# A02014087 张文滔 复杂网络第四次作业

# 第一题

* 生成LFR网络（可以直接用程序包生成）；

import networkx as nx

from networkx.generators.community import LFR\_benchmark\_graph

n = 250

tau1 = 3

tau2 = 1.5

mu = 0.1

G = LFR\_benchmark\_graph(

    n, tau1, tau2, mu, average\_degree=5, min\_community=20, seed=10

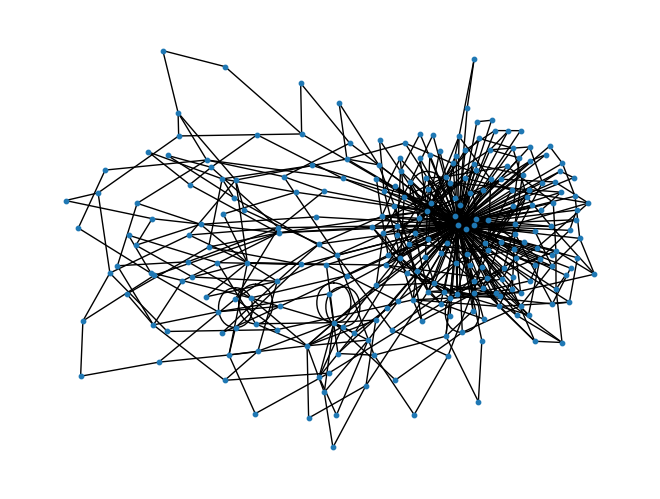
)

# 可视化网络

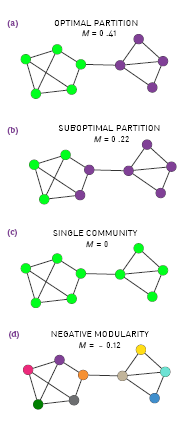
pos = nx.spring\_layout(G)

nx.draw(G, pos, node\_size=10)

## 运行结果：



# 第二题

* 编程求右上图中Q值；
* 

import networkx as nx

from networkx.algorithms import community

import numpy as np

# 计算网络的模块度

def modularity(G, communities):

    m = G.number\_of\_edges()

    Q = 0

    for comm in communities:

        comm = list(comm)

        commG = G.subgraph(comm)

        Q += len(list(commG.edges())) / m - (sum(dict(G.degree(comm)).values()) / (2 \* m)) \*\* 2

    print("Modularity:", Q)

G = nx.Graph()

# 创建一个简单的网络图

G.add\_edges\_from([(0, 1), (0, 7), (0, 8), (1, 2), (1, 8), (2, 7), (2, 3), (3, 4), (3, 5), (3, 6), (4, 5), (5, 6), (7, 8)])

# 计算网络的社区划分

# communities = community.greedy\_modularity\_communities(G)

communitiesA =[{0, 1, 2, 7, 8}, {3, 4, 5, 6}]

communitiesB =[{0, 1, 8}, {2, 3, 4, 5, 6, 7}]

communitiesC =[{0, 1, 8, 2, 3, 4, 5, 6, 7}]

communitiesD =[{0}, {1}, {8}, {2}, {3}, {4}, {5}, {6}, {7}]

modularity(G, communitiesA)

print(f'NetworkX result:{community.modularity(G, communitiesA)}')

modularity(G, communitiesB)

print(f'NetworkX result:{community.modularity(G, communitiesB)}')

modularity(G, communitiesC)

print(f'NetworkX result:{community.modularity(G, communitiesC)}')

modularity(G, communitiesD)

print(f'NetworkX result:{community.modularity(G, communitiesD)}')

## 运行结果：

Modularity: 0.41124260355029596

NetworkX result:0.41124260355029585

Modularity: 0.22189349112426032

NetworkX result:0.22189349112426038

Modularity: 0.0

NetworkX result:0.0

Modularity: -0.11538461538461539

NetworkX result:-0.11538461538461539

# 第三题

若某个算法得到分类为：A = [1 1 1 1, 2 2 2 2 2 , 3 3 3 3 3, 4 4 4]，而真实的分类结果为 B = [1 2 4 1, 1 2 3 2 2, 1 2 4 3 3, 4 4 2]，求NMI值；

import numpy as np

import networkx as nx

from networkx.algorithms import community

import community as community\_louvain

from sklearn.metrics import normalized\_mutual\_info\_score

import matplotlib.pyplot as plt

def compute\_NMI(A, B):  # 定义计算归一化互信息的函数，输入两个分类结果列表A和B

    assert len(A) == len(B), "The length of both clusterings should be equal."  # 确保A和B的长度相等

    cluster\_labels\_A = np.unique(A)  # 获取分类结果A中的唯一标签

    cluster\_labels\_B = np.unique(B)  # 获取分类结果B中的唯一标签

    # Compute joint probability matrix

    joint\_matrix = np.zeros((len(cluster\_labels\_A), len(cluster\_labels\_B)))  # 初始化联合概率矩阵

    for a, b in zip(A, B):  # 遍历分类结果A和B中的每个样本

        joint\_matrix[np.where(cluster\_labels\_A == a), np.where(cluster\_labels\_B == b)] += 1  # 更新联合概率矩阵中对应标签的计数

    joint\_matrix /= len(A)  # 将联合概率矩阵中的计数除以样本总数，得到联合概率分布

    # Compute marginal probabilities

    p\_A = joint\_matrix.sum(axis=1)  # 计算分类结果A中每个标签的边缘概率

    p\_B = joint\_matrix.sum(axis=0)  # 计算分类结果B中每个标签的边缘概率

    # Compute MI

    MI = np.sum([joint\_matrix[i, j] \* np.log(joint\_matrix[i, j] / (p\_A[i] \* p\_B[j])) for i in range(len(cluster\_labels\_A)) for j in range(len(cluster\_labels\_B)) if joint\_matrix[i, j] > 0])  # 计算互信息

    # Compute entropies

    H\_A = -np.sum(p\_A[p\_A > 0] \* np.log(p\_A[p\_A > 0]))  # 计算分类结果A的熵

    H\_B = -np.sum(p\_B[p\_B > 0] \* np.log(p\_B[p\_B > 0]))  # 计算分类结果B的熵

    # Compute NMI

    NMI = MI / np.sqrt(H\_A \* H\_B)  # 计算归一化互信息

    return NMI  # 返回归一化互信息的值

A = [1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4]  # 初始化聚类结果A

B = [1, 2, 4, 1, 1, 2, 3, 2, 2, 1, 2, 4, 3, 3, 4, 4, 2]  # 初始化聚类结果B

result = compute\_NMI(A, B)  # 调用函数计算A和B的归一化互信息

print(result)  # 输出A和B的归一化互信息

print(f"NetworkX: {normalized\_mutual\_info\_score(A, B)}")  # 输出A和B的归一化互信息

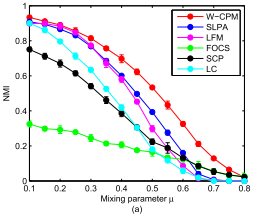
## 运行结果：

0.2398445188528549

NetworkX: 0.23984224652884514

# 第四题

选择几个社团划分的算法，如GN算法、 Louvain算法、FN算法、标签传播算法对不同参数u的LFR网络进行社团划分，并计算NMI值，结果展示类似右下图所示（可以调用工具包）；



import matplotlib.pyplot as plt

import networkx as nx

from networkx.algorithms import community

import numpy as np

from sklearn.metrics import normalized\_mutual\_info\_score

def generate\_lfr\_network(initial\_mu):

    """

    使用networkx生成LFR网络。如果在最大尝试次数内未能成功生成，则逐步增加mu值尝试生成。

    """

    n = 250  # 网络中的节点数量

    tau1 = 3  # 度分布的幂律指数

    tau2 = 1.5  # 社区大小分布的幂律指数

    max\_attempts = 10  # 最大尝试次数

    mu = initial\_mu  # 混合参数，表示社区内部和外部边的比例

    for attempt in range(max\_attempts):

        print(f"尝试生成LFR网络，当前mu值为{mu}...")

        try:

            G = nx.LFR\_benchmark\_graph(n, tau1, tau2, mu, average\_degree=5, min\_community=20, seed=10, max\_iters=100)

            return G  # 如果生成成功，则返回生成的网络

        except Exception as e:

            print(f"生成LFR网络失败，尝试次数：{attempt+1}")

    raise RuntimeError(f"在{max\_attempts}次尝试后，仍无法生成LFR网络，初始mu值为{initial\_mu}。")

def perform\_community\_detection(G, method):

    """

    根据指定的方法进行社区检测，并返回节点到社区的映射。

    """

    if method == 'GN':

        communities\_generator = community.girvan\_newman(G)

        top\_level\_communities = next(communities\_generator)

        communities = {node: i for i, community in enumerate(top\_level\_communities) for node in community}

    elif method == 'Louvain':

        import community as louvain

        communities = louvain.best\_partition(G)

    elif method == 'FN':

        communities\_generator = community.greedy\_modularity\_communities(G)

        communities = {node: i for i, community in enumerate(communities\_generator) for node in community}

    elif method == 'LabelPropagation':

        communities\_generator = list(community.asyn\_lpa\_communities(G))

        communities = {node: i for i, community in enumerate(communities\_generator) for node in community}

    else:

        raise ValueError('未知的社区检测方法')

    return communities

def calculate\_nmi(G, communities):

    """

    计算真实社区标签和预测社区标签之间的归一化互信息(NMI)。

    """

    true\_communities = nx.get\_node\_attributes(G, 'community')  # 获取真实的社区标签

    nmi = normalized\_mutual\_info\_score(list(true\_communities.values()), list(communities.values()))  # 计算NMI

    return nmi

def plot\_results(mu\_values, nmi\_values, methods):

    """

    对于每种社区检测方法，绘制其NMI值随mu值变化的折线图。

    """

    for i, method in enumerate(methods):

        plt.plot(mu\_values, nmi\_values[i], label=method, marker='o')

    plt.xlabel('mu')

    plt.ylabel('NMI')

    plt.legend()

    plt.show()

mu\_values = [0.1, 0.2, 0.3, 0.4, 0.5]

methods = ['GN', 'Louvain', 'FN', 'LabelPropagation']

nmi\_values = []

n\_repeats = 10  # 进行10次重复

for method in methods:

    method\_nmi\_values = []

    for mu in mu\_values:

        repeat\_nmi\_values = []  # 用于存储重复操作的NMI值

        for \_ in range(n\_repeats):

            G = generate\_lfr\_network(mu)

            print(f"已经生成LFR网络，μ = {mu}, method = {method}")

            communities = perform\_community\_detection(G, method)

            nmi = calculate\_nmi(G, communities)

            repeat\_nmi\_values.append(nmi)

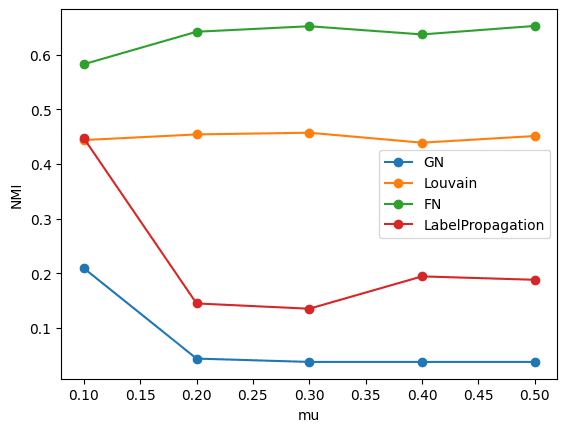
        average\_nmi = np.mean(repeat\_nmi\_values)  # 计算10次的平均NMI值

        method\_nmi\_values.append(average\_nmi)

    nmi\_values.append(method\_nmi\_values)

plot\_results(mu\_values, nmi\_values, methods)

## 运行结果



# 第五题

在Karate、Dolphins、Email三个真实网络上利用不同社团划分的算法，如GN算法、 Louvain算法、FN算法、标签传播算法，计算不同算法对应的模块度Q值（可以调用工具包）。

import networkx as nx

import community as community\_louvain

import igraph as ig

import numpy as np

from networkx.algorithms import community

# 定义函数 load\_network，从文件中读取网络数据

def load\_network(file\_path):

    G = nx.read\_edgelist(file\_path, nodetype=int, data=(('weight', int),))

    return G

# 定义函数 newman\_method，实现基于Newman的快速聚类方法

def newman\_method(A):

    e = A.copy()

    e[e == 1] = 1 / A.sum()   # 计算转移概率

    a = e.sum(axis=0)         # 计算每一列的总和

    n = A.shape[0]            # 获取邻接矩阵的大小

    b = [[i] for i in range(n)]  # 初始化b为嵌套列表，储存每个节点所在的社区

    k = 0                       # 初始化社区编号

    Q = []                      # 初始化模块度列表

    # 循环条件：邻接矩阵大小大于1

    while e.shape[0] > 1:

        # 初始化detaQ矩阵

        lg = e.shape[0]

        detaQ = -np.inf \* np.ones((n-k, n-k))

        # 计算detaQ矩阵

        for i in range(lg-1):

            for j in range(i+1, lg):

                if e[i, j] != 0:

                    detaQ[i, j] = 2 \* (e[i, j] - a[i] \* a[j])

        # 当所有的detaQ都为负无穷时，终止循环

        if np.all(detaQ == -np.inf):

            break

        # 找到最大的detaQ的坐标

        I, J = np.where(detaQ == detaQ.max())

        # 合并社区并更新邻接矩阵e

        for ii in range(len(I)):

            e[J[ii], :] = e[I[ii], :] + e[J[ii], :]

            e[I[ii], :] = 0

            e[:, J[ii]] = e[:, I[ii]] + e[:, J[ii]]

            e[:, I[ii]] = 0

            b[J[ii]] = b[I[ii]] + b[J[ii]]

            b[I[ii]] = []

        # 删除合并后的多余社区，并更新矩阵e和社区列表b

        e = np.delete(e, I, axis=0)

        e = np.delete(e, I, axis=1)

        b = [b[i] for i in range(len(b)) if b[i] != []]  # 删除空列表

        a = e.sum(axis=0)

        k += 1

        # 计算并保存当前模块度Q

        tmp = 0

        for jj in range(len(e)):

            tmp += e[jj, jj] - a[jj] \* a[jj]

        Q.append(tmp)

    # 返回最大模块度Q和对应的社区划分b

    return Q, b

# 定义函数 compute\_communities，计算并输出四种方法的平均模块度

def compute\_communities(G, iterations=10):

    gn\_modularities = []  # Girvan-Newman方法的模块度

    lp\_modularities = []  # 标签传播法的模块度

    lv\_modularities = []  # Louvain方法的模块度

    fn\_modularities = []  # Fastgreedy Newman方法的模块度

    # 对每种方法进行iterations次迭代

    for \_ in range(iterations):

        # Girvan-Newman

        gn\_comm = community.girvan\_newman(G)

        top\_level\_communities = next(gn\_comm)

        gn\_partition = {node: cid for cid, community in enumerate(top\_level\_communities) for node in community}

        gn\_modularity = community\_louvain.modularity(gn\_partition, G)

        gn\_modularities.append(gn\_modularity)

        # Label Propagation

        lp\_comm = community.asyn\_lpa\_communities(G, seed=42)

        lp\_partition = {node: cid for cid, community in enumerate(lp\_comm) for node in community}

        lp\_modularity = community\_louvain.modularity(lp\_partition, G)

        lp\_modularities.append(lp\_modularity)

        # Louvain

        lv\_partition = community\_louvain.best\_partition(G)

        lv\_modularity = community\_louvain.modularity(lv\_partition, G)

        lv\_modularities.append(lv\_modularity)

        # Fastgreedy Newman

        adjacency\_matrix = nx.to\_numpy\_array(G)

        Q, \_ = newman\_method(adjacency\_matrix)

        fn\_modularity = max(Q)

        fn\_modularities.append(fn\_modularity)

    # 输出四种方法的平均模块度

    print('Average GN Modularity:', sum(gn\_modularities)/iterations)

    print('Average LP Modularity:', sum(lp\_modularities)/iterations)

    print('Average LV Modularity:', sum(lv\_modularities)/iterations)

    print('Average FN Modularity:', sum(fn\_modularities)/iterations)

# 加载网络并计算社区

file\_path = 'karate.txt'

G = load\_network(file\_path)

compute\_communities(G)

file\_path = 'dolphins.txt'

G = load\_network(file\_path)

compute\_communities(G)

file\_path = 'email.txt'

G = load\_network(file\_path)

compute\_communities(G)

## 运行结果

运行10次平均

Karate

Average GN Modularity: 0.35996055226824464

Average LP Modularity: 0.3712195923734385

Average LV Modularity: 0.41978961209730437

Average FN Modularity: 0.3806706114398422

Dolphins

Average GN Modularity: 0.3787033740753926

Average LP Modularity: 0.4923658083145445

Average LV Modularity: 0.5241090146750524

Average FN Modularity: 0.4954906847039279

Email

Average GN Modularity: 0.0007332045227673073

Average LP Modularity: 0.4129712381292199

Average LV Modularity: 0.5739126766404156

Average FN Modularity: 0.5091428948170562