Random graph model:

To implement the graph, we made use of the Python library NetworkX.

In the E-R G(n,p) model, the graph is constructed by connecting nodes randomly. Each pair of distinct nodes is connected by an edge with an independent probability p.

The probability can be obtained by $p = \frac{avgDegree}{n-1}$

Because the E-R model is probabilistic, different simulations with the same parameters can yield graphs with varying structures. However, across many runs, the statistical properties of the graphs align with the expected values dictated by the model or atleast yield close results.

```python
import networkx as nx
import matplotlib.pyplot as plt

# Number of nodes in the graph
num_nodes = 282

# Average degree of the nodes
average_degree = 7.35

# Calculate the probability of edge creation
prob_edge_creation = (average_degree) / (num_nodes - 1)
# Generate a random model graph
```

(variable) largest_cc_graph: ...

```python
random_graph = nx.erdos_renyi_graph(num_nodes, prob_edge_creation)
if nx.is_connected(random_graph):
        avg_path_length = nx.average_shortest_path_length(random_graph)
else:
        largest_cc = max(nx.connected_components(random_graph), key=len)
        largest_cc_graph = random_graph.subgraph(largest_cc)
        avg_path_length = nx.average_shortest_path_length(largest_cc_graph)

print(avg_path_length)
# Calculate the clustering coefficient
clustering_coefficient = nx.average_clustering(random_graph)
print(clustering_coefficient)

# Calculate the clustering coefficient for each node
node_clustering_coefficients = nx.clustering(random_graph)

# Average the clustering coefficients to find the overall clustering coefficient
overall_clustering_coefficient = sum(node_clustering_coefficients.values()) / len(node_clustering_coefficients)
print (overall_clustering_coefficient)
```

✓ 0.0s

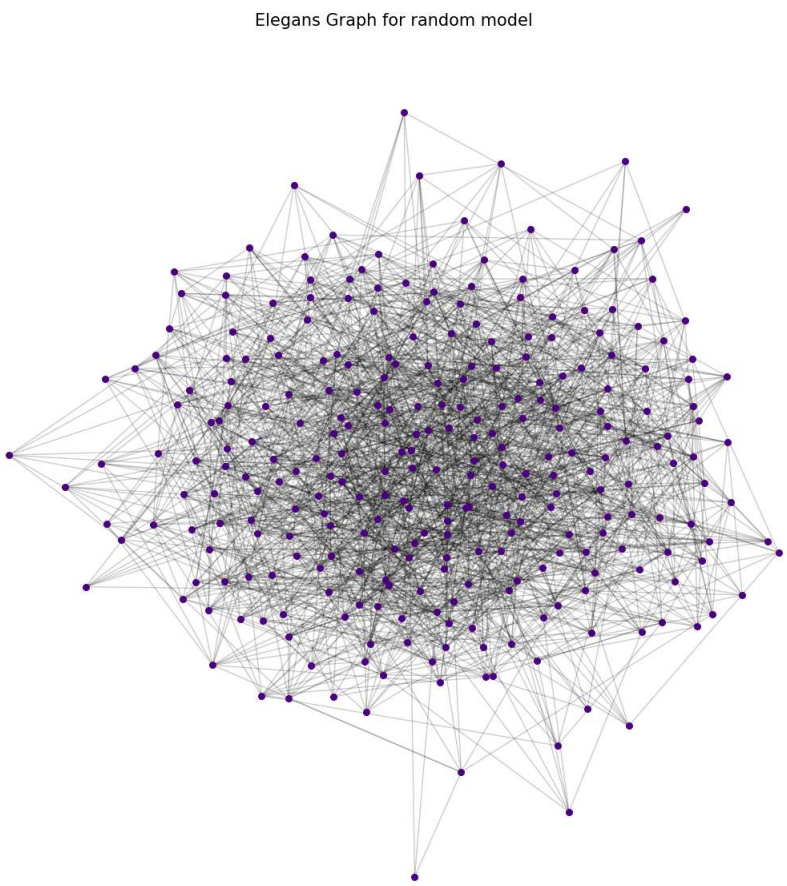
```
3.0452537795613437
0.025017634060187234
0.025017634060187234
```

Ecoli Graph for random model

```python
import networkx as nx
import matplotlib.pyplot as plt

# Number of nodes in the graph
num_nodes = 282

# Average degree of the nodes
average_degree = 14

# Calculate the probability of edge creation
prob_edge_creation = (average_degree) / (num_nodes - 1)
# Generate a random model graph

random_graph = nx.erdos_renyi_graph(num_nodes, prob_edge_creation)
if nx.is_connected(random_graph):
        avg_path_length = nx.average_shortest_path_length(random_graph)
else:
        largest_cc = max(nx.connected_components(random_graph), key=len)
        largest_cc_graph = random_graph.subgraph(largest_cc)
        avg_path_length = nx.average_shortest_path_length(largest_cc_graph)

print(avg_path_length)
# Calculate the clustering coefficient
clustering_coefficient = nx.average_clustering(random_graph)
print(clustering_coefficient)

# Calculate the clustering coefficient for each node
node_clustering_coefficients = nx.clustering(random_graph)

# Average the clustering coefficients to find the overall clustering coefficient
overall_clustering_coefficient = sum(node_clustering_coefficients.values()) / len(node_clustering_coefficients)
print (overall_clustering_coefficient)
```

✓ 0.0s

```
2.4046591454026904
0.0538140921878674
0.0538140921878674
```



Comparision:

Given the input/output table:

Network	Size	Avg. Degree	Original Avg. Path Length	Original C	Simulated Avg. Path Length	Simulated C	My Avg path length	My C
E.Coli	282	7.35	2.9	0.32	3.04	0.026	3.045	0.025
C.Elegans	282	14	2.65	0.28	2.387	0.054	2.404	0.053

The E-R random graph model tends to produce a logarithmic average path length relative to the number of nodes, which often aligns with the short path lengths seen in real-world

networks. This is why both simulated results (mine and the textbook's) are fairly close to the original network's average path length.

Where the Erdős-Rényi model often diverges significantly from real-world networks is in the clustering coefficient. In a G(n,p) graph, the clustering coefficient is also p, which tends to be much lower than that of most real-world networks, especially social and biological networks, which exhibit higher levels of clustering due to community structures and other non-random connections.

Small world model:

The starting point for the small-world model is a regular lattice, where each node has the same number of connections (c/2 on either side if the nodes are arranged in a ring). This regularity models an egalitarian scenario where every individual (node) has the same number of friends (connections). Although it's an oversimplification of real-world social structures, this regular lattice serves as a foundation for introducing small-world characteristics.

Since the Watts-Strogatz model requires an even number of edges for each node to maintain symmetry in the regular lattice, you round the mean degree to the nearest even integer. This ensures that every node will start with the same number of edges, distributed equally on either side.

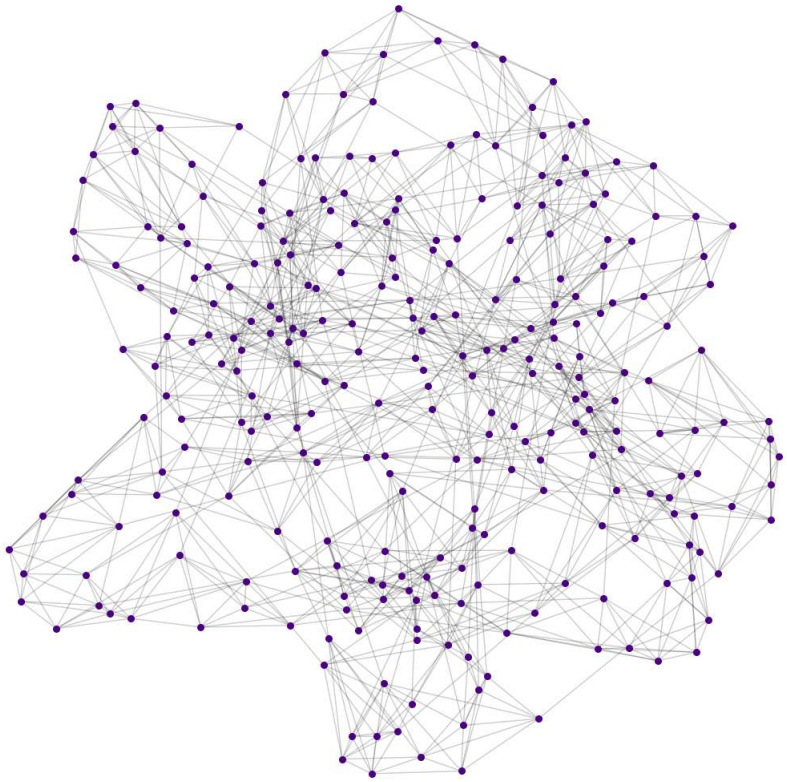
The nx.watts_strogatz_graph function is used to generate the network. It starts by creating a ring lattice where each node is connected to its k nearest neighbors (k/2 on each side), and then it rewires each edge with probability beta. The rewiring introduces randomness into the network, creating shortcuts that decrease the average path length while potentially preserving a high clustering coefficient.

```python
#small world Ecoli

def generate_small_world_graph(num_nodes, mean_degree, beta):
    # Round mean_degree to the nearest even integer
    mean_degree = int(2 * round(mean_degree / 2))

    # Create a regular ring lattice with num_nodes nodes and mean_degree neighbors on each side
    G = nx.watts_strogatz_graph(num_nodes, mean_degree, beta)

    return G

# Parameters
num_nodes = 282  # This is |V| in the algorithm.
mean_degree = 7.35  # This is the mean degree c in the algorithm.
beta = 0.1  # This is the rewiring probability β in the algorithm.

# Generate the graph
try:
    small_world_graph = generate_small_world_graph(num_nodes, mean_degree, beta)

    # Output the nodes and edges of the graph
    print("Nodes:", small_world_graph.nodes())
    print("Edges:", small_world_graph.edges())

    # Calculate the average path length
    avg_path_length = nx.average_shortest_path_length(small_world_graph)
    print(f"Average Path Length: {avg_path_length}")

    # Calculate the clustering coefficient
    clustering_coefficient = nx.average_clustering(small_world_graph)
    print(f"Clustering Coefficient: {clustering_coefficient}")

except Exception as e:
    print(e)
```

✓ 0.0s Python

```
Nodes: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 4
Edges: [(0, 1), (0, 281), (0, 280), (0, 3), (0, 279), (0, 4), (0, 52), (0, 210), (1, 2), (1, 3), (1, 281), (1, 4), (1, 280), (1, 5), (1, 279), (2, 3), (2, 4), (2, 5), (2,
Average Path Length: 3.939981323035764
Clustering Coefficient: 0.46342707725686494
```

Ecoli Graph for small world model

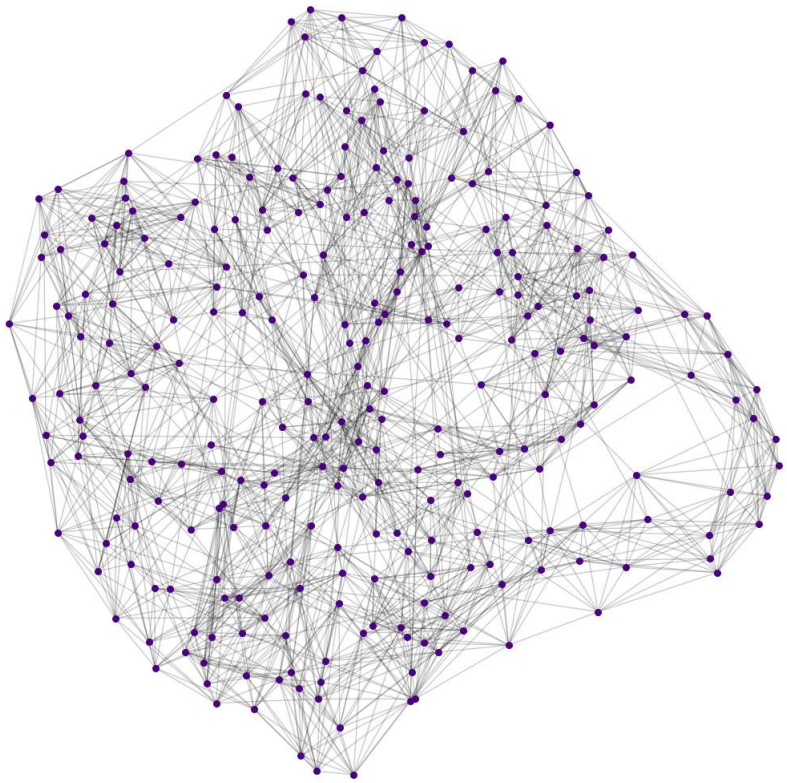


```python
#elegans small world
def generate_small_world_graph(num_nodes, mean_degree, beta):
    # Round mean_degree to the nearest even integer
    mean_degree = int(2 * round(mean_degree / 2))
    # Create a regular ring lattice with num_nodes nodes and mean_degree neighbors on each side
    G = nx.watts_strogatz_graph(num_nodes, mean_degree, beta)

    return G
# Parameters
num_nodes = 282  # This is |V| in the algorithm.
mean_degree = 14  # This is the mean degree c in the algorithm.
beta = 0.1  # This is the rewiring probability β in the algorithm.
# Generate the graph
try:
    small_world_graph = generate_small_world_graph(num_nodes, mean_degree, beta)

    # Output the nodes and edges of the graph
    print("Nodes:", small_world_graph.nodes())
    print("Edges:", small_world_graph.edges())

    # Calculate the average path length
    avg_path_length = nx.average_shortest_path_length(small_world_graph)
    print(f"Average Path Length: {avg_path_length}")

    # Calculate the clustering coefficient
    clustering_coefficient = nx.average_clustering(small_world_graph)
    print(f"Clustering Coefficient: {clustering_coefficient}")

except Exception as e:
    print(e)
```

✓ 0.0s Python

```
Nodes: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 4
Edges: [(0, 281), (0, 2), (0, 280), (0, 3), (0, 279), (0, 4), (0, 278), (0, 277), (0, 276), (0, 7), (0, 275), (0, 35), (0, 209), (0, 130), (1, 2), (1, 3), (1, 281), (1, 4)
Average Path Length: 3.0334418616390297
Clustering Coefficient: 0.533541000929405
```

Elegans Graph for small world model



Comparision:

Network	Size	Avg. Degree	Original Avg. Path Length	Original C	Simulated Avg. Path Length	Simulated C	My Avg path length	My C
E.Coli	282	7.35	2.9	0.32	4.46	0.31	3.93	0.46
C.Elegans	282	14	2.65	0.28	3.49	0.37	3.03	0.53

The differences between our small-world simulation results and those from the original and textbook's are likely due to the distinct mechanisms of edge formation in each model.

The original networks have evolved structures that may be driven by functional and evolutionary constraints, leading to their specific properties of path lengths and clustering.

Our small-world simulations use the Watts-Strogatz model, which adds a degree of randomness to an otherwise regular lattice. The specific value of β we've chosen seems to increase local clustering significantly, possibly more than what is found in real biological networks. It might also be the case that the chosen β is creating more short-cuts that decrease the average path length compared to a random graph but not to the extent of the actual biological networks.

Preferential attachment model:

This code implements a network generation algorithm that models preferential attachment, a process that is characteristic of scale-free networks.

The code calculates 'm' as the floor of half the average degree, which represents the number of edges that each new node will attempt to make when it is added to the network. This is based on the characteristic of scale-free networks where the average degree is roughly twice the number of edges made by each new node due to the undirected nature of edges.

A complete graph with m+1 nodes is created to serve as the initial network. A complete graph is chosen so that early nodes have a high likelihood of being selected for new

connections, as they will have a higher degree by virtue of being part of the initial complete graph.

The network grows by adding one new node at a time until the desired number of nodes n is reached. For each new node, it:

Calculates the current sum of all node degrees in the graph.

Selects m existing nodes to connect to the new node. The probability of selecting any existing node is proportional to its degree.

Adds the new node to the graph and creates edges to the m selected nodes.

The process is repeated until the graph has n nodes. The resulting graph is expected to display the scale-free property, with a few nodes (hubs) having a very high degree and most nodes having a low degree.

Once the graph is generated, the code calculates and prints two important metrics:

Average Path Length: The average number of steps along the shortest paths for all possible pairs of network nodes.

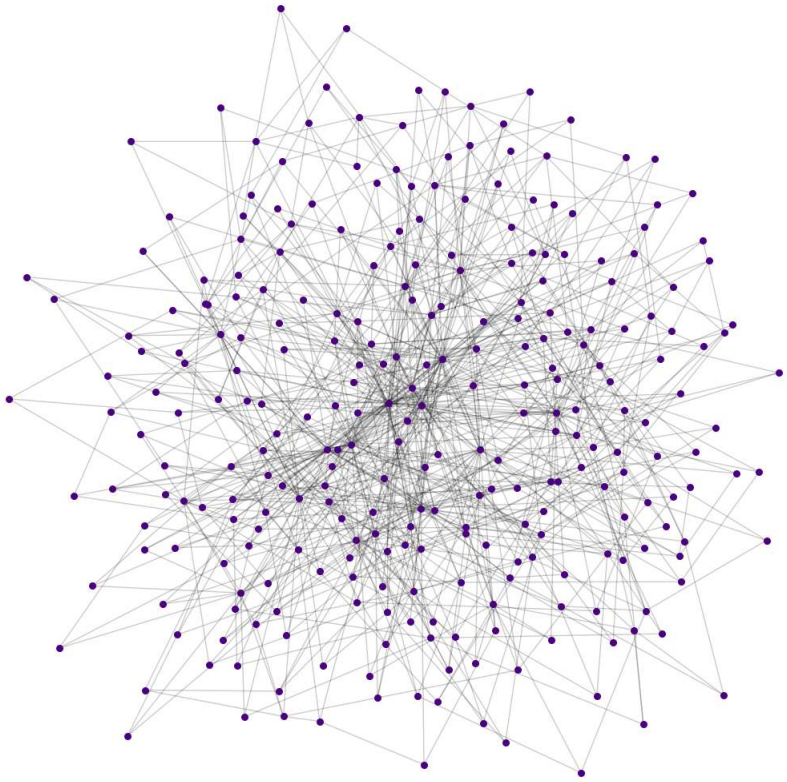
Clustering Coefficient: A measure of the degree to which nodes in the graph tend to cluster together.

```python
def generate_preferential_attachment_graph(n, avg_degree):

    # Calculate 'm' as half of the average degree, rounded down to the nearest integer
    m = int(avg_degree // 2)

    # Ensure that m is at least 1
    if m < 1:
        raise ValueError("Average degree must be at least 2 to form a connected graph.")

    # Create initial complete graph with m+1 nodes to ensure an initial degree of at least m
    G = nx.complete_graph(m + 1)

    # Add the remaining nodes using preferential attachment
    for i in range(m + 1, n):
        # Get the sum of degrees of all nodes in the graph
        sum_of_degrees = sum(dict(G.degree()).values())
        # Choose m nodes to connect to the new node based on their degree
        nodes = set()
        while len(nodes) < m:
            chosen_node = random.choices(
                population=list(G.nodes),
                weights=[degree for _, degree in G.degree()],
                k=1
            )[0]
            nodes.add(chosen_node)

        # Add the new node and connect it to the chosen nodes
        G.add_node(i)
        G.add_edges_from((i, node) for node in nodes)

    return G

# Parameters
num_nodes = 282  # The desired number of nodes in the graph
avg_degree = 7.35   # The desired average degree
# Generate the graph
try:
    pa_graph = generate_preferential_attachment_graph(num_nodes, avg_degree)
    print(f"Generated graph with {pa_graph.number_of_nodes()} nodes and {pa_graph.number_of_edges()} edges.")
        # Calculate the average path length
    avg_path_length = nx.average_shortest_path_length(pa_graph)
    print(f"Average Path Length: {avg_path_length}")
    m = int(avg_degree // 2)
    # graph = nx.barabasi_albert_graph(num_nodes, m)
    # avg_path_length_2 = nx.average_shortest_path_length(graph)
    # clustering_coefficient_1 = nx.average_clustering(graph)

    # Calculate the clustering coefficient
    clustering_coefficient = nx.average_clustering(pa_graph)
    print(f"Clustering Coefficient: {clustering_coefficient}")
    # print(f"Generated graph with {graph.number_of_nodes()} nodes and {graph.number_of_edges()} edges.")

    # print(avg_path_length_2)
    # print(clustering_coefficient_1)
except ValueError as e:

    print(e)
```

✓ 0.0s

```
Generated graph with 282 nodes and 840 edges.
Average Path Length: 2.966709573206128
Clustering Coefficient: 0.09486947935186936
```

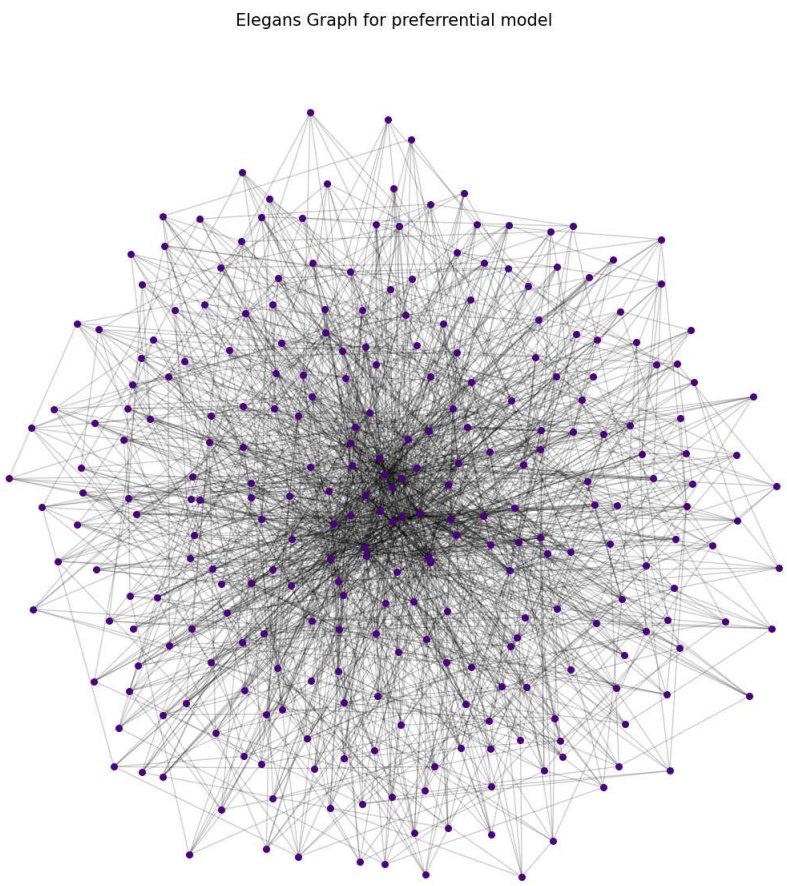
Ecoli Graph for preferrential model



```python
import random
def generate_preferential_attachment_graph(n, avg_degree):

    # Calculate 'm' as half of the average degree, rounded down to the nearest integer
    m = int(avg_degree // 2)

    # Ensure that m is at least 1
    if m < 1:
        raise ValueError("Average degree must be at least 2 to form a connected graph.")

    # Create initial complete graph with m+1 nodes to ensure an initial degree of at least m
    G = nx.complete_graph(m + 1)

    # Add the remaining nodes using preferential attachment
    for i in range(m + 1, n):
        # Get the sum of degrees of all nodes in the graph
        sum_of_degrees = sum(dict(G.degree()).values())
        # Choose m nodes to connect to the new node based on their degree
        nodes = set()
        while len(nodes) < m:
            chosen_node = random.choices(
                population=list(G.nodes),
                weights=[degree for _, degree in G.degree()],
                k=1
            )[0]
            nodes.add(chosen_node)

        # Add the new node and connect it to the chosen nodes
        G.add_node(i)
        G.add_edges_from((i, node) for node in nodes)

    return G

# Parameters
num_nodes = 282  # The desired number of nodes in the graph
avg_degree = 14   # The desired average degree
# Generate the graph
try:
    pa_graph = generate_preferential_attachment_graph(num_nodes, avg_degree)
    print(f"Generated graph with {pa_graph.number_of_nodes()} nodes and {pa_graph.number_of_edges()} edges.")
        # Calculate the average path length
    avg_path_length = nx.average_shortest_path_length(pa_graph)
    print(f"Average Path Length: {avg_path_length}")
    # m = int(avg_degree // 2)
    # graph = nx.barabasi_albert_graph(num_nodes, m)
    # avg_path_length_2 = nx.average_shortest_path_length(graph)
    # clustering_coefficient_1 = nx.average_clustering(graph)
    # Calculate the clustering coefficient
    clustering_coefficient = nx.average_clustering(pa_graph)
    print(f"Clustering Coefficient: {clustering_coefficient}")
    # print(f"Generated graph with {graph.number_of_nodes()} nodes and {graph.number_of_edges()} edges.")
       # print(avg_path_length_2)
    # print(clustering_coefficient_1)
except ValueError as e:
    print(e)
```

✓ 0.1s

```
Generated graph with 282 nodes and 1946 edges.
Average Path Length: 2.3523131672597866
Clustering Coefficient: 0.1380837420239287
```



Comparision:

Network	Size	Avg. Degree	Original Avg. Path Length	Original C	Simulated Avg. Path Length	Simulated C	My Avg path length	My C
E.Coli	282	7.35	2.9	0.32	2.37	0.03	2.96	0.09
C.Elegans	282	14	2.65	0.28	1.99	0.05	2.35	0.13

This model is known to generate networks with a scale-free degree distribution but does not inherently produce high clustering. The differences in clustering coefficients across all simulations versus the original networks highlight this limitation.

The E-R random graph model used in the textbook simulations generally produces lower clustering coefficients and shorter average path lengths compared to real-world networks

because it assumes all connections are equally likely, ignoring the clustering that arises from real-world interactions.

Neither model captures the full small-world nature of real networks, which have both high clustering and short average path lengths. The values from our simulation suggest that while it does not perfectly model the original networks, it may incorporate some aspects of the small-world property more effectively than the textbook's E-R simulations.

Comparing all three models:

Random Graph Model: This model generates edges with a uniform probability between all pairs of nodes, without any preference or structure. This randomness can produce networks with short average path lengths due to the high chance of long-range connections, but it usually fails to model the high clustering coefficients seen in real-world networks.

Small World Model: The Watts-Strogatz model, which generates small-world networks, starts with a regular lattice (where nodes are connected to their nearest neighbors) and introduces randomness through a rewiring process controlled by the parameter β. The result is a network that often has both a high clustering coefficient, due to the initial regular lattice structure, and short average path lengths, due to the random rewiring. It balances the regularity of the lattice with the randomness of edge creation to mimic the small-world properties observed in many real-world networks.

Preferential Attachment Model: In the Barabási–Albert model, nodes are added to the network over time and preferentially attach to existing nodes that have higher degrees. This model captures the scale-free property of many real-world networks, where a few nodes (hubs) have many connections, and most nodes have only a few. While it can replicate the average path lengths found in real-world networks, the model often generates lower clustering coefficients than what is observed in reality.

References:

http://www.socialmediamining.info/

https://networkx.org/

https://chat.openai.com/chat