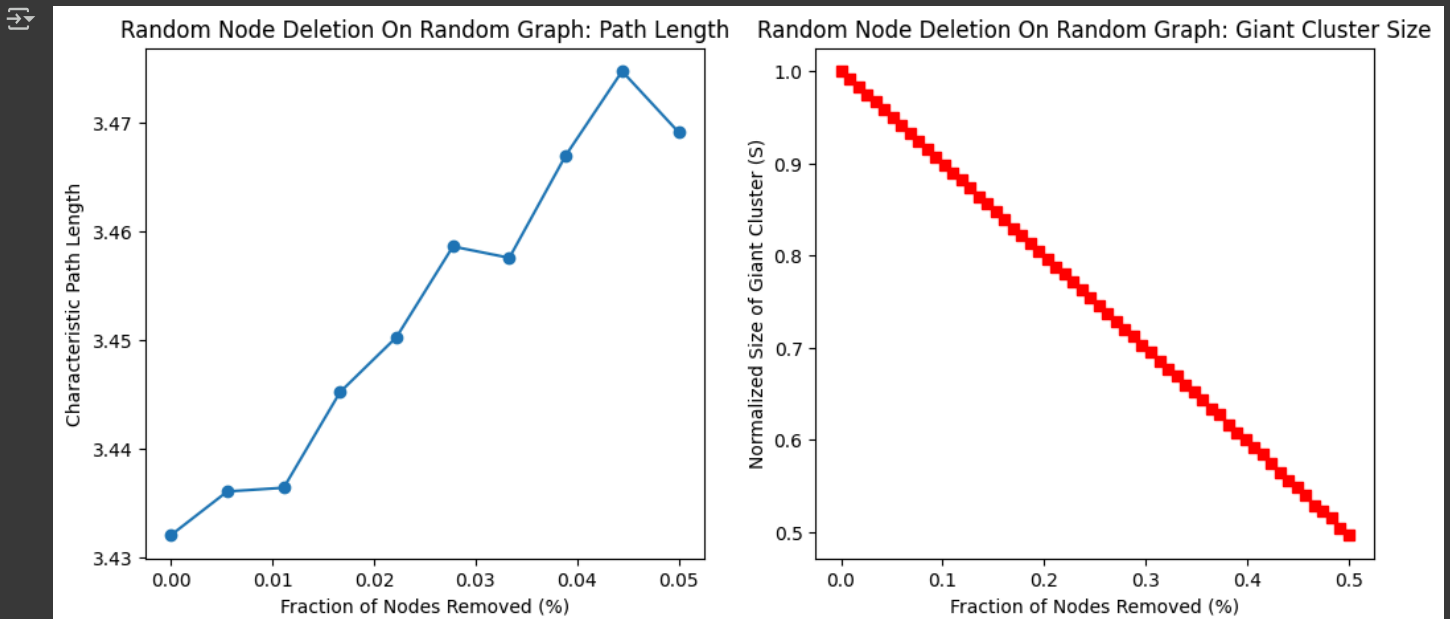
```

```

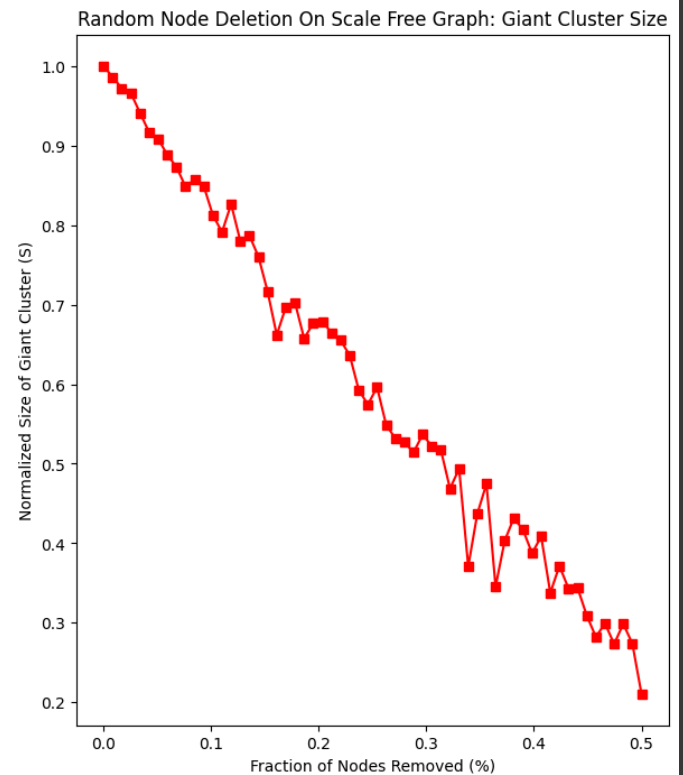
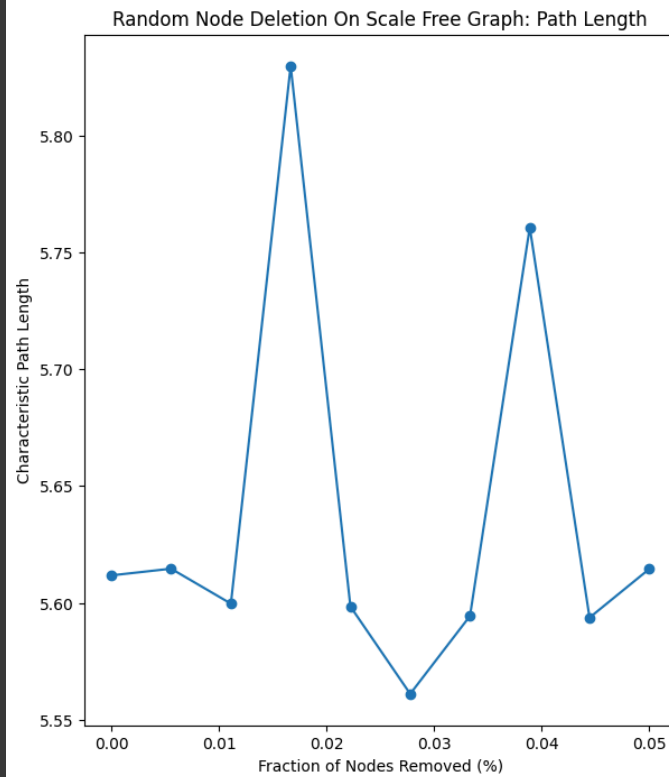
1 NUM_NODES = 1500
2 AVG_DEGREE = 10
3 P = AVG_DEGREE / NUM_NODES
4
5 # Generate Erdős-Rényi random graph
6 random_graph = nx.erdos_renyi_graph(NUM_NODES, P)
7
8 adj_matrix = nx.to_numpy_array(random_graph)
9
10 # chlo random k liye
11 path_lengths, path_fractions = deleteRandomNode(adj_matrix,0)
12 cluster_sizes, cluster_fractions = deleteRandomNode(adj_matrix,1)
13
14 plt.figure(figsize=(12, 5))
15
16 plt.subplot(1, 2, 1)
17 plt.plot(path_fractions, path_lengths, marker='o', linestyle='--')
18 plt.xlabel("Fraction of Nodes Removed (%)")
19 plt.ylabel("Characteristic Path Length")
20 plt.title("Random Node Deletion On Random Graph: Path Length")
21
22 # Giant Cluster Size Plot
23 plt.subplot(1, 2, 2)
24 plt.plot(cluster_fractions, cluster_sizes, marker='s', color='r', linestyle='--')
25 plt.xlabel("Fraction of Nodes Removed (%)")
26 plt.ylabel("Normalized Size of Giant Cluster (S)")
27 plt.title("Random Node Deletion On Random Graph: Giant Cluster Size")

```

```
28
29 plt.show()
```



```
1 NUM_NODES = 1000
2 M = 2
3
4 real_network = nx.read_edgelist("numerical_yeast_data.csv", nodetype=int, delimiter=",")
5 adj_matrix = nx.to_numpy_array(real_network)
6
7 # chlo scale free k liye
8 path_lengths, path_fractions = deleteRandomNode(adj_matrix, 0)
9 cluster_sizes, cluster_fractions = deleteRandomNode(adj_matrix, 1)
10
11 plt.figure(figsize=(15, 8))
12
13 plt.subplot(1, 2, 1)
14 plt.plot(path_fractions, path_lengths, marker='o', linestyle='--')
15 plt.xlabel("Fraction of Nodes Removed (%)")
16 plt.ylabel("Characteristic Path Length")
17 plt.title("Random Node Deletion On Scale Free Graph: Path Length")
18
19 # Giant Cluster Size Plot
20 plt.subplot(1, 2, 2)
21 plt.plot(cluster_fractions, cluster_sizes, marker='s', color='r', linestyle='--')
22 plt.xlabel("Fraction of Nodes Removed (%)")
23 plt.ylabel("Normalized Size of Giant Cluster (S)")
24 plt.title("Random Node Deletion On Scale Free Graph: Giant Cluster Size")
25
26 plt.show()
```



## ▼ PART-B

This part is basically the extended version of the first part, we'll use both the `deleteRandomNode` function and `deleteTargetNode` (we'll make this in this part) function.

Objective of this part is To compare the impact of node deletions on random and scale-free networks under two strategies Random Deletion: Nodes are removed uniformly at random. Targeted Deletion: High-degree nodes are removed first.

First generate two kinds of networks: Random Graph: Erdős-Rényi model (`nx.erdos_renyi_graph`) Scale-Free Graph: Barabási-Albert model (`nx.barabasi_albert_graph`)

Then we apply both the deletion strategy to both the graphs Random Deletion: Use `deleteRandomNode()` function. Targeted Deletion: Implement `deleteTargetNode()` function.

`deleteTargetNode` Function: Inputs: `G`: A NetworkX graph. `fractions`: List of fractions of nodes to remove (e.g., [0.1, 0.2, 0.5]). Outputs: `path_lengths`: Average shortest path length of the largest connected component after each fraction of node removal. `giant_cluster_sizes`: Size of the largest connected component after each fraction. Process: Sort nodes by degree (highest first). For each fraction: Remove corresponding nodes. Find the largest connected component. Record its size and average shortest path length (or NaN if disconnected).

```

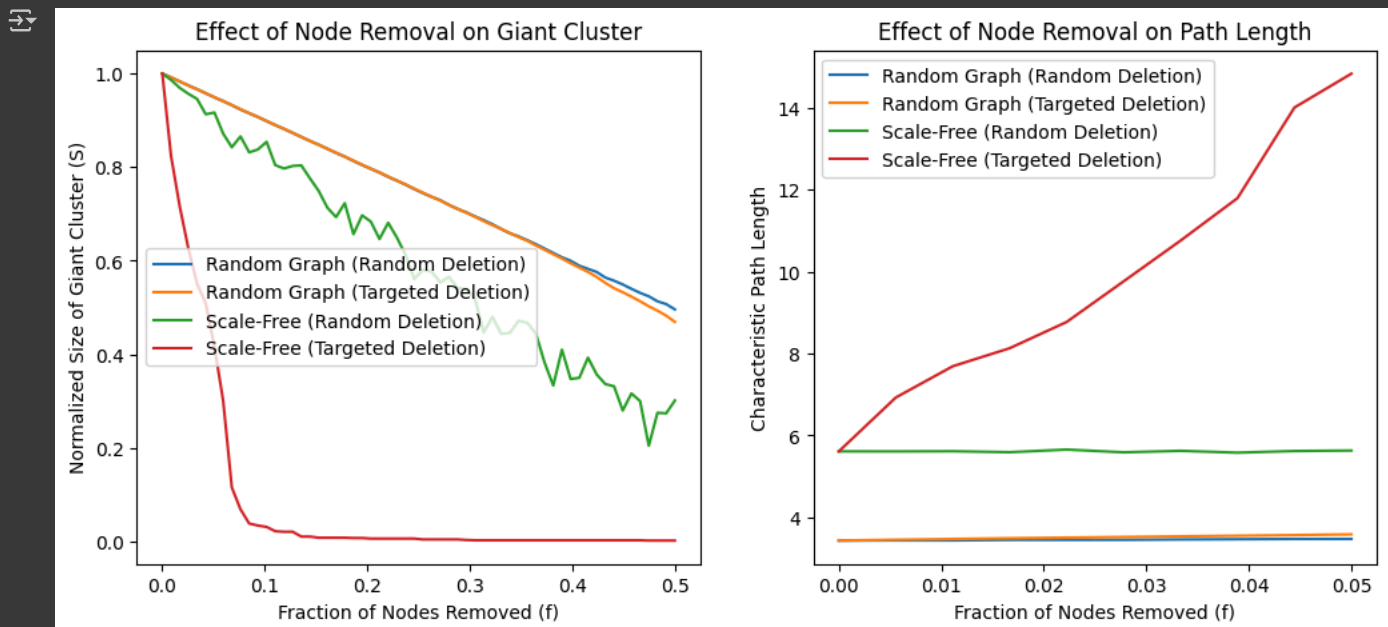
1 adj_matrix_random = nx.to_numpy_array(random_graph)
2 adj_matrix_real = nx.to_numpy_array(real_network)
3
4 # chlo random k liye
5 path_lengths_random_random, path_fractions = deleteRandomNode(adj_matrix_random,0)
6 cluster_sizes_random_random, cluster_fractions = deleteRandomNode(adj_matrix_random,1)
7 path_lengths_random_target, path_fractions = deleteTargetNode(random_graph,0)
8 cluster_sizes_random_target, cluster_fractions = deleteTargetNode(random_graph,1)
9
10
11 #chlo real k liye
12 path_lengths_real_random, path_fractions = deleteRandomNode(adj_matrix_real,0)
13 cluster_sizes_real_random, cluster_fractions = deleteRandomNode(adj_matrix_real,1)
14 path_lengths_real_target, path_fractions = deleteTargetNode(real_network,0)
15 cluster_sizes_real_target, cluster_fractions = deleteTargetNode(real_network,1)
16
17
18
19 plt.figure(figsize=(12, 5))
20 plt.subplot(1, 2, 1)

```

```

21 plt.plot(cluster_fractions, cluster_sizes_random_random, label="Random Graph (Random Deletion)")
22 plt.plot(cluster_fractions, cluster_sizes_random_target, label="Random Graph (Targeted Deletion)")
23 plt.plot(cluster_fractions, cluster_sizes_real_random, label="Scale-Free (Random Deletion)")
24 plt.plot(cluster_fractions, cluster_sizes_real_target, label="Scale-Free (Targeted Deletion)")
25 plt.xlabel("Fraction of Nodes Removed (f)")
26 plt.ylabel("Normalized Size of Giant Cluster (S)")
27 plt.title("Effect of Node Removal on Giant Cluster")
28 plt.legend()
29
30 plt.subplot(1, 2, 2)
31 plt.plot(path_fractions, path_lengths_random_random, label="Random Graph (Random Deletion)")
32 plt.plot(path_fractions, path_lengths_random_target, label="Random Graph (Targeted Deletion)")
33 plt.plot(path_fractions, path_lengths_real_random, label="Scale-Free (Random Deletion)")
34 plt.plot(path_fractions, path_lengths_real_target, label="Scale-Free (Targeted Deletion)")
35 plt.xlabel("Fraction of Nodes Removed (f)")
36 plt.ylabel("Characteristic Path Length")
37 plt.title("Effect of Node Removal on Path Length")
38 plt.legend()
39
40 plt.show()
41
42

```



Analysis and Plots(Of all 4 pairs):

Plot characteristic path length vs. fraction of nodes removed. Plot giant cluster size vs. fraction of nodes removed.

Observations: - Random graphs and scale-free networks show a gradual decrease under random deletion. Targeted deletion in scale-free networks causes a rapid breakdown, reducing the giant cluster size sharply.

- Random deletion causes path length to remain constant for both network types.  
Targeted deletion in scale-free networks causes an exponential rise, indicating severe disruption in connectivity.

Conclusions: - Scale-free networks are highly vulnerable to targeted attacks, while random graphs degrade gradually under both deletion strategies. - Hubs in scale-free networks are critical—removing them rapidly fragments the network and increases path length. - This supports the Albert et al. (2000) findings—random failures have less impact, while targeted deletions quickly disintegrate scale-free networks.

## ✓ PART-C

In this, we are performing above analysis on Facebook scale-free network.

Methodology: Dataset: Load a real-world network (e.g., facebook\_combined.txt). Apply Deletion Strategies: Random Node Deletion. Targeted Node Deletion.

Implementation: Load the network using nx.read\_edgelist(). Perform random and targeted deletions. Plot the results for giant cluster size and characteristic path length.

## PART-D

Alignment with Albert et al., Nature (2000)

Characteristic Path Length: Random graphs: Increases gradually in both strategies. Scale-free networks: Remain stable under random deletion but grow rapidly under targeted deletion.

Size of Giant Cluster (S): Random graphs: Gradual reduction for both strategies. Scale-free networks: Stable under random deletion but collapse under targeted deletion, confirming findings by Albert et al.

```

1
2 # Load real-world dataset (Example: Power Grid Network)
3 real_world_nw = nx.read_edgelist("facebook_combined.txt", nodetype=int) # Adjust file path
4 adj_matrix_real = nx.to_numpy_array(real_world_nw)
5
6 path_lengths_real_random, path_fractions = deleteRandomNode(adj_matrix_real,0)
7 cluster_sizes_real_random, cluster_fractions = deleteRandomNode(adj_matrix_real,1)
8 path_lengths_real_target, path_fractions = deleteTargetNode(real_world_nw,0)
9 cluster_sizes_real_target, cluster_fractions = deleteTargetNode(real_world_nw,1)
10
11
12
13 plt.figure(figsize=(12, 5))
14 plt.subplot(1, 2, 1)
15 plt.plot(cluster_fractions, cluster_sizes_real_random, label="Random Deletion")
16 plt.plot(cluster_fractions, cluster_sizes_real_target, label="Targeted Deletion")
17 plt.xlabel("Fraction of Nodes Removed (f)")
18 plt.ylabel("Normalized Size of Giant Cluster (S)")
19 plt.title("Real-World Network: Giant Cluster Size")
20 plt.legend()
21
22 plt.subplot(1, 2, 2)
23 plt.plot(path_fractions, path_lengths_real_random, label="Random Deletion")
24 plt.plot(path_fractions, path_lengths_real_target, label="Targeted Deletion")
25 plt.xlabel("Fraction of Nodes Removed (f)")
26 plt.ylabel("Characteristic Path Length")
27 plt.title("Real-World Network: Path Length")
28 plt.legend()
29
30 plt.show()
31

```



## Question 5

- Here the network we are using is: Yeast Interactome.
- Here's the link for the same: [http://interactome.dfci.harvard.edu/S\\_cerevisiae/download/Y2H\\_union.txt](http://interactome.dfci.harvard.edu/S_cerevisiae/download/Y2H_union.txt)

### RAW DATA TO CSV DATA

We are given the network in a `.txt` file containing raw data in the format: `node1 node2`, representing an undirected edge between `node1` and `node2`.

To process this data efficiently, we convert it into a `.csv` file with two columns: `u` and `v`.

```
1 import pandas as pd
2
3 with open("yeast_interactome_dataset.txt") as file:
4     filtered_data=[]
5     data=file.readlines()
6     for item in data:
7         item=item.strip()
8         item=item.split(' ')
9         filtered_data.append([item[0],item[1]])
10    dataframe=pd.DataFrame(filtered_data)
11    dataframe.columns=["u","v"]
12    dataframe.to_csv("yeast_data.csv",index=False)
13
```

## PART-A

### Representing the Yeast Interactome In a Adjacency Matrix

```
1 dataframe = pd.read_csv("yeast_data.csv")
2
3 PROTEIN_TO_NUM_MAP = {}
4 counter=0
5 for index,row in dataframe.iterrows():
6     u=row.iloc[0]
7     v=row.iloc[1]
8     if(PROTEIN_TO_NUM_MAP.get(u)==None):
9         PROTEIN_TO_NUM_MAP[u]=counter
10        counter+=1
11    if(PROTEIN_TO_NUM_MAP.get(v)==None):
12        PROTEIN_TO_NUM_MAP[v]=counter
13        counter+=1
14
15 ADJ_SIZE = counter
16 ADJ_MATRIX=[]
17 ADJ_ROW=[]
18 for i in range(ADJ_SIZE):
19     ADJ_ROW.append(0)
20
21 for i in range(ADJ_SIZE):
22     ADJ_MATRIX.append(list(ADJ_ROW))
23
24 for index,row in dataframe.iterrows():
25     u=row.iloc[0]
26     v=row.iloc[1]
27     u_num = PROTEIN_TO_NUM_MAP[u]
28     v_num = PROTEIN_TO_NUM_MAP[v]
29     ADJ_MATRIX[u_num][v_num]=1
30     ADJ_MATRIX[v_num][u_num]=1
```

## Plotting the Degree Distribution on a log-log Scale

To obtain the probabilities that a node has degree `k`, we first find the maximum possible degree in our network, by simply summing across each row in adjacency matrix and then finding the maximum across them.

Next, we precompute the degrees of each node by the same logic as used in the previous step and store them at `kth` index in a array.

Now, we run a simple for loop across the array and get the total number of nodes for that degree and then we divide this by total number of nodes present in our network.

Effective Formula:  $P(k) = N_k/N$

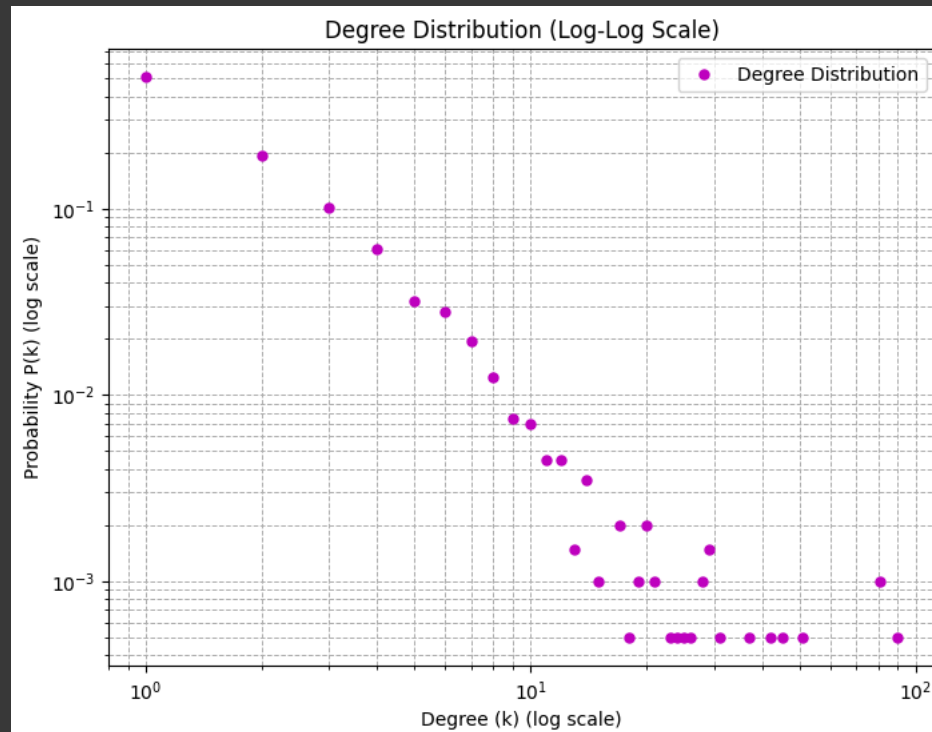
Next, we simply plot this data using matplotlib and label the axes as well.

The degree distribution confirms the presence of hubs in the Yeast Network.

```

1 # degree distribution
2 import matplotlib.pyplot as plt
3
4
5 def max_possible_degree(adj_matrix):
6     degrees=[]
7     for i in range(len(adj_matrix)):
8         my_degree=0
9         for j in range(len(adj_matrix)):
10             if(adj_matrix[i][j]==1):
11                 my_degree+=1
12         degrees.append(my_degree)
13     max_val = max(degrees)
14     return max_val
15
16 def calculate_nodes_with_degree(adj_matrix):
17     degrees=[0]*len(adj_matrix)
18     for i in range(len(adj_matrix)):
19         for j in range(len(adj_matrix)):
20             if(adj_matrix[i][j]==1):
21                 degrees[i]+=1
22     return degrees
23
24 max_degree_in_network=max_possible_degree(ADJ_MATRIX)
25
26 node_degrees=calculate_nodes_with_degree(ADJ_MATRIX)
27 probabiltiy_and_degree_data=[]
28
29 degree_count = [0] * (max_degree_in_network + 1)
30
31 for i in range(len(node_degrees)):
32     degree_count[node_degrees[i]] += 1
33
34 for i in range(max_degree_in_network+1):
35     pk = degree_count[i] / len(ADJ_MATRIX)
36     probabiltiy_and_degree_data.append([pk, i])
37
38 probabilities, degrees = zip(*probabiltiy_and_degree_data)
39
40 plt.figure(figsize=(8, 6))
41 plt.loglog(degrees, probabilities, 'mo', linewidth=1, markersize=5, label='Degree Distribution')
42
43 plt.title('Degree Distribution (Log-Log Scale)')
44 plt.xlabel('Degree (k) (log scale)')
45 plt.ylabel('Probability P(k) (log scale)')
46 plt.grid(True, which="both", linestyle='--', linewidth=0.7)
47 plt.legend()
48 plt.savefig('q5_part(a)_degree_distribution_loglog.png', dpi=1000, bbox_inches='tight')
49 plt.show()
50

```



### Plotting Clustering Coefficient vs Degree

Formula used:-

- Clustering Coefficient (C) =  $(2 * \text{Number of Actual Edges Between Neighbors}) / (\text{Degree of Node} * (\text{Degree of Node} - 1))$

Where:

- Number of Actual Edges Between Neighbors** is the number of edges that actually exist between the neighbors of the node.
- Degree of Node** is the number of neighbors connected to the node.

After Obtaining the Clustering Coefficient for a set of degrees, we simply stored them and then plotted them using matplotlib.

```
1 import matplotlib.pyplot as plt
2 from collections import defaultdict
3 import numpy as np
4
5 def find_clustering_coefficient(adj_matrix):
6     num_nodes=len(adj_matrix)
7
8     all_data=[] # coefficient,degree
9     for src in range(num_nodes):
10         degree=0
11         neighbours=[]
12         common_edge=0
13         for dst in range(num_nodes):
14             if(adj_matrix[src][dst]==1):
15                 degree+=1
16                 neighbours.append(dst)
17
18         if(degree < 2):
19             all_data.append([0,degree])
20             continue
21
22         for i in range(degree):
23             for j in range(i + 1, degree):
24                 if adj_matrix[neighbours[i]][neighbours[j]] == 1:
25                     common_edge += 1
26
27         coefficient=(2*(common_edge))/(degree*(degree-1))
28         all_data.append([coefficient,degree])
29
30     return all_data
31
32
```

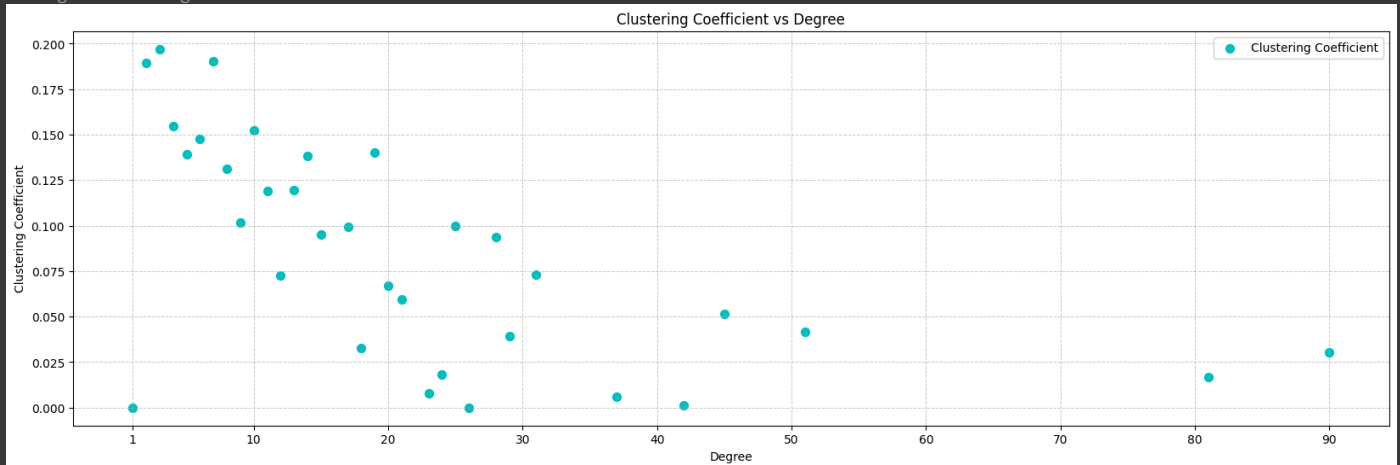


```

33 coefficients=find_clustering_coefficient(ADJ_MATRIX)
34 sorted_list = sorted(coefficients, key=lambda x: x[1])
35 sum_coefficient=0
36 sorted_coefficinetns=[]
37 sorted_degrees=[]
38 for i in sorted_list:
39     sum_coefficient+=i[0]
40     sorted_coefficinetns.append(i[0])
41     sorted_degrees.append(i[1])
42
43
44 grouped = defaultdict(lambda: [0, 0])
45
46 for first, second in sorted_list:
47     grouped[second][0] += first
48     grouped[second][1] += 1
49 means = {key: total / count for key, (total, count) in grouped.items()}
50
51 avg_clustering_coefficient=sum_coefficient/len(coefficients)
52 print("Average Clustering Coefficient is: ",avg_clustering_coefficient)
53 sorted_degrees=[]
54 sorted_coefficinetns=[]
55 for key,val in means.items():
56     sorted_degrees.append(key)
57     sorted_coefficinetns.append(val)
58 plt.figure(figsize=(20, 6))
59
60 plt.scatter(sorted_degrees, sorted_coefficinetns, color='c', s=50, label='Clustering Coefficient')
61
62 x_min, x_max = min(sorted_degrees), max(sorted_degrees)
63 plt.xticks(np.linspace(x_min, x_max, num=10, dtype=int))
64 plt.title('Clustering Coefficient vs Degree')
65 plt.xlabel('Degree')
66 plt.ylabel('Clustering Coefficient')
67 plt.grid(True, linestyle='--', linewidth=0.7, alpha=0.7)
68 plt.legend()
69
70 plt.savefig('q5_part(a)_clustering_coefficient_distribution.png', dpi=1000, bbox_inches='tight')
71 plt.show()

```

↗ Average Clustering Coefficient is: 0.08397718749419945



## ✓ Visualization In Cytoscape: Yeast Interactome

 Visualization In Cytoscape: Yeast Interactome

## ✓ Finding the Max Degree Protein


- To find the maximum degree protein, we simply run a for loop over the adjacency matrix and for each row we sum all the values.

- This gives us the degree for the particular node.
- Next, we simply lookup in the dictionary created earlier to find the protein name.

```

1 MAX_DEGREE = -1
2 MAX_DEGREE_PROTEIN = ""
3
4 for node in range(len(ADJ_MATRIX)):
5     degree_of_this_node = sum(ADJ_MATRIX[node])
6     if(degree_of_this_node > MAX_DEGREE):
7         MAX_DEGREE=degree_of_this_node
8         for key,value in PROTEIN_TO_NUM_MAP.items():
9             if(value==node):
10                 MAX_DEGREE_PROTEIN=key
11
12 print("The maximum Degree in the network is: ",MAX_DEGREE)
13 print("The Max Degree Protein is: ",MAX_DEGREE_PROTEIN)
14
15

```

 The maximum Degree in the network is: 90  
The Max Degree Protein is: YLR291C

### Importance of the Highest degree protein: YLR291C

- The protein encoded by the YLR291C is known as: GDC7.
- This protein is a subunit of eukaryotic translation initiation factor 2B (eIF2B) complex.
- This helps in initiation phase of the protein synthesis.
- Specific functions of the protein is to work as a guanine nucleotide exchange factor (GEF), and thus it helps in the exchange of the GDP for GTP.
- Thus, this protein plays a huge role in the initiation of translation.
- Sources: <https://www.yeastgenome.org/locus/YLR291C>

## PART-B

This was removed from the questions.

## ✓ PART-C

For this part we reused the code of q4, and simply created the numerical yeast network csv, which had numbers instead of protein. And then we simply used the functions of q4 and plotted the graphs.

```

1 dataframe = pd.read_csv("yeast_data.csv")
2
3 data = []
4 for index, row in dataframe.iterrows():
5     u = row.iloc[0]
6     v = row.iloc[1]
7     data.append([PROTEIN_TO_NUM_MAP[u], PROTEIN_TO_NUM_MAP[v]])
8 data = pd.DataFrame(data)
9 data.to_csv("numerical_yeast_data.csv", index=False)

```

```

1 import networkx as nx
2 import numpy as np
3 import random
4 import matplotlib.pyplot as plt
5 import pandas as pd
6
7 def deleteRandomNode(adj_matrix,flag):
8     if(flag==0):
9         fractions = np.linspace(0, 0.05, 20)
10    else:
11        fractions = np.linspace(0, 0.5, 60)
12    G = nx.from_numpy_array(adj_matrix)
13    nodes = list(G.nodes())
14
15    path_lengths = []
16    giant_cluster_sizes = []
17    initial Largest cc size = len(max(nx.connected_components(G), key=len))

```

```

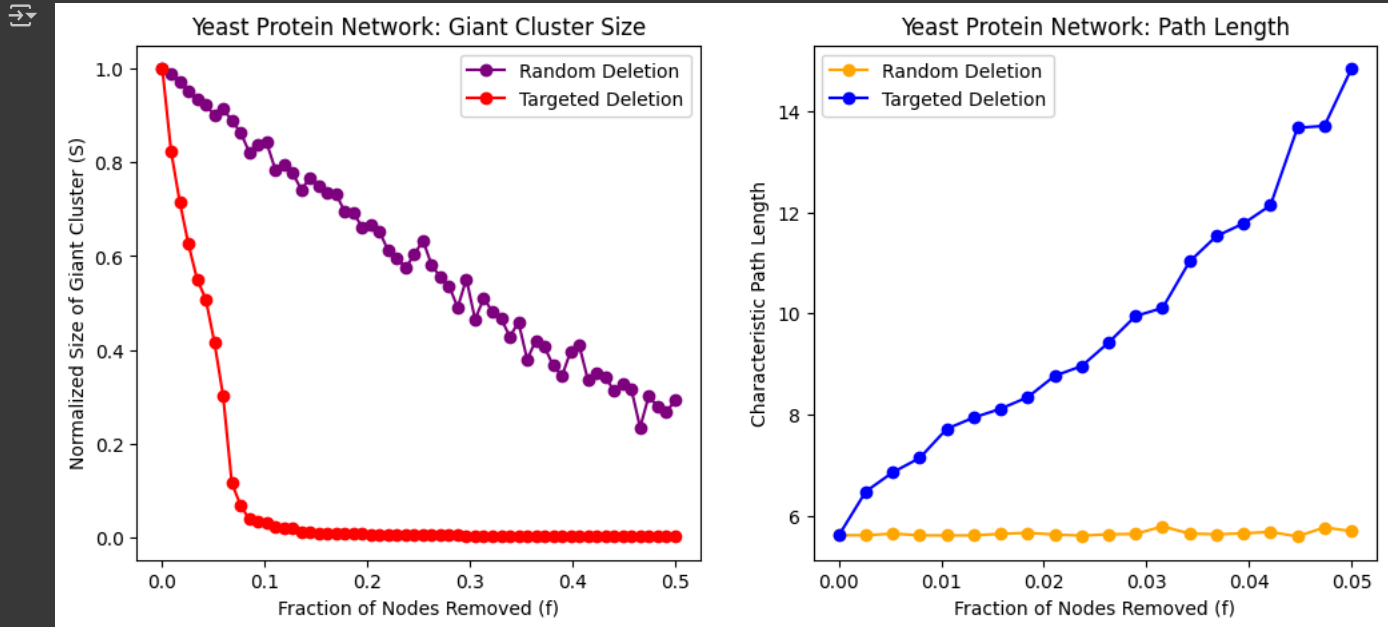
17 initial_largest_cc_size = len(max(nx.connected_components(G), key=len))
18
19 for f in fractions:
20     G_copy = G.copy()
21     num_remove = int(f * len(nodes))
22     remove_nodes = random.sample(nodes, num_remove)
23     G_copy.remove_nodes_from(remove_nodes)
24
25     b = nx.connected_components(G_copy)
26     components = list(b)
27     if components:
28         largest_cc = max(components, key=len)
29         H = G_copy.subgraph(largest_cc)
30         giant_cluster_sizes.append(len(largest_cc)/initial_largest_cc_size)
31         if nx.is_connected(H):
32             a = nx.average_shortest_path_length(H)
33             path_lengths.append(a)
34         else:
35             path_lengths.append(np.nan)
36     else:
37         giant_cluster_sizes.append(0)
38         path_lengths.append(np.nan)
39 if(flag==0):
40     return path_lengths, fractions
41 else:
42     return giant_cluster_sizes, fractions
43
44
45 def deleteTargetNode(G,flag):
46     if(flag==0):
47         fractions = np.linspace(0, 0.05, 20)
48     else:
49         fractions = np.linspace(0, 0.5, 60)
50     nodes = sorted(G.nodes(), key=lambda x: G.degree[x], reverse=True) # Sort by degree
51
52     path_lengths = []
53     giant_cluster_sizes = []
54     initial_largest_cc_size = len(max(nx.connected_components(G), key=len))
55
56     for f in fractions:
57         G_copy = G.copy()
58         num_remove = int(f * len(nodes))
59         remove_nodes = nodes[:num_remove] # Remove highest-degree nodes first
60         G_copy.remove_nodes_from(remove_nodes)
61
62         components = list(nx.connected_components(G_copy))
63         if components:
64             largest_cc = max(components, key=len)
65             H = G_copy.subgraph(largest_cc)
66             giant_cluster_sizes.append(len(largest_cc)/initial_largest_cc_size)
67             if nx.is_connected(H):
68                 path_lengths.append(nx.average_shortest_path_length(H))
69             else:
70                 path_lengths.append(np.nan)
71         else:
72             giant_cluster_sizes.append(0)
73             path_lengths.append(np.nan)
74     if(flag==0):
75         return path_lengths, fractions
76     else:
77         return giant_cluster_sizes, fractions
78
79
80
81
82 real_network = nx.read_edgelist("numerical_yeast_data.csv", nodetype=int,delimiter=",")
83
84 # chlo phele path length nikal lu
85 real_random_path, path_fractions = deleteRandomNode(nx.to_numpy_array(real_network),0)
86 real_target_path, path_fractions = deleteTargetNode(real_network,0)
87
88 # chlo ab cluster size nikal lu
89 real_random_giant, cluster_fractions = deleteRandomNode(nx.to_numpy_array(real_network),1)
90 real_target_giant, cluster_fractions = deleteTargetNode(real_network,1)
91
92
93 plt.figure(figsize=(12, 5))
94 plt.subplot(1, 2, 1)

```

```

95 plt.plot(cluster_fractions, real_random_giant,'o-',color='purple', label="Random Deletion")
96 plt.plot(cluster_fractions, real_target_giant,'o-', color='red',label="Targeted Deletion")
97 plt.xlabel("Fraction of Nodes Removed (f)")
98 plt.ylabel("Normalized Size of Giant Cluster (S)")
99 plt.title("Yeast Protein Network: Giant Cluster Size")
100 plt.legend()
101
102 plt.subplot(1, 2, 2)
103 plt.plot(path_fractions, real_random_path,'o-', color='orange',label="Random Deletion")
104 plt.plot(path_fractions, real_target_path, 'o-',color='blue',label="Targeted Deletion")
105 plt.xlabel("Fraction of Nodes Removed (f)")
106 plt.ylabel("Characteristic Path Length")
107 plt.title("Yeast Protein Network: Path Length")
108 plt.legend()
109
110 plt.show()
111

```



As expected both the graphs follow the same trend of the Albert et al., Nature, 406, 378 (2000).

- The path length increases for targeted attack as we increase the number of nodes removed and remains constant for random deletion.
- The size of giant cluster decreases sharply for targeted attack and the critical fraction of node point( $f_c$ ) occurs just before 0.1, and it shows a linear decrease pattern for random deletion.
- Both of these aligns with the trends of the scale free graphs.
- Thus, it proves tha Yeast Protein Network is also a scale free network.

#### ✓ Avg Size of Fragmented Components

```

1 import networkx as nx
2 import numpy as np
3 import random
4 import matplotlib.pyplot as plt
5
6 def avg_fragmented_component_size(adj_matrix):
7     G = nx.from_numpy_array(adj_matrix)
8     nodes = list(G.nodes())
9     fractions = np.linspace(0, 0.5, 20)
10
11     avg_fragment_sizes = []
12
13     for f in fractions:
14         G_temp = G.copy()
15         b = f * len(nodes)
16         num_remove = int(b)
17         remove_nodes = random.sample(nodes, num_remove)
18         G_temp.remove_nodes_from(remove_nodes)
19

```

```

20     components = list(nx.connected_components(G_temp))
21
22     if components:
23         sizes = []
24         for c in components:
25             sizes.append(len(c))
26
27         largestSize = max(sizes)
28         fragmented_sizes = []
29
30         for s in sizes:
31             if s != largestSize:
32                 fragmented_sizes.append(s)
33
34         if fragmented_sizes:
35             a = np.mean(fragmented_sizes)
36             avg_fragment_sizes.append(a)
37         else:
38             avg_fragment_sizes.append(0)
39     else:
40         avg_fragment_sizes.append(0)
41
42     return avg_fragment_sizes, fractions
43
44 def avg_fragmented_component_size_targeted(adj_matrix):
45     G = nx.from_numpy_array(adj_matrix)
46     nodes = list(G.nodes())
47     fractions = np.linspace(0, 0.5, 20)
48
49     avg_fragment_sizes = []
50
51     for f in fractions:
52         G_temp = G.copy()
53         b = f * len(nodes)
54         num_remove = int(b)
55
56         x = G_temp.degree()
57         degree_sorted_nodes = sorted(x, key=lambda x: x[1], reverse=True)
58         remove_nodes = []
59
60         for i in range(num_remove):
61             node, _ = degree_sorted_nodes[i]
62             remove_nodes.append(node)
63
64         G_temp.remove_nodes_from(remove_nodes)
65
66         components = list(nx.connected_components(G_temp))
67
68         if components:
69             sizes = []
70             for c in components:
71                 size = len(c)
72                 sizes.append(size)
73
74             if sizes:
75                 largest_size = max(sizes)
76             else:
77                 largest_size = 0
78
79             fragmented_sizes = []
80
81             for s in sizes:
82                 if s != largest_size:
83                     fragmented_sizes.append(s)
84
85             avg_fragment_sizes.append(np.mean(fragmented_sizes) if fragmented_sizes else 0)
86         else:
87             avg_fragment_sizes.append(0)
88
89     return avg_fragment_sizes, fractions

```

```

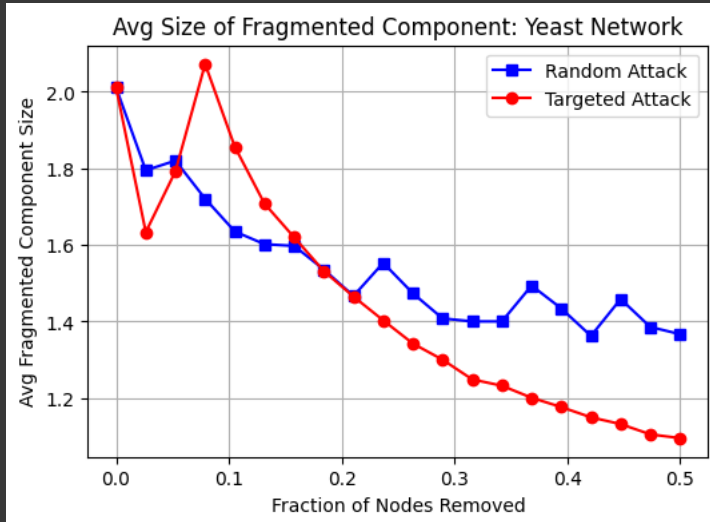
1 def plot_fragmentation_combined(adj_matrix):
2     avg_sizes_random, fractions = avg_fragmented_component_size(adj_matrix)
3     avg_sizes_targeted, _ = avg_fragmented_component_size_targeted(adj_matrix)
4
5     plt.figure(figsize=(6, 4))

```

```

6 plt.plot(fractions, avg_sizes_random, 's-', color='blue', label="Random Attack")
7 plt.plot(fractions, avg_sizes_targeted, 'o-', color='red', label="Targeted Attack")
8 plt.xlabel("Fraction of Nodes Removed")
9 plt.ylabel("Avg Fragmented Component Size")
10 plt.title("Avg Size of Fragmented Component: Yeast Network")
11 plt.legend()
12 plt.grid(True)
13 plt.show()
14
15 real_network = nx.read_edgelist("numerical_yeast_data.csv", nodetype=int, delimiter=",")
16 adj_matrix = nx.to_numpy_array(real_network, nodelist=sorted(real_network.nodes()))
17 plot_fragmentation_combined(adj_matrix)

```



- In case of random deletion the the avg size of fragmented component `<s>` decreases slowly, there are some fluctuations in the graph but overall decreasing pattern is linear.
- In case of targeted deletion, the size first increases, and then it decreases sharply.
- This suggests that scale free networks are tolerable to random node deletions, but for a targeted attack, they are vulnerable.
- Thus, this is in accordance with the Albert et al (2000), which proves that scale free networks are vulnerable to targeted node deletions.