基本信息

In [ ]:
```
import networkx as nx

# 读取 facebook_combined.txt 文件并创建无向图
def create_undirected_graph(file_path):
    G = nx.Graph()

    with open(file_path, 'r') as file:
        for line in file:
            edge = line.strip().split()
            if len(edge) == 2:
                G.add_edge(int(edge[0]), int(edge[1]))

    return G

# 指定文件路径
file_path = 'facebook_combined.txt'

# 创建无向图
G = create_undirected_graph(file_path)

# 打印图的基本信息
print("Number of nodes:", G.number_of_nodes())
print("Number of edges:", G.number_of_edges())
```

```
Number of nodes: 4039
Number of edges: 88234
```

In [ ]:
```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

# 1. Connectedness
is_connected = nx.is_connected(G)
print("connected")
# 2. Components
components = list(nx.connected_components(G))
num_components = len(components)
largest_component_size = len(max(components, key=len))
print("component")
# 3. Clustering
average_clustering_coefficient = nx.average_clustering(G)
print("clustering")
# 4. Degree distribution
degree_distribution = [d for n, d in G.degree()]
print("degree distribution")
# 5. Degree correlation (assortativity)
degree_assortativity = nx.degree_assortativity_coefficient(G)
print("correlation")
# 6. Community detection (using Girvan-Newman method as an example)
# from networkx.algorithms.community import girvan_newman
# communities = girvan_newman(G)
# top_level_communities = next(communities)
# sorted_communities = sorted(map(sorted, top_level_communities))
# print("community")
# Output structural information
print(f"Is graph connected: {is_connected}")
print(f"Number of components: {num_components}")
print(f"Largest component size: {largest_component_size}")
```

```
print(f"Average clustering coefficient: {average_clustering_coefficient:.4f}")
print(f"Degree assortativity: {degree_assortativity:.4f}")
# print("Communities (first level):", sorted_communities)
```

```
connected
component
clustering
degree distribution
correlation
Is graph connected: True
Number of components: 1
Largest component size: 4039
Average clustering coefficient: 0.6055
Degree assortativity: 0.0636
```

这个暂时不用

In [ ]:
```
import networkx as nx

# 创建一个示例图
G = nx.erdos_renyi_graph(100, 0.05)

# 计算所有节点对之间的最短路径长度
lengths = dict(nx.all_pairs_shortest_path_length(G))

# 打印从节点0到其他节点的最短路径长度
for target, length in lengths[0].items():
    print(f"Shortest path length from node 0 to node {target}: {length}")
```

In [ ]:
```
import networkx as nx

# 创建一个示例图
G = nx.erdos_renyi_graph(100, 0.05)

# 计算平均最短路径长度
average_length = nx.average_shortest_path_length(G)
print(f"Average shortest path length: {average_length}")
```

```
Average shortest path length: 2.9585858585858587
```

In [ ]:
```
# 1. Number of nodes
num_nodes = G.number_of_nodes()

# 2. Number of edges
num_edges = G.number_of_edges()

# 3. Average degree
average_degree = sum(dict(G.degree()).values()) / num_nodes

# 4. Hubs (nodes with highest degree)
degree_sequence = sorted(G.degree(), key=lambda x: x[1], reverse=True)
hubs = degree_sequence[:5]  # Top 5 hubs

# 5. Shortest path lengths
shortest_path_lengths = dict(nx.shortest_path_length(G))

# 6. Diameter (maximum shortest path length)
# Diameter is only defined for connected graphs
if nx.is_connected(G):
    diameter = nx.diameter(G)
else:
    diameter = max([max(lengths.values()) for lengths in shortest_path_lengths.valu

# Output basic statistics
```

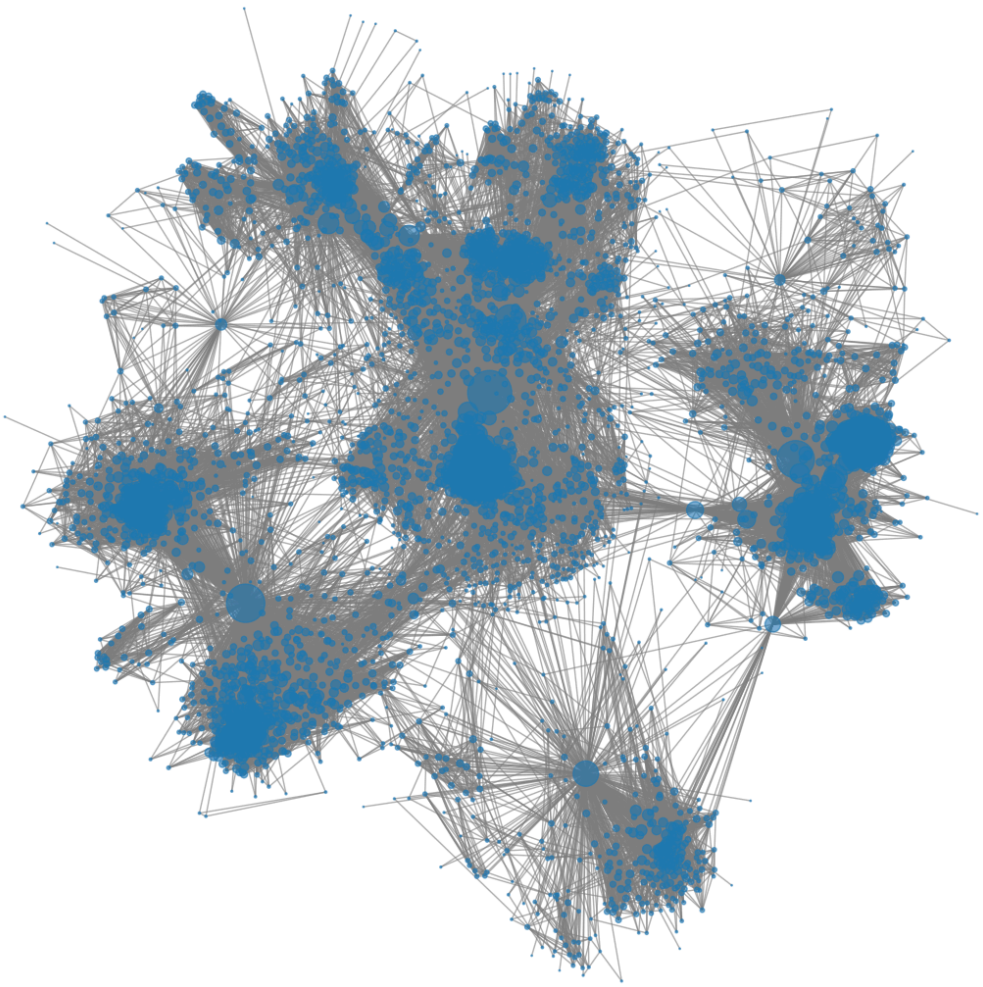
```
print(f"Number of nodes: {num_nodes}")
print(f"Number of edges: {num_edges}")
print(f"Average degree: {average_degree:.2f}")
print("Top 5 hubs (node, degree):", hubs)
print(f"Diameter: {diameter}")
```

```
Number of nodes: 4039
Number of edges: 88234
Average degree: 43.69
Top 5 hubs (node, degree): [(107, 1045), (1684, 792), (1912, 755), (3437, 547), (0, 347)]
Diameter: 8
```

画一下图

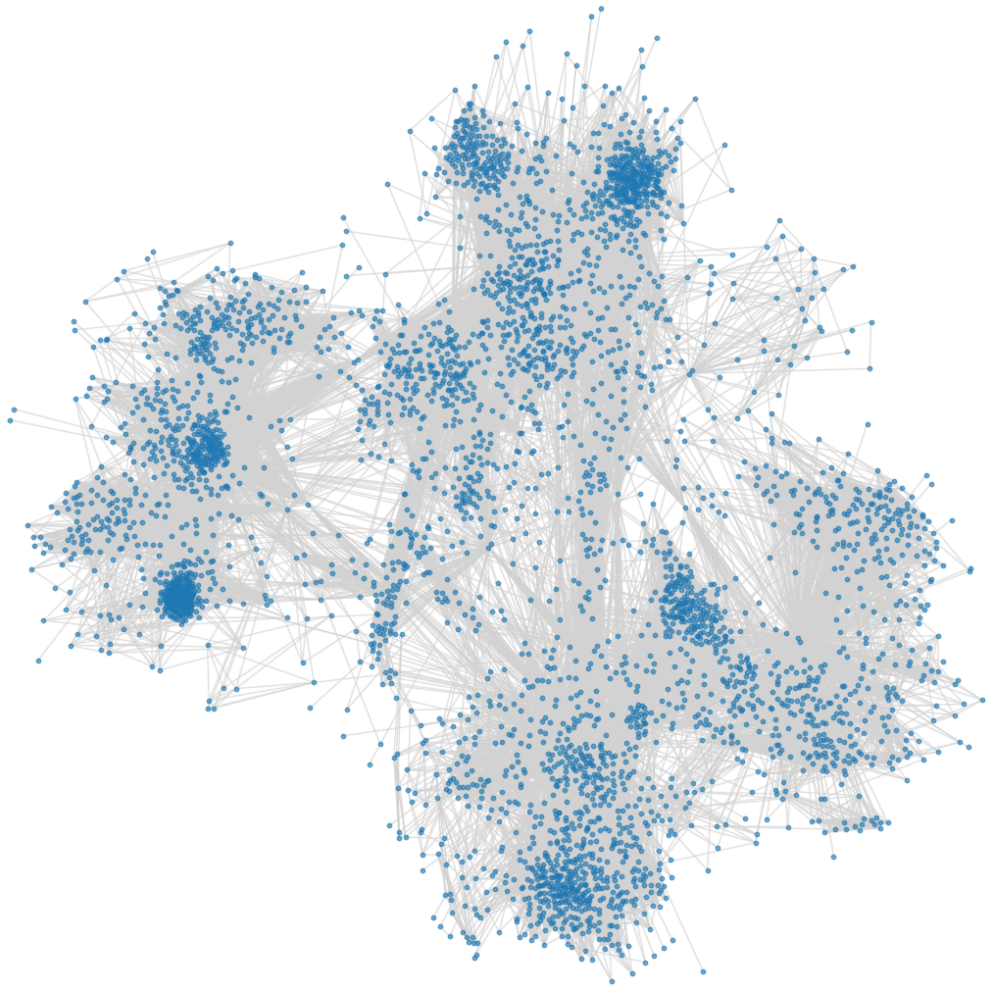
In [ ]:

```
# 绘制节点大小与度数相关的网络图
def draw_graph_with_degree_size(G):
    pos = nx.spring_layout(G, k=0.1, iterations=50)  # 调整参数以增加节点之间的距离
    degrees = dict(G.degree())
    node_size = [v for v in degrees.values()]  # 调整节点大小的比例因子

    plt.figure(figsize=(12, 12))
    nx.draw(G, pos, node_size=node_size, with_labels=False, alpha=0.6, edge_color='
    plt.title('Network Visualization with Node Size Proportional to Degree')
    plt.show()

# 绘制网络图
draw_graph_with_degree_size(G)
```



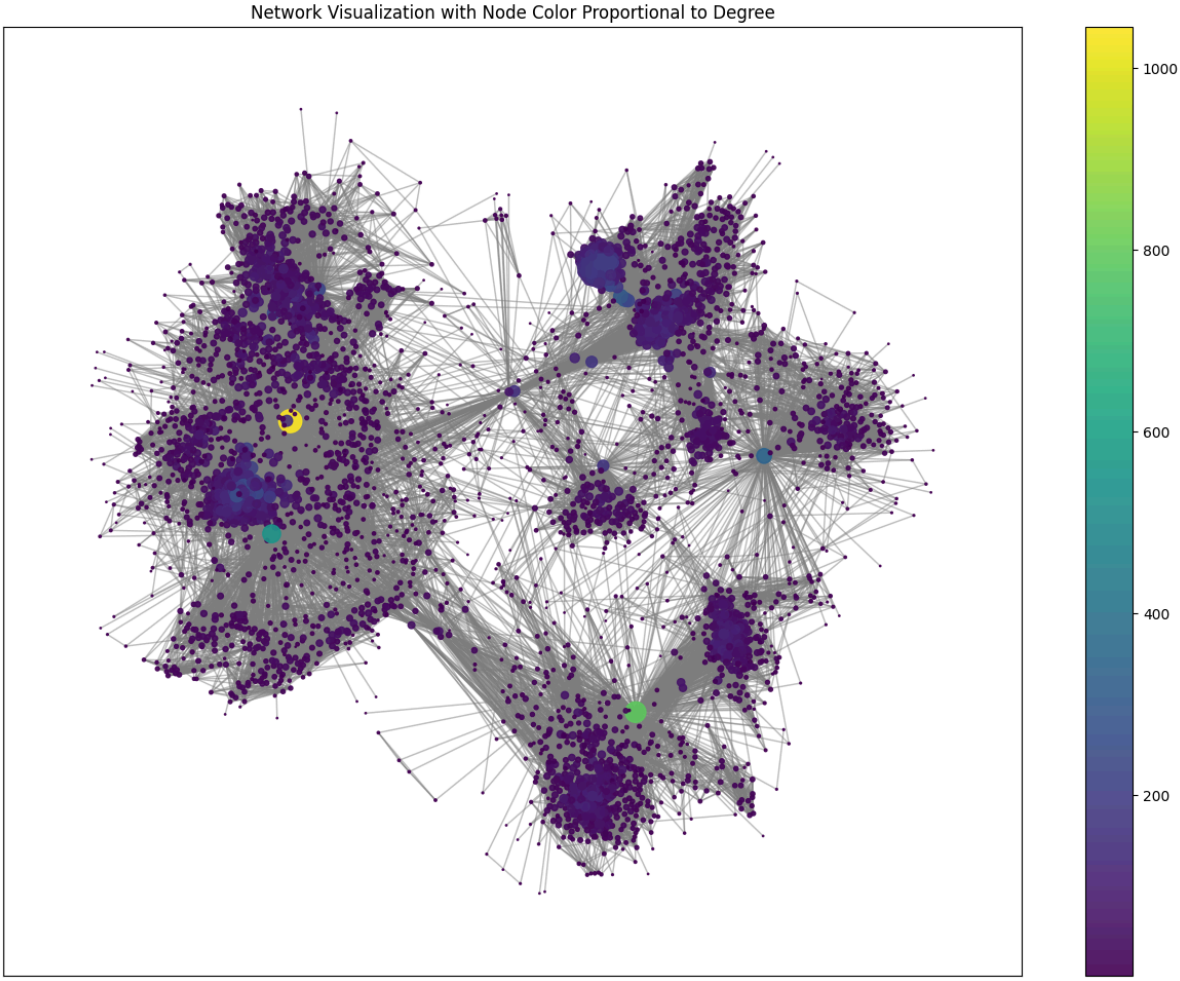
In [ ]:

```
plt.figure(figsize=(12, 12))
pos = nx.spring_layout(G, k=0.1) # 使用spring布局
nx.draw(G, pos, node_size=10, with_labels=False, alpha=0.6, edge_color='lightgray')
plt.title('Facebook Combined Network')
plt.show()
```



In [ ]:

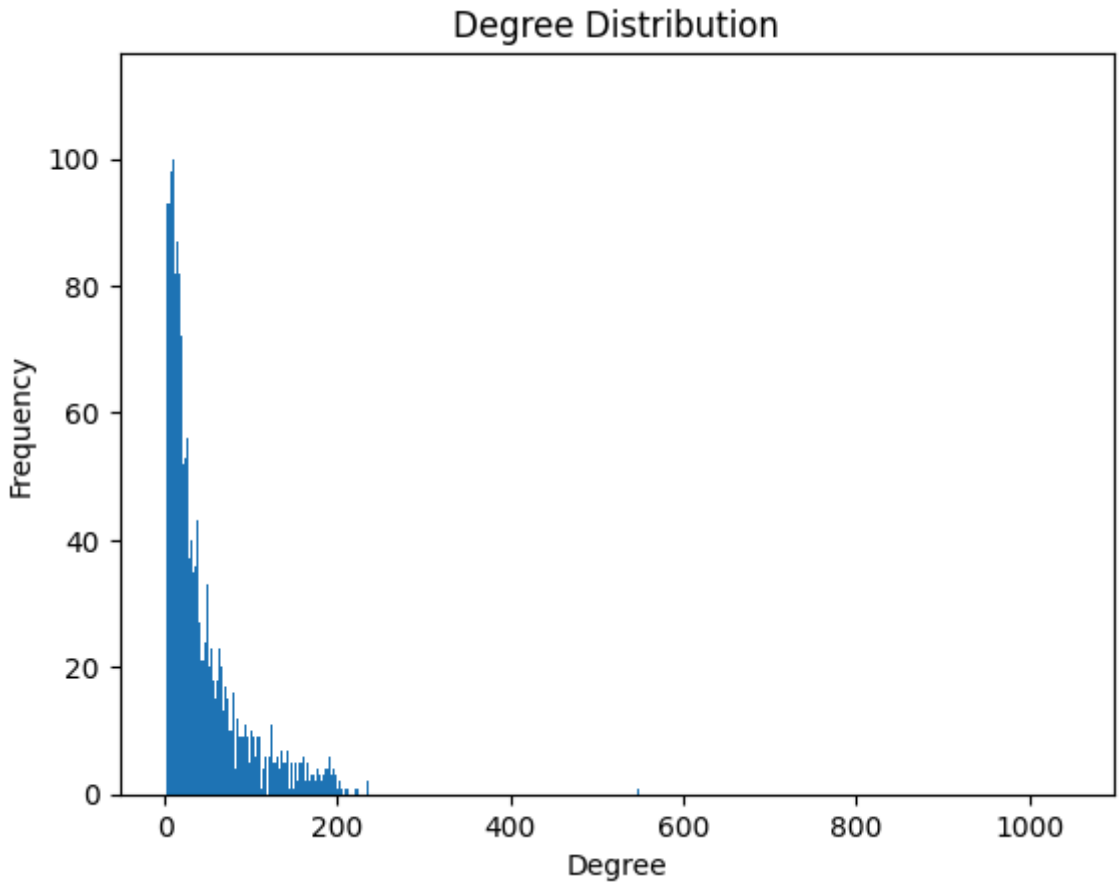
```
import networkx as nx
import matplotlib.pyplot as plt
import random

def draw_graph_with_degree_color(G, seed=None):
    # 设置随机种子
    if seed is not None:
        random.seed(seed)

    pos = nx.spring_layout(G, k=0.1)  # 调整参数以增加节点之间的距离
    degrees = dict(G.degree())
    node_color = [degrees[node] for node in G.nodes()]  # 使用节点度数作为颜色值
    node_size = [v** 0.8 for v in degrees.values()]  # 调整节点大小的比例因子

    plt.figure(figsize=(16, 12))
    nodes = nx.draw_networkx_nodes(G, pos, node_size=node_size, node_color=node_colo
    edges = nx.draw_networkx_edges(G, pos, alpha=0.5, edge_color='gray')
    plt.colorbar(nodes)
    plt.title('Network Visualization with Node Color Proportional to Degree')
    plt.show()

# 设置随机种子为 42，并绘制网络图
# 78989
# 10086
draw_graph_with_degree_color(G, seed=99)
```



degree distribution

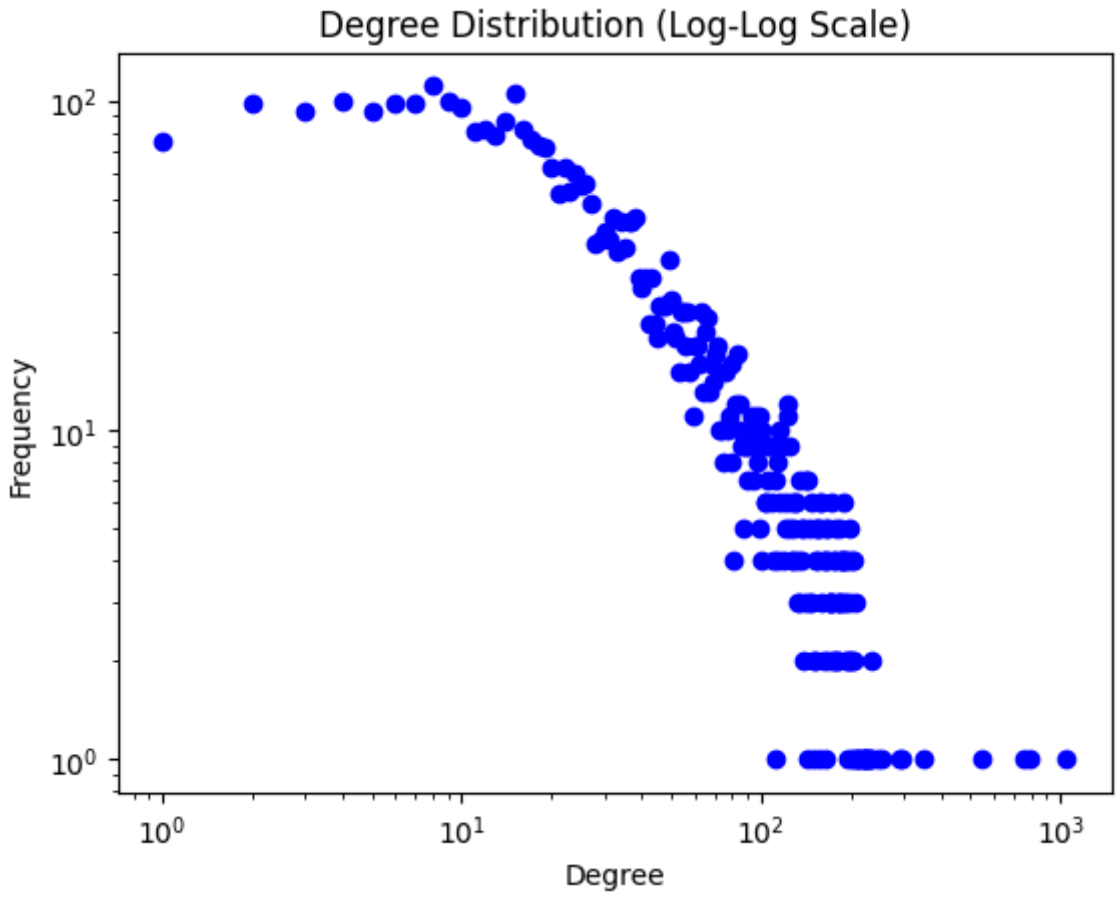
In [ ]:
```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

# 4. Degree distribution
degree_distribution = [d for n, d in G.degree()]
print("degree distribution")

plt.figure()
plt.hist(degree_distribution, bins=range(max(degree_distribution)+1))
plt.title('Degree Distribution')
plt.xlabel('Degree')
plt.ylabel('Frequency')
plt.show()

# Plot degree distribution on log-log scale
degree_counts = np.bincount(degree_distribution)
degrees = np.nonzero(degree_counts)[0]
counts = degree_counts[degrees]

plt.figure()
plt.loglog(degrees, counts, 'bo')
plt.title('Degree Distribution (Log-Log Scale)')
plt.xlabel('Degree')
plt.ylabel('Frequency')
plt.show()
```

```
degree distribution
```



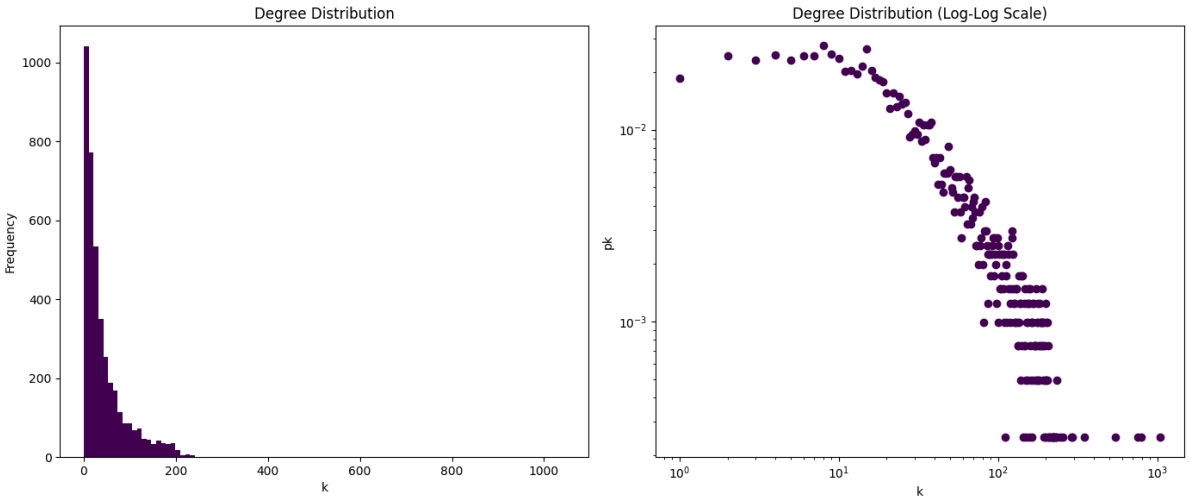
In [ ]:

```
# Plot degree distribution on log-log scale
degree_counts = np.bincount(degree_distribution)
degrees = np.nonzero(degree_counts)[0]
counts = degree_counts[degrees]

plt.figure()
plt.loglog(degrees, counts, 'bo')
plt.title('Degree Distribution (Log-Log Scale)')
plt.xlabel('Degree')
plt.ylabel('Frequency')
plt.show()
```



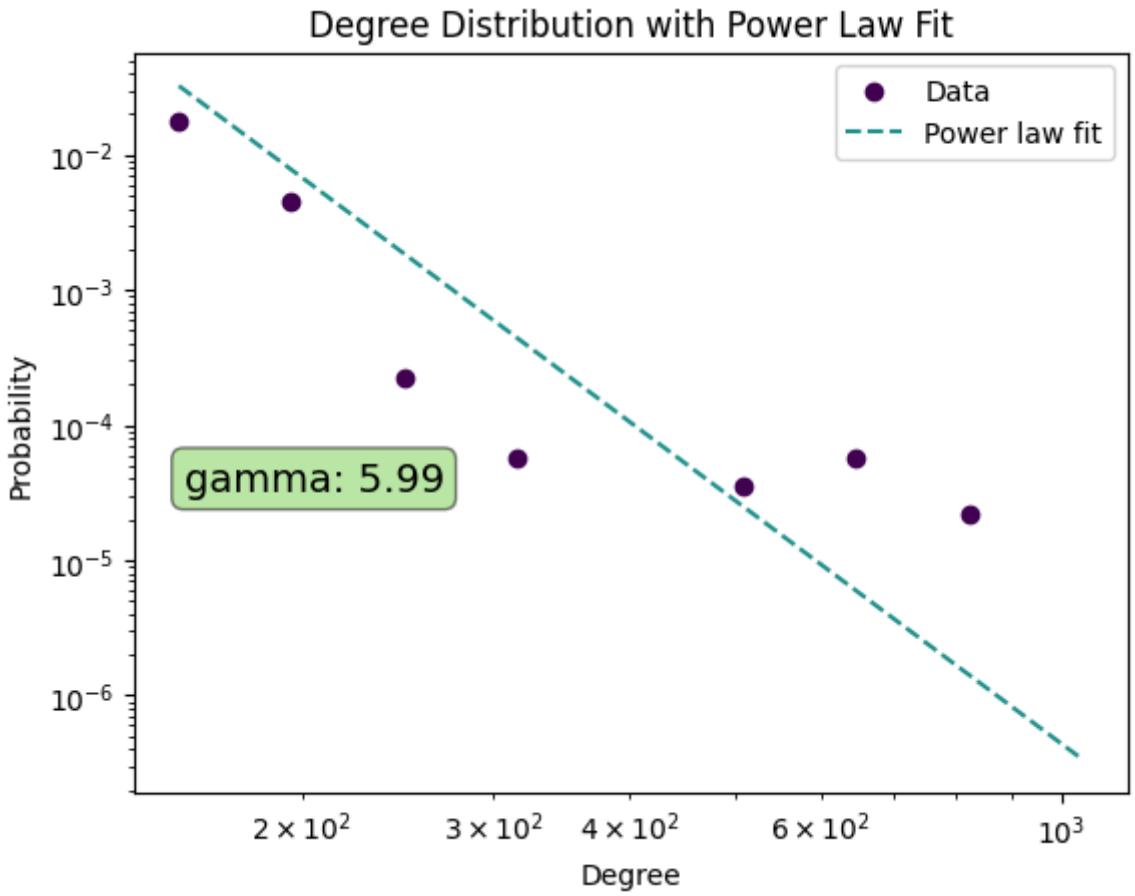
In [ ]:

```
color = plt.cm.viridis(0.0)

# 创建子图
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

# 度分布直方图
ax1.hist(degree_distribution, bins=100, color=color)
ax1.set_title('Degree Distribution')
ax1.set_xlabel('k')
ax1.set_ylabel('Frequency')

# 度分布对数-对数图
degree_counts = np.bincount(degree_distribution)
degrees = np.nonzero(degree_counts)[0]
counts = degree_counts[degrees]

ax2.loglog(degrees, counts/sum(counts), 'o', color=color)
ax2.set_title('Degree Distribution (Log-Log Scale)')
ax2.set_xlabel('k')
ax2.set_ylabel('pk')

# 显示图形
plt.tight_layout()
plt.show()
```



In [ ]:
```
degree_distribution = [d for n, d in G.degree()]
# 使用 powerlaw 库拟合幂律分布
fit = powerlaw.Fit(degree_distribution)

# 绘制度分布的散点图和拟合的幂律分布
fig, ax = plt.subplots()
data_points = fit.pdf()

# 手动过滤零概率的度数
filtered_x = []
filtered_y = []
for x, y in zip(data_points[0], data_points[1]):
    if y > 0:
        filtered_x.append(x)
        filtered_y.append(y)

gamma=5.99
ax.scatter(filtered_x, filtered_y, color=plt.cm.viridis(0.0), label='Data')
textstr = f'gamma: {gamma:.2f}'
props = dict(boxstyle='round', facecolor=plt.cm.viridis(0.8), alpha=0.5)
ax.text(0.05, 0.45, textstr, transform=ax.transAxes, fontsize=14,
        verticalalignment='top', bbox=props)

# 绘制拟合的幂律分布
fit.power_law.plot_pdf(color=plt.cm.viridis(0.5), linestyle='--', ax=ax, label='Pow

# 设置图表标题和标签
ax.set_title('Degree Distribution with Power Law Fit')
ax.set_xlabel('Degree')
ax.set_ylabel('Probability')
ax.legend()
plt.show()


print(f"Alpha (幂律指数): {fit.power_law.alpha}")
print(f"Xmin (最小值): {fit.power_law.xmin}")
# print(f"Log-likelihood: {fit.power_law.loglikelihood}")
```

```
Calculating best minimal value for power law fit
xmin progress: 99%
```

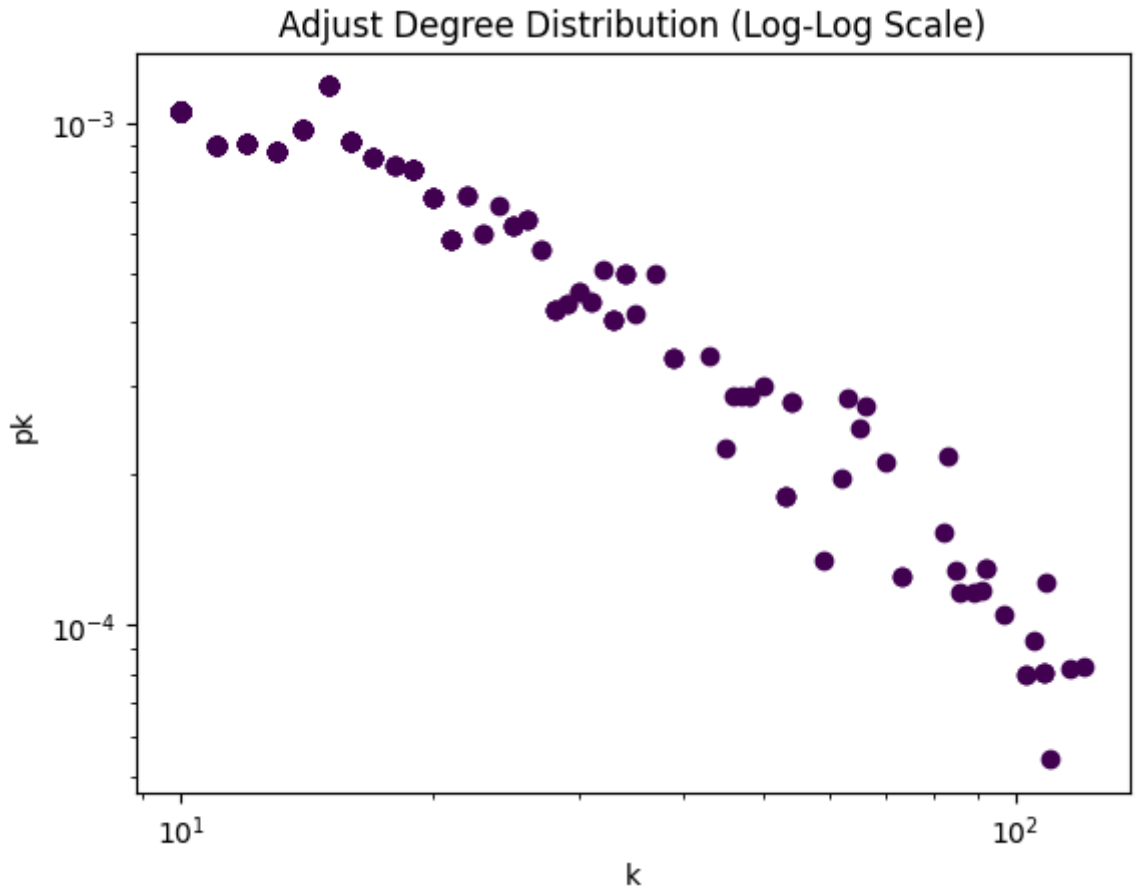



```
Alpha (幂律指数): 5.993294854823435
Xmin (最小值): 154.0
```

In [ ]:

```
ksat=10
kcut=500
# Plot degree distribution on log-log scale
degree_distribution = [d for n, d in G.degree()]
degree_counts = np.bincount(degree_distribution)
degrees = degree_counts+ksat
counts = degree_counts[degrees]
tilde=counts/sum(counts)*np.exp(degrees/kcut)
plt.figure()
plt.loglog(degrees, tilde, 'bo',color=color)
plt.title('Adjust Degree Distribution (Log-Log Scale)')
plt.xlabel('k')
plt.ylabel('pk')
plt.show()
```

```
C:\Users\12137\AppData\Local\Temp\ipykernel_21240\3342627413.py:10: UserWarning: col
or is redundantly defined by the 'color' keyword argument and the fmt string "bo" (-
> color='b'). The keyword argument will take precedence.
  plt.loglog(degrees, tilde, 'bo',color=color)
```

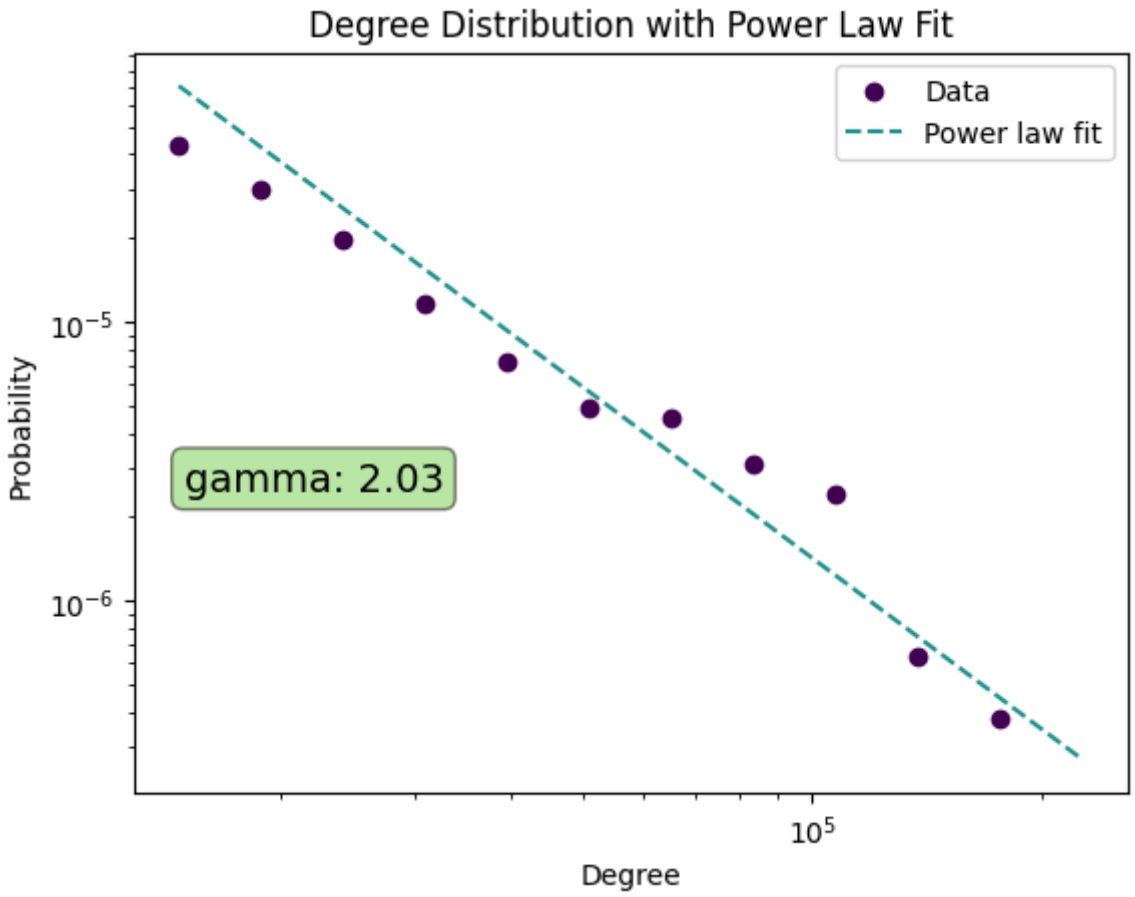


In [ ]:

```
degree_distribution = [d for n, d in G.degree()]
random.seed(42)
ksat=10
kcut=500
degree_distribution = [d * np.exp((degree+ksat)/kcut) for degree, d in enumerate(de

# 使用 powerlaw 库拟合幂律分布
fit = powerlaw.Fit(degree_distribution)

# 绘制度分布的散点图和拟合的幂律分布
fig, ax = plt.subplots()
data_points = fit.pdf()

# 手动过滤零概率的度数
filtered_x = []
filtered_y = []
for x, y in zip(data_points[0], data_points[1]):
    if y > 0:
        filtered_x.append(x)
        filtered_y.append(y)

gamma=2.03
ax.scatter(filtered_x, filtered_y, color=plt.cm.viridis(0.0), label='Data')
textstr = f'gamma: {gamma:.2f}'
props = dict(boxstyle='round', facecolor=plt.cm.viridis(0.8), alpha=0.5)
ax.text(0.05, 0.45, textstr, transform=ax.transAxes, fontsize=14,
        verticalalignment='top', bbox=props)

# 绘制拟合的幂律分布
fit.power_law.plot_pdf(color=plt.cm.viridis(0.5), linestyle='--', ax=ax, label='Pow

# 设置图表标题和标签
ax.set_title('Degree Distribution with Power Law Fit')
ax.set_xlabel('Degree')
ax.set_ylabel('Probability')
```

```
ax.legend()
plt.show()


print(f"Alpha (幂律指数): {fit.power_law.alpha}")
print(f"Xmin (最小值): {fit.power_law.xmin}")
# print(f"Log-likelihood: {fit.power_law.loglikelihood}")
```

```
Calculating best minimal value for power law fit
xmin progress: 99%
```



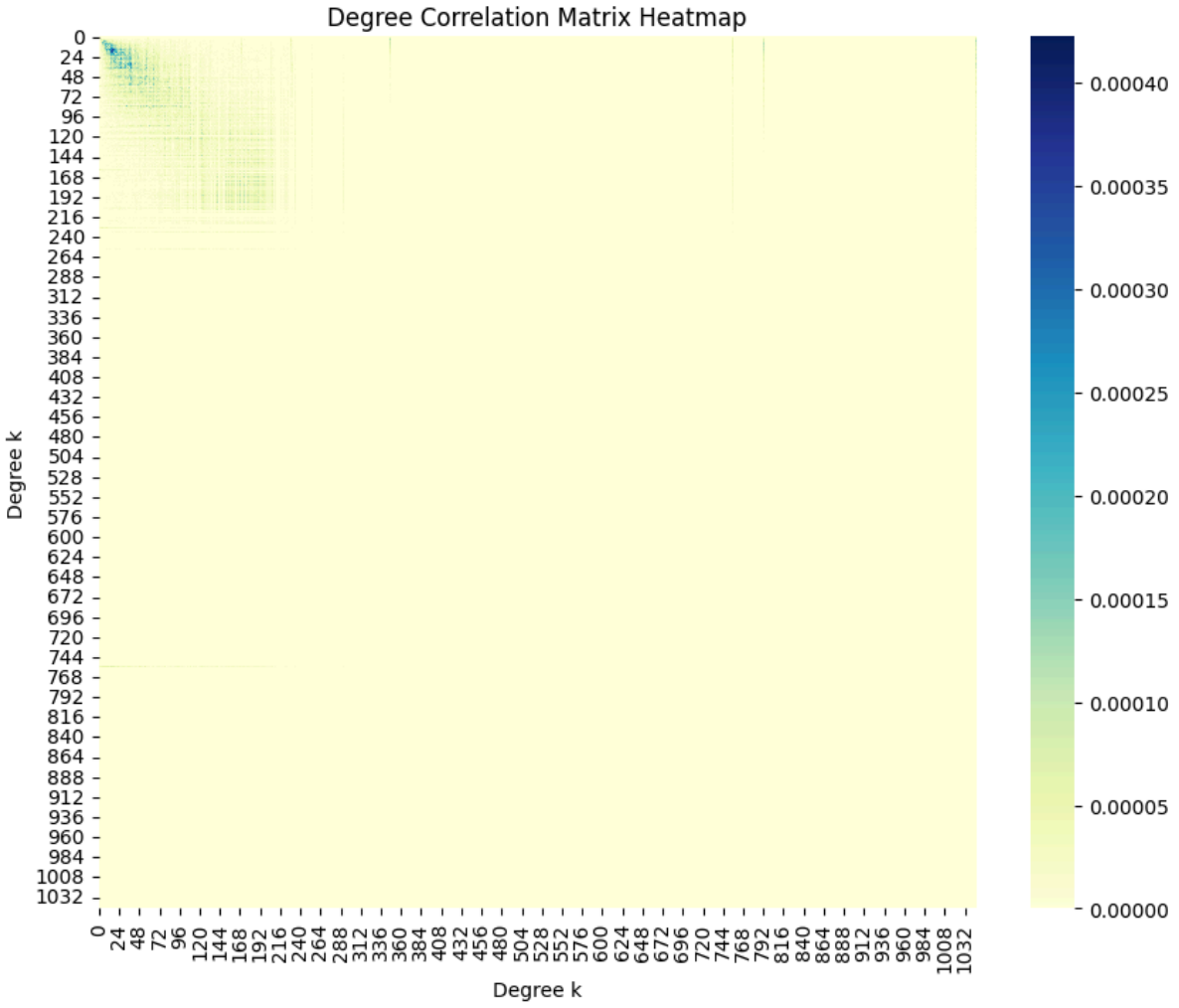

```
Alpha (幂律指数): 2.030095183278526
Xmin (最小值): 14652.41613322138
```

degree correlation

In [ ]:

```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

def degree_correlation_matrix(G):
    max_degree = max(dict(G.degree()).values())
    matrix = np.zeros((max_degree + 1, max_degree + 1))

    for u, v in G.edges():
        d_u = G.degree(u)
        d_v = G.degree(v)
        matrix[d_u][d_v] += 1
        if d_u != d_v:  # If u and v have different degrees, increment symmetric ele
            matrix[d_v][d_u] += 1

    # Normalize by the number of edges
    matrix /= matrix.sum()
    return matrix
```

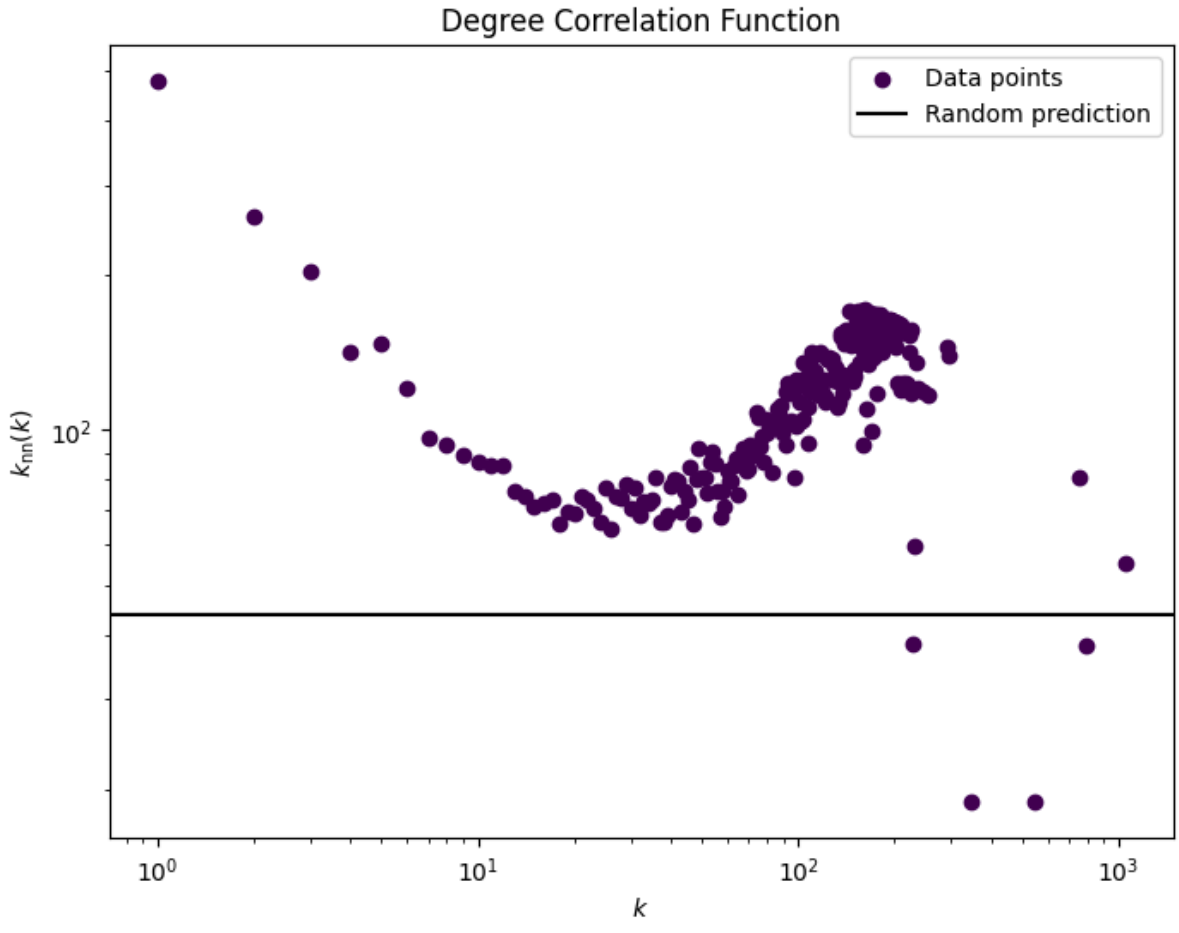
```
# Step 2: Calculate the degree correlation matrix
corr_matrix = degree_correlation_matrix(G)

# Step 3: Plot the heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(corr_matrix, cmap="YlGnBu", norm=plt.Normalize(vmin=0, vmax=corr_matrix.
plt.title('Degree Correlation Matrix Heatmap')
plt.xlabel('Degree k')
plt.ylabel('Degree k')
plt.show()
```



In [ ]:

```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt

import networkx as nx

# 读取 facebook_combined.txt 文件并创建无向图
def create_undirected_graph(file_path):
    G = nx.Graph()

    with open(file_path, 'r') as file:
        for line in file:
            edge = line.strip().split()
            if len(edge) == 2:
                G.add_edge(int(edge[0]), int(edge[1]))

    return G

# 指定文件路径
```

```
file_path = 'facebook_combined.txt'

# 创建无向图
G = create_undirected_graph(file_path)
# Calculate the degree of each node
degrees = dict(G.degree())
degree_values = np.array(list(degrees.values()))

# Calculate k_nn(k)
knn = {}
for node in G.nodes():
    k = G.degree(node)
    knn.setdefault(k, []).append(np.mean([G.degree(neigh) for neigh in G.neighbors(

knn_avg = {k: np.mean(knn_list) for k, knn_list in knn.items()}

# Sort the results for plotting
k_vals = np.array(list(knn_avg.keys()))
knn_vals = np.array(list(knn_avg.values()))

# Plot the results
plt.figure(figsize=(8, 6))
plt.scatter(k_vals, knn_vals, color=color, label='Data points')
# plt.plot(k_vals, knn_vals, color='green', linestyle='--', label=r'$k_{\text{nn}} \s
plt.axhline(y=np.mean(degree_values), color='black', linestyle='-', label='Random pr
plt.xscale('log')
plt.yscale('log')
plt.xlabel(r'$k$')
plt.ylabel(r'$k_{\text{nn}}(k)$')
plt.legend()
plt.title('Degree Correlation Function')
plt.show()
```



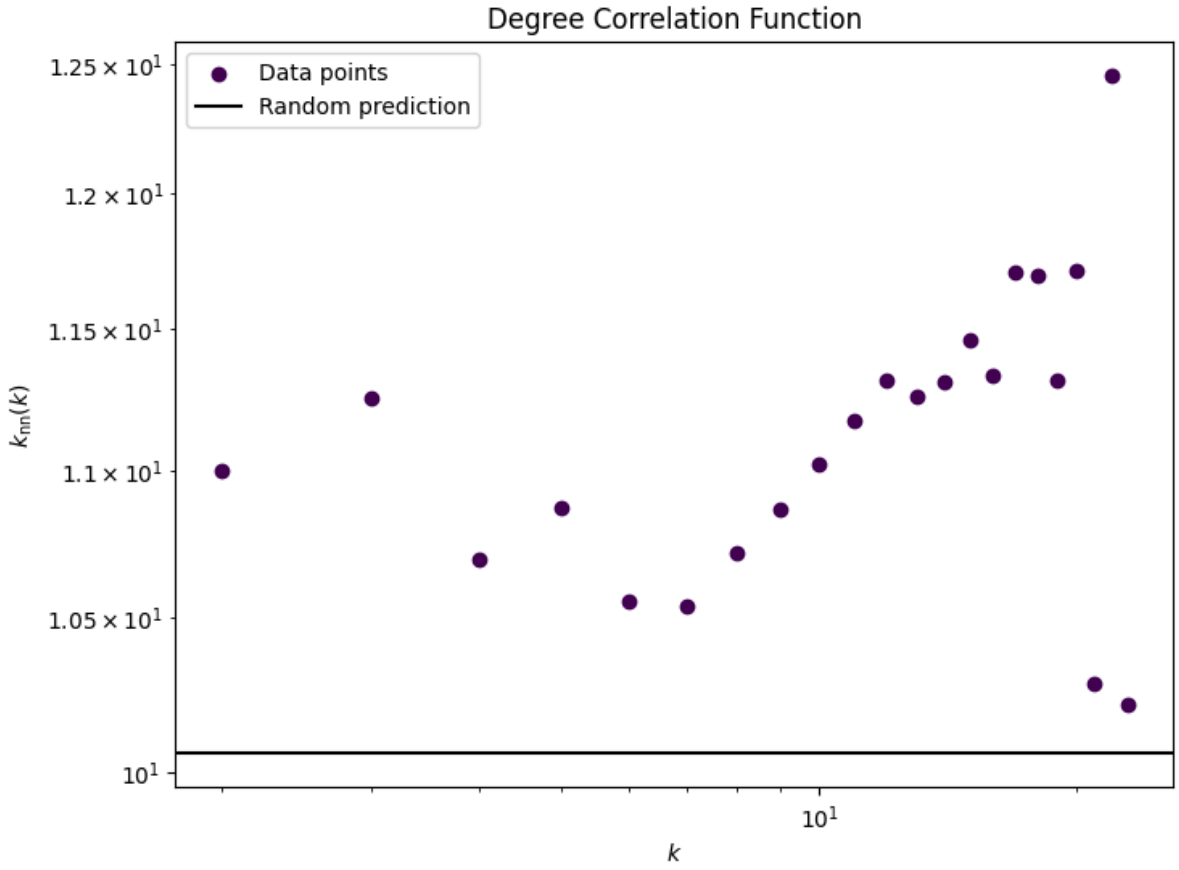
画一个随机生成的，暂时不要

In [ ]:

```
import networkx as nx
import random

def assortative_rewire(G, iterations=1000):
    for _ in range(iterations):
        # 将边转换为列表
        edges = list(G.edges())

        # 随机选择两条边 (u, v) 和 (x, y)
        u, v = random.sample(edges, 1)[0]
        x, y = random.sample(edges, 1)[0]

        # 确保有四个唯一的节点
        if len({u, v, x, y}) < 4:
            continue

        # 计算当前的同配性系数
        current_assortativity = nx.degree_assortativity_coefficient(G)

        # 尝试重连边以增加同配性
        G.remove_edge(u, v)
        G.remove_edge(x, y)
        G.add_edge(u, x)
        G.add_edge(v, y)

        # 计算新的同配性系数
        new_assortativity = nx.degree_assortativity_coefficient(G)

        # 如果新的同配性系数没有增加，则还原重连
        if new_assortativity <= current_assortativity:
            G.remove_edge(u, x)
            G.remove_edge(v, y)
            G.add_edge(u, v)
            G.add_edge(x, y)

    return G

# 生成一个随机网络
n = 1000  # 节点数
p = 0.01  # 边的概率
G = nx.erdos_renyi_graph(n, p)

# 增加同配性
G = assortative_rewire(G, iterations=1000)

# 计算并打印同配性系数
assortativity = nx.degree_assortativity_coefficient(G)
print("Degree assortativity coefficient:", assortativity)

import networkx as nx
import numpy as np
import matplotlib.pyplot as plt


# Calculate the degree of each node
degrees = dict(G.degree())
degree_values = np.array(list(degrees.values()))

# Calculate k_nn(k)
knn = {}
for node in G.nodes():
    k = G.degree(node)
    knn.setdefault(k, []).append(np.mean([G.degree(neigh) for neigh in G.neighbors(
```

```
knn_avg = {k: np.mean(knn_list) for k, knn_list in knn.items()}

# Sort the results for plotting
k_vals = np.array(list(knn_avg.keys()))
knn_vals = np.array(list(knn_avg.values()))

# Plot the results
plt.figure(figsize=(8, 6))
plt.scatter(k_vals, knn_vals, color=color, label='Data points')
# plt.plot(k_vals, knn_vals, color='green', linestyle='--', label=r'$k_{\text{nn}} \
plt.axhline(y=np.mean(degree_values), color='black', linestyle='-', label='Random pr
plt.xscale('log')
plt.yscale('log')
plt.xlabel(r'$k$')
plt.ylabel(r'$k_{\text{nn}}(k)$')
plt.legend()
plt.title('Degree Correlation Function')
plt.show()
```

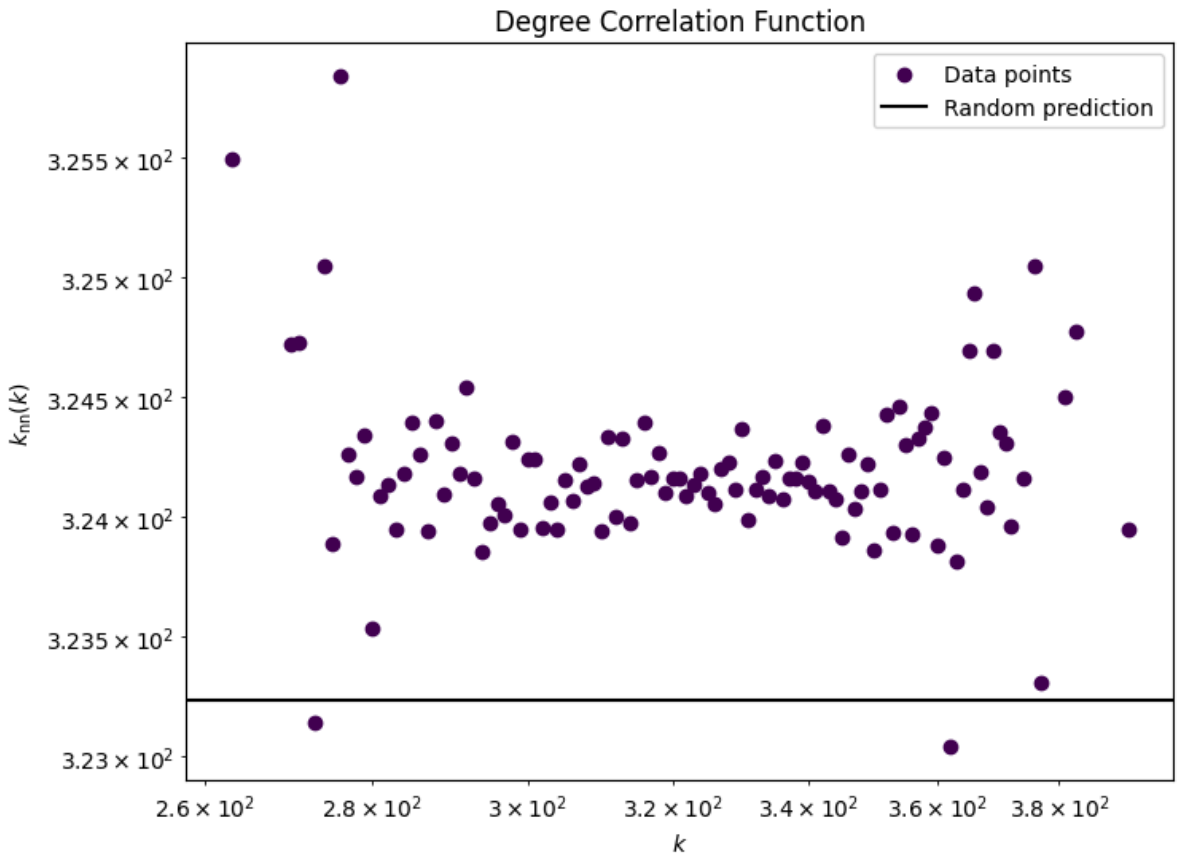
```
Degree assortativity coefficient: 0.07883648824942502
```



In [ ]:

```
GS=G
degrees = dict(GS.degree())
degree_values = np.array(list(degrees.values()))

# Calculate k_nn(k)
knn = {}
for node in GS.nodes():
    k = GS.degree(node)
    knn.setdefault(k, []).append(np.mean([GS.degree(neigh) for neigh in GS.neighbor

knn_avg = {k: np.mean(knn_list) for k, knn_list in knn.items()}

# Sort the results for plotting
k_vals = np.array(list(knn_avg.keys()))
knn_vals = np.array(list(knn_avg.values()))
```

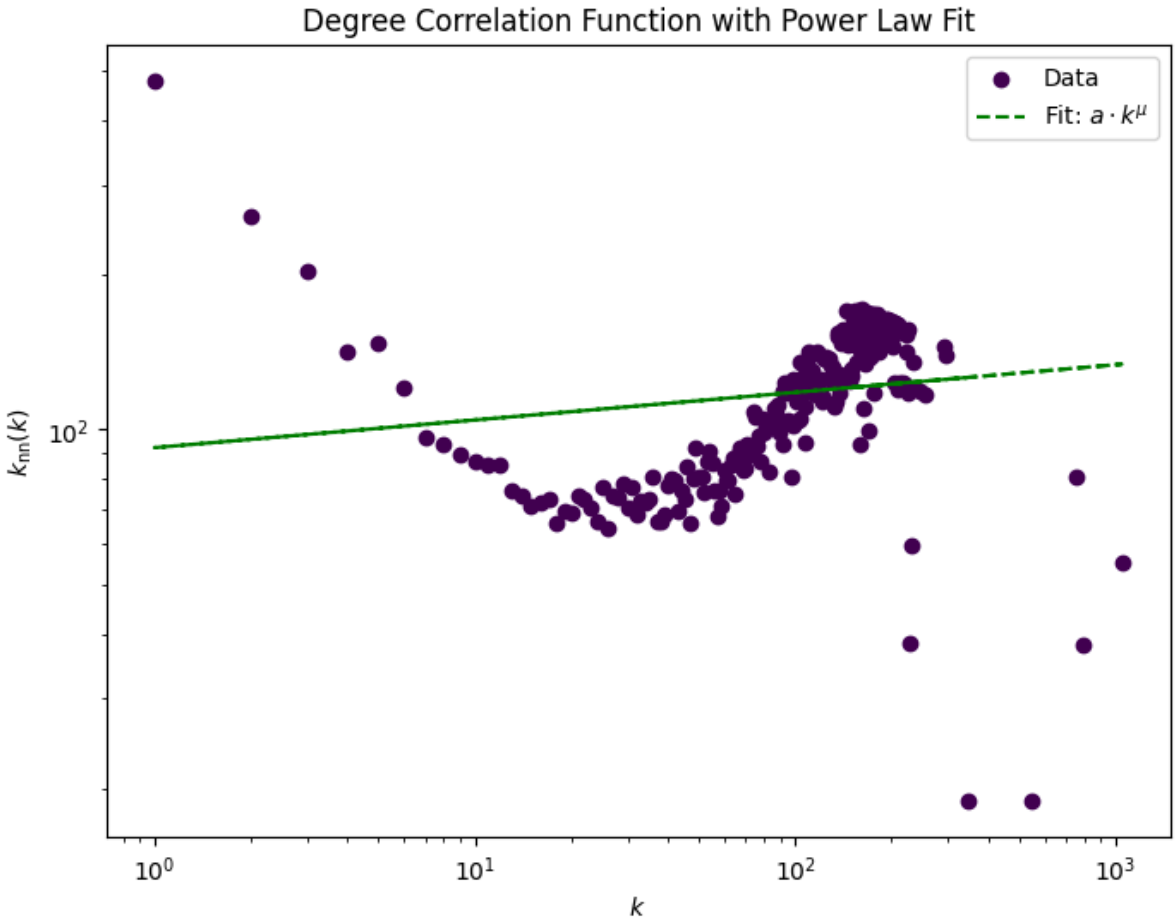
```
# Plot the results
plt.figure(figsize=(8, 6))
plt.scatter(k_vals, knn_vals, color=color, label='Data points')
# plt.plot(k_vals, knn_vals, color='green', linestyle='--', label=r'$k_{\text{nn}} \
plt.axhline(y=np.mean(degree_values), color='black', linestyle='-', label='Random pr
plt.xscale('log')
plt.yscale('log')
plt.xlabel(r'$k$')
plt.ylabel(r'$k_{\text{nn}}(k)$')
plt.legend()
plt.title('Degree Correlation Function')
plt.show()
```



暂时不要

In [ ]:
```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def average_neighbor_degree(G):
    knn = {}
    for node in G.nodes():
        k = G.degree(node)
        if k > 0:  # Avoid division by zero
            knn.setdefault(k, []).append(np.mean([G.degree(neigh) for neigh in G.ne
    knn_avg = {k: np.mean(knn_list) for k, knn_list in knn.items()}
    return knn_avg

def power_law(x, a, mu):
    return a * np.power(x, mu)

# Step 2: Calculate k_nn(k) for the original network
knn_original = average_neighbor_degree(G)
```

```
# Step 3: Extract data for fitting
k_vals = np.array(list(knn_original.keys()))
knn_vals = np.array(list(knn_original.values()))

# Step 4: Fit k_nn(k) to a*k^mu
popt, pcov = curve_fit(power_law, k_vals, knn_vals)

# Step 5: Plot the results
plt.figure(figsize=(8, 6))
plt.scatter(k_vals, knn_vals, color=color, label='Data')
plt.plot(k_vals, power_law(k_vals, *popt), color='green', linestyle='--', label=r'Fi

plt.xscale('log')
plt.yscale('log')
plt.xlabel(r'$k$')
plt.ylabel(r'$k_{\text{nn}}(k)$')
plt.legend()
plt.title('Degree Correlation Function with Power Law Fit')
plt.show()

print(f"Fitted parameters: a = {popt[0]}, mu = {popt[1]}")
```




```
Fitted parameters: a = 92.2630687839182, mu = 0.053877352341481236
```

cut

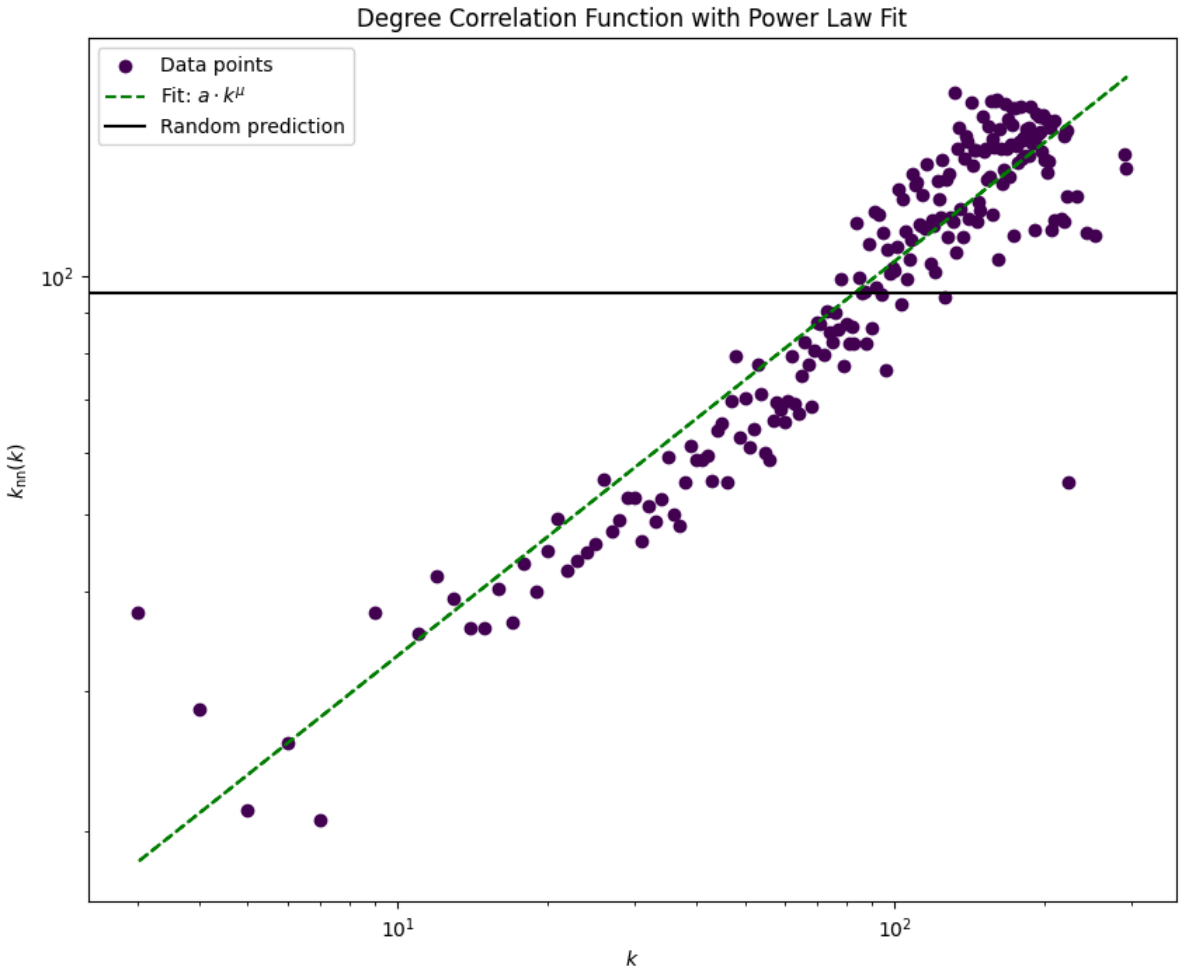
In [ ]:

```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
# 读取 facebook_combined.txt 文件并创建无向图
def create_undirected_graph(file_path):
    G = nx.Graph()
```

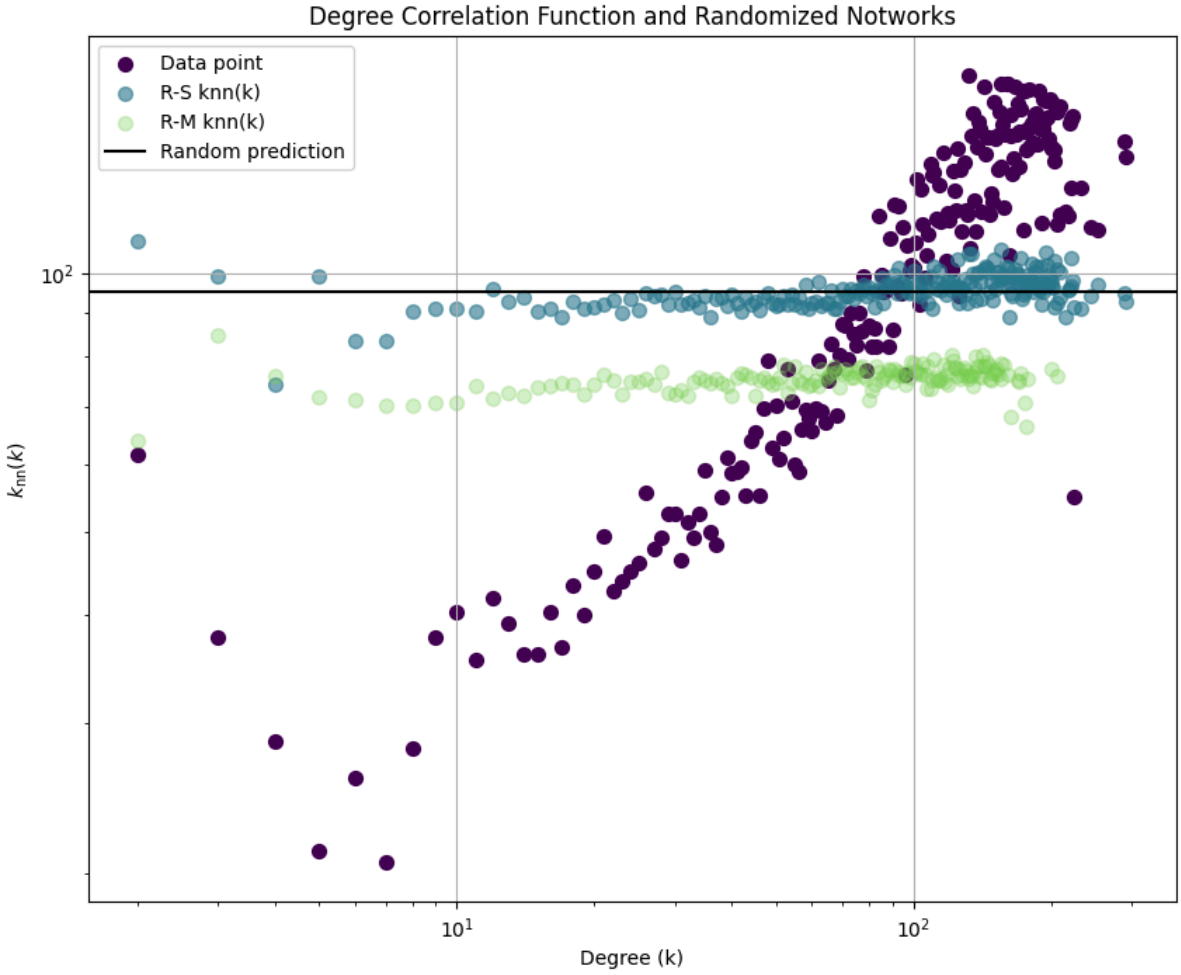
```
    with open(file_path, 'r') as file:
        for line in file:
            edge = line.strip().split()
            if len(edge) == 2:
                G.add_edge(int(edge[0]), int(edge[1]))
    return G

# 指定文件路径
file_path = 'facebook_combined.txt'

# 创建无向图
G = create_undirected_graph(file_path)

def simple_link_preserving_randomization(G, iterations=100000):
    H = G.copy()
    edges = list(H.edges())

    for _ in range(iterations):
        # Pick two random edges
        e1, e2 = np.random.choice(len(edges), 2, replace=False)
        u1, v1 = edges[e1]
        u2, v2 = edges[e2]

        # Ensure the nodes are unique and we do not create multiple edges or self-loo
        if len({u1, v1, u2, v2}) == 4:
            if not (H.has_edge(u1, v2) or H.has_edge(u2, v1)):
                H.remove_edge(u1, v1)
                H.remove_edge(u2, v2)
                H.add_edge(u1, v2)
                H.add_edge(u2, v1)
                edges[e1] = (u1, v2)
                edges[e2] = (u2, v1)
    return H
def multiple_link_preserving_randomization(G, iterations=100000):
    H = G.copy()
    edges = list(H.edges())

    for _ in range(iterations):
        # Pick two random edges
        e1, e2 = np.random.choice(len(edges), 2, replace=False)
        u1, v1 = edges[e1]
        u2, v2 = edges[e2]

        # Ensure the nodes are unique
        if len({u1, v1, u2, v2}) == 4:
            # Remove edges if they exist
            if H.has_edge(u1, v1) and H.has_edge(u2, v2):
                H.remove_edge(u1, v1)
                H.remove_edge(u2, v2)
                H.add_edge(u1, v2)
                H.add_edge(u2, v1)
                edges[e1] = (u1, v2)
                edges[e2] = (u2, v1)
    return H

def average_neighbor_degree(G):
    knn = {}
    for node in G.nodes():
        k = G.degree(node)
        if k > 10 & k<500:  # Avoid division by zero
            knn.setdefault(k, []).append(np.mean([G.degree(neigh) for neigh in G.ne
    knn_avg = {k: np.mean(knn_list) for k, knn_list in knn.items()}
    return knn_avg
```

```
def power_law(x, a, mu):
    return a * np.power(x, mu)

import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

def remove_nodes_by_degree(G, min_degree=10, max_degree=500):
    nodes_to_remove = [node for node, degree in dict(G.degree()).items() if degree
    G.remove_nodes_from(nodes_to_remove)
    return G

# 指定文件路径
file_path = 'facebook_combined.txt'

# 创建无向图
G = create_undirected_graph(file_path)

# 删除度小于10和大于500的点
G = remove_nodes_by_degree(G, min_degree=10, max_degree=500)


# Step 2: Calculate k_nn(k) for the original network
knn_original = average_neighbor_degree(G)

# Step 3: Extract data for fitting
k_vals = np.array(list(knn_original.keys()))
knn_vals = np.array(list(knn_original.values()))

# Step 4: Fit k_nn(k) to a*k^mu
popt, pcov = curve_fit(power_law, k_vals, knn_vals)

# Step 5: Calculate <k> and <k^2>
degrees = np.array([d for n, d in G.degree()])
k_mean = np.mean(degrees)
k_square_mean = np.mean(degrees**2)
k_ratio = k_square_mean / k_mean

# Step 6: Plot the results
plt.figure(figsize=(10, 8))
plt.scatter(k_vals, knn_vals, color=plt.cm.viridis(0.0), label='Data points')
plt.plot(k_vals, power_law(k_vals, *popt), color='green', linestyle='--', label=r'Fi
plt.axhline(y=k_ratio, color='black', linestyle='-', label='Random prediction')
plt.xscale('log')
plt.yscale('log')
plt.xlabel(r'$k$')
plt.ylabel(r'$k_{\text{nn}}(k)$')
plt.legend()
plt.title('Degree Correlation Function with Power Law Fit')
plt.show()

# Print fitted parameters
print(f"Fitted parameters: a = {popt[0]}, mu = {popt[1]}")
print(f"<k^2>/<k> = {k_ratio}")
G_rs = simple_link_preserving_randomization(G, iterations=100000)
G_rm = multiple_link_preserving_randomization(G, iterations=100000)
# Step 2: Calculate k_nn(k) for the original network
knn_original = average_neighbor_degree(G)
# Function to calculate knn(k)
def calculate_knn(G):
    knn = nx.average_degree_connectivity(G)
    degrees = np.array(list(knn.keys()))
    knn_values = np.array(list(knn.values()))
    return degrees, knn_values
```

```python
# Calculate knn(k) for original and randomized networks
degrees_orig, knn_orig = calculate_knn(G)
degrees_rs, knn_rs = calculate_knn(G_rs)
degrees_rm, knn_rm = calculate_knn(G_rm)


# Calculate <k^2>/<k>
degree_sequence = [d for n, d in G.degree()]
k_sq_avg = np.mean(np.array(degree_sequence) ** 2)
k_avg = np.mean(degree_sequence)
k2_div_k = k_sq_avg / k_avg




# Plot knn(k) vs k
plt.figure(figsize=(10, 8))
plt.scatter(degrees_orig, knn_orig, color=plt.cm.viridis(0.0), label='Data point', s
plt.scatter(degrees_rs, knn_rs, color=plt.cm.viridis(0.4), label='R-S knn(k)', s=50,
plt.scatter(degrees_rm, knn_rm, color=plt.cm.viridis(0.8), label='R-M knn(k)', s=50,
plt.axhline(y=k2_div_k, color='k', linestyle='-', label='Random prediction')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Degree (k)')
plt.ylabel(r'$k_{\text{nn}}(k)$')
plt.title('Degree Correlation Function and Randomized Notworks')
plt.legend()
plt.grid(True)
plt.show()
```




```
Fitted parameters: a = 10.57466353617884, mu = 0.4971805920225443
<k^2>/<k> = 95.37279988257886
```

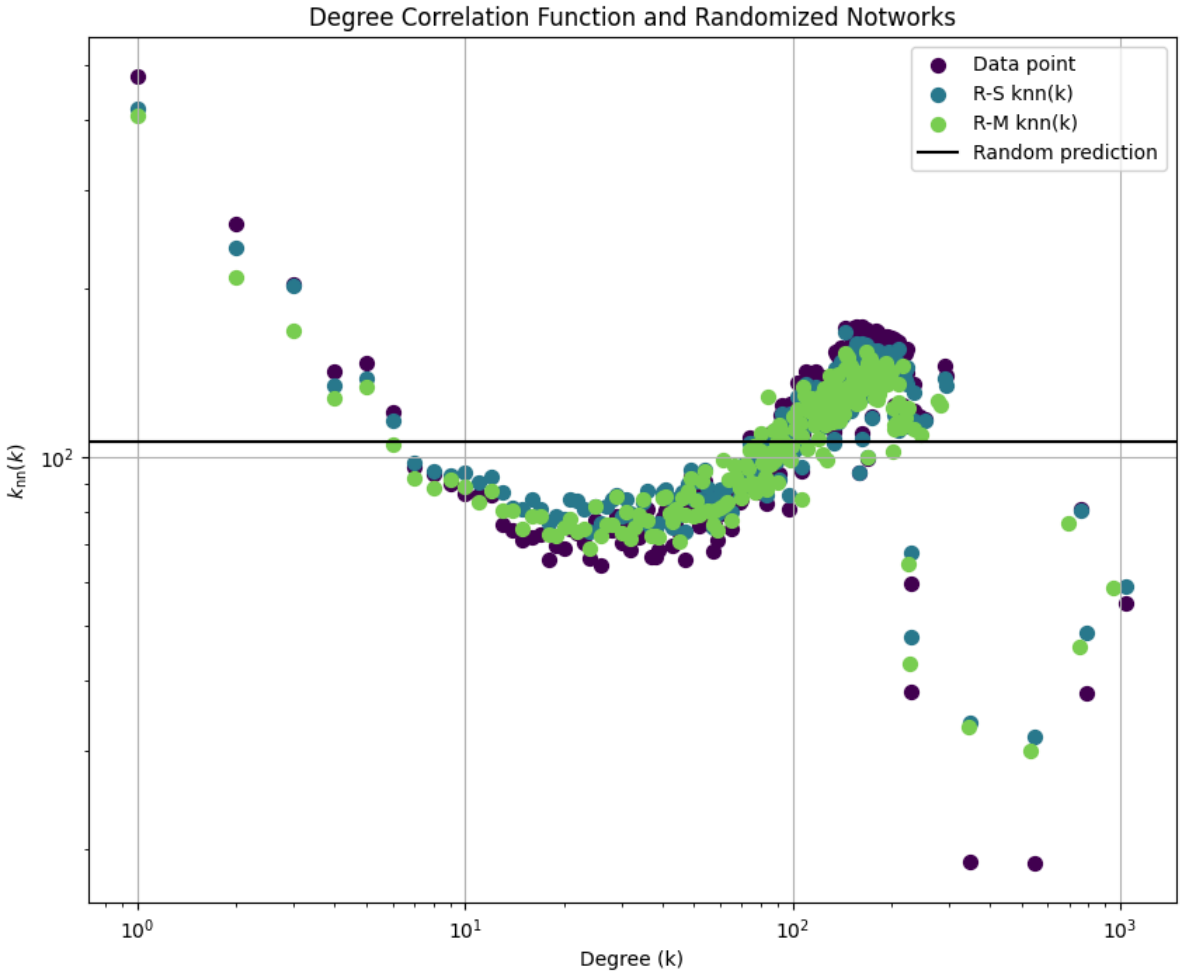


原网络

In [ ]:

```
# Perform randomizations
G_rs = simple_link_preserving_randomization(G, iterations=10000)
G_rm = multiple_link_preserving_randomization(G, iterations=10000)

# Function to calculate knn(k)
def calculate_knn(G):
    knn = nx.average_degree_connectivity(G)
    degrees = np.array(list(knn.keys()))
    knn_values = np.array(list(knn.values()))
    return degrees, knn_values

# Calculate knn(k) for original and randomized networks
degrees_orig, knn_orig = calculate_knn(G)
degrees_rs, knn_rs = calculate_knn(G_rs)
degrees_rm, knn_rm = calculate_knn(G_rm)


# Calculate <k^2>/<k>
degree_sequence = [d for n, d in G.degree()]
k_sq_avg = np.mean(np.array(degree_sequence) ** 2)
k_avg = np.mean(degree_sequence)
k2_div_k = k_sq_avg / k_avg




# Plot knn(k) vs k
plt.figure(figsize=(10, 8))
plt.scatter(degrees_orig, knn_orig, color=color, label='Data point', s=50)
```

```
plt.scatter(degrees_rs, knn_rs, color=plt.cm.viridis(0.4), label='R-S knn(k)', s=50)
plt.scatter(degrees_rm, knn_rm, color=plt.cm.viridis(0.8), label='R-M knn(k)', s=50)
plt.axhline(y=k2_div_k, color='k', linestyle='-', label='Random prediction')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Degree (k)')
plt.ylabel(r'$k_{\text{nn}}(k)$')
plt.title('Degree Correlation Function and Randomized Notworks')
plt.legend()
plt.grid(True)
plt.show()
```

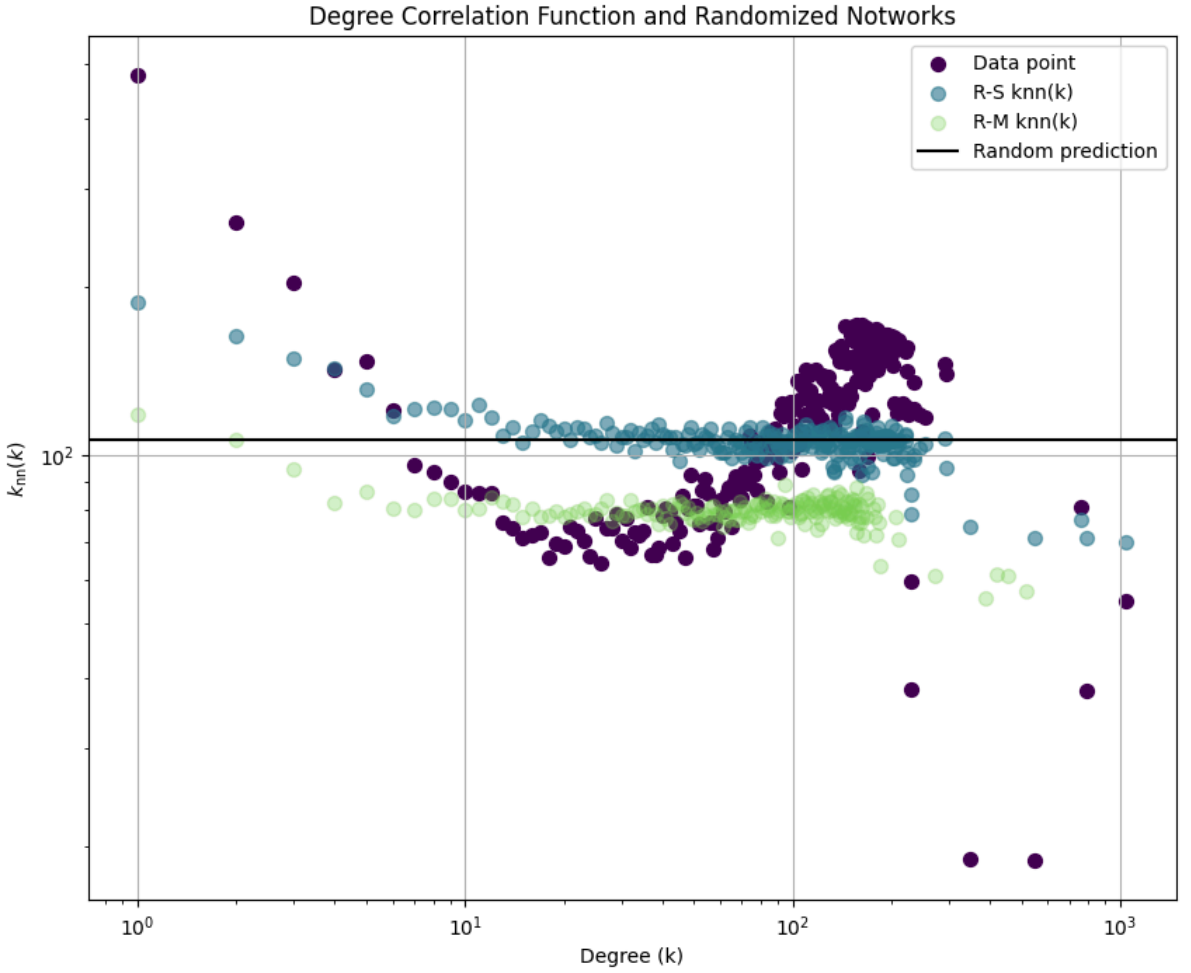


In [ ]:

```
# 读取 facebook_combined.txt 文件并创建无向图
def create_undirected_graph(file_path):
    G = nx.Graph()
    with open(file_path, 'r') as file:
        for line in file:
            edge = line.strip().split()
            if len(edge) == 2:
                G.add_edge(int(edge[0]), int(edge[1]))
    return G

# 指定文件路径
file_path = 'facebook_combined.txt'

# 创建无向图
G = create_undirected_graph(file_path)

def simple_link_preserving_randomization(G, iterations=100000):
    H = G.copy()
    edges = list(H.edges())

    for _ in range(iterations):
```

```
        # Pick two random edges
        e1, e2 = np.random.choice(len(edges), 2, replace=False)
        u1, v1 = edges[e1]
        u2, v2 = edges[e2]

        # Ensure the nodes are unique and we do not create multiple edges or self-loo
        if len({u1, v1, u2, v2}) == 4:
            if not (H.has_edge(u1, v2) or H.has_edge(u2, v1)):
                H.remove_edge(u1, v1)
                H.remove_edge(u2, v2)
                H.add_edge(u1, v2)
                H.add_edge(u2, v1)
                edges[e1] = (u1, v2)
                edges[e2] = (u2, v1)
    return H

def multiple_link_preserving_randomization(G, iterations=100000):
    H = G.copy()
    edges = list(H.edges())

    for _ in range(iterations):
        # Pick two random edges
        e1, e2 = np.random.choice(len(edges), 2, replace=False)
        u1, v1 = edges[e1]
        u2, v2 = edges[e2]

        # Ensure the nodes are unique
        if len({u1, v1, u2, v2}) == 4:
            # Remove edges if they exist
            if H.has_edge(u1, v1) and H.has_edge(u2, v2):
                H.remove_edge(u1, v1)
                H.remove_edge(u2, v2)
                H.add_edge(u1, v2)
                H.add_edge(u2, v1)
                edges[e1] = (u1, v2)
                edges[e2] = (u2, v1)
    return H

# Step 2: Calculate k_nn(k) for the original network
knn_original = average_neighbor_degree(G)
# Function to calculate knn(k)
def calculate_knn(G):
    knn = nx.average_degree_connectivity(G)
    degrees = np.array(list(knn.keys()))
    knn_values = np.array(list(knn.values()))
    return degrees, knn_values
G_rs = simple_link_preserving_randomization(G, iterations=100000)
G_rm = multiple_link_preserving_randomization(G, iterations=100000)
# Calculate knn(k) for original and randomized networks
degrees_orig, knn_orig = calculate_knn(G)
degrees_rs, knn_rs = calculate_knn(G_rs)
degrees_rm, knn_rm = calculate_knn(G_rm)


# Calculate <k^2>/<k>
degree_sequence = [d for n, d in G.degree()]
k_sq_avg = np.mean(np.array(degree_sequence) ** 2)
k_avg = np.mean(degree_sequence)
k2_div_k = k_sq_avg / k_avg

# Plot knn(k) vs k
plt.figure(figsize=(10, 8))
plt.scatter(degrees_orig, knn_orig, color=color, label='Data point', s=50)
plt.scatter(degrees_rs, knn_rs, color=plt.cm.viridis(0.4), label='R-S knn(k)', s=50,
```

```
plt.scatter(degrees_rm, knn_rm, color=plt.cm.viridis(0.8), label='R-M knn(k)', s=50,
plt.axhline(y=k2_div_k, color='k', linestyle='-', label='Random prediction')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Degree (k)')
plt.ylabel(r'$k_{\text{nn}}(k)$')
plt.title('Degree Correlation Function and Randomized Notworks')
plt.legend()
plt.grid(True)
plt.show()
```



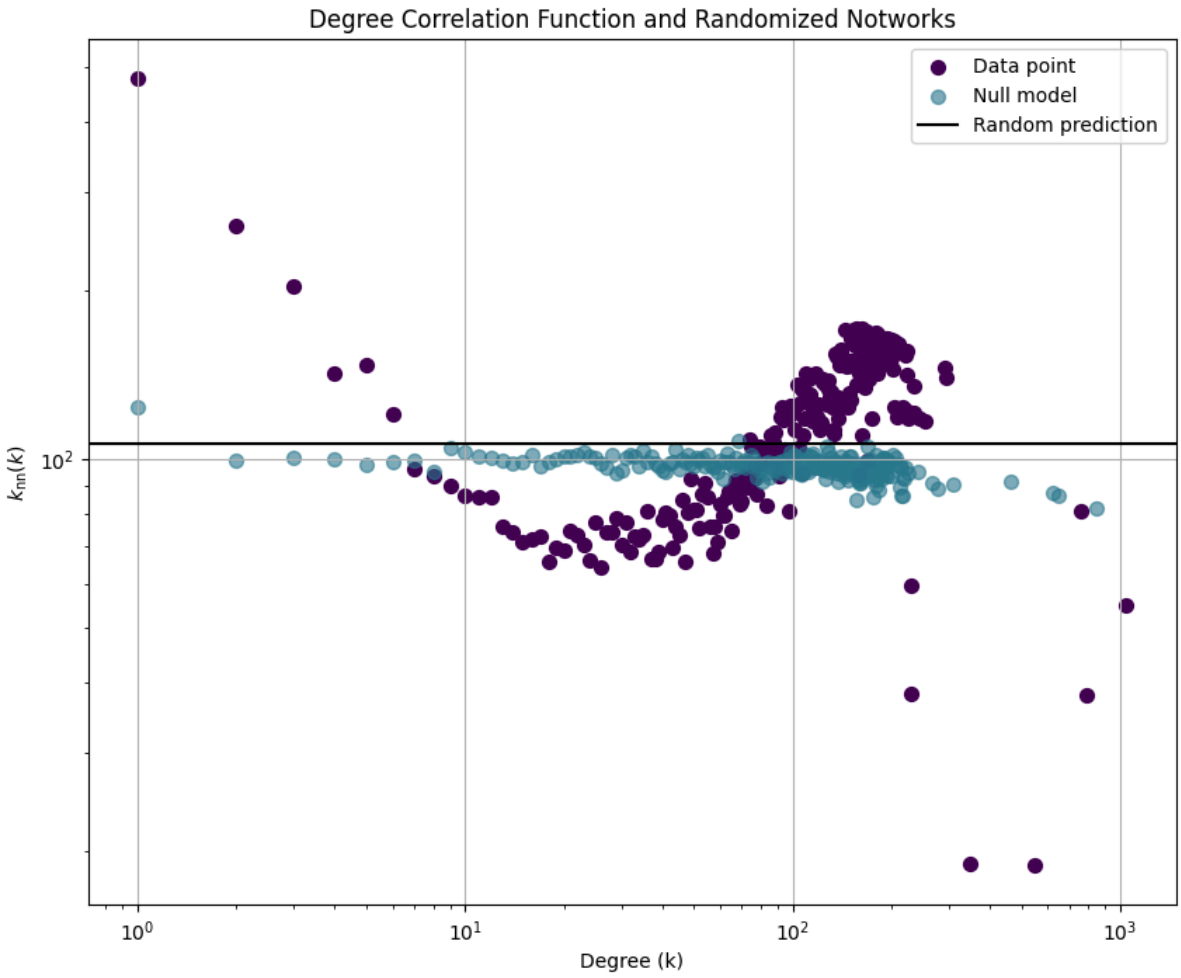
In [ ]:

```
import networkx as nx
import numpy as np


def create_undirected_graph(file_path):
    G = nx.Graph()
    with open(file_path, 'r') as file:
        for line in file:
            edge = line.strip().split()
            if len(edge) == 2:
                G.add_edge(int(edge[0]), int(edge[1]))
    return G

# 指定文件路径
file_path = 'facebook_combined.txt'

# 创建无向图
G = create_undirected_graph(file_path)


def create_configuration_model(G):
    """
```

```
    创建一个配置模型（Configuration Model），保持节点度分布不变。

    参数：
    G: 初始网络

    返回：
    H: 一个具有相同度分布的零模型网络
    """
    # 获取初始网络的度序列
    degree_sequence = [d for n, d in G.degree()]

    # 使用度序列生成配置模型
    H = nx.configuration_model(degree_sequence)

    # 去除多重边和自环
    H = nx.Graph(H)  # 去除多重边
    H.remove_edges_from(nx.selfloop_edges(H))  # 去除自环

    return H


def simple_link_preserving_randomization(G, iterations=1000):
    H = G.copy()
    edges = list(H.edges())

    for _ in range(iterations):
        # Pick two random edges
        e1, e2 = np.random.choice(len(edges), 2, replace=False)
        u1, v1 = edges[e1]
        u2, v2 = edges[e2]

        # Ensure the nodes are unique and we do not create multiple edges or self-loo
        if len({u1, v1, u2, v2}) == 4:
            if not (H.has_edge(u1, v2) or H.has_edge(u2, v1)):
                H.remove_edge(u1, v1)
                H.remove_edge(u2, v2)
                H.add_edge(u1, v2)
                H.add_edge(u2, v1)
                edges[e1] = (u1, v2)
                edges[e2] = (u2, v1)
    return H

def multiple_link_preserving_randomization(G, iterations=100000):
    H = G.copy()
    edges = list(H.edges())

    for _ in range(iterations):
        # Pick two random edges
        e1, e2 = np.random.choice(len(edges), 2, replace=False)
        u1, v1 = edges[e1]
        u2, v2 = edges[e2]

        # Ensure the nodes are unique
        if len({u1, v1, u2, v2}) == 4:
            # Remove edges if they exist
            if H.has_edge(u1, v1) and H.has_edge(u2, v2):
                H.remove_edge(u1, v1)
                H.remove_edge(u2, v2)
                H.add_edge(u1, v2)
                H.add_edge(u2, v1)
                edges[e1] = (u1, v2)
                edges[e2] = (u2, v1)
    return H
```

```python
# 创建配置模型零网络
H = create_configuration_model(G)

# 计算并比较网络特性
print("Original network assortativity:", nx.degree_assortativity_coefficient(G))
print("Null model assortativity:", nx.degree_assortativity_coefficient(H))


# Step 2: Calculate k_nn(k) for the original network
knn_original = average_neighbor_degree(G)
# Function to calculate knn(k)
def calculate_knn(G):
    knn = nx.average_degree_connectivity(G)
    degrees = np.array(list(knn.keys()))
    knn_values = np.array(list(knn.values()))
    return degrees, knn_values

# Calculate knn(k) for original and randomized networks
degrees_orig, knn_orig = calculate_knn(G)
degrees_rs, knn_rs = calculate_knn(H)

# Calculate <k^2>/<k>
degree_sequence = [d for n, d in G.degree()]
k_sq_avg = np.mean(np.array(degree_sequence) ** 2)
k_avg = np.mean(degree_sequence)
k2_div_k = k_sq_avg / k_avg

# Plot knn(k) vs k
plt.figure(figsize=(10, 8))
plt.scatter(degrees_orig, knn_orig, color=color, label='Data point', s=50)
plt.scatter(degrees_rs, knn_rs, color=plt.cm.viridis(0.4), label='Null model', s=50,
# plt.scatter(degrees_rm, knn_rm, color=plt.cm.viridis(0.8), label='R-M knn(k)', s=5(
plt.axhline(y=k2_div_k, color='k', linestyle='-', label='Random prediction')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Degree (k)')
plt.ylabel(r'$k_{\text{nn}}(k)$')
plt.title('Degree Correlation Function and Randomized Notworks')
plt.legend()
plt.grid(True)
plt.show()
```

```
Original network assortativity: 0.06357722918564943
Null model assortativity: -0.023640879354844904
```



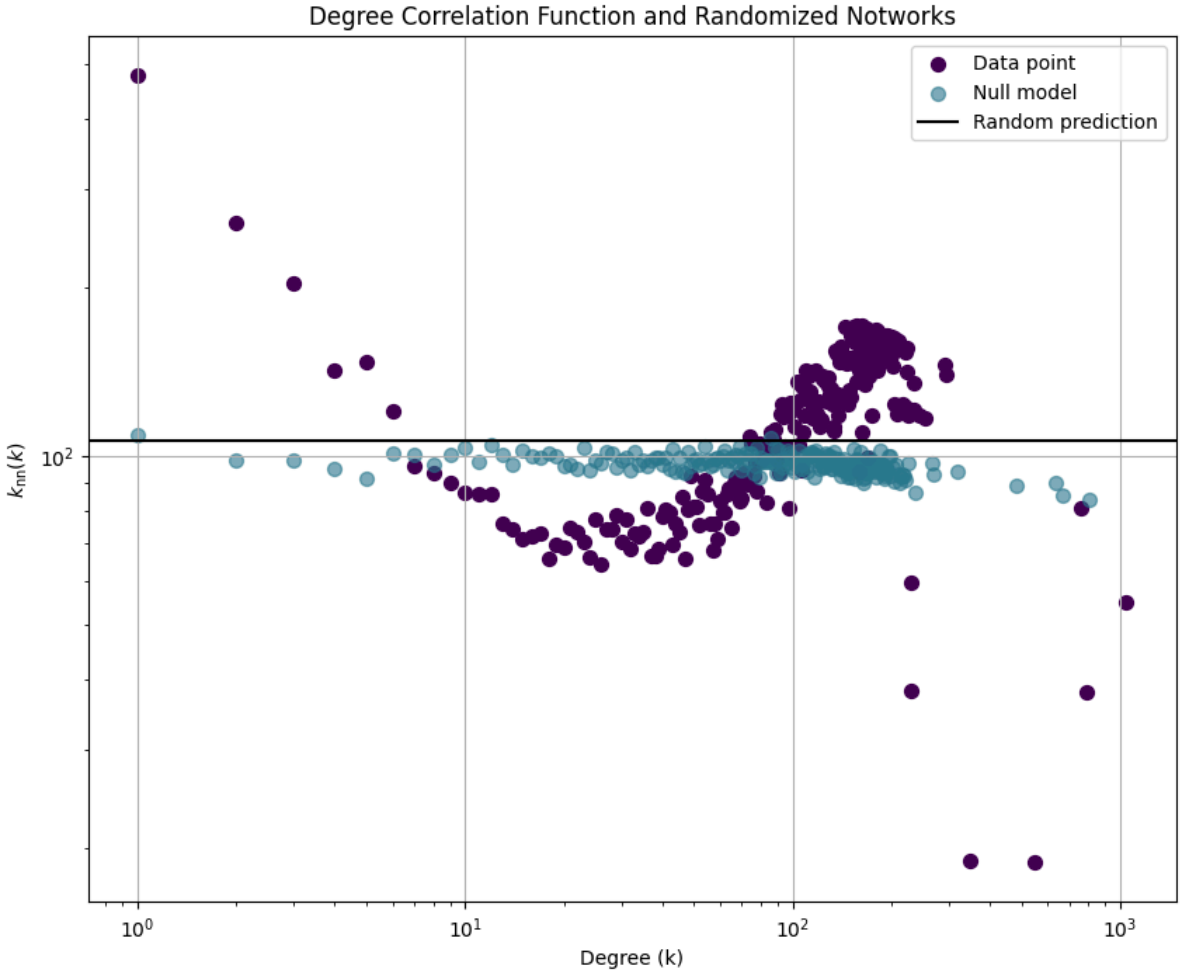
In [ ]:

```python
import networkx as nx
import numpy as np


def create_undirected_graph(file_path):
    G = nx.Graph()
    with open(file_path, 'r') as file:
        for line in file:
            edge = line.strip().split()
            if len(edge) == 2:
                G.add_edge(int(edge[0]), int(edge[1]))
    return G

# 指定文件路径
file_path = 'facebook_combined.txt'

# 创建无向图
G = create_undirected_graph(file_path)


def create_configuration_model(G):
    """
    创建一个配置模型（Configuration Model），保持节点度分布不变。

    参数:
    G: 初始网络

    返回:
    H: 一个具有相同度分布的零模型网络
    """
    # 获取初始网络的度序列
    degree_sequence = [d for n, d in G.degree()]
```

```
    # 使用度序列生成配置模型
    H = nx.configuration_model(degree_sequence)

    # 去除多重边和自环
    H = nx.Graph(H)  # 去除多重边
    H.remove_edges_from(nx.selfloop_edges(H))  # 去除自环

    return H


def simple_link_preserving_randomization(G, iterations=1000):
    H = G.copy()
    edges = list(H.edges())

    for _ in range(iterations):
        # Pick two random edges
        e1, e2 = np.random.choice(len(edges), 2, replace=False)
        u1, v1 = edges[e1]
        u2, v2 = edges[e2]

        # Ensure the nodes are unique and we do not create multiple edges or self-loo
        if len({u1, v1, u2, v2}) == 4:
            if not (H.has_edge(u1, v2) or H.has_edge(u2, v1)):
                H.remove_edge(u1, v1)
                H.remove_edge(u2, v2)
                H.add_edge(u1, v2)
                H.add_edge(u2, v1)
                edges[e1] = (u1, v2)
                edges[e2] = (u2, v1)
    return H

def multiple_link_preserving_randomization(G, iterations=100000):
    H = G.copy()
    edges = list(H.edges())

    for _ in range(iterations):
        # Pick two random edges
        e1, e2 = np.random.choice(len(edges), 2, replace=False)
        u1, v1 = edges[e1]
        u2, v2 = edges[e2]

        # Ensure the nodes are unique
        if len({u1, v1, u2, v2}) == 4:
            # Remove edges if they exist
            if H.has_edge(u1, v1) and H.has_edge(u2, v2):
                H.remove_edge(u1, v1)
                H.remove_edge(u2, v2)
                H.add_edge(u1, v2)
                H.add_edge(u2, v1)
                edges[e1] = (u1, v2)
                edges[e2] = (u2, v1)
    return H

# 创建配置模型零网络
H = create_configuration_model(G)

# 计算并比较网络特性
print("Original network assortativity:", nx.degree_assortativity_coefficient(G))
print("Null model assortativity:", nx.degree_assortativity_coefficient(H))


# Step 2: Calculate k_nn(k) for the original network
knn_original = average_neighbor_degree(G)
```

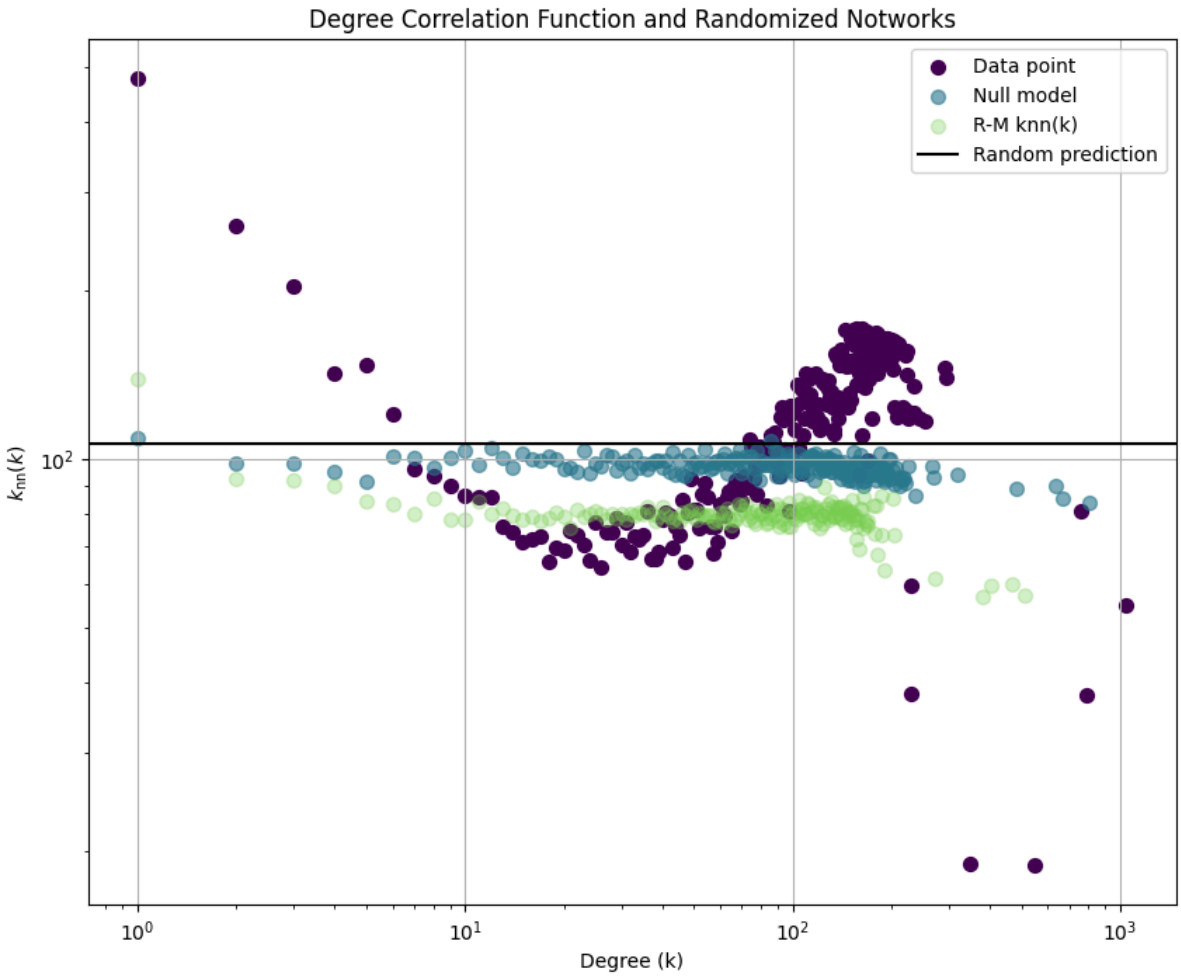
```
# Function to calculate knn(k)
def calculate_knn(G):
    knn = nx.average_degree_connectivity(G)
    degrees = np.array(list(knn.keys()))
    knn_values = np.array(list(knn.values()))
    return degrees, knn_values
# Perform randomizations
G_rs = simple_link_preserving_randomization(G, iterations=10000)
G_rm = multiple_link_preserving_randomization(G, iterations=100000)

# Function to calculate knn(k)
def calculate_knn(G):
    knn = nx.average_degree_connectivity(G)
    degrees = np.array(list(knn.keys()))
    knn_values = np.array(list(knn.values()))
    return degrees, knn_values

# Calculate knn(k) for original and randomized networks
degrees_orig, knn_orig = calculate_knn(G)
degrees_rs, knn_rs = calculate_knn(H)
degrees_rm, knn_rm = calculate_knn(G_rm)
# # Calculate knn(k) for original and randomized networks
# degrees_orig, knn_orig = calculate_knn(G)
# degrees_rs, knn_rs = calculate_knn(H)

# Calculate <k^2>/<k>
degree_sequence = [d for n, d in G.degree()]
k_sq_avg = np.mean(np.array(degree_sequence) ** 2)
k_avg = np.mean(degree_sequence)
k2_div_k = k_sq_avg / k_avg

# Plot knn(k) vs k
plt.figure(figsize=(10, 8))
plt.scatter(degrees_orig, knn_orig, color=color, label='Data point', s=50)
plt.scatter(degrees_rs, knn_rs, color=plt.cm.viridis(0.4), label='Null model', s=50,
plt.scatter(degrees_rm, knn_rm, color=plt.cm.viridis(0.8), label='R-M knn(k)', s=50,
plt.axhline(y=k2_div_k, color='k', linestyle='-', label='Random prediction')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Degree (k)')
plt.ylabel(r'$k_{\text{nn}}(k)$')
plt.title('Degree Correlation Function and Randomized Notworks')
plt.legend()
plt.grid(True)
plt.show()
```

```
Original network assortativity: 0.06357722918564943
Null model assortativity: -0.020639187166377988
```



In [ ]:

```
plt.figure(figsize=(10, 8))
plt.scatter(degrees_orig, knn_orig, color=color, label='Data point', s=50)
plt.scatter(degrees_rs, knn_rs, color=plt.cm.viridis(0.4), label='Null model', s=50,
plt.scatter(degrees_rm, knn_rm, color=plt.cm.viridis(0.8), label='R-M knn(k)', s=50,
plt.axhline(y=k2_div_k, color='k', linestyle='-', label='Random prediction')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Degree (k)')
plt.ylabel(r'$k_{\text{nn}}(k)$')
plt.title('Degree Correlation Function and Randomized Notworks')
plt.legend()
plt.grid(True)
plt.show()
```

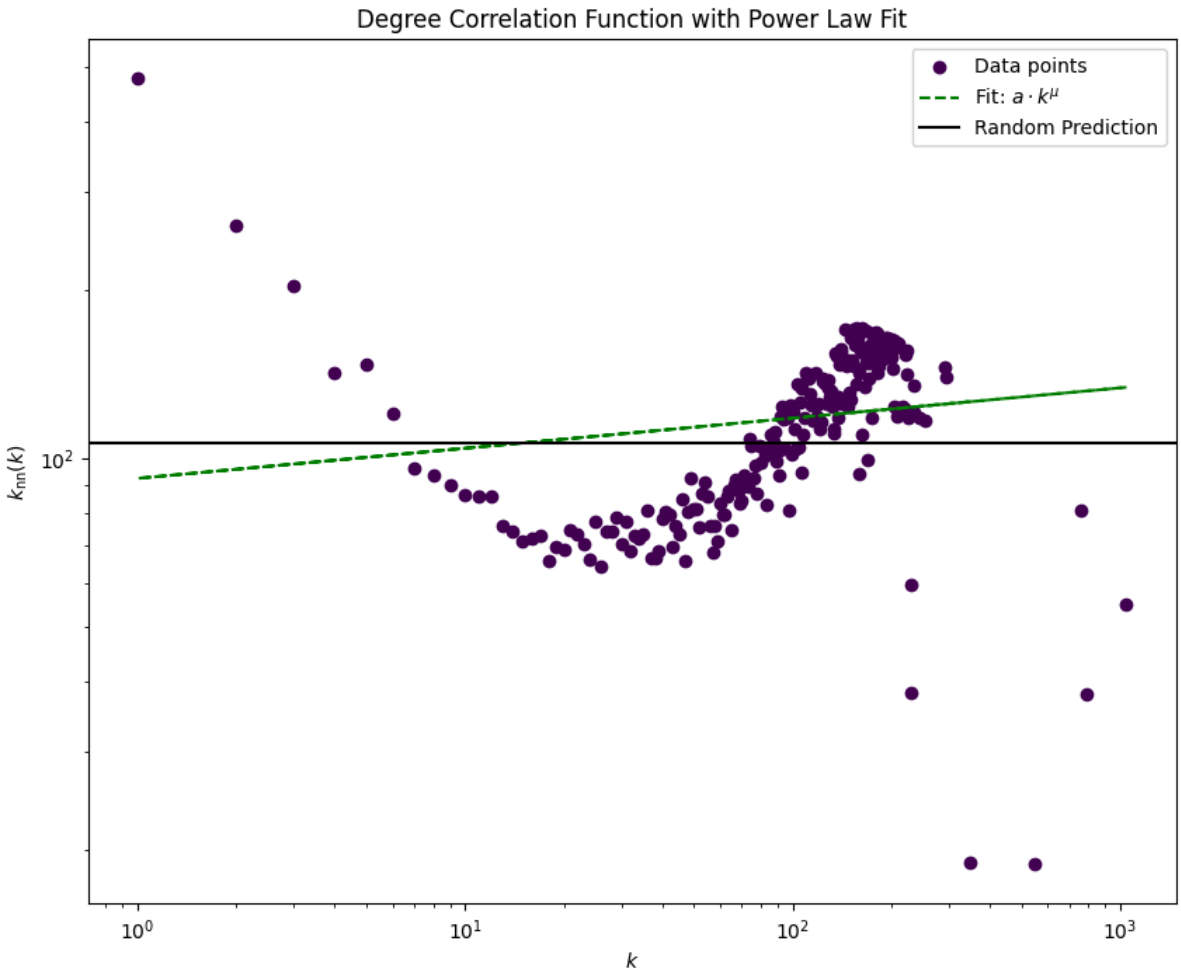


fit

In [ ]:

```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def average_neighbor_degree(G):
    knn = {}
    for node in G.nodes():
        k = G.degree(node)
        if k > 0:  # Avoid division by zero
            knn.setdefault(k, []).append(np.mean([G.degree(neigh) for neigh in G.ne
    knn_avg = {k: np.mean(knn_list) for k, knn_list in knn.items()}
    return knn_avg

def power_law(x, a, mu):
    return a * np.power(x, mu)

import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

# 读取 facebook_combined.txt 文件并创建无向图
def create_undirected_graph(file_path):
    G = nx.Graph()
    with open(file_path, 'r') as file:
        for line in file:
            edge = line.strip().split()
            if len(edge) == 2:
                G.add_edge(int(edge[0]), int(edge[1]))
```

```
    return G

# 指定文件路径
file_path = 'facebook_combined.txt'

# 创建无向图
G = create_undirected_graph(file_path)


# Step 2: Calculate k_nn(k) for the original network
knn_original = average_neighbor_degree(G)

# Step 3: Extract data for fitting
k_vals = np.array(list(knn_original.keys()))
knn_vals = np.array(list(knn_original.values()))

# Step 4: Fit k_nn(k) to a*k^mu
popt, pcov = curve_fit(power_law, k_vals, knn_vals)

# Step 5: Calculate <k> and <k^2>
degrees = np.array([d for n, d in G.degree()])
k_mean = np.mean(degrees)
k_square_mean = np.mean(degrees**2)
k_ratio = k_square_mean / k_mean

# Step 6: Plot the results in log-log scale
plt.figure(figsize=(10, 8))
plt.scatter(k_vals, knn_vals, color=color, label='Data points')
plt.plot(k_vals, power_law(k_vals, *popt), color='green', linestyle='--', label=r'Fi
plt.axhline(y=k_ratio, color='black', linestyle='-', label='Random Prediction')

plt.xscale('log')
plt.yscale('log')
plt.xlabel(r'$k$')
plt.ylabel(r'$k_{\text{nn}}(k)$')
plt.legend()
plt.title('Degree Correlation Function with Power Law Fit')
plt.show()

# Print fitted parameters
print(f"Fitted parameters: a = {popt[0]}, mu = {popt[1]}")
print(f"<k^2>/<k> = {k_ratio}")
```




```
Fitted parameters: a = 92.2630687839182, mu = 0.053877352341481236
<k^2>/<k> = 106.56983702427635
```

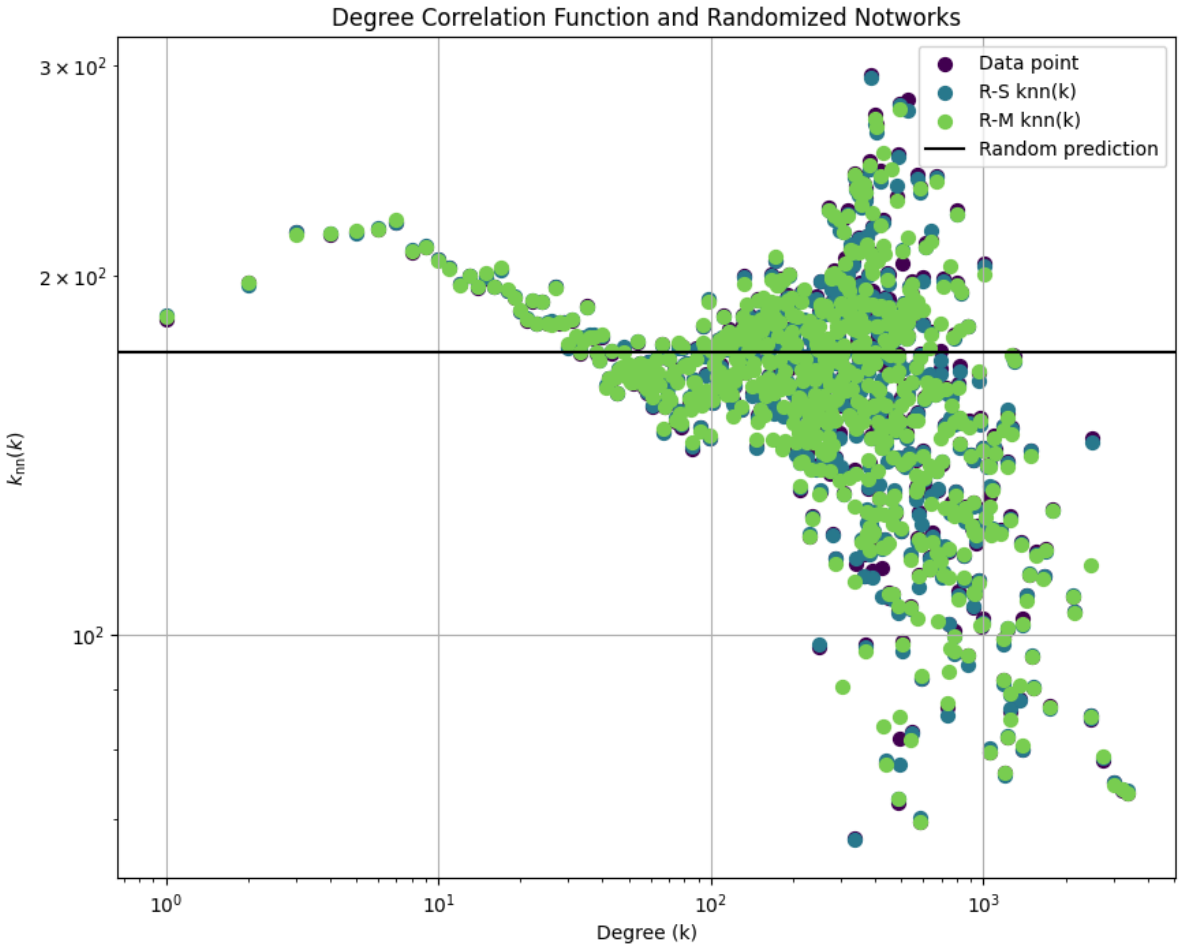
其他网络

In [ ]:

```python
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def simple_link_preserving_randomization(G, iterations=1000):
    H = G.copy()
    edges = list(H.edges())

    for _ in range(iterations):
        # Pick two random edges
        e1, e2 = np.random.choice(len(edges), 2, replace=False)
        u1, v1 = edges[e1]
        u2, v2 = edges[e2]

        # Ensure the nodes are unique and we do not create multiple edges or self-loo
        if len({u1, v1, u2, v2}) == 4:
            if not (H.has_edge(u1, v2) or H.has_edge(u2, v1)):
                H.remove_edge(u1, v1)
                H.remove_edge(u2, v2)
                H.add_edge(u1, v2)
                H.add_edge(u2, v1)
                edges[e1] = (u1, v2)
                edges[e2] = (u2, v1)
    return H

def multiple_link_preserving_randomization(G, iterations=1000):
```

```
    H = G.copy()
    edges = list(H.edges())

    for _ in range(iterations):
        # Pick two random edges
        e1, e2 = np.random.choice(len(edges), 2, replace=False)
        u1, v1 = edges[e1]
        u2, v2 = edges[e2]

        # Ensure the nodes are unique
        if len({u1, v1, u2, v2}) == 4:
            # Remove edges if they exist
            if H.has_edge(u1, v1) and H.has_edge(u2, v2):
                H.remove_edge(u1, v1)
                H.remove_edge(u2, v2)
                H.add_edge(u1, v2)
                H.add_edge(u2, v1)
                edges[e1] = (u1, v2)
                edges[e2] = (u2, v1)
    return H

# # Define parameters
# n = 100  # number of nodes
# p = 0.1  # average degree

# # Create assortative network
# G = create_assortative_network(n, p)



# N = 1000
# k = 5
# G = nx.barabasi_albert_graph(N, k)

# 读取 facebook_combined.txt 文件并创建无向图
def create_undirected_graph(file_path):
    G = nx.Graph()
    with open(file_path, 'r') as file:
        for line in file:
            edge = line.strip().split()
            if len(edge) == 2:
                G.add_edge(int(edge[0]), int(edge[1]))
    return G

# 指定文件路径
file_path = 'twitter_combined.txt'

# 创建无向图
GT = create_undirected_graph(file_path)

# Perform randomizations
GT_rs = simple_link_preserving_randomization(GT, iterations=10000)
GT_rm = multiple_link_preserving_randomization(GT, iterations=10000)

# Function to calculate knn(k)
def calculate_knn(G):
    knn = nx.average_degree_connectivity(G)
    degrees = np.array(list(knn.keys()))
    knn_values = np.array(list(knn.values()))
    return degrees, knn_values

# Calculate knn(k) for original and randomized networks
degrees_orig, knn_orig = calculate_knn(GT)
degrees_rs, knn_rs = calculate_knn(GT_rs)
```

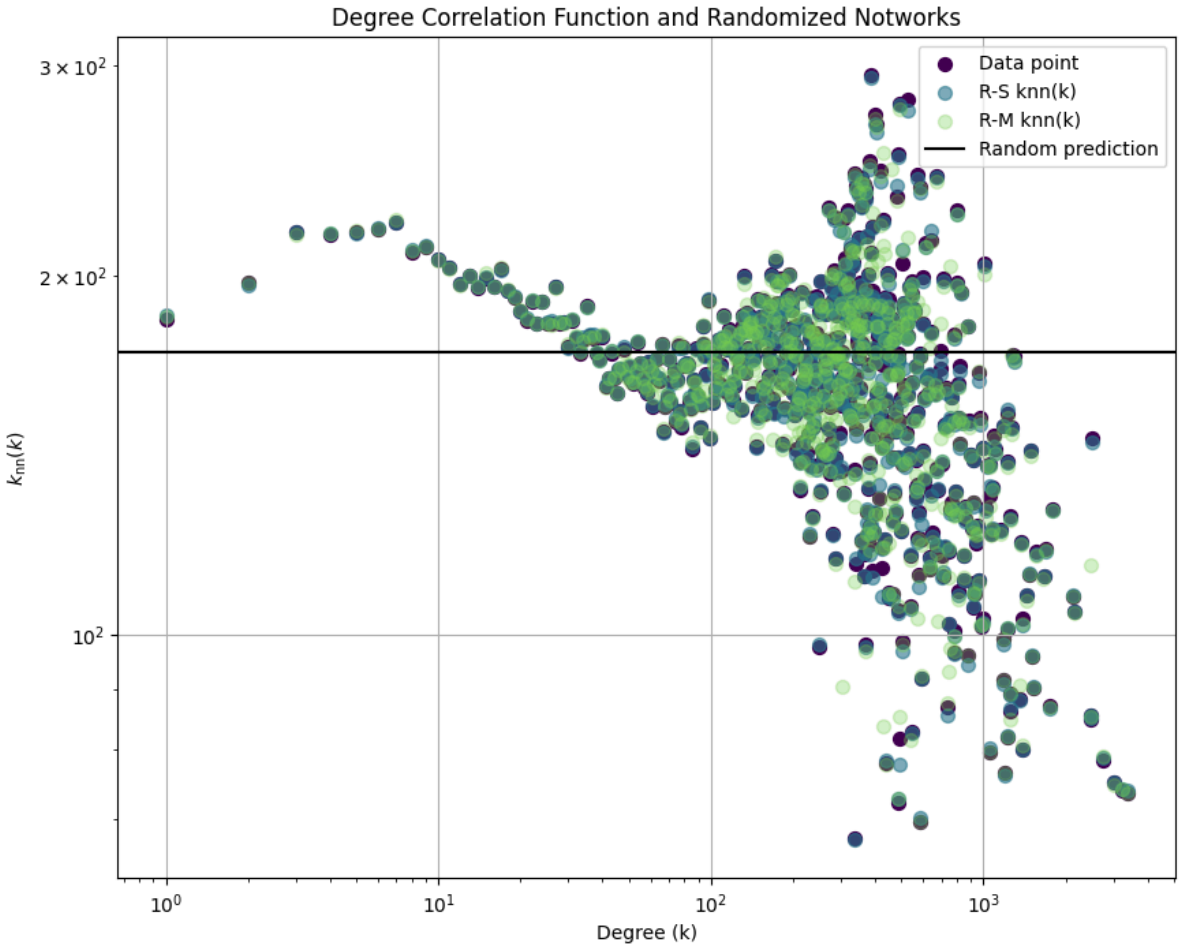
```
degrees_rm, knn_rm = calculate_knn(GT_rm)


# Calculate <k^2>/<k>
degree_sequence = [d for n, d in GT.degree()]
k_sq_avg = np.mean(np.array(degree_sequence) ** 2)
k_avg = np.mean(degree_sequence)
k2_div_k = k_sq_avg / k_avg


plt.figure(figsize=(10, 8))
plt.scatter(degrees_orig, knn_orig, color=color, label='Data point', s=50)
plt.scatter(degrees_rs, knn_rs, color=plt.cm.viridis(0.4), label='R-S knn(k)', s=50,
plt.scatter(degrees_rm, knn_rm, color=plt.cm.viridis(0.8), label='R-M knn(k)', s=50,
plt.axhline(y=k2_div_k, color='k', linestyle='-', label='Random prediction')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Degree (k)')
plt.ylabel(r'$k_{\text{nn}}(k)$')
plt.title('Degree Correlation Function and Randomized Notworks')
plt.legend()
plt.grid(True)
plt.show()
```



In [ ]:

```
plt.figure(figsize=(10, 8))
plt.scatter(degrees_orig, knn_orig, color=color, label='Data point', s=50)
plt.scatter(degrees_rs, knn_rs, color=plt.cm.viridis(0.4), label='R-S knn(k)', s=50,
plt.scatter(degrees_rm, knn_rm, color=plt.cm.viridis(0.8), label='R-M knn(k)', s=50,
plt.axhline(y=k2_div_k, color='k', linestyle='-', label='Random prediction')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Degree (k)')
plt.ylabel(r'$k_{\text{nn}}(k)$')
```

```
plt.title('Degree Correlation Function and Randomized Notworks')
plt.legend()
plt.grid(True)
plt.show()
```



NWS Project demo2

June 14, 2024

1 Facebook

1.1 Community Detection

[ ]:
```
import community.community_louvain as community_louvain
import seaborn as sns
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
from networkx.algorithms.community import label_propagation_communities
import numpy as np
import time

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

[ ]:
```
nx.__version__
```

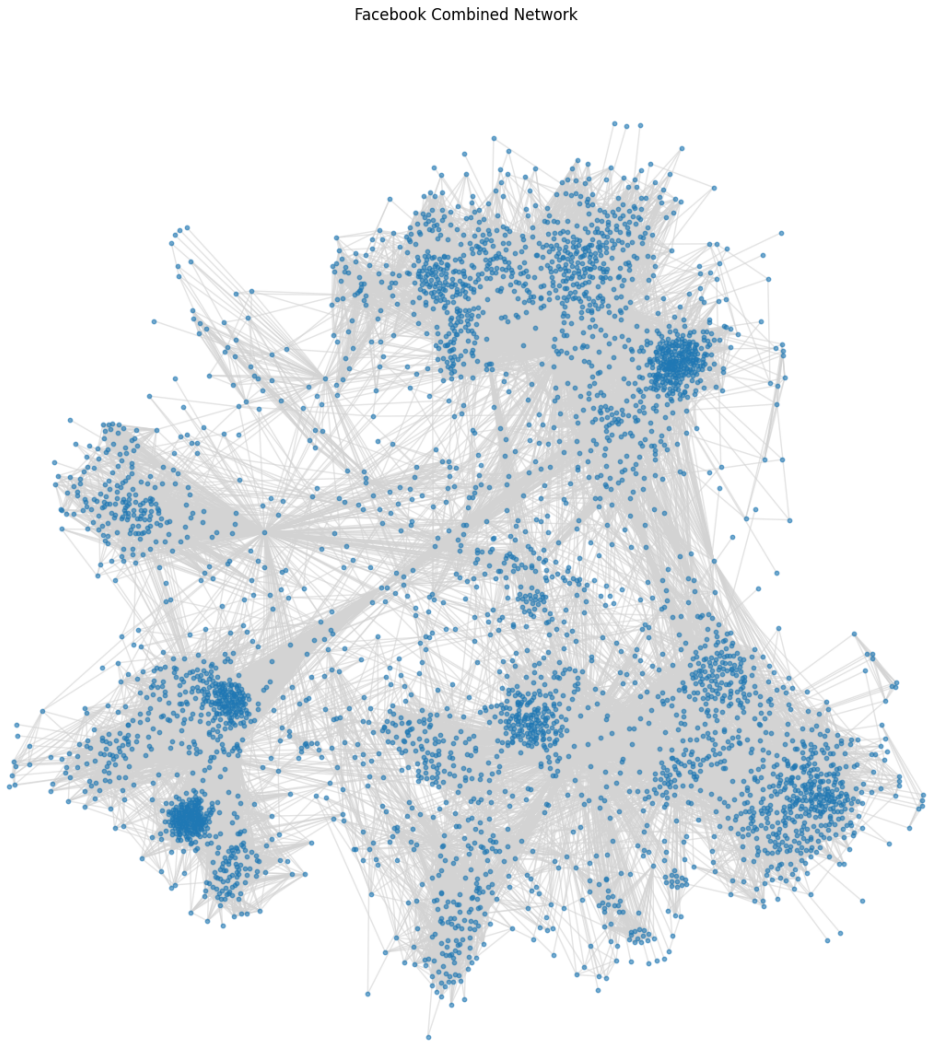
[ ]:
```
'3.1'
```

[ ]:
```
def read_graph():
    file_path = 'facebook_combined.txt'
    edges = []

    with open(file_path, 'r') as file:
        for line in file:
            edge = tuple(map(int, line.strip().split()))
            edges.append(edge)

    #
    G = nx.Graph()
    G.add_edges_from(edges)
    return G
```

[ ]:
```
G = read_graph()
```

[ ]:
```
#
plt.figure(figsize=(12, 12))
pos = nx.spring_layout(G, k=0.1,seed=1)
```

```
nx.draw(G, pos, node_size=10, with_labels=False, alpha=0.6,␣
 ↪edge_color='lightgray')
plt.title('Facebook Combined Network')
plt.show()
```



1.1.1 ;

[ ]:
```
#
def compute_modularity(G, partition):
    communities = {}
    for node, community in partition.items():
        if community not in communities:
```

```
            communities[community] = []
        communities[community].append(node)
    modularity = nx.algorithms.community.quality.modularity(G, communities.
 ↪values())
    return modularity

#
def draw_communities(G, partition, title):
    t1 = time.time()
    pos = nx.spring_layout(G, k=0.1,seed=1)
    cmap = plt.get_cmap('viridis')
    plt.figure(figsize=(10, 10))
    for node, community in partition.items():
        nx.draw_networkx_nodes(G, pos, [node], node_size=10,␣
 ↪node_color=[cmap(community / max(partition.values()))])
    nx.draw_networkx_edges(G, pos, alpha=0.5, edge_color='lightgray')
    plt.title(title)
    plt.show()
    t2 = time.time()
    t = round(t2-t1,2)
    print(f"Draawing running time {t}s, {round(t/60, 2)}min")

partitions = {}
modularities = {}
```

[ ]:
```
def community_size_distri(partition, algo):
    #
    community_sizes = [len([node for node in partition if partition[node] ==␣
 ↪com]) for com in set(partition.values())]

    #
    plt.figure(figsize=(8, 4))
    sns.histplot(community_sizes, kde=True)
    plt.title(f"Community Size Distribution ({algo})")
    plt.xlabel("Community Size")
    plt.ylabel("Frequency")
    plt.show()
```

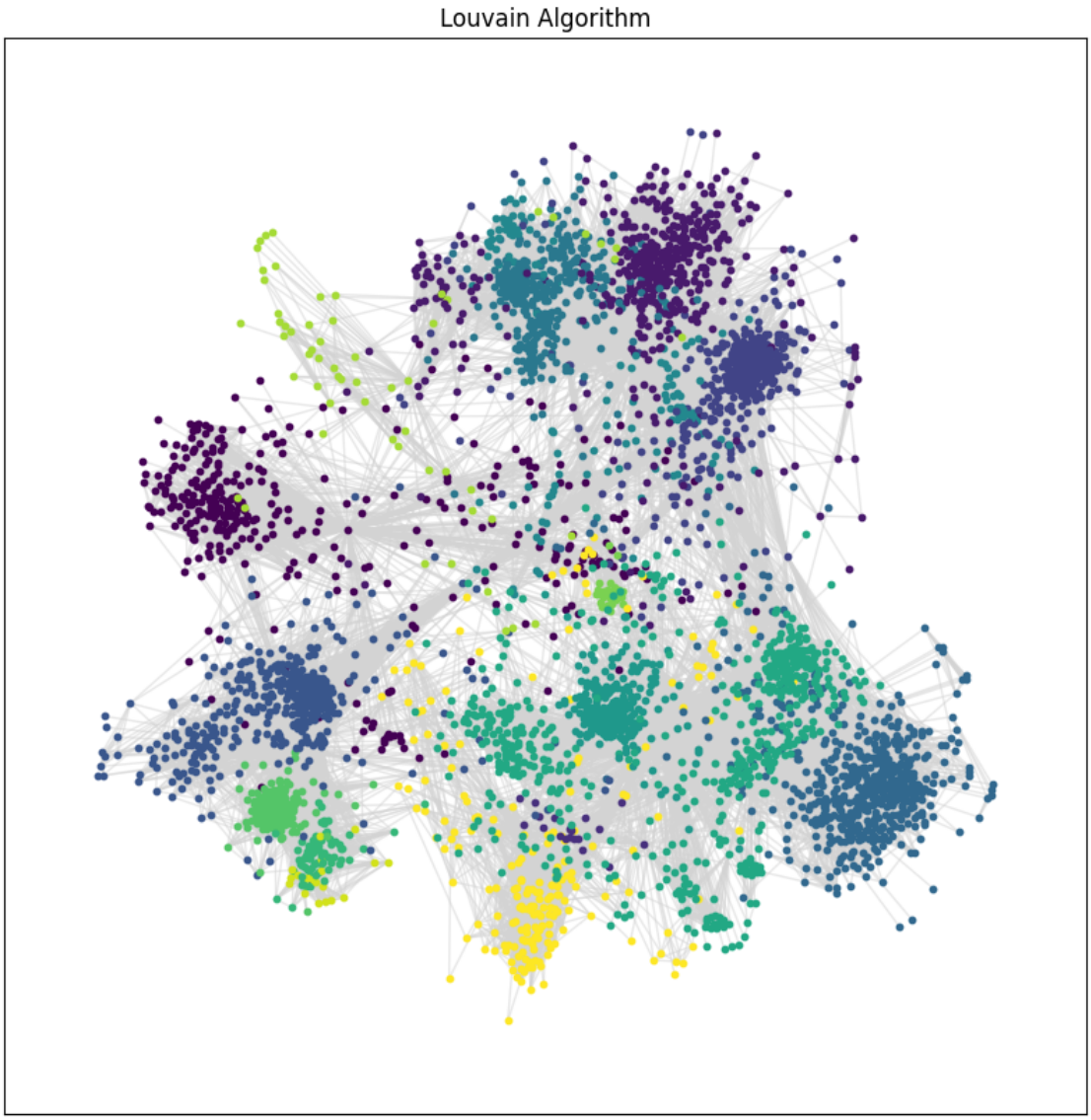
1.1.2 Louvain

Key Point: Modularity optimization
(log)
https://arxiv.org/abs/0803.0476

[ ]:
```
def louvain_community_detection(G):
    partition = community_louvain.best_partition(G)
    return partition
```

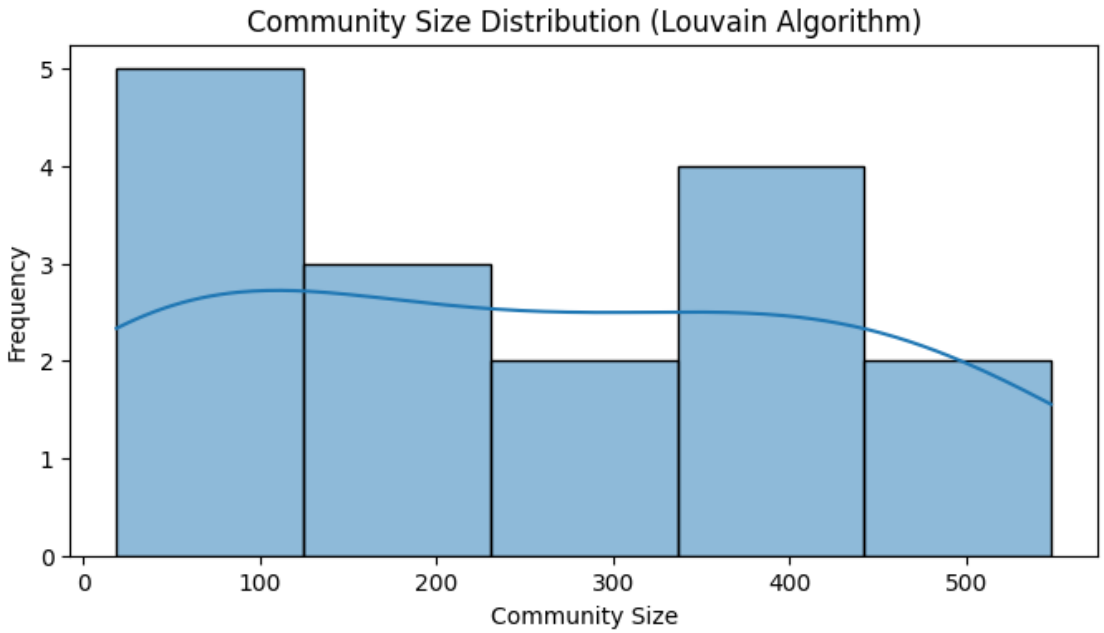
[ ]:
```
#
time_start = time.time()  #
# Louvain
partition_louvain = louvain_community_detection(G)
time_end = time.time()  #
time_sum = time_end - time_start  #           /s
print(f"Done louvain community detection, running time {round(time_sum,2)}s")
partitions['Louvain'] = partition_louvain
modularities['Louvain'] = compute_modularity(G, partition_louvain)
print(f"Done louvain modularity, {modularities['Louvain']}, \nNow start Drawing.
  ↪..")
draw_communities(G, partition_louvain, "Louvain Algorithm")
```

```
Done louvain community detection, running time 1.34s
Done louvain modularity, 0.8349405290865815,
Now start Drawing…
```




```
Draawing running time 92.98s, 1.55min
```

[ ]:
```
#
community_size_distri(partition_louvain, "Louvain Algorithm")
```



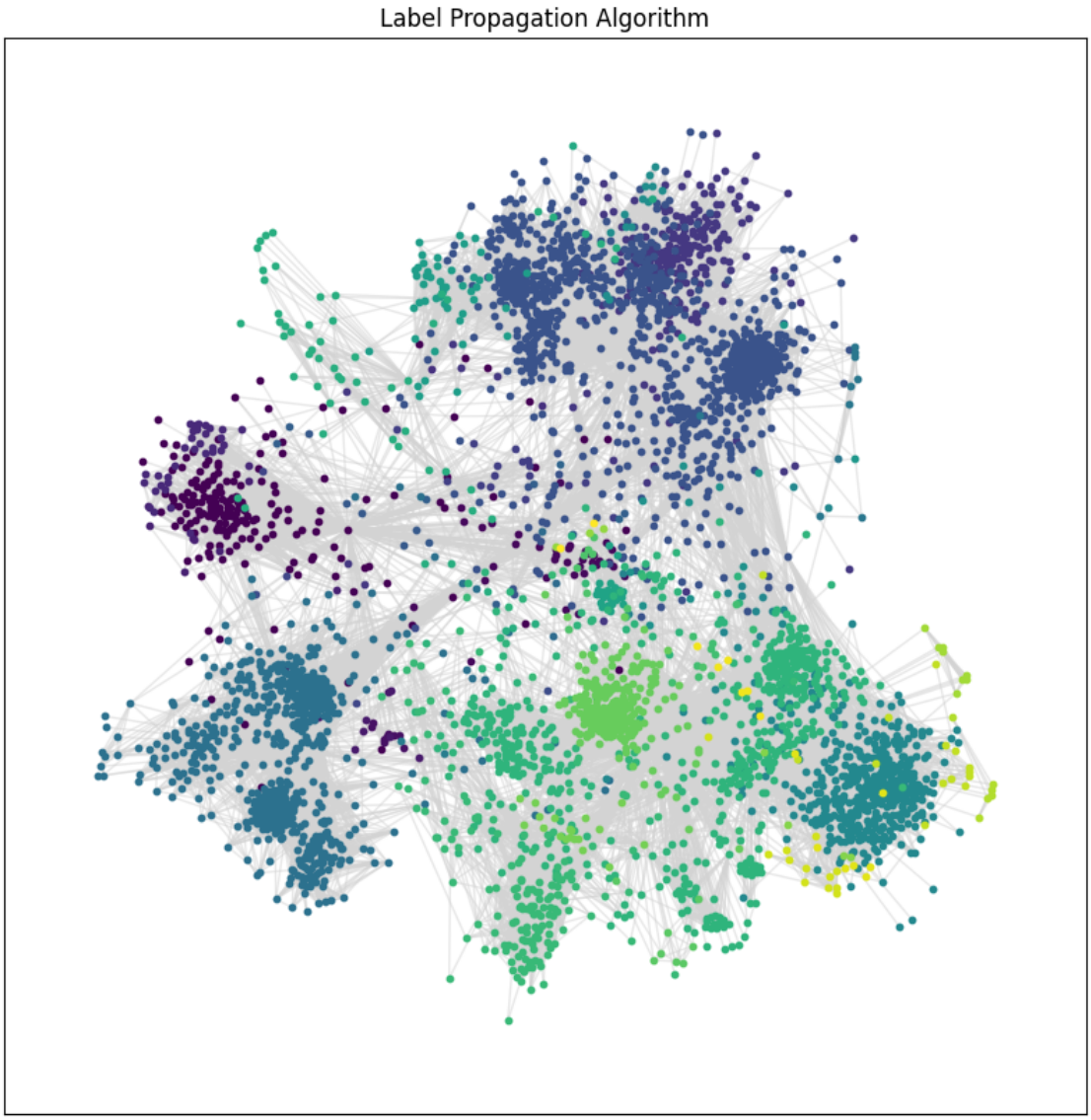
1.1.3 Label Propagation Community Detection Algorithm

https://arxiv.org/abs/1103.4550

[ ]:
```python
def label_propagation_community_detection(G):
    communities = list(label_propagation_communities(G))
    partition = {}
    for i, community in enumerate(communities):
        for node in community:
            partition[node] = i
    return partition
```

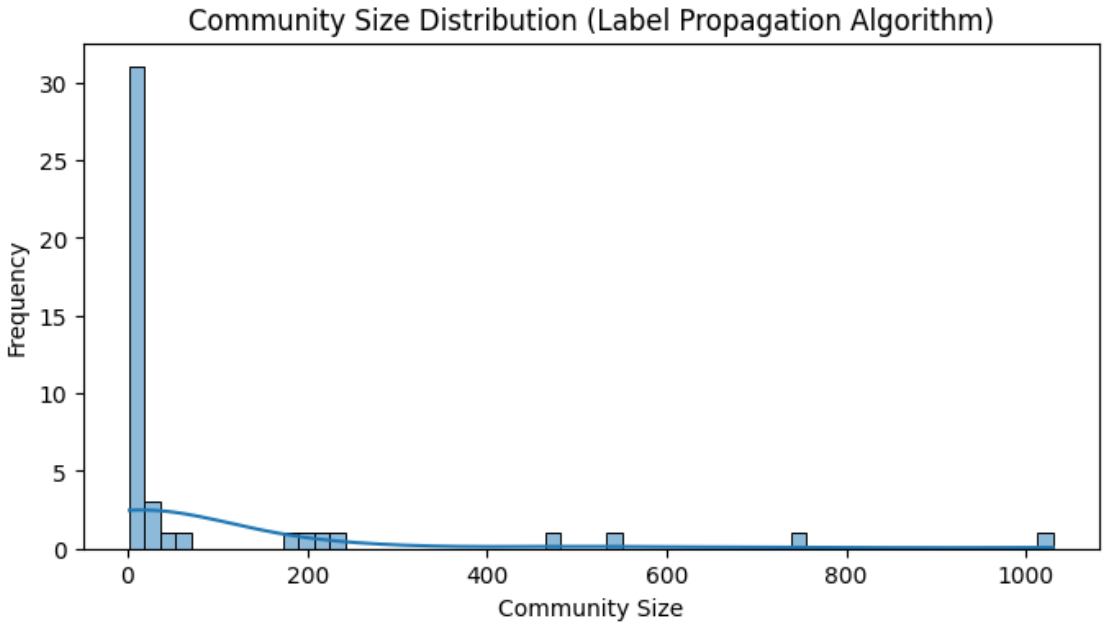
[ ]:
```
# LP
t1 = time.time()
partition_lpa = label_propagation_community_detection(G)
t2 = time.time()
t = round(t2-t1,2)
print(f"Done label propagation algorithm, running time {t}s")
partitions['LPA'] = partition_lpa
modularities['LPA'] = compute_modularity(G, partition_lpa)
print(f"Done compute LPA modularity, {modularities['LPA']}, \nNow start Drawing.
  ↪..")
draw_communities(G, partition_lpa, "Label Propagation Algorithm")
```

```
Done label propagation algorithm, running time 0.42s
Done compute LPA modularity, 0.7368407345348218,
Now start Drawing…
```



```
Draawing running time 92.22s, 1.54min
```

```
[ ]: #
community_size_distri(partition_lpa, "Label Propagation Algorithm")
```



1.1.4 Infomap

Minimum Description Length, MDL
The Minimum Description Length principle (MDL) in information theory

Paper: https://arxiv.org/abs/0707.0609

https://www.mapequation.org/infomap

Code: https://github.com/mapequation/infomap

Infomap

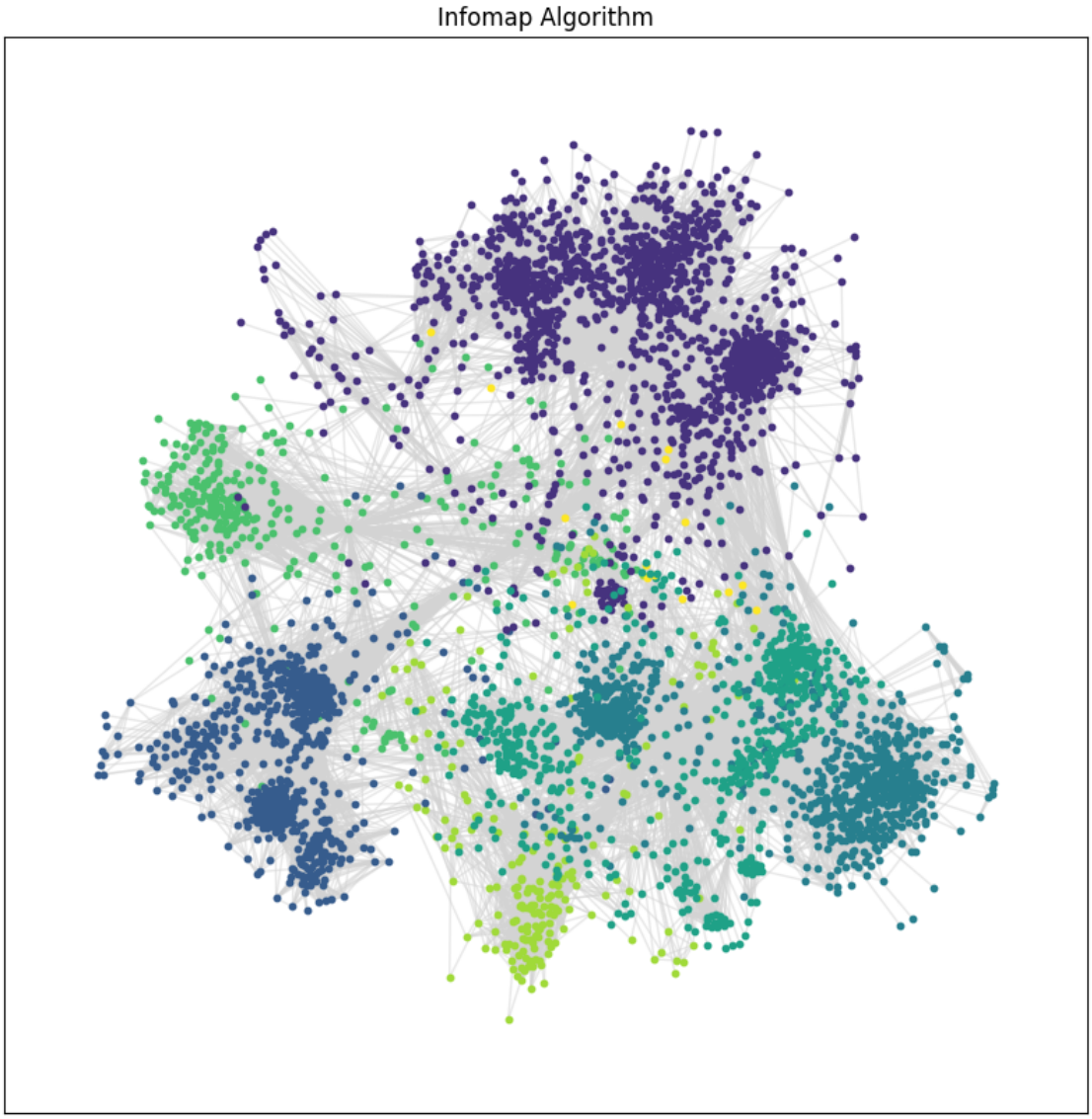
Infomap

[ ]:
```
import infomap
#  Infomap
def infomap_community_detection(G):
    im = infomap.Infomap()
    for e in G.edges():
        im.add_link(e[0], e[1])
    im.run()
    communities = im.get_modules()
    partition = {node: communities[node] for node in communities}
    return partition
```

[ ]:
```
# Infomap
t1 = time.time()
partition_info = infomap_community_detection(G)
t2 = time.time()
t = round(t2-t1,2)
print(f"Done Infomap community detection, running time {t}s")
partitions['Infomap'] = partition_info
modularities['Infomap'] = compute_modularity(G, partition_info)
print(f"Done compute infomap modularity, {modularities['Infomap']},\nNow start␣
 ↪Drawing...")
draw_communities(G, partition_info, "Infomap Algorithm")
```

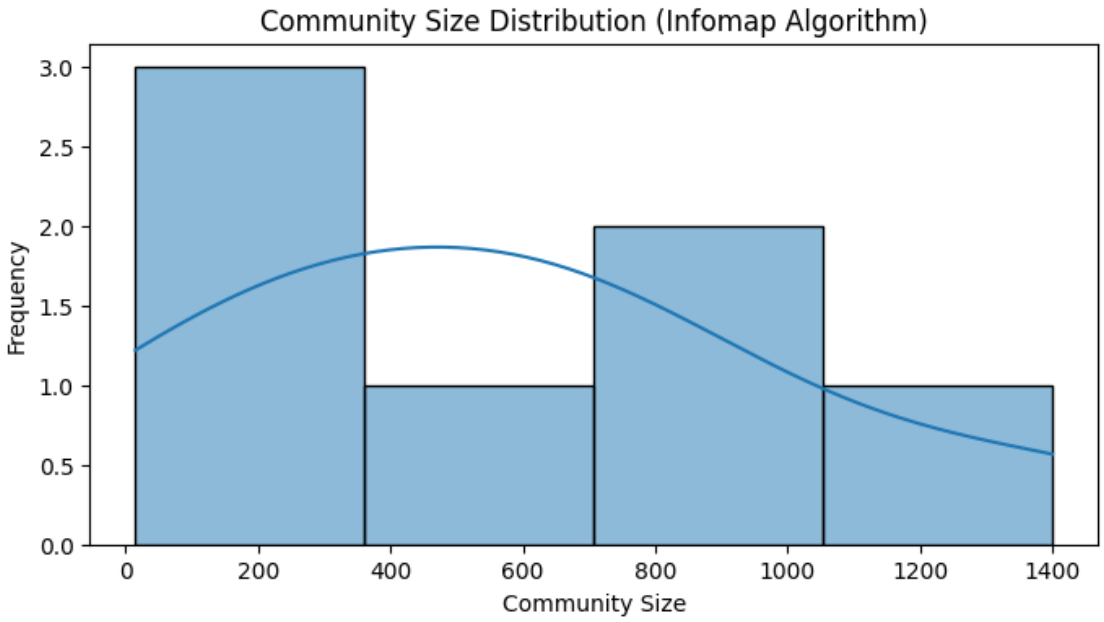
```
=============================================================
  Infomap v2.7.1 starts at 2024-06-04 19:31:52
  -> Input network:
  -> No file output!
=============================================================
  OpenMP 201511 detected with 96 threads…
  -> Ordinary network input, using the Map Equation for first order network
flows
Calculating global network flow using flow model 'undirected'…
  -> Using undirected links.
  => Sum node flow: 1, sum link flow: 1
Build internal network with 4039 nodes and 88234 links…
  -> One-level codelength: 11.2456758

================================================
Trial 1/1 starting at 2024-06-04 19:31:52
================================================
Two-level compression: 28% 0.42% 0.15326333% 0.00273817443%
Partitioned to codelength 0.371498905 + 7.70601404 = 8.077512945 in 76 modules.
Super-level compression: 1.38841111% to codelength 7.965363858 in 7 top modules.

Recursive sub-structure compression: 12.2611435% 2.98432497% 0.00446560267% 0%
```

```
Done Infomap community detection, running time 0.55s
Done compute infomap modularity, 0.7061782532697447,
Now start Drawing…
```




```
Draawing running time 93.7s, 1.56min
```

[ ]:
```
#
community_size_distri(partition_info, "Infomap Algorithm")
```



1.1.5 GEMSEC

Graph Embedding for Structural Clustering

Paper: https://arxiv.org/abs/1802.03997
Code: https://github.com/benedekrozemberczki/GEMSEC

GEMSEC

DeepWalk Node2Vec
K-means

GEMSEC

GEMSEC

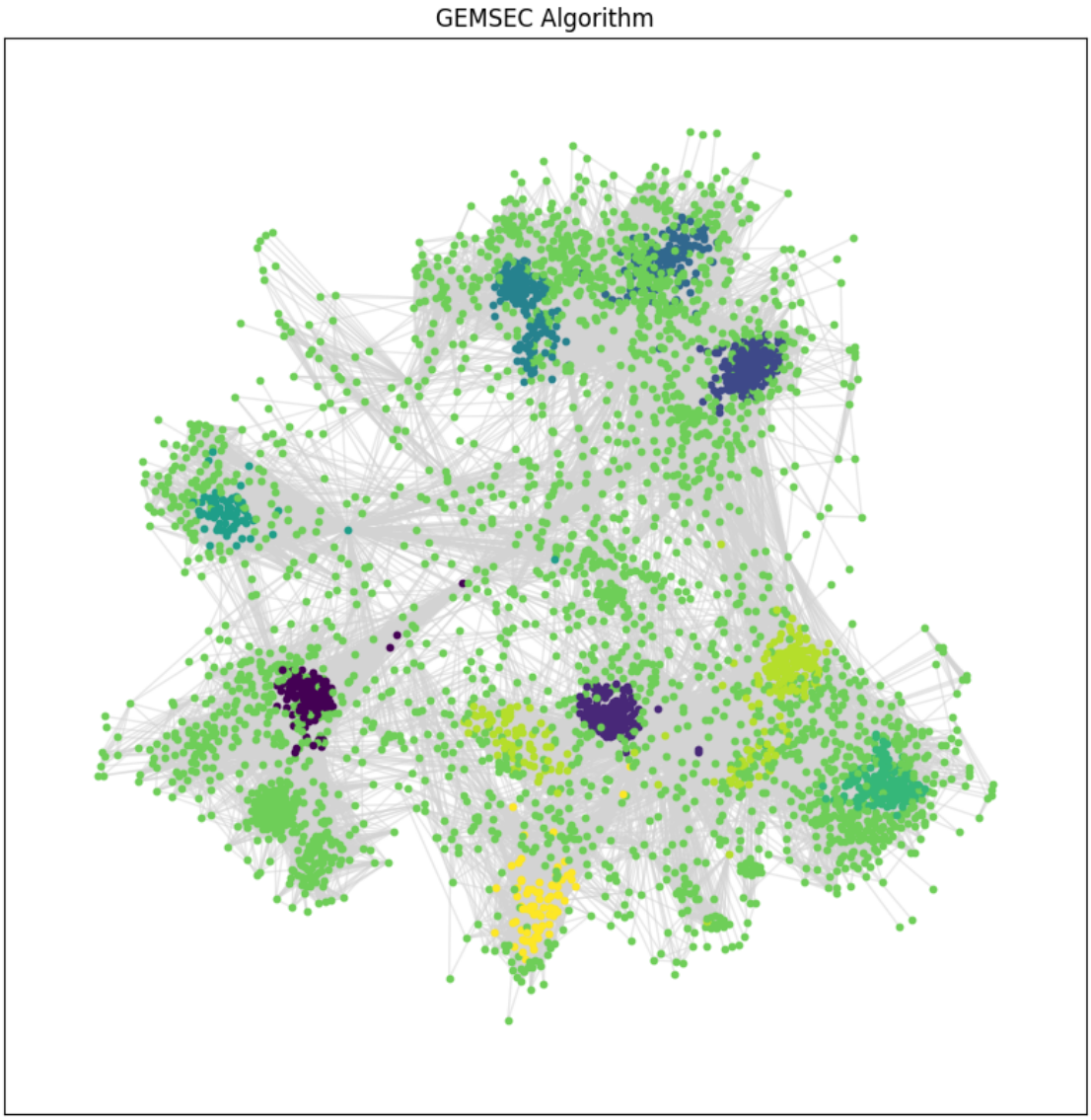
Non-overlapping

[ ]:
```
import karateclub
```

[ ]:
```python
from karateclub import GEMSEC
# 1.  GEMSEC
def gemsec_community_detection(G):
    #
    G.remove_edges_from(nx.selfloop_edges(G))

    model = GEMSEC()
    model.fit(G)
    communities = model.get_memberships()
    return communities
```

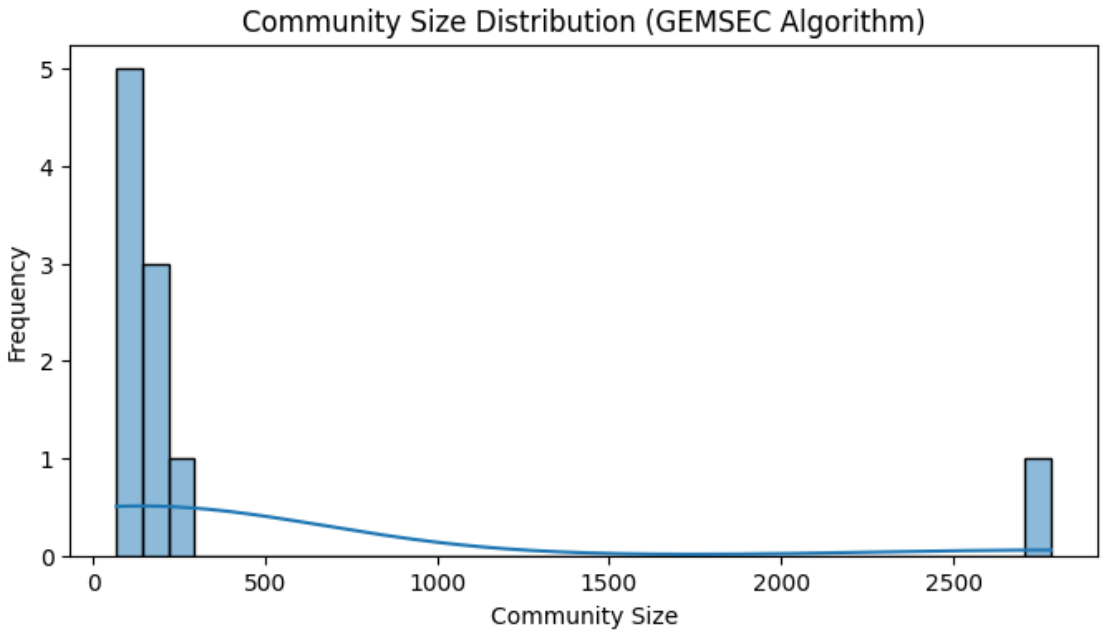
[ ]:
```python
#  GEMSEC
print("Now start GEMSEC community detection...")
t1 = time.time()
partition_gemsec = gemsec_community_detection(G)
t2 = time.time()
t = round(t2-t1, 2)
print(f"Done GEMSEC community detection, running time {t}s, {round(t/60,␣
 ↪2)}min")
#
modularity_gemsec = compute_modularity(G, partition_gemsec)
print(f"Modularity (GEMSEC): {modularity_gemsec:.4f}")
#

G = read_graph()
print("Now start drawing...")
draw_communities(G, partition_gemsec, "GEMSEC Algorithm")
```

```
Now start GEMSEC community detection…
Done GEMSEC community detection, running time 1323.04s, 22.05min
Modularity (GEMSEC): 0.5398
Now start drawing…
```




```
Draawing running time 92.95s, 1.55min
```

[ ]:
```
#
community_size_distri(partition_gemsec, "GEMSEC Algorithm")
```



1.1.6 MNMF

Multiview Nonnegative Matrix Factorization

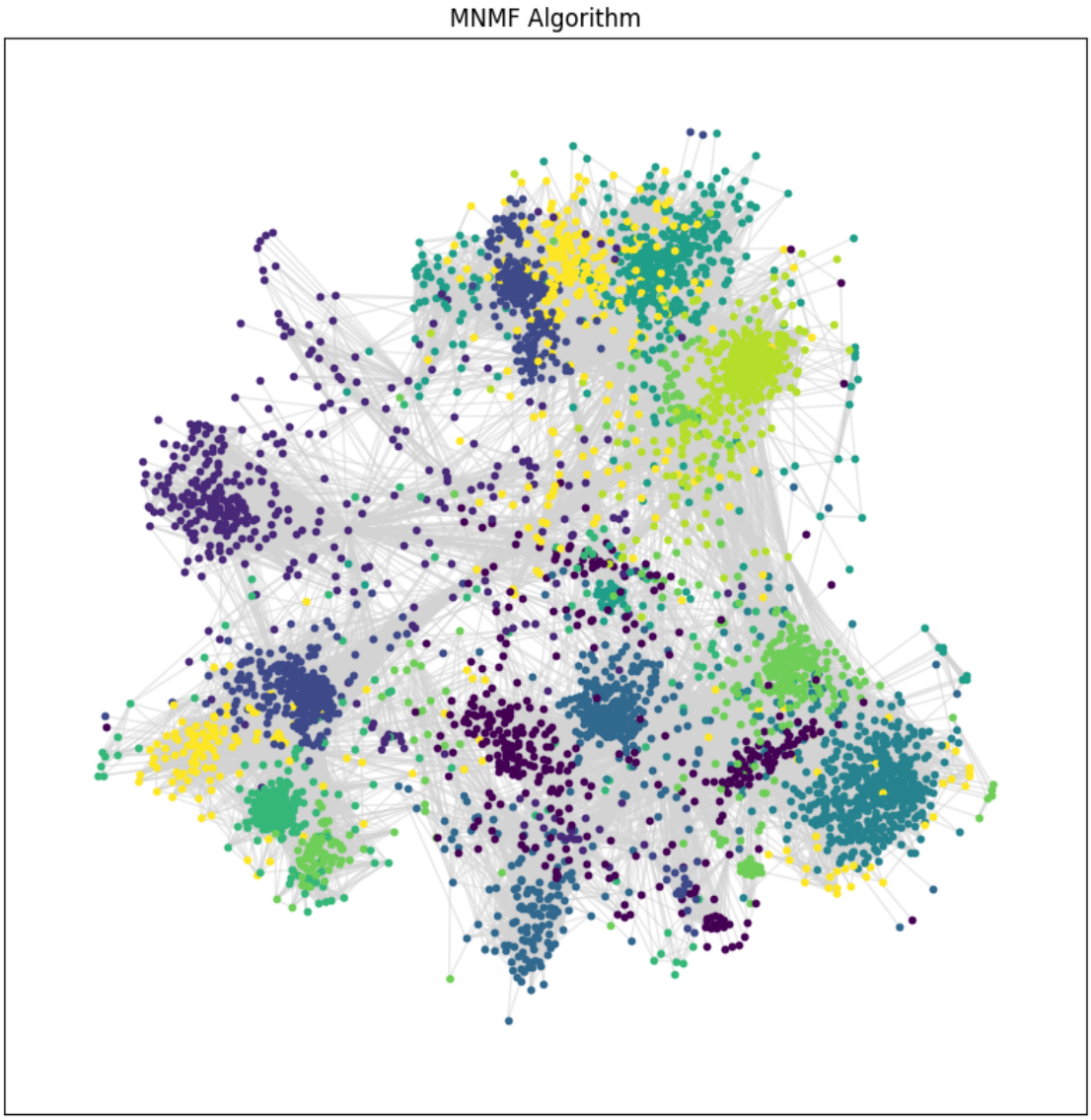
Paper: https://ojs.aaai.org/index.php/AAAI/article/view/10488
Code: https://github.com/benedekrozemberczki/M-NMF

[ ]:
```
from karateclub import MNMF

# 1.  MNMF
def mnmf_community_detection(G):
    model = MNMF()
    model.fit(G)
    communities = model.get_memberships()
    return communities
```

[ ]:
```
print("Now start MNMF community detection...")
t1 = time.time()
#  MNMF
partition_mnmf = mnmf_community_detection(G)
t2 = time.time()
t = round(t2-t1, 2)
print(f"Done MNMF community detection, running time {t}s, {round(t/60, 2)}min")
#
modularity_mnmf = compute_modularity(G, partition_mnmf)
```

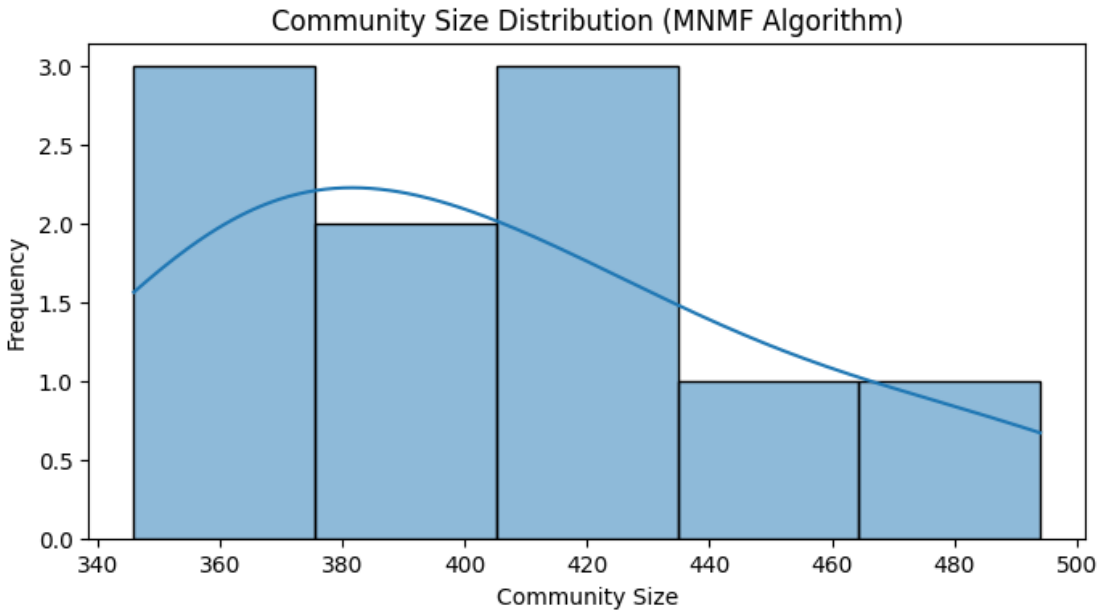
```
print(f"Modularity (MNMF): {modularity_mnmf:.4f}")

G = read_graph()
def draw_communities(G, partition, title):
    t1 = time.time()
    pos = nx.spring_layout(G, k=0.1,seed=1)
    cmap = plt.get_cmap('viridis')
    plt.figure(figsize=(10, 10))
    for node, community in partition.items():
        nx.draw_networkx_nodes(G, pos, [node], node_size=10,␣
 ↪node_color=[cmap(community / max(partition.values()))])
    nx.draw_networkx_edges(G, pos, alpha=0.5, edge_color='lightgray')
    plt.title(title)
    plt.show()
    t2 = time.time()
    t = round(t2-t1,2)
    print(f"Draawing running time {t}s, {round(t/60, 2)}min")


print("Now start drawing...")
#
draw_communities(G, partition_mnmf, "MNMF Algorithm")
```

```
Now start MNMF community detection…
Done MNMF community detection, running time 18.59s, 0.31min
Modularity (MNMF): 0.8109
Now start drawing…
```




```
Draawing running time 96.12s, 1.6min
```

[ ]:
```
#
community_size_distri(partition_mnmf, "MNMF Algorithm")
```



1.1.7 EgoNetSplitter

Paper: https://dl.acm.org/doi/10.1145/3097983.3098054
Code: https://github.com/benedekrozemberczki/EgoSplitting

EgoNetSplitter Ego Networks

EgoNetSplitter

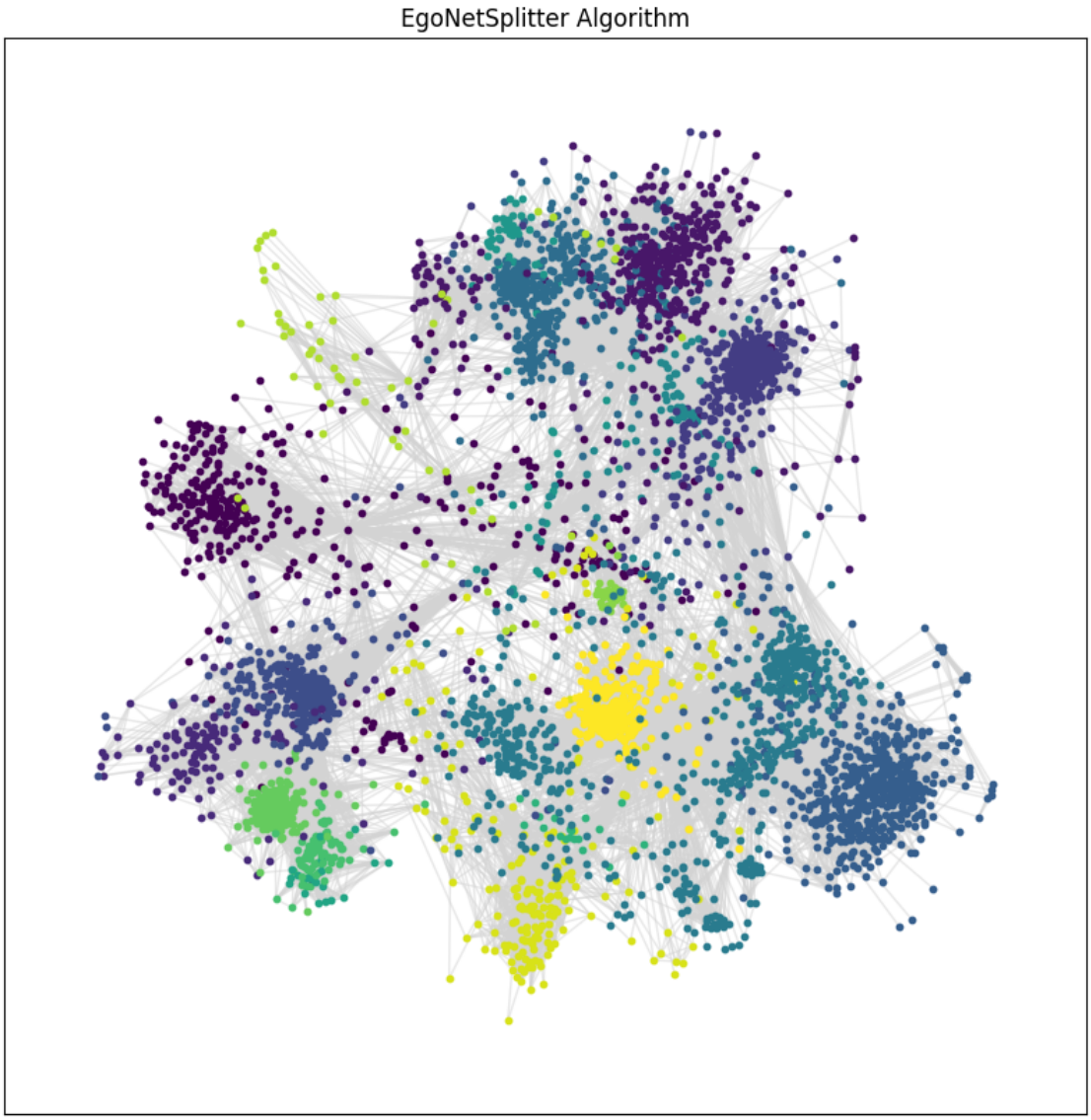
[ ]:

[ ]:

[ ]:
```
from karateclub import EgoNetSplitter
# 1.  EgoNetSplitter
def egonet_splitter_community_detection(G):
    model = EgoNetSplitter()
    model.fit(G)
    communities = model.get_memberships()
    return communities
```

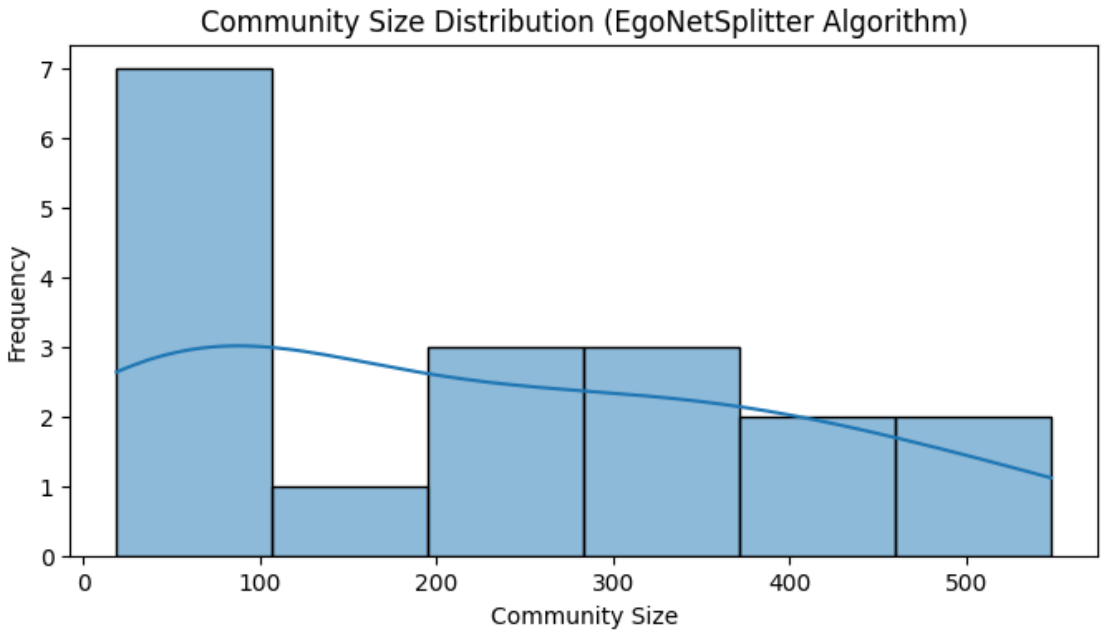
[ ]:
```
print("Now start EgoNetSplitter community detection...")
t1 = time.time()
#  EgoNetSplitter
partition_egonet_splitter = egonet_splitter_community_detection(G)
t2 = time.time()
t = round(t2-t1, 2)
print(f"Done EgoNetSplitter community detection, running time {t}s, {round(t/
 ↪60, 2)}min")

# membership
communities_egonet_splitter = {key: value[0] for key, value in␣
 ↪partition_egonet_splitter.items()}

#
modularity_egonet_splitter = compute_modularity(G, communities_egonet_splitter)
print(f"Modularity (EgoNetSplitter): {modularity_egonet_splitter:.4f}")

G = read_graph()
print("Now start drawing...")
#
draw_communities(G, communities_egonet_splitter, "EgoNetSplitter Algorithm")
```

```
Now start EgoNetSplitter community detection…
Done EgoNetSplitter community detection, running time 9.86s, 0.16min
Modularity (EgoNetSplitter): 0.8396
Now start drawing…
```




```
Draawing running time 93.28s, 1.55min
```

[ ]:
```
#
community_size_distri(communities_egonet_splitter, "EgoNetSplitter Algorithm")
```


Community Size Distribution (EgoNetSplitter Algorithm)
Frequency
Community Size
0
1
2
3
4
5
6
7
0
100
200
300
400
500

[ ]:

[ ]:

[ ]:

[ ]:

In [1]:
```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
from networkx.algorithms import bipartite
```

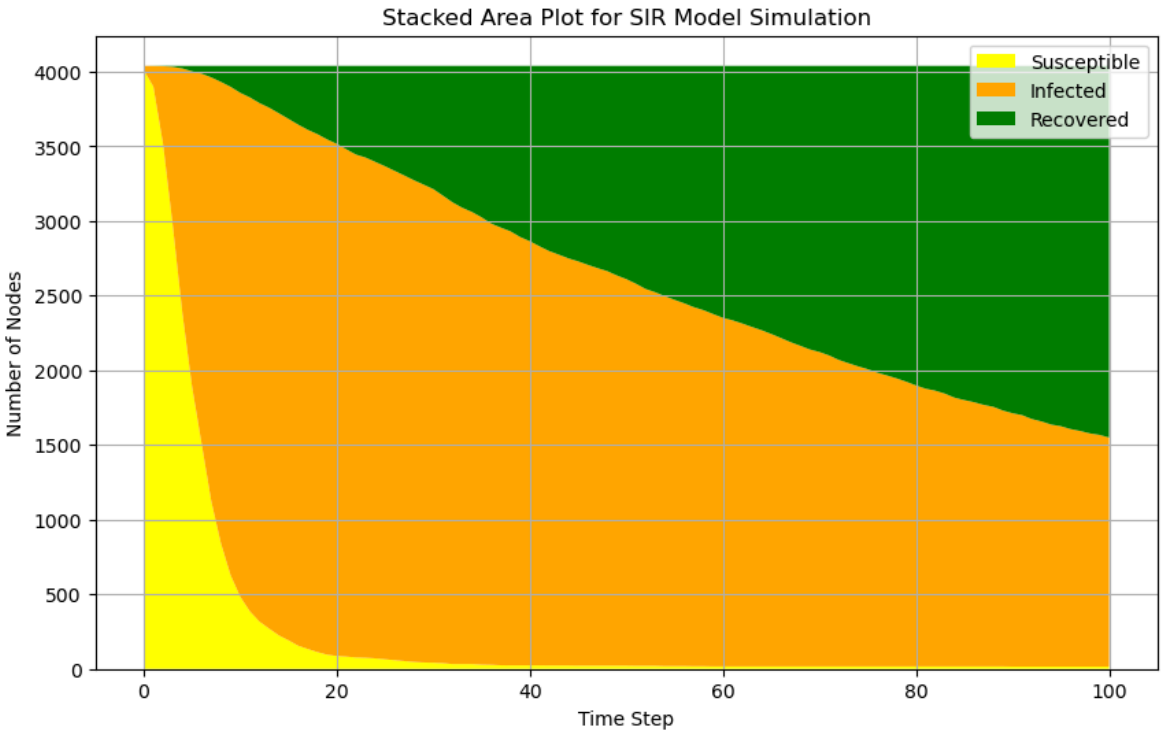
In [2]:
```
G = nx.read_edgelist('facebook_combined.txt', create_using=nx.Graph(), no
```

SIR Modeling for Spreading Phenomenon

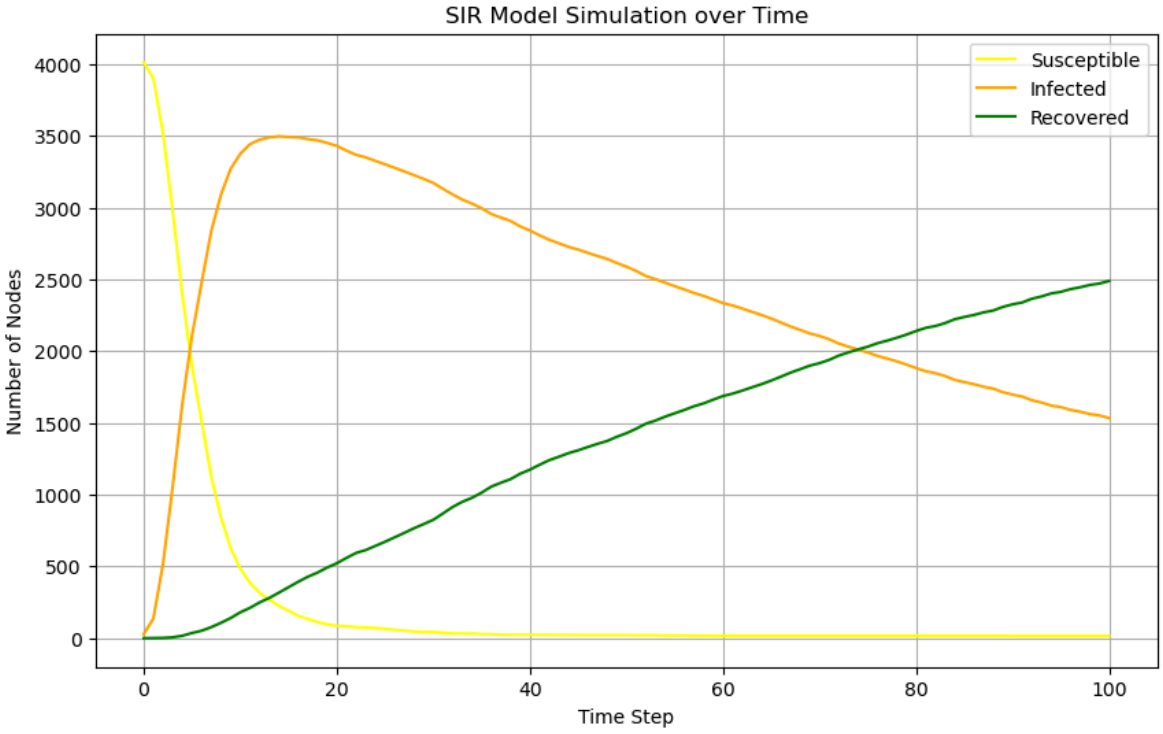
In [3]:
```
initial_infected = 10
infection_probability = 0.05
recovery_rate = 0.01
steps = 100

susceptible = np.ones(G.number_of_nodes(), dtype=bool)
infected = np.zeros(G.number_of_nodes(), dtype=bool)
infected[np.random.choice(G.nodes(), initial_infected, replace=False)] =
susceptible[infected] = False
recovered = np.zeros(G.number_of_nodes(), dtype=bool)

S, I, R = [], [], []
```

In [4]:
```
# Simulation loop
for _ in range(steps + 1):
    new_infected = np.zeros_like(infected)
    new_recovered = np.zeros_like(infected)

    for node in np.where(infected)[0]:
        new_recovered[node] = np.random.rand() < recovery_rate
        for neighbor in G.neighbors(node):
            if susceptible[neighbor] and np.random.rand() < infection_pro
                new_infected[neighbor] = True

    infected[new_infected] = True
    susceptible[new_infected] = False
    infected[new_recovered] = False
    recovered[new_recovered] = True

    S.append(np.sum(susceptible))
    I.append(np.sum(infected))
    R.append(np.sum(recovered))
```

In [5]:
```
plt.figure(figsize=(10, 6))
plt.stackplot(range(steps + 1), S, I, R, labels=['Susceptible', 'Infected
plt.title('Stacked Area Plot for SIR Model Simulation')
plt.xlabel('Time Step')
plt.ylabel('Number of Nodes')
plt.legend(loc='upper right')
plt.grid(True)
plt.show()
```



In [6]:
```
plt.figure(figsize=(10, 6))
plt.plot(S, label='Susceptible', color='yellow')
plt.plot(I, label='Infected', color='orange')
plt.plot(R, label='Recovered', color='green')
plt.title('SIR Model Simulation over Time')
plt.xlabel('Time Step')
plt.ylabel('Number of Nodes')
plt.legend(loc='upper right')
plt.grid(True)
plt.show()
```



The Degree Preserve Null Models

In [7]:
```
# Original network clustering coefficient
original_clustering = nx.average_clustering(G)

# Generate a null model by randomizing the network while preserving the d
null_model = nx.configuration_model([d for n, d in G.degree()])
null_model = nx.Graph(null_model)  # Remove parallel edges and self-loops
null_model.remove_edges_from(nx.selfloop_edges(null_model))

is_connected = nx.is_connected(null_model)
is_connected
```

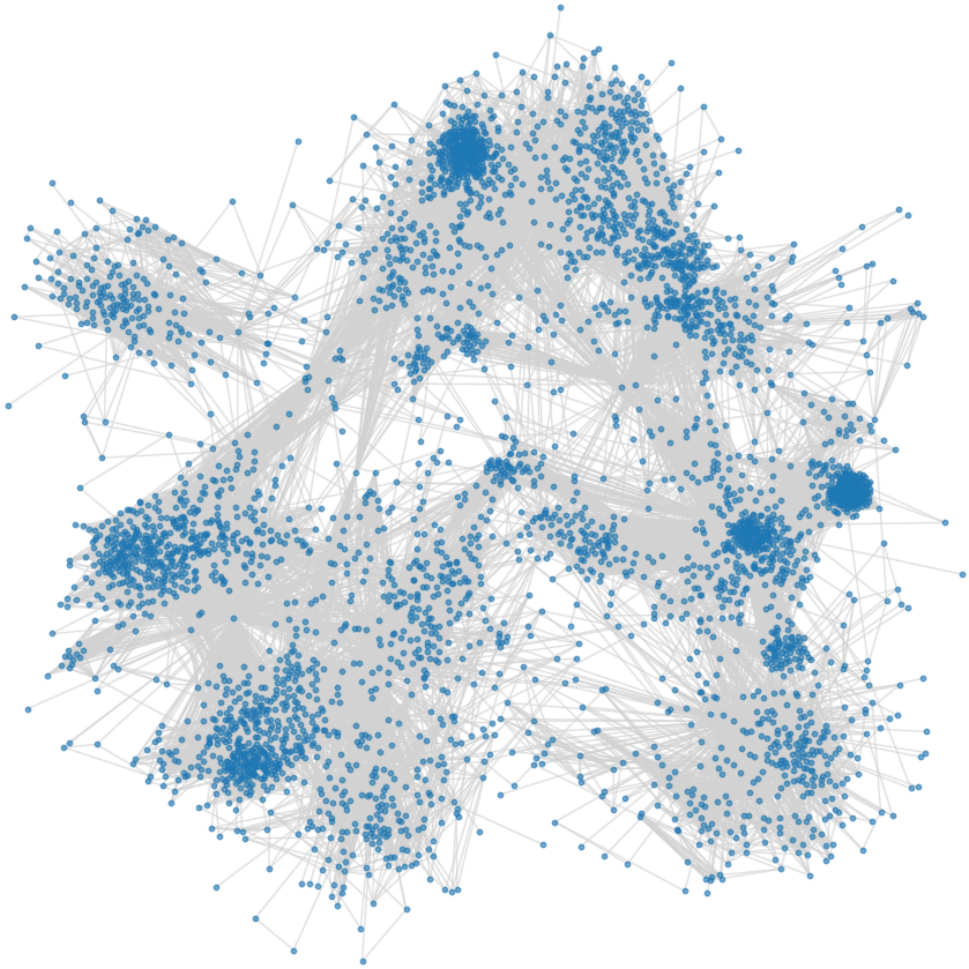
Out[7]: True

In [8]:
```
null_clustering = nx.average_clustering(null_model)

print("original_clustering:",original_clustering)
print("null_clustering:",null_clustering)
def calculate_average_path_length_largest_component(graph):
    if nx.is_connected(graph):
        return nx.average_shortest_path_length(graph)
    else:
        largest_cc = max(nx.connected_components(graph), key=len)
        subgraph = graph.subgraph(largest_cc)
        return nx.average_shortest_path_length(subgraph)

avg_path_length_original = calculate_average_path_length_largest_componen

avg_path_length_null = calculate_average_path_length_largest_component(nu


print("Average Path Length of Original Network:", avg_path_length_origina
print("Average Path Length of Null Model:", avg_path_length_null)

degree_ass_ori = nx.degree_assortativity_coefficient(G)
print("Degree Assortativity of the original social network:", degree_ass_
degree_ass_null = nx.degree_assortativity_coefficient(null_model)
print("Degree Assortativity of the Null Model:", degree_ass_null)
```

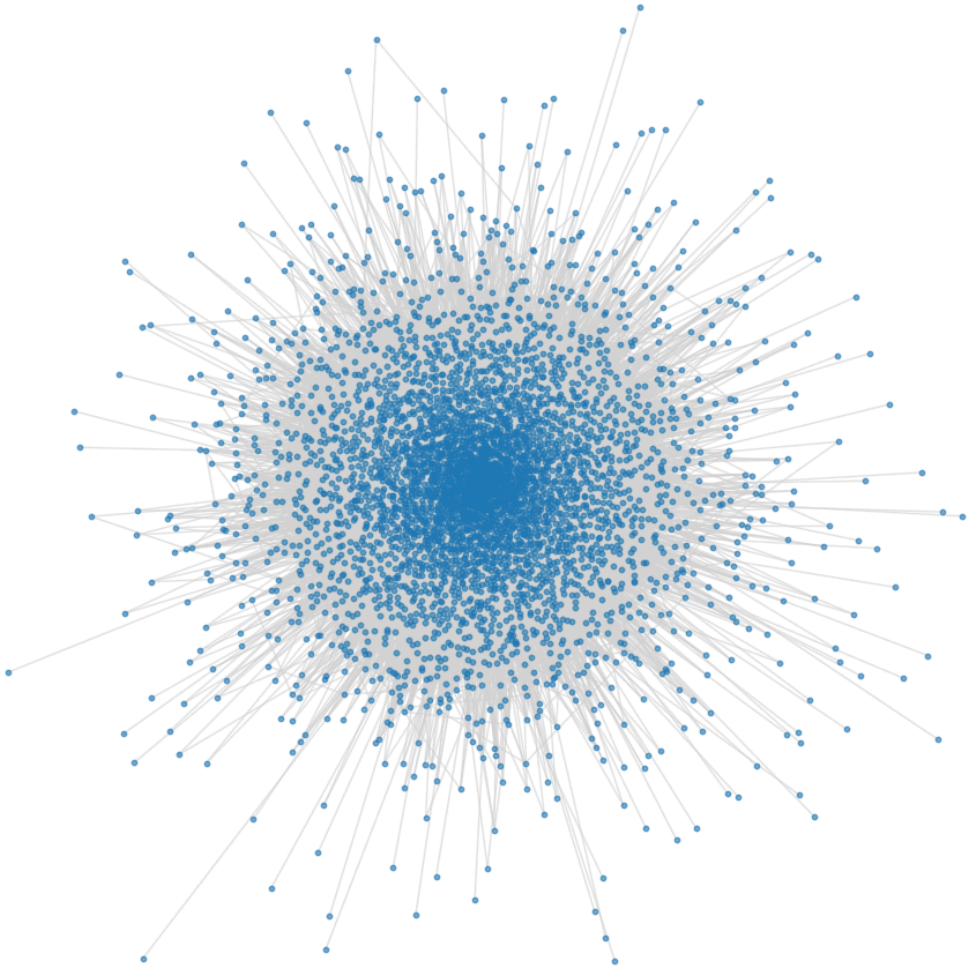
```
original_clustering: 0.6055467186200876
null_clustering: 0.05431102920088585
Average Path Length of Original Network: 3.6925068496963913
Average Path Length of Null Model: 2.628268390130355
Degree Assortativity of the original social network: 0.06357722918564943
Degree Assortativity of the Null Model: -0.024690919649239047
```

In [9]:
```
plt.figure(figsize=(10, 10))
pos = nx.spring_layout(G, k=0.1)
nx.draw(G, pos, node_size=10, with_labels=False, alpha=0.6, edge_color='l
plt.title('Original Network')
plt.show()
plt.figure(figsize=(10, 10))
pos = nx.spring_layout(null_model, k=0.1)
nx.draw(null_model, pos, node_size=10, with_labels=False, alpha=0.6, edge
plt.title('The Null Model')
plt.show()
```

Original Network



The Null Model

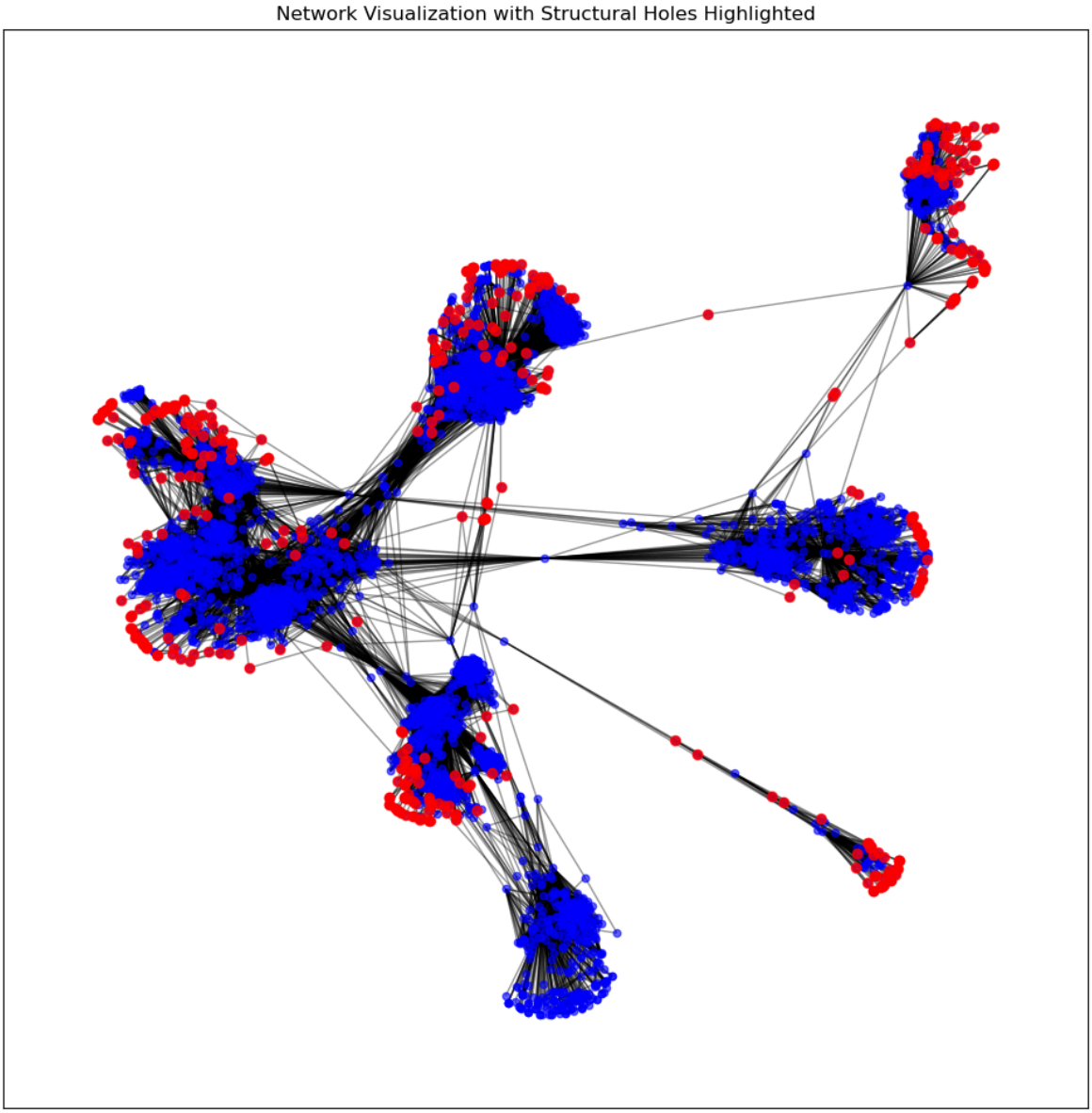


In [10]:

```
def calculate_network_constraint(G):
    constraints = {}
    for i in G.nodes():
        constraint = 0
        for j in G.nodes():
            if j != i:
                if G.has_edge(i, j):
                    pij = G[i][j]['weight'] if 'weight' in G[i][j] else 1
                else:
                    pij = 0
                sum_piq_pqj = sum(G[i][q]['weight'] * G[q][j]['weight'] i
                                  for q in G.neighbors(i) if q != j)
                sj = sum(G[j][q]['weight'] for q in G.neighbors(j))
                if sj != 0:
                    constraint += (pij + sum_piq_pqj / sj) ** 2
        constraints[i] = constraint
    return constraints

for u, v in G.edges():
    G[u][v]['weight'] = 1
node_constraints = calculate_network_constraint(G)
min_constraint_node = min(node_constraints, key=node_constraints.get)
print("Node with minimal constraint (potential structural hole):", min_co
```

```
Node with minimal constraint (potential structural hole): 692
```

In [11]:
```
low_constraint_nodes = sorted(node_constraints, key=node_constraints.get)

pos = nx.spring_layout(G)
plt.figure(figsize=(12, 12))
nx.draw_networkx_nodes(G, pos, node_size=20, node_color='blue', alpha=0.6
nx.draw_networkx_nodes(G, pos, nodelist=low_constraint_nodes, node_size=3
nx.draw_networkx_edges(G, pos, alpha=0.4)
plt.title('Network Visualization with Structural Holes Highlighted')
plt.show()
```



Bridges

In [12]:
```
nx.has_bridges(G)
```

Out[12]:
```
True
```

In [13]:
```
bridges = list(nx.bridges(G))
len(bridges)
```

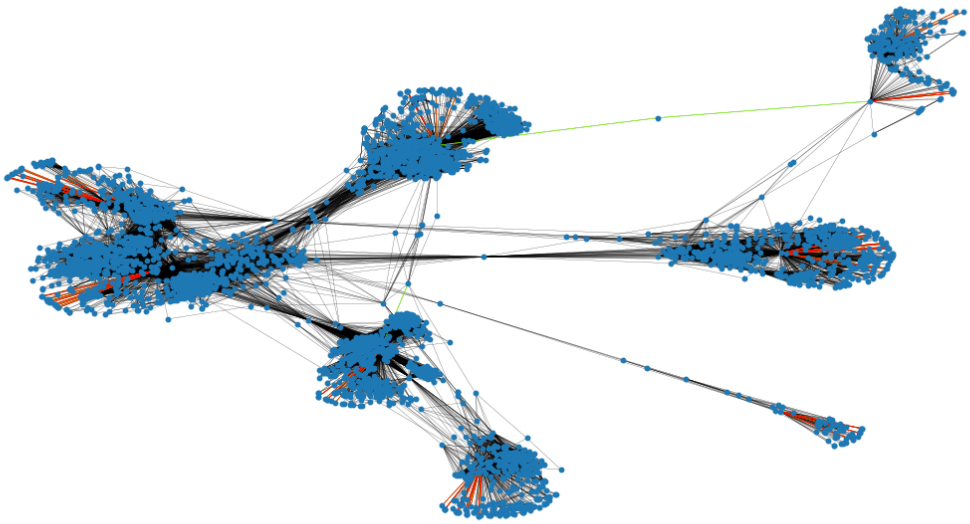
Out[13]: 75

In [14]:
```
local_bridges = list(nx.local_bridges(G, with_span=False))
len(local_bridges)
```

Out[14]: 78

In [15]:
```
plt.figure(figsize=(15, 8))
nx.draw_networkx(G, pos=pos, node_size=10, with_labels=False, width=0.15)
nx.draw_networkx_edges(
    G, pos, edgelist=local_bridges, width=0.5, edge_color="lawngreen"
)  # green color for local bridges
nx.draw_networkx_edges(
    G, pos, edgelist=bridges, width=0.5, edge_color="r"
)  # red color for bridges
plt.axis("off")
```

Out[15]:
```
(-0.9158058530092239,
 1.1820449000597002,
 -1.1815252473950386,
 1.1715664437413216)
```



Potential Links

In [16]:
```
potential_links = {}
for u in G.nodes():
    for v in G.nodes():
        if u != v and not G.has_edge(u, v):
            common_neighbors = len(list(nx.common_neighbors(G, u, v)))
            if common_neighbors > 0:
                potential_links[(u, v)] = common_neighbors

top_links = sorted(potential_links.items(), key=lambda x: x[1], reverse=T

pos = nx.spring_layout(G, scale=2)

plt.figure(figsize=(14, 10))
nx.draw_networkx_nodes(G, pos, node_size=20, node_color='blue', alpha=0.2
```

```
nx.draw_networkx_edges(G, pos, alpha=0.1)

for link, count in top_links:
    nx.draw_networkx_edges(G, pos, edgelist=[link], width=4, edge_color='
    midpoint = ((pos[link[0]][0] + pos[link[1]][0]) / 2, (pos[link[0]][1]
    plt.text(midpoint[0], midpoint[1], f'{count}', color='red', fontsize=

plt.title('Network Visualization with Top Predicted Links Highlighted')
plt.axis('off')
plt.show()
```