# A02014087 张文滔 复杂网络平时作二

1.生成N=1000，连边概率p=0.05的随机网络，计算最大连通子图的规模。求最大连通子图的平均路径长度，平均聚类系数，统计度分布。

## 代码

from itertools import combinations

import random

import networkx as nx

import matplotlib.pyplot as plt

import numpy as np

from collections import Counter

from collections import deque

def clustering\_coefficient(G):

    '''在上述代码中, 我们首先将聚类系数 coefficient 初始化为 0 , 然后遍历图中的每个节点。

对于每个节点, 我们获取其所有邻居节点, 并计 算该节点的度数 $k$ 。

如果该节点的度数大于 1 , 则计算该节点的聚类系数 num\_edges, 并将其加入总聚类系数中。

在计算聚类系数时, 我们使用 combinations 函数生成该节点所有邻居节点中的两两组合, 然后计算相邻节点之间是否存在边。

最后, 我们将总聚类系数除以节点数, 得到平均聚类系数, 并返回该值。

这样做的时间复杂度为 $O\left(n^2\right)$, 其中 $n$ 是节点数。由于 combinations 函数的空间复杂度为 $O\left(k^2\right)$, 其中 $k$ 是节点的平均度数, 因此总的空间复杂度为 $O\left(n k^2\right)$ 。'''

    # 计算图对象 G 的平均聚类系数,输入是图对象，输出是平均聚类系数

    coefficient = 0  # 初始化聚类系数为0

    for node in G.nodes:  # 遍历图中每个节点

        neighbors = list(G.neighbors(node))  # 获取该节点的所有邻居节点

        k = len(neighbors)  # 该节点的度数

        if k > 1:  # 只有度数大于1的节点才有聚类系数

            # 计算该节点的聚类系数

            num\_edges = sum(1 for v, w in combinations(neighbors, 2) if G.has\_edge(v, w))

            coefficient += num\_edges / (k \* (k - 1))

    print(f"\nAverage Clustering Coefficient:{2 \* coefficient / G.number\_of\_nodes()}")

    return 2 \* coefficient / G.number\_of\_nodes()  # 返回平均聚类系数

def degree\_distribution(G):

    # 获取所有节点的度并创建Counter对象，记录每个度出现的次数

    counts = Counter(deg for \_, deg in G.degree)

    # 找到最小和最大的度

    \_, max\_degree = min(counts), max(counts)

    max\_degree = max(counts.keys())

    degree\_counts = [0] \* (max\_degree + 1)

    for degree, count in counts.items():

        degree\_counts[degree] = count

    print('\nDegree Distribution:')

    print(degree\_counts)

    # 绘制度分布直方图

    plt.bar(counts.keys(), counts.values())

    plt.xlabel('Degree')

    plt.ylabel('Count')

    plt.title('Degree Distribution')

    # plt.xticks(range(min\_degree, max\_degree + 1))

    plt.show()

def breadth\_first\_search(G, source, target=None):

    '''

    输入：

    G：一个图，表示节点及其相互之间的连接关系。在这里，它应该是一个 NetworkX 图对象。

    source：源节点，是 BFS 算法的起点。

    target（可选）：目标节点，如果提供了这个参数，函数将返回源节点到目标节点的最短距离。

    输出：

    如果没有指定目标节点，函数将返回一个字典，键为图中的每个节点，值为一个元组，包含两个元素：该节点到源节点的最短距离和该节点是否已被访问过。

    如果指定了目标节点，函数将返回一个整数，表示源节点到目标节点的最短距离。

    '''

    # 初始化队列，将源节点加入队列

    queue = deque([source])

    # 为图中的每个节点创建一个字典，键为节点，值为一个包含两个元素的元组：距离和是否访问过的布尔值

    node\_info = {node: (0, False) for node in G.nodes}

    # 将源节点的距离设置为 0，并将其访问状态设置为 True

    node\_info[source] = (0, True)

    # 当队列非空时，继续执行循环

    while queue:

        # 从队列左侧移除并返回一个节点，将其作为当前节点

        current\_node = queue.popleft()

        # 如果目标节点不为空，且当前节点等于目标节点，退出循环

        if target is not None and current\_node == target:

            break

        # 获取当前节点的相邻节点列表

        neighbors = list(G.neighbors(current\_node))

        # 遍历相邻节点

        for neighbor in neighbors:

            # 获取相邻节点的距离和访问状态

            distance, visited = node\_info[neighbor]

            # 如果相邻节点未访问过

            if not visited:

                # 更新相邻节点的距离和访问状态

                node\_info[neighbor] = (node\_info[current\_node][0] + 1, True)

                # 将相邻节点添加到队列的右侧

                queue.append(neighbor)

    # 如果没有指定目标节点，返回包含所有节点信息的字典

    if target is None:

        return node\_info

    # 如果指定了目标节点，返回目标节点的距离

    else:

        return node\_info[target][0]

def bfs(graph, start\_node, visited):

    queue = [start\_node]

    visited.add(start\_node)

    while queue:

        current\_node = queue.pop(0)

        for neighbor in graph[current\_node]:

            if neighbor not in visited:

                visited.add(neighbor)

                queue.append(neighbor)

def is\_connected(G):

    start = next(iter(G.nodes))  # 选择任意一个节点作为起点

    target = None  # 设置目标节点为 None，因为我们只关心访问的节点数量

    node\_distances = breadth\_first\_search(G, start, target)

    visited\_nodes = [node for node, (distance, visited) in node\_distances.items() if visited]

    if len(visited\_nodes) == len(G.nodes):

        print("Graph is connected")

        return True

    else:

        print("Graph is not connected")

        return False

def get\_largest\_connected\_component(graph):

    visited = set()

    largest\_cc = set()

    for node in graph:

        if node not in visited:

            current\_cc = set()

            bfs(graph, node, current\_cc)

            if len(current\_cc) > len(largest\_cc):

                largest\_cc = current\_cc

    return largest\_cc

def random\_network(n, p):

    # 创建一个空的网络

    G = nx.Graph()

    # 添加节点

    G.add\_nodes\_from(range(1, n+1))

    # 添加边

    for i in range(1, n + 1):

        for j in range(i + 1, n + 1):

            if random.random() < p:

                G.add\_edge(i, j)

                G.add\_edge(j, i)

    return G

def average\_path\_length(G):

    n = G.number\_of\_nodes()

    matrix = [np.zeros((n, n))]

    shortest\_path = np.zeros((n, n))

    adj\_matrix = np.array(nx.to\_numpy\_matrix(G))

    matrix = adj\_matrix

    np.fill\_diagonal(shortest\_path, np.inf)

    for i in range(1, n):

        if shortest\_path.all() != 0:

            break

        for p in range(n):

            for q in range(n):

                if q != p:

                    if matrix[p][q] != 0 and shortest\_path[p][q] == 0:

                        shortest\_path[p][q] = i

        matrix = matrix @ adj\_matrix

    # 将对角线上的值设置为0，因为节点到自身的距离为0

    np.fill\_diagonal(shortest\_path, 0)

    # 计算平均最短路径长度

    average = np.sum(shortest\_path) / (n \* (n - 1))

    print(f'Average Path Length is {average}')

    return average

G = random\_network(1000, 0.05)

# 判断是否连通，如果不连通，删除不连通的节点

if not is\_connected(G):

    G\_set = get\_largest\_connected\_component(G)

    G\_list = list(G\_set)

    print(G\_list)

    for i in list(G.nodes()):

        if i not in G\_list:

            G.remove\_node(i)

# 输出平均最短路径长度

average\_path\_length(G)

print(f"python NetworkX results:{nx.average\_shortest\_path\_length(G)}")

# 输出平均聚类系数

clustering\_coefficient(G)

print(f"python NetworkX results:{nx.average\_clustering(G)}")

# 输出度分布序列

degree\_distribution(G)

print(f"python NetworkX results:\n{nx.degree\_histogram(G)}")

## 运行结果

Graph is connected

Average Path Length is 2.0283023023023024

python NetworkX results:2.0283023023023024

Average Clustering Coefficient:0.04980091553303759

python NetworkX results:0.04980091553303759

Degree Distribution:

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 2, 6, 5, 5, 9, 14, 11, 24, 28, 28, 37, 38, 42, 59, 50, 64, 64, 56, 70, 57, 50, 49, 36, 39, 21, 31, 18, 24, 19, 15, 6, 4, 3, 3, 6, 4, 1, 1]

python NetworkX results:

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 2, 6, 5, 5, 9, 14, 11, 24, 28, 28, 37, 38, 42, 59, 50, 64, 64, 56, 70, 57, 50, 49, 36, 39, 21, 31, 18, 24, 19, 15, 6, 4, 3, 3, 6, 4, 1, 1]

Chart, histogram

Description automatically generated

2.生成N=1000，平均度为6，重连概率p=0.02的WS小世界网络，计算最大连通子图的规模。求最大连通子图的平均路径长度，平均聚类系数，统计度分布。

## 代码

from itertools import combinations

import random

import networkx as nx

import matplotlib.pyplot as plt

import numpy as np

from collections import Counter

from collections import deque

def clustering\_coefficient(G):

    '''在上述代码中, 我们首先将聚类系数 coefficient 初始化为 0 , 然后遍历图中的每个节点。

对于每个节点, 我们获取其所有邻居节点, 并计 算该节点的度数 $k$ 。

如果该节点的度数大于 1 , 则计算该节点的聚类系数 num\_edges, 并将其加入总聚类系数中。

在计算聚类系数时, 我们使用 combinations 函数生成该节点所有邻居节点中的两两组合, 然后计算相邻节点之间是否存在边。

最后, 我们将总聚类系数除以节点数, 得到平均聚类系数, 并返回该值。

这样做的时间复杂度为 $O\left(n^2\right)$, 其中 $n$ 是节点数。由于 combinations 函数的空间复杂度为 $O\left(k^2\right)$, 其中 $k$ 是节点的平均度数, 因此总的空间复杂度为 $O\left(n k^2\right)$ 。'''

    # 计算图对象 G 的平均聚类系数,输入是图对象，输出是平均聚类系数

    coefficient = 0  # 初始化聚类系数为0

    for node in G.nodes:  # 遍历图中每个节点

        neighbors = list(G.neighbors(node))  # 获取该节点的所有邻居节点

        k = len(neighbors)  # 该节点的度数

        if k > 1:  # 只有度数大于1的节点才有聚类系数

            # 计算该节点的聚类系数

            num\_edges = sum(1 for v, w in combinations(neighbors, 2) if G.has\_edge(v, w))

            coefficient += num\_edges / (k \* (k - 1))

    print(f"\nAverage Clustering Coefficient:{2 \* coefficient / G.number\_of\_nodes()}")

    return 2 \* coefficient / G.number\_of\_nodes()  # 返回平均聚类系数

def degree\_distribution(G):

    # 获取所有节点的度并创建Counter对象，记录每个度出现的次数

    counts = Counter(deg for \_, deg in G.degree)

    # 找到最小和最大的度

    min\_degree, max\_degree = min(counts), max(counts)

    max\_degree = max(counts.keys())

    degree\_counts = [0] \* (max\_degree + 1)

    for degree, count in counts.items():

        degree\_counts[degree] = count

    print('\nDegree Distribution:')

    print(degree\_counts)

    # 绘制度分布直方图

    plt.bar(counts.keys(), counts.values())

    plt.xlabel('Degree')

    plt.ylabel('Count')

    plt.title('Degree Distribution')

    # plt.xticks(range(min\_degree, max\_degree + 1))

    plt.show()

def breadth\_first\_search(G, source, target=None):

    '''

    输入：

    G：一个图，表示节点及其相互之间的连接关系。在这里，它应该是一个 NetworkX 图对象。

    source：源节点，是 BFS 算法的起点。

    target（可选）：目标节点，如果提供了这个参数，函数将返回源节点到目标节点的最短距离。

    输出：

    如果没有指定目标节点，函数将返回一个字典，键为图中的每个节点，值为一个元组，包含两个元素：该节点到源节点的最短距离和该节点是否已被访问过。

    如果指定了目标节点，函数将返回一个整数，表示源节点到目标节点的最短距离。

    '''

    # 初始化队列，将源节点加入队列

    queue = deque([source])

    # 为图中的每个节点创建一个字典，键为节点，值为一个包含两个元素的元组：距离和是否访问过的布尔值

    node\_info = {node: (0, False) for node in G.nodes}

    # 将源节点的距离设置为 0，并将其访问状态设置为 True

    node\_info[source] = (0, True)

    # 当队列非空时，继续执行循环

    while queue:

        # 从队列左侧移除并返回一个节点，将其作为当前节点

        current\_node = queue.popleft()

        # 如果目标节点不为空，且当前节点等于目标节点，退出循环

        if target is not None and current\_node == target:

            break

        # 获取当前节点的相邻节点列表

        neighbors = list(G.neighbors(current\_node))

        # 遍历相邻节点

        for neighbor in neighbors:

            # 获取相邻节点的距离和访问状态

            distance, visited = node\_info[neighbor]

            # 如果相邻节点未访问过

            if not visited:

                # 更新相邻节点的距离和访问状态

                node\_info[neighbor] = (node\_info[current\_node][0] + 1, True)

                # 将相邻节点添加到队列的右侧

                queue.append(neighbor)

    # 如果没有指定目标节点，返回包含所有节点信息的字典

    if target is None:

        return node\_info

    # 如果指定了目标节点，返回目标节点的距离

    else:

        return node\_info[target][0]

def bfs(graph, start\_node, visited):

    queue = [start\_node]

    visited.add(start\_node)

    while queue:

        current\_node = queue.pop(0)

        for neighbor in graph[current\_node]:

            if neighbor not in visited:

                visited.add(neighbor)

                queue.append(neighbor)

def is\_connected(G):

    start = next(iter(G.nodes))  # 选择任意一个节点作为起点

    target = None  # 设置目标节点为 None，因为我们只关心访问的节点数量

    node\_distances = breadth\_first\_search(G, start, target)

    visited\_nodes = [node for node, (distance, visited) in node\_distances.items() if visited]

    if len(visited\_nodes) == len(G.nodes):

        print("Graph is connected")

        return True

    else:

        print("Graph is not connected")

        return False

def get\_largest\_connected\_component(graph):

    visited = set()

    largest\_cc = set()

    for node in graph:

        if node not in visited:

            current\_cc = set()

            bfs(graph, node, current\_cc)

            if len(current\_cc) > len(largest\_cc):

                largest\_cc = current\_cc

    return largest\_cc

def average\_path\_length(G):

    n = G.number\_of\_nodes()

    matrix = [np.zeros((n, n))]

    shortest\_path = np.zeros((n, n))

    adj\_matrix = np.array(nx.to\_numpy\_matrix(G))

    matrix = adj\_matrix

    np.fill\_diagonal(shortest\_path, np.inf)

    for i in range(1, n):

        if shortest\_path.all() != 0:

            break

        for p in range(n):

            for q in range(n):

                if q != p:

                    if matrix[p][q] != 0 and shortest\_path[p][q] == 0:

                        shortest\_path[p][q] = i

        matrix = matrix @ adj\_matrix

    # 将对角线上的值设置为0，因为节点到自身的距离为0

    np.fill\_diagonal(shortest\_path, 0)

    # 计算平均最短路径长度

    average = np.sum(shortest\_path) / (n \* (n - 1))

    print(f'Average Path Length is {average}')

    return average

def generate\_ws\_small\_world\_network(N, k, p):

    # 初始化网络，创建一个环形的规则图

    G = nx.Graph()

    adj = np.zeros((N, N))

    for i in range(N):

        for j in range(1, k // 2 + 1):

            adj[i, (i + j) % N] = 1

            adj[i, (i - j) % N] = 1

    # 将下三角全部置为零

    adj1 = np.triu(adj)

    # 将邻接矩阵表示为G对象

    G = nx.from\_numpy\_matrix(adj1)

    avg\_degree = sum(dict(G.degree()).values()) / len(G)

    print(G.degree())

    print("平均度数:", avg\_degree)

    edges = list(G.edges())

    print(edges)

    for m, e in enumerate(edges):

        i, j = e

        if random.random() > 1 - p:

            non\_one\_indices = np.where((adj[i, :] != 1) & (np.arange(N) != i))[0]

            if len(non\_one\_indices) > 0:

                k = random.choice(non\_one\_indices)

                adj[i, k] = 1

                adj[k, i] = 1

                adj[i, j] = 0

                adj[j, i] = 0

    # 将下三角全部置为零

    adj2 = np.triu(adj)

    print(adj)

    G = nx.from\_numpy\_matrix(adj2)

    nx.draw

    return G

N = 1000

k = 6

p = 0.02

G = generate\_ws\_small\_world\_network(N, k, p)

# 判断是否是连通图

if not is\_connected(G):

    G\_set = get\_largest\_connected\_component(G)

    G\_list = list(G\_set)

    print(G\_list)

    for i in list(G.nodes()):

        if i not in G\_list:

            G.remove\_node(i)

# 输出平均最短路径长度

average\_path\_length(G)

print(f"python NetworkX results:{nx.average\_shortest\_path\_length(G)}")

# 输出平均聚类系数

clustering\_coefficient(G)

print(f"python NetworkX results:{nx.average\_clustering(G)}")

# 输出度分布序列

degree\_distribution(G)

print(f"python NetworkX results:\n{nx.degree\_histogram(G)}")

## 运行结果

Graph is connected

Average Path Length is 12.00619019019019

python NetworkX results:12.00619019019019

Average Clustering Coefficient:0.5690119047619137

python NetworkX results:0.5690119047619137

Degree Distribution:

[0, 0, 0, 0, 0, 50, 901, 48, 1]

python NetworkX results:

[0, 0, 0, 0, 0, 50, 901, 48, 1]

Chart, bar chart

Description automatically generated

3.生成N=1000，m=3的BA网络，求网络的平均路径长度，平均聚类系数，统计度分布。

# 代码

from itertools import combinations

import random

import networkx as nx

import matplotlib.pyplot as plt

import numpy as np

from collections import Counter

from collections import deque

def generate\_ba\_small\_world\_network(N, m0, m):

    G = nx.Graph()

    G.add\_nodes\_from(range(1, m0+1))

    G.add\_edges\_from([[1,edge] for edge in [node for node in G.nodes() if node != 1]])

    for i in range(m0+1, N+1):

        for \_ in range(m):

            degree = dict(G.degree())

            degree = {k: v / sum(degree.values()) for k, v in degree.items()}

            sorted\_degree = sorted(degree.items(), key=lambda x: x[1], reverse=True)

            # Connect the new node to 'm' existing nodes

            node = random.choices([node[0] for node in sorted\_degree], degree.values())

            G.add\_edge(i, node[0])

    return G

def clustering\_coefficient(G):

    '''在上述代码中, 我们首先将聚类系数 coefficient 初始化为 0 , 然后遍历图中的每个节点。

对于每个节点, 我们获取其所有邻居节点, 并计 算该节点的度数 $k$ 。

如果该节点的度数大于 1 , 则计算该节点的聚类系数 num\_edges, 并将其加入总聚类系数中。

在计算聚类系数时, 我们使用 combinations 函数生成该节点所有邻居节点中的两两组合, 然后计算相邻节点之间是否存在边。

最后, 我们将总聚类系数除以节点数, 得到平均聚类系数, 并返回该值。

这样做的时间复杂度为 $O\left(n^2\right)$, 其中 $n$ 是节点数。由于 combinations 函数的空间复杂度为 $O\left(k^2\right)$, 其中 $k$ 是节点的平均度数, 因此总的空间复杂度为 $O\left(n k^2\right)$ 。'''

    # 计算图对象 G 的平均聚类系数,输入是图对象，输出是平均聚类系数

    coefficient = 0  # 初始化聚类系数为0

    for node in G.nodes:  # 遍历图中每个节点

        neighbors = list(G.neighbors(node))  # 获取该节点的所有邻居节点

        k = len(neighbors)  # 该节点的度数

        if k > 1:  # 只有度数大于1的节点才有聚类系数

            # 计算该节点的聚类系数

            num\_edges = sum(1 for v, w in combinations(neighbors, 2) if G.has\_edge(v, w))

            coefficient += num\_edges / (k \* (k - 1))

    print(f"\nAverage Clustering Coefficient:{2 \* coefficient / G.number\_of\_nodes()}")

    return 2 \* coefficient / G.number\_of\_nodes()  # 返回平均聚类系数

def degree\_distribution(G):

    # 获取所有节点的度并创建Counter对象，记录每个度出现的次数

    counts = Counter(deg for \_, deg in G.degree)

    # 找到最小和最大的度

    min\_degree, max\_degree = min(counts), max(counts)

    max\_degree = max(counts.keys())

    degree\_counts = [0] \* (max\_degree + 1)

    for degree, count in counts.items():

        degree\_counts[degree] = count

    print('\nDegree Distribution:')

    print(degree\_counts)

    # 绘制度分布直方图

    plt.bar(counts.keys(), counts.values())

    plt.xlabel('Degree')

    plt.ylabel('Count')

    plt.title('Degree Distribution')

    # plt.xticks(range(min\_degree, max\_degree + 1))

    plt.show()

def breadth\_first\_search(G, source, target=None):

    '''

    输入：

    G：一个图，表示节点及其相互之间的连接关系。在这里，它应该是一个 NetworkX 图对象。

    source：源节点，是 BFS 算法的起点。

    target（可选）：目标节点，如果提供了这个参数，函数将返回源节点到目标节点的最短距离。

    输出：

    如果没有指定目标节点，函数将返回一个字典，键为图中的每个节点，值为一个元组，包含两个元素：该节点到源节点的最短距离和该节点是否已被访问过。

    如果指定了目标节点，函数将返回一个整数，表示源节点到目标节点的最短距离。

    '''

    # 初始化队列，将源节点加入队列

    queue = deque([source])

    # 为图中的每个节点创建一个字典，键为节点，值为一个包含两个元素的元组：距离和是否访问过的布尔值

    node\_info = {node: (0, False) for node in G.nodes}

    # 将源节点的距离设置为 0，并将其访问状态设置为 True

    node\_info[source] = (0, True)

    # 当队列非空时，继续执行循环

    while queue:

        # 从队列左侧移除并返回一个节点，将其作为当前节点

        current\_node = queue.popleft()

        # 如果目标节点不为空，且当前节点等于目标节点，退出循环

        if target is not None and current\_node == target:

            break

        # 获取当前节点的相邻节点列表

        neighbors = list(G.neighbors(current\_node))

        # 遍历相邻节点

        for neighbor in neighbors:

            # 获取相邻节点的距离和访问状态

            distance, visited = node\_info[neighbor]

            # 如果相邻节点未访问过

            if not visited:

                # 更新相邻节点的距离和访问状态

                node\_info[neighbor] = (node\_info[current\_node][0] + 1, True)

                # 将相邻节点添加到队列的右侧

                queue.append(neighbor)

    # 如果没有指定目标节点，返回包含所有节点信息的字典

    if target is None:

        return node\_info

    # 如果指定了目标节点，返回目标节点的距离

    else:

        return node\_info[target][0]

def bfs(graph, start\_node, visited):

    queue = [start\_node]

    visited.add(start\_node)

    while queue:

        current\_node = queue.pop(0)

        for neighbor in graph[current\_node]:

            if neighbor not in visited:

                visited.add(neighbor)

                queue.append(neighbor)

def is\_connected(G):

    start = next(iter(G.nodes))  # 选择任意一个节点作为起点

    target = None  # 设置目标节点为 None，因为我们只关心访问的节点数量

    node\_distances = breadth\_first\_search(G, start, target)

    visited\_nodes = [node for node, (distance, visited) in node\_distances.items() if visited]

    if len(visited\_nodes) == len(G.nodes):

        print("Graph is connected")

        return True

    else:

        print("Graph is not connected")

        return False

def get\_largest\_connected\_component(graph):

    visited = set()

    largest\_cc = set()

    for node in graph:

        if node not in visited:

            current\_cc = set()

            bfs(graph, node, current\_cc)

            if len(current\_cc) > len(largest\_cc):

                largest\_cc = current\_cc

    return largest\_cc

def average\_path\_length(G):

    n = G.number\_of\_nodes()

    matrix = [np.zeros((n, n))]

    shortest\_path = np.zeros((n, n))

    adj\_matrix = np.array(nx.to\_numpy\_matrix(G))

    matrix = adj\_matrix

    np.fill\_diagonal(shortest\_path, np.inf)

    for i in range(1, n):

        if shortest\_path.all() != 0:

            break

        for p in range(n):

            for q in range(n):

                if q != p:

                    if matrix[p][q] != 0 and shortest\_path[p][q] == 0:

                        shortest\_path[p][q] = i

        matrix = matrix @ adj\_matrix

    # 将对角线上的值设置为0，因为节点到自身的距离为0

    np.fill\_diagonal(shortest\_path, 0)

    # 计算平均最短路径长度

    average = np.sum(shortest\_path) / (n \* (n - 1))

    print(f'Average Path Length is {average}')

    return average

N = 1000

m0 = 3

m = 3

G = generate\_ba\_small\_world\_network(N, m0, m)

average\_path\_length(G)

print(f"NetworkX average\_shortest\_path\_length{nx.average\_shortest\_path\_length(G)}")

clustering\_coefficient(G)

print(f"NetworkX average\_clustering:{nx.average\_clustering(G)}")

degree\_distribution(G)

print(f"NetworkX degree\_distribution:{nx.degree\_histogram(G)}")

# 运行结果

Average Path Length is 3.6052652652652655

NetworkX average\_shortest\_path\_length3.6052652652652655

Average Clustering Coefficient:0.0209988456608948

NetworkX average\_clustering:0.0209988456608948

Degree Distribution:

[0, 0, 4, 363, 210, 117, 78, 45, 42, 24, 25, 18, 10, 8, 3, 5, 3, 4, 7, 3, 5, 1, 1, 3, 1, 2, 2, 0, 2, 1, 1, 0, 1, 0, 0, 0, 0, 1, 2, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

NetworkX Degree Distribution:

[0, 0, 4, 363, 210, 117, 78, 45, 42, 24, 25, 18, 10, 8, 3, 5, 3, 4, 7, 3, 5, 1, 1, 3, 1, 2, 2, 0, 2, 1, 1, 0, 1, 0, 0, 0, 0, 1, 2, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

Chart, histogram

Description automatically generated

4.生成一个配置网络，使得其节点规模数以及度分布和Email网络相同。

# 代码

from collections import Counter

import networkx as nx

import random

def degree\_distribution(G):

    # 获取所有节点的度并创建 Counter 对象，记录每个度出现的次数

    counts = Counter(deg for \_, deg in G.degree)

    # 初始化一个空列表来存储输出

    output = []

    # 遍历 Counter 对象

    for degree, count in counts.items():

        # 将每个度数添加到输出列表中，次数等于其在 Counter 中的计数值

        output.extend([degree] \* count)

    return output

def create\_configuration\_model(N, degree\_sequence):

    # 创建一个空的无向图

    G = nx.Graph()

    # 添加 N 个节点

    G.add\_nodes\_from(range(N))

    # 生成每个节点的桩子

    stubs = [node for node, degree in enumerate(degree\_sequence) for \_ in range(degree)]

    # 随机连接桩子，生成边

    while len(stubs) > 1:

        stub1 = random.choice(stubs)

        stub2 = random.choice(list(filter(lambda x: x != stub1, stubs)))  # 避免自环

        if not G.has\_edge(stub1, stub2):  # 避免多重边

            G.add\_edge(stub1, stub2)

        else:

            continue

        stubs.remove(stub1)

        stubs.remove(stub2)

    return G

# 定义文件路径

file\_path = 'email.txt'

# 从文件中读取边列表，创建一个图，其中节点的类型为整数，边的权重为整数

G = nx.read\_edgelist(

    file\_path,

    nodetype=int,

    data=(('weight', int),)

)

# 获取原始图的节点数和度分布

N = G.number\_of\_nodes()

degree\_sequence = degree\_distribution(G)

# 创建一个具有email网络节点个数和给定度序列的配置模型

G\_config = create\_configuration\_model(N, degree\_sequence)

# 计算配置模型的度直方图并删除值为 0 的元素

histogram = nx.degree\_histogram(G\_config)

histogram = [x for x in histogram if x != 0]

# 计算配置模型的度分布

counts = Counter(deg for \_, deg in G\_config.degree)

degree\_counts = list(counts.values())

# 比较度直方图和度分布是否相同

comparison\_result = sorted(histogram, reverse=True) == sorted(degree\_counts, reverse=True)

print(f"度分布进行比较：{comparison\_result}")

# 代码运行结果

度分布进行比较：True