A02014087 张文滔 复杂网络平时作业一

Contents

[Karate网络简介 1](#_Toc131362945)

[第一题 分别用邻接矩阵、三元组表示网络 2](#_Toc131362946)

[代码 2](#_Toc131362947)

[运行结果 2](#_Toc131362948)

[第二题 计算网络的聚类系数，网络的度分布，平均路径长度 3](#_Toc131362949)

[代码 3](#_Toc131362950)

[运行结果 6](#_Toc131362951)

[第三题 利用广度优先算法计算任意两点之间的距离 6](#_Toc131362952)

[代码 6](#_Toc131362953)

[运行结果 8](#_Toc131362954)

[第四题 判断给定一个网络是否连通？如不连通，计算网络的最大连通巨片 8](#_Toc131362955)

[代码 8](#_Toc131362956)

[运行结果 8](#_Toc131362957)

[第五题 利用余平均度和标准化的同配系数r刻画一个网络的同配性 9](#_Toc131362958)

[代码 9](#_Toc131362959)

[运行结果 10](#_Toc131362960)

[第六题 利用python中的networkx绘制精美网络图(见：https://blog.csdn.net/qq951127336/article/details/54586869) 11](#_Toc131362961)

[代码 11](#_Toc131362962)

[运行结果 12](#_Toc131362963)

[第七题 绘制富人俱乐部系数vs节点度的关系图，如下图所示 12](#_Toc131362964)

[代码 12](#_Toc131362965)

[运行结果 15](#_Toc131362966)

# Karate网络简介

Karate网络（Karate Club Network）是一个常用于复杂网络社区发现研究的数据集。该网络来源于20世纪70年代初期，社会学家Zachary观察了美国一所大学中空手道俱乐部34名成员间的社会关系，通过这些成员在俱乐部内外的交往情况，构造了成员之间的社会关系网。该网络共有34个节点和78条边，其中34个节点表示某空手道俱乐部的34名成员，节点之间的边表示两个成员相互认识。这个数据集是一个真实的数据集，对应于美国的一个空手道俱乐部的人物关系研究。

# 第一题 分别用邻接矩阵、三元组表示网络

## 代码

import networkx as nx

import numpy as np

# 定义文件路径

file\_path = 'karate.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),)

)

# 将图转换为 NumPy 矩阵

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

# 使用格式化字符串打印邻接矩阵

print(f"邻接矩阵为:\n{np.array(adj\_matrix)}\n")

# 生成三元组，表示节点之间的边和权重，如 (1, 2, 1) 表示节点 1 和节点 2 之间有一条权重为 1 的边

edges = [(x[0], x[1], 1) for x in G.edges]

# 使用格式化字符串打印三元组

print(f"三元组为: {edges}")

## 运行结果

邻接矩阵为:

[0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
1 0 1 1 0 0 0 1 0 0 0 0 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
1 1 0 1 0 0 0 1 1 0 0 0 1 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0  
1 1 1 0 0 0 0 1 0 0 0 1 1 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 1 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  
1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0  
1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0  
1 1 1 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 0 0  
1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0  
1 0 0 0 1 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 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 0 0 0 0 0  
1 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 0 0  
1 1 1 1 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 0 0 0 0  
1 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 0 0 0 0  
1 1 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 0 0 0 0  
1 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 0 0 0 0  
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0 0 1 0 1 0  
0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 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 1 0 0 0 0 0 0 0 0 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 1 0 0 0 0 0 1 0 0 1 0  
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0  
0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 1 1 1 1 1 1 0 1 0 0  
0 0 0 0 0 1 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 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 1 0 1  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 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 0 0 0 1 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 0 0 0 1 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 0 0 0 1 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 0 0 0 1 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 0 1 0 1 0 1 0 0 0 0 0 0 1 1 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 0 0 0 0 0 0 1 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 1 0 1 0 0 0 0 0 1 0 0 0 1  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 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 1 0 0 0 0 0 0 0 1 0 0]

三元组为: [(1, 2, 1), (1, 3, 1), (1, 4, 1), (1, 5, 1), (1, 6, 1), (1, 7, 1), (1, 8, 1), (1, 9, 1), (1, 11, 1), (1, 12, 1), (1, 13, 1), (1, 14, 1), (1, 18, 1), (1, 20, 1), (1, 22, 1), (1, 32, 1), (2, 3, 1), (2, 4, 1), (2, 8, 1), (2, 14, 1), (2, 18, 1), (2, 20, 1), (2, 22, 1), (2, 31, 1), (3, 4, 1), (3, 8, 1), (3, 9, 1), (3, 10, 1), (3, 14, 1), (3, 28, 1), (3, 29, 1), (3, 33, 1), (4, 8, 1), (4, 13, 1), (4, 14, 1), (5, 7, 1), (5, 11, 1), (6, 7, 1), (6, 11, 1), (6, 17, 1), (7, 17, 1), (9, 31, 1), (9, 33, 1), (9, 34, 1), (14, 34, 1), (20, 34, 1), (32, 25, 1), (32, 26, 1), (32, 29, 1), (32, 33, 1), (32, 34, 1), (31, 33, 1), (31, 34, 1), (10, 34, 1), (28, 24, 1), (28, 25, 1), (28, 34, 1), (29, 34, 1), (33, 15, 1), (33, 16, 1), (33, 19, 1), (33, 21, 1), (33, 23, 1), (33, 24, 1), (33, 30, 1), (33, 34, 1), (34, 15, 1), (34, 16, 1), (34, 19, 1), (34, 21, 1), (34, 23, 1), (34, 24, 1), (34, 27, 1), (34, 30, 1), (24, 26, 1), (24, 30, 1), (26, 25, 1), (30, 27, 1)]

# 第二题 计算网络的聚类系数，网络的度分布，平均路径长度

## 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):

    # 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('\nAverage Clustering Coefficient:')

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

'''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 $O\left(n k^2\right)$ 。'''

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)

    # 遍历所有可能的度，输出对应的出现次数

    for d in range(min\_degree, max\_degree + 1):

        count = counts[d] # 如果d不存在于counts中，返回0

    # 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.show()

def average\_path\_length(G):

    # 获取图中节点的数量

    n = G.number\_of\_nodes()

    # 初始化矩阵列表和最短路径矩阵

    matrix = [np.zeros((n, n))]

    shortest\_path = np.zeros((n, n))

    # 从图中获取邻接矩阵

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

    # 将邻接矩阵添加到矩阵列表

    matrix[0] = adj\_matrix

    # 计算邻接矩阵的幂，并更新最短路径矩阵

    for i in range(n):

        # 计算下一个矩阵

        next\_matrix = matrix[i] @ adj\_matrix

        # 将计算结果添加到矩阵列表

        matrix.append(next\_matrix)

        # 更新最短路径矩阵

        for p in range(n):

            for q in range(n):

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

                    shortest\_path[p][q] = i + 1

    # 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))

    # 打印平均最短路径长度

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

    # 返回平均最短路径长度

    return average

# 输出平均聚类系数

print(clustering\_coefficient(G), "\n")

# 输出度分布序列

degree\_distribution(G)

# 输出平均最短路径长度

average\_path\_length(np.array(nx.to\_numpy\_matrix(G)))

## 运行结果

Average Clustering Coefficient:

0.5706384782076823

Chart, histogram

Description automatically generated

Average Path Length is 1.5668449197860963

# 第三题 利用广度优先算法计算任意两点之间的距离

## 代码

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

    # 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]

'''

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。

'''

source, target = 1, 31

print(f'Breadth First Search Distance between {source} and {target}:')

print(breadth\_first\_search(G, 1, 31))

## 运行结果

Breadth First Search Distance between 1 and 31:

2

# 第四题 判断给定一个网络是否连通？如不连通，计算网络的最大连通巨片

## 代码

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

print('Is Connected?:')

print(is\_connected(G))

## 运行结果

Is Connected?:

Graph is connected

True

# 第五题 利用余平均度和标准化的同配系数r刻画一个网络的同配性

## 代码

def compute\_yu\_average(G):

    # 定义一个字典，用于存储每个节点的 Yu 平均度数

    yu\_average = {}

    # 迭代处理每个节点

    for node in G.nodes():

        # 获取当前节点的邻居节点列表

        neighbors = list(G.neighbors(node))

        # 如果当前节点有邻居，则计算其邻居节点的度数总和并除以邻居节点数，得到平均度数

        if len(neighbors) > 0:

            yu\_average[node] = sum(d for n, d in G.degree(neighbors)) / len(neighbors)

        # 如果当前节点没有邻居，则将其平均度数设置为 0

        else:

            yu\_average[node] = 0

    # 返回每个节点的 Yu 平均度数

    return yu\_average

def normalized\_assortativity(G):

    # 计算网络中每个节点的度数

    d = dict(G.degree())

    # 提取网络中的边

    edges = list(G.edges())

    # 计算边的数量

    K = len(edges)

    # 提取与边关联的节点度数

    di = [d[edge[0]] for edge in edges]

    dj = [d[edge[1]] for edge in edges]

    # 计算同配系数

    di\_times\_dj\_sum = sum(di\_i \* dj\_i for di\_i, dj\_i in zip(di, dj))

    di\_plus\_dj\_sum = sum(di\_i + dj\_i for di\_i, dj\_i in zip(di, dj))

    di\_squared\_plus\_dj\_squared\_sum = sum(di\_i\*\*2 + dj\_i\*\*2 for di\_i, dj\_i in zip(di, dj))

    r = (di\_times\_dj\_sum / K - (di\_plus\_dj\_sum / (2 \* K))\*\*2) / (di\_squared\_plus\_dj\_squared\_sum / (2 \* K) - (di\_plus\_dj\_sum / (2 \* K))\*\*2)

    return r

# 计算网络的余平均度

yu\_average = compute\_yu\_average(G)

# 打印结果

for node, yu\_aver in yu\_average.items():

    print(f"Node {node}: Yu Average = {yu\_aver}")

# 计算网络的标准化同配系数

print("Normalized Assortativity Coefficient:", normalized\_assortativity(G))

nx.degree\_assortativity\_coefficient(G)

## 运行结果

Node 1: Yu Average = 4.3125

Node 2: Yu Average = 5.777777777777778

Node 3: Yu Average = 6.6

Node 4: Yu Average = 7.666666666666667

Node 5: Yu Average = 7.666666666666667

Node 6: Yu Average = 6.25

Node 7: Yu Average = 6.25

Node 8: Yu Average = 10.25

Node 9: Yu Average = 11.8

Node 11: Yu Average = 7.666666666666667

Node 12: Yu Average = 16.0

Node 13: Yu Average = 11.0

Node 14: Yu Average = 11.6

Node 18: Yu Average = 12.5

Node 20: Yu Average = 14.0

Node 22: Yu Average = 12.5

Node 32: Yu Average = 9.0

Node 31: Yu Average = 10.75

Node 10: Yu Average = 13.5

Node 28: Yu Average = 8.75

Node 29: Yu Average = 11.0

Node 33: Yu Average = 5.083333333333333

Node 17: Yu Average = 4.0

Node 34: Yu Average = 3.823529411764706

Node 15: Yu Average = 14.5

Node 16: Yu Average = 14.5

Node 19: Yu Average = 14.5

Node 21: Yu Average = 14.5

Node 23: Yu Average = 14.5

Node 24: Yu Average = 8.0

Node 26: Yu Average = 4.666666666666667

Node 30: Yu Average = 9.0

Node 25: Yu Average = 4.333333333333333

Node 27: Yu Average = 10.5

Normalized Assortativity Coefficient: -0.47561309768461435

# 第六题 利用python中的networkx绘制精美网络图(见：<https://blog.csdn.net/qq951127336/article/details/54586869>)

## 代码

# 绘制图形

pos = nx.spring\_layout(G, seed=14)  # 使用弹簧布局

# 绘制节点

nx.draw\_networkx\_nodes(G, pos, node\_size=500, node\_color="orange", alpha=0.7, edgecolors="black")

# 绘制边

nx.draw\_networkx\_edges(G, pos, edge\_color='gray', alpha=0.6)

# 绘制节点标签

nx.draw\_networkx\_labels(G, pos, font\_size=16, font\_family='sans-serif', font\_weight='bold')

# 关闭坐标轴

plt.axis('off')

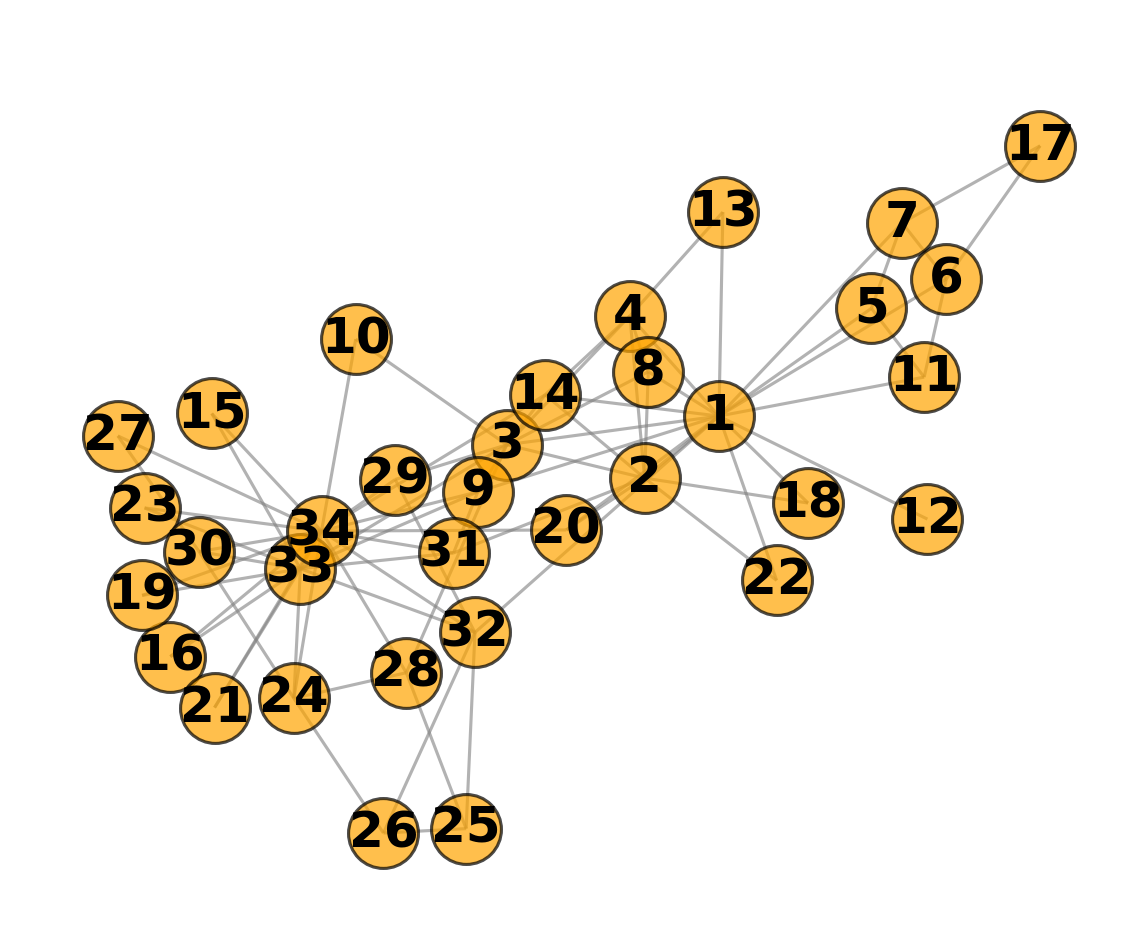
# 设置背景颜色

plt.gca().set\_facecolor("white")

# 显示图形

plt.show()

## 运行结果



# 第七题 绘制富人俱乐部系数vs节点度的关系图，如下图所示

Chart, line chart

Description automatically generated

## 代码

import random

import numpy as np

import networkx as nx

import matplotlib.pyplot as plt

"""

Create a random network with the same number of nodes as G1 and

a given probability p for creating edges.

:param G1: A networkx graph

:param p: The probability of creating an edge between two nodes

:return: A random networkx graph

"""

def random\_network(G1, p):

    # 创建一个空的网络

    G = nx.Graph()

    # 添加节点

    G.add\_nodes\_from(range(1, G1.number\_of\_nodes() + 1))

    # 添加边

    for i in range(1, G1.number\_of\_nodes()):

        for j in range(i + 1, G1.number\_of\_nodes() + 1):

            if random.random() < p:

                G.add\_edge(i, j)

                G.add\_edge(j, i)

    return G

def rich\_club\_coefficient(G):

"""

Calculate the rich club coefficient for each degree in the given graph.

:param G: A networkx graph

:return: A dictionary with degrees as keys and their rich club coefficients as values

"""

    # 获得网络中每个节点的度数，转化为字典

    d = dict(G.degree())

    rich\_club = {}

    for i in range(0, max(d.values())):

        # 提取网络中度数大于等于 i 的节点

        nodes = [node for node in d if d[node] > i]

        sub\_maj\_matrix = nx.to\_numpy\_matrix(G, nodelist=nodes)

        # 统计sub\_maj\_matrix中1的个数

        sum = np.sum(sub\_maj\_matrix)

        if len(nodes) == 1:

            rich\_club.update({i: 1})

        else:

            rich\_club.update({i: sum / (len(nodes) \* (len(nodes) - 1))})

    return rich\_club

def compare\_rich\_club\_coefficients(G, p):

"""

Compare the rich club coefficients of the given graph G and a random network

with the same number of nodes and probability p for creating edges. Plot the

results.

:param G: A networkx graph

:param p: The probability of creating an edge between two nodes in the random network

"""

    random\_G = random\_network(G, p)

    rich\_club\_G = rich\_club\_coefficient(G)

    rich\_club\_random\_G = rich\_club\_coefficient(random\_G)

    plt.plot(list(rich\_club\_G.keys()), list(rich\_club\_G.values()), 'b', label='Original Network')

    plt.plot(list(rich\_club\_random\_G.keys()), list(rich\_club\_random\_G.values()), 'r', label='Random Network')

    plt.xlabel('k')

    plt.ylabel('Rich Club Coefficient')

    plt.legend()

    plt.show()

compare\_rich\_club\_coefficients(G, p=0.5)

## 运行结果

Chart, line chart

Description automatically generated