# A02014087 张文滔 复杂网络平时作业三

我和同一组的A02014002陈子雄同学合作写的作业三，我写了

第一题的给定几个不同的真实网络，分别计算每个节点的度中心性、接近中心性、介数中心性、K-壳分解、特征向量中心性。

第三题选取Email网络，比较度中心性方法（从度最高的节点中选择k个节点）和度折扣方法在ICM上的效果。参数设置，IC模型中的传播率设为p=0.1, 分别比较节点种子数k=5,10,15,20,25的情况，类似下图所示。

σ是感染比例，每次情况做十次，一千次平均。

A picture containing line, text, diagram, plot

Description automatically generated

## 度中心性

### 代码：

def degree\_centrality(G):

    degree = [d[1] for d in G.degree()]  # 获取图中每个节点的度（连接的边数）

    # 绘制每个节点的度中心化分布的柱状图

    plt.bar(G.nodes(), np.divide(degree, (len(G.nodes()) - 1)))

    plt.xlabel('Degree')  # 设置x轴标签为“Degree”

    plt.ylabel('Count')  # 设置y轴标签为“Count”

    plt.title('Degree Centrality')  # 设置图表标题为“Degree Centrality”

    plt.show()  # 显示图表

    # 返回每个节点的度中心化分布值

    return np.divide(degree, (len(G.nodes()) - 1))

### 运行结果：

Karate网络：

A picture containing text, screenshot, diagram, plot

Description automatically generated

[0.48484848 0.27272727 0.3030303 0.18181818 0.09090909 0.12121212

0.12121212 0.12121212 0.15151515 0.09090909 0.03030303 0.06060606

0.15151515 0.06060606 0.09090909 0.06060606 0.18181818 0.12121212

0.06060606 0.12121212 0.09090909 0.36363636 0.06060606 0.51515152

0.06060606 0.06060606 0.06060606 0.06060606 0.06060606 0.15151515

0.09090909 0.12121212 0.09090909 0.06060606]

Dolphins网络：

A picture containing text, screenshot, plot, line

Description automatically generated

[0.09836066 0.04918033 0.1147541 0.06557377 0.09836066 0.08196721

0.09836066 0.06557377 0.13114754 0.19672131 0.1147541 0.09836066

0.14754098 0.13114754 0.1147541 0.06557377 0.08196721 0.14754098

0.09836066 0.01639344 0.09836066 0.04918033 0.04918033 0.08196721

0.08196721 0.14754098 0.08196721 0.01639344 0.04918033 0.16393443

0.01639344 0.08196721 0.01639344 0.1147541 0.04918033 0.18032787

0.13114754 0.03278689 0.13114754 0.08196721 0.09836066 0.1147541

0.06557377 0.18032787 0.03278689 0.09836066 0.03278689 0.1147541

0.16393443 0.01639344 0.01639344 0.06557377 0.03278689 0.1147541

0.03278689 0.03278689 0.14754098 0.01639344 0.01639344 0.08196721

0.01639344 0.04918033]

## 接近中心性

### 代码：

def Closeness\_Centrality(G):

    # 初始化一个零矩阵，用于存储图中所有节点间的最短路径长度

    shoretest\_path\_matrix = np.zeros((len(G.nodes()), len(G.nodes())))

    # 创建一个字典，将图中的节点映射到它们在矩阵中的索引

    dic = dict(zip(G.nodes(), range(len(G.nodes()))))

    # 遍历图中的所有节点，计算它们之间的最短路径长度

    for start in G.nodes:

        for end in G.nodes:

            if start != end:

                # 如果当前节点对的最短路径尚未计算

                if shoretest\_path\_matrix[dic.get(start), dic.get(end)] == 0:

                    # 使用广度优先搜索算法计算最短路径长度，并将其存储在矩阵中

                    shoretest\_path\_matrix[dic.get(start), dic.get(end)] = breadth\_first\_search(G, start, end)

                    continue

    # 打印最短路径矩阵

    print(shoretest\_path\_matrix)

    # 计算并返回接近中心性

    return (len(G.nodes()) - 1) / shoretest\_path\_matrix.sum(axis = 1)

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]

### 运行结果：

Karate网络：

[0.56896552 0.48529412 0.55932203 0.46478873 0.37931034 0.38372093

0.38372093 0.44 0.515625 0.37931034 0.36666667 0.37078652

0.515625 0.375 0.5 0.375 0.54098361 0.45833333

0.43421053 0.45833333 0.45205479 0.515625 0.28448276 0.55

0.37078652 0.37078652 0.37078652 0.37078652 0.37078652 0.39285714

0.375 0.38372093 0.375 0.36263736]

{1: 0.5689655172413793, 2: 0.4852941176470588, 3: 0.559322033898305, 4: 0.4647887323943662, 5: 0.3793103448275862, 6: 0.38372093023255816, 7: 0.38372093023255816, 8: 0.44, 9: 0.515625, 11: 0.3793103448275862, 12: 0.36666666666666664, 13: 0.3707865168539326, 14: 0.515625, 18: 0.375, 20: 0.5, 22: 0.375, 32: 0.5409836065573771, 31: 0.4583333333333333, 10: 0.4342105263157895, 28: 0.4583333333333333, 29: 0.4520547945205479, 33: 0.515625, 17: 0.28448275862068967, 34: 0.55, 15: 0.3707865168539326, 16: 0.3707865168539326, 19: 0.3707865168539326, 21: 0.3707865168539326, 23: 0.3707865168539326, 24: 0.39285714285714285, 26: 0.375, 30: 0.38372093023255816, 25: 0.375, 27: 0.3626373626373626}

Dolphins网络：

[0.36309524 0.30808081 0.25206612 0.23828125 0.26754386 0.31282051

0.34659091 0.28240741 0.27111111 0.37654321 0.33888889 0.32972973

0.30964467 0.37195122 0.33701657 0.31606218 0.36526946 0.39102564

0.33333333 0.23735409 0.31282051 0.25523013 0.27727273 0.31606218

0.36526946 0.32275132 0.32275132 0.23735409 0.21631206 0.36526946

0.26872247 0.31606218 0.24497992 0.41780822 0.33333333 0.39869281

0.32972973 0.33516484 0.40397351 0.29756098 0.32972973 0.33701657

0.31122449 0.34659091 0.25630252 0.33888889 0.24696356 0.35057471

0.32972973 0.24897959 0.24897959 0.34269663 0.25957447 0.32446809

0.27111111 0.21328671 0.3019802 0.23282443 0.24897959 0.34269663

0.17836257 0.3019802 ]

{8: 0.3630952380952381, 3: 0.30808080808080807, 9: 0.25206611570247933, 5: 0.23828125, 6: 0.2675438596491228, 10: 0.3128205128205128, 0: 0.3465909090909091, 2: 0.2824074074074074, 13: 0.27111111111111114, 14: 0.3765432098765432, 15: 0.3388888888888889, 16: 0.32972972972972975, 17: 0.3096446700507614, 1: 0.3719512195121951, 18: 0.3370165745856354, 19: 0.3160621761658031, 7: 0.3652694610778443, 20: 0.391025641025641, 21: 0.3333333333333333, 22: 0.23735408560311283, 24: 0.3128205128205128, 25: 0.25523012552301255, 26: 0.2772727272727273, 27: 0.3160621761658031, 28: 0.3652694610778443, 29: 0.32275132275132273, 30: 0.32275132275132273, 31: 0.23735408560311283, 32: 0.21631205673758866, 33: 0.3652694610778443, 12: 0.2687224669603524, 34: 0.3160621761658031, 35: 0.24497991967871485, 36: 0.4178082191780822, 23: 0.3333333333333333, 37: 0.39869281045751637, 38: 0.32972972972972975, 39: 0.33516483516483514, 40: 0.40397350993377484, 41: 0.2975609756097561, 42: 0.32972972972972975, 43: 0.3370165745856354, 44: 0.3112244897959184, 45: 0.3465909090909091, 46: 0.25630252100840334, 47: 0.3388888888888889, 49: 0.24696356275303644, 50: 0.3505747126436782, 51: 0.32972972972972975, 4: 0.24897959183673468, 11: 0.24897959183673468, 52: 0.34269662921348315, 53: 0.25957446808510637, 54: 0.324468085106383, 55: 0.27111111111111114, 56: 0.21328671328671328, 57: 0.30198019801980197, 48: 0.23282442748091603, 58: 0.24897959183673468, 59: 0.34269662921348315, 60: 0.1783625730994152, 61: 0.30198019801980197}

## 介数中心性

### 代码：

def bfs\_shortest\_paths(G, source):

    visited = {source: 0}  # 初始化已访问节点字典，将起始节点的访问深度设为0

    queue = [source]  # 初始化队列，将起始节点加入队列

    paths = {node: [] for node in G.nodes}  # 初始化路径字典，用于存储从起始节点到每个节点的最短路径

    paths[source] = [[source]]  # 将起始节点的路径设为包含自身的列表

    # 当队列非空时，继续执行循环

    while queue:

        current = queue.pop(0)  # 从队列中取出第一个节点，并将其从队列中移除

        neighbors = G[current]  # 获取当前节点的邻居节点

        # 遍历邻居节点

        for neighbor in neighbors:

            # 如果邻居节点没有被访问过

            if neighbor not in visited:

                visited[neighbor] = visited[current] + 1  # 将邻居节点的访问深度设为当前节点访问深度加1

                queue.append(neighbor)  # 将邻居节点加入队列

            # 如果邻居节点的访问深度等于当前节点访问深度加1

            if visited[neighbor] == visited[current] + 1:

                # 更新从起始节点到邻居节点的最短路径

                paths[neighbor] += [path + [neighbor] for path in paths[current]]

    return paths  # 返回从起始节点到每个节点的最短路径字典

def betweenness\_centrality(G):

    shoretest\_path\_matrix = shortest\_path\_number(G)  # 计算网络图中所有节点对之间的最短路径数量

    # np.savetxt("homework3.3.txt", shoretest\_path\_matrix, fmt='%d')  # 将最短路径数量矩阵以整数形式写入文件

    dic = dict(zip(G.nodes(), range(len(G.nodes()))))  # 创建一个将节点映射到其索引的字典

    # 初始化一个用于存储分子（通过特定节点的最短路径数量）的三维数组

    Numerator = [np.zeros((len(G.nodes()), len(G.nodes()))) for \_ in range(len(G.nodes()))]

    # 遍历网络图中的每个节点

    for node in G.nodes:

        # 遍历除当前节点以外的所有起始节点

        for start in [n for n in G.nodes if n != node]:

            # 遍历除当前节点以外的所有结束节点

            for end in [n for n in G.nodes if n != node]:

                if start != end:

                    # 计算从起始节点到结束节点的最短路径中通过当前节点的路径数量

                    shortest\_paths = bfs\_shortest\_paths(G, start)

                    count = sum(node in path for path in shortest\_paths[end])

                    Numerator[dic.get(node)][dic.get(start)][dic.get(end)] += count

    # 将 shoretest\_path\_matrix 转换为 NumPy 数组

    shoretest\_path\_matrix\_np = np.array(shoretest\_path\_matrix)

    Numerator\_np = np.array(Numerator)

    re = []  # 初始化用于存储结果的列表

    mask = shoretest\_path\_matrix\_np != 0  # 创建一个掩码，标识 shoretest\_path\_matrix\_np 中非零元素的位置

    for i in Numerator\_np:

        # 初始化一个与 i 相同形状的全零矩阵

        result = np.zeros\_like(i)

        # 只对 shoretest\_path\_matrix\_np 中非零元素对应的位置进行除法操作

        result[mask] = i[mask] / shoretest\_path\_matrix\_np[mask]

        re.append(np.sum(result))

    re = np.array(re)

    # 输出网络图的介数中心性（使用 NetworkX 计算）

    print("betweenness\_centrality NetworkX:", nx.betweenness\_centrality(G,normalized=True))

    # 输出结果

    result = dict(zip(G.nodes(), re/((len(G.nodes()) - 1) \* (len(G.nodes()) - 2))))

    return result

def shortest\_path\_number(G):

    n = G.number\_of\_nodes()  # 获取图 G 的节点数

    matrix = [np.zeros((n, n))]  # 初始化一个 n\*n 的零矩阵

    shortest\_path\_number = np.zeros((n, n))  # 初始化一个 n\*n 的零矩阵

    np.fill\_diagonal(shortest\_path\_number, np.inf)  # 将 shortest\_path\_number 的对角线上的值设置为正无穷

    adj\_matrix = np.array(nx.to\_numpy\_array(G))  # 将图 G 转换成邻接矩阵

    matrix = adj\_matrix  # 将邻接矩阵赋值给 matrix

    np.fill\_diagonal(shortest\_path\_number, np.inf)  # 将 shortest\_path\_number 的对角线上的值设置为正无穷

    for i in range(1, n):  # 进行 n-1 次迭代

        for p in range(n):  # 遍历所有节点对

            for q in range(n):

                if q != p:  # 排除节点到自身的情况

                    if matrix[p][q] != 0 and shortest\_path\_number[p][q] == 0:  # 如果节点 p 和节点 q 之间有边，并且它们之间的最短路径还没有被计算出来

                        shortest\_path\_number[p][q] = matrix[p][q]  # 将它们之间的距离作为它们之间的最短路径

        matrix = matrix @ adj\_matrix  # 计算 matrix 的下一次幂

    # 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\_number, 0)

    return shortest\_path\_number

### 运行结果：

Karate网络：

betweenness\_centrality NetworkX: {1: 0.4376352813852815, 2: 0.05393668831168831, 3: 0.14365680615680615, 4: 0.011909271284271283, 5: 0.0006313131313131313, 6: 0.02998737373737374, 7: 0.029987373737373736, 8: 0.0, 9: 0.05592682780182782, 11: 0.0006313131313131313, 12: 0.0, 13: 0.0, 14: 0.045863395863395856, 18: 0.0, 20: 0.03247504810004811, 22: 0.0, 32: 0.13827561327561327, 31: 0.014411976911976905, 10: 0.0008477633477633478, 28: 0.022333453583453587, 29: 0.0017947330447330447, 33: 0.14524711399711404, 17: 0.0, 34: 0.30407497594997596, 15: 0.0, 16: 0.0, 19: 0.0, 21: 0.0, 23: 0.0, 24: 0.017613636363636363, 26: 0.0038404882154882162, 30: 0.0029220779220779218, 25: 0.0022095959595959595, 27: 0.0}

own code：

{1: 0.4376352813852814, 2: 0.05393668831168831, 3: 0.14365680615680615, 4: 0.011909271284271284, 5: 0.0006313131313131313, 6: 0.029987373737373736, 7: 0.029987373737373736, 8: 0.0, 9: 0.05592682780182779, 11: 0.0006313131313131313, 12: 0.0, 13: 0.0, 14: 0.04586339586339586, 18: 0.0, 20: 0.032475048100048094, 22: 0.0, 32: 0.13827561327561325, 31: 0.014411976911976912, 10: 0.0008477633477633477, 28: 0.02233345358345358, 29: 0.0017947330447330447, 33: 0.14524711399711399, 17: 0.0, 34: 0.3040749759499759, 15: 0.0, 16: 0.0, 19: 0.0, 21: 0.0, 23: 0.0, 24: 0.017613636363636363, 26: 0.0038404882154882154, 30: 0.0029220779220779218, 25: 0.0022095959595959595, 27: 0.0}

Dolphins网络：

betweenness\_centrality NetworkX: {8: 0.022365737598409235, 3: 0.0023737965131407756, 9: 0.02089438036159347, 5: 0.004380300179480508, 6: 0.029372536747686685, 10: 0.016092020911693046, 0: 0.01908259621374376, 2: 0.00907281243346817, 13: 0.0528463284386915, 14: 0.06197200484885411, 15: 0.03329222098223321, 16: 0.0033047098620869104, 17: 0.11430016291546968, 1: 0.213324435532811, 18: 0.0148548997169549, 19: 0.013314394166853183, 7: 0.11823861926938342, 20: 0.10264573972090967, 21: 0.012700653930162124, 22: 0.0, 24: 0.0073830434896008665, 25: 0.0016441148408361526, 26: 0.004362477231329691, 27: 0.029236860493157976, 28: 0.06675695466395656, 29: 0.06552928249649563, 30: 0.03305046077177225, 31: 0.0, 32: 0.03278688524590164, 33: 0.057166440117259784, 12: 0.0, 34: 0.032694759702956426, 35: 0.0, 36: 0.24823719602893804, 23: 0.04218278563875618, 37: 0.13856978865859435, 38: 0.04535223883584539, 39: 0.070516778539934, 40: 0.14314951834261747, 41: 0.023251600670186168, 42: 0.029157951944837193, 43: 0.06283060442896508, 44: 0.012037805849281259, 45: 0.040670440596470174, 46: 0.003008466942893172, 47: 0.0232014761195089, 49: 0.0009289617486338798, 50: 0.03341103492742837, 51: 0.08467725475022554, 4: 0.0, 11: 0.0, 52: 0.0192343448900826, 53: 0.0011930783242258653, 54: 0.09912164676351938, 55: 0.0008774695250105086, 56: 0.0001366120218579235, 57: 0.08420468343495603, 48: 0.0, 58: 0.0, 59: 0.020332774690384588, 60: 0.0, 61: 0.014194982613015397}

{8: 0.022365737598409235, 3: 0.0023737965131407756, 9: 0.020894380361593478, 5: 0.004380300179480508, 6: 0.02937253674768669, 10: 0.01609202091169304, 0: 0.019082596213743756, 2: 0.009072812433468172, 13: 0.0528463284386915, 14: 0.0619720048488541, 15: 0.03329222098223321, 16: 0.0033047098620869113, 17: 0.11430016291546972, 1: 0.213324435532811, 18: 0.014854899716954894, 19: 0.013314394166853184, 7: 0.11823861926938344, 20: 0.10264573972090964, 21: 0.012700653930162127, 22: 0.0, 24: 0.007383043489600867, 25: 0.0016441148408361523, 26: 0.004362477231329691, 27: 0.029236860493157973, 28: 0.06675695466395656, 29: 0.0655292824964956, 30: 0.03305046077177225, 31: 0.0, 32: 0.03278688524590164, 33: 0.05716644011725979, 12: 0.0, 34: 0.032694759702956426, 35: 0.0, 36: 0.24823719602893812, 23: 0.04218278563875619, 37: 0.13856978865859432, 38: 0.04535223883584539, 39: 0.07051677853993399, 40: 0.14314951834261755, 41: 0.023251600670186178, 42: 0.02915795194483719, 43: 0.06283060442896508, 44: 0.012037805849281259, 45: 0.04067044059647018, 46: 0.0030084669428931724, 47: 0.0232014761195089, 49: 0.0009289617486338799, 50: 0.03341103492742836, 51: 0.08467725475022556, 4: 0.0, 11: 0.0, 52: 0.019234344890082595, 53: 0.0011930783242258653, 54: 0.09912164676351942, 55: 0.0008774695250105087, 56: 0.0001366120218579235, 57: 0.08420468343495603, 48: 0.0, 58: 0.0, 59: 0.020332774690384588, 60: 0.0, 61: 0.014194982613015397}

## K-壳分解

### 代码：

def k\_shell\_decomposition(graph):

    k\_shell = {}  # 创建一个空字典，用于存储每个节点的k-壳值

    k = 1  # 初始化k的值为1，用于表示当前计算的k-壳层级

    # 当图中仍然有节点时，继续执行循环

    while graph.nodes():

        nodes\_to\_remove = []  # 创建一个空列表，用于存储本轮迭代中需要移除的节点

        # 遍历图中的所有节点

        for node in graph.nodes():

            # 如果当前节点的度（连接的边数）小于等于k

            if graph.degree(node) <= k:

                nodes\_to\_remove.append(node)  # 将当前节点添加到需要移除的节点列表中

                k\_shell[node] = k  # 将当前节点的k-壳值设置为k

        # 如果本轮迭代没有需要移除的节点

        if not nodes\_to\_remove:

            k += 1  # 将k的值加1，计算下一个k-壳层级

        else:

            graph.remove\_nodes\_from(nodes\_to\_remove)  # 从图中移除本轮迭代中所有需要移除的节点

    return k\_shell  # 返回字典k\_shell，其中包含每个节点的k-壳值

### 运行结果：

Karate网络：

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

Dolphins网络：

{22: 1, 31: 1, 12: 1, 35: 1, 4: 1, 11: 1, 48: 1, 58: 1, 60: 1, 32: 2, 39: 2, 46: 2, 49: 2, 53: 2, 55: 2, 56: 2, 61: 2, 3: 3, 5: 3, 2: 3, 25: 3, 26: 3, 23: 3, 27: 3, 44: 3, 34: 3, 10: 4, 19: 4, 7: 4, 52: 4, 59: 4, 8: 4, 30: 4, 36: 4, 42: 4, 0: 4, 1: 4, 28: 4, 47: 4, 15: 4, 17: 4, 20: 4, 40: 4, 41: 4, 54: 4, 9: 4, 6: 4, 13: 4, 38: 4, 57: 4, 16: 4, 43: 4, 14: 4, 33: 4, 37: 4, 50: 4, 21: 4, 24: 4, 18: 4, 29: 4, 45: 4, 51: 4}

## 特征向量中心性

### 代码：

### 运行结果：

Karate网络：

自己代码的结果

[[0.35549162]

[0.26596009]

[0.3171926 ]

[0.21117995]

[0.07596896]

[0.0794832 ]

[0.0794832 ]

[0.17095998]

[0.22740397]

[0.07596896]

[0.0528558 ]

[0.08425477]

[0.22647294]

[0.0923997 ]

[0.14791266]

[0.0923997 ]

[0.19103388]

[0.1747583 ]

[0.10267427]

[0.13347712]

[0.13107779]

[0.3086438 ]

[0.02363567]

[0.3733629 ]

[0.10140323]

[0.10140323]

[0.10140323]

[0.10140323]

[0.10140323]

[0.15011845]

[0.05920637]

[0.1349607 ]

[0.05705236]

[0.07557936]]

NetworkX结果：

{1: 0.35548349418519426, 2: 0.26595387045450236, 3: 0.3171893899684447, 4: 0.21117407832057056, 5: 0.0759664588165738, 6: 0.07948057788594245, 7: 0.07948057788594245, 8: 0.1709551149803543, 9: 0.22740509147166046, 11: 0.0759664588165738, 12: 0.05285416945233646, 13: 0.08425192086558085, 14: 0.22646969838808148, 18: 0.0923967566684595, 20: 0.14791134007618667, 22: 0.0923967566684595, 32: 0.19103626979791702, 31: 0.17476027834493088, 10: 0.10267519030637758, 28: 0.13347932684333308, 29: 0.13107925627221215, 33: 0.3086510477336959, 17: 0.02363479426059687, 34: 0.37337121301323495, 15: 0.10140627846270833, 16: 0.10140627846270833, 19: 0.10140627846270833, 21: 0.10140627846270833, 23: 0.10140627846270833, 24: 0.15012328691726787, 26: 0.05920820250279008, 30: 0.13496528673866567, 25: 0.057053735638028055, 27: 0.07558192219009326}

Dolphins网络：

自己的代码：

[[0.14310214]

[0.07933473]

[0.01219912]

[0.00656029]

[0.01219284]

[0.07525347]

[0.12850353]

[0.03975712]

[0.01500666]

[0.31578271]

[0.16417574]

[0.20799418]

[0.01751416]

[0.04207731]

[0.20249457]

[0.02067537]

[0.04290152]

[0.18447743]

[0.20735127]

[0.00243473]

[0.19321344]

[0.00594653]

[0.00894436]

[0.01631792]

[0.06822377]

[0.21176293]

[0.04074751]

[0.00243473]

[0.00385658]

[0.28109856]

[0.03907612]

[0.13882803]

[0.02943762]

[0.13276236]

[0.08736213]

[0.30056197]

[0.19661756]

[0.02087142]

[0.20787197]

[0.01524498]

[0.08095015]

[0.19033924]

[0.07780235]

[0.28500505]

[0.02971649]

[0.08037073]

[0.02342974]

[0.21769169]

[0.21068206]

[0.02928737]

[0.02928737]

[0.12956424]

[0.03368691]

[0.02300356]

[0.05210979]

[0.00260699]

[0.0173781 ]

[0.00241583]

[0.02733223]

[0.11181863]

[0.00053613]

[0.05199139]]

NetworkX结果：

{8: 0.14310221675120963, 3: 0.07933447607445522, 9: 0.012220169764468436, 5: 0.006572752164517534, 6: 0.01221198028654735, 10: 0.07525346435103515, 0: 0.12850351911087216, 2: 0.03975713309801053, 13: 0.015030415487480693, 14: 0.3157810764804676, 15: 0.16417491138339668, 16: 0.20799316926734238, 17: 0.01753496833868235, 1: 0.042091441435724825, 18: 0.20249300124247624, 19: 0.020682545955984944, 7: 0.042908020056302676, 20: 0.18447787335323979, 21: 0.20734961722376985, 22: 0.002438237539825345, 24: 0.19321180983300906, 25: 0.005952326426510388, 26: 0.008949192287688042, 27: 0.01632668732073103, 28: 0.06822697202643725, 29: 0.21176109464474221, 30: 0.04075065098782638, 31: 0.002438237539825345, 32: 0.0038643437589662884, 33: 0.281097017160587, 12: 0.03907586116163528, 34: 0.1388272511035687, 35: 0.02943731400967758, 36: 0.13276550630017522, 23: 0.08736202260403907, 37: 0.3005609284704956, 38: 0.1966165389283655, 39: 0.020875970745552558, 40: 0.20787263130203712, 41: 0.015262103099078316, 42: 0.08095068341031326, 43: 0.1903379603455067, 44: 0.0778021296401005, 45: 0.28500310473240437, 46: 0.02971624484960024, 47: 0.08037195763313358, 49: 0.023429567002119712, 50: 0.21769051331081501, 51: 0.2106802089208981, 4: 0.029287057142628196, 11: 0.029287057142628196, 52: 0.12956359101537382, 53: 0.03368665459890723, 54: 0.023022376520473405, 55: 0.052109343930712856, 56: 0.002612314190108688, 57: 0.017401973681970627, 48: 0.002419847393008891, 58: 0.02733206208697856, 59: 0.11181865699755933, 60: 0.0005374384753918985, 61: 0.05199117201156755}

## 第三题：

### 代码：

import networkx as nx

import numpy as np

import matplotlib.pyplot as plt

import random

# 定义度折扣函数

def degree\_discount(G, k, p):

    n = len(G.nodes())  # n为节点数量

    degree = dict(G.degree())  # degree记录每个节点的度数

    dd = degree.copy()  # dd记录每个节点的度折扣值，初始值为节点度数

    t = {node: 0 for node in G.nodes()}  # t记录每个节点在已选节点中的邻居数

    selected\_nodes = []  # selected\_nodes记录已选节点集合

    for \_ in range(k):

        # 选择度折扣最大的节点

        u = max(dd, key=lambda node: dd[node] if node not in selected\_nodes else float('-inf'))

        selected\_nodes.append(u)  # 将选中的节点加入已选节点集合

        # 更新度折扣

        for v in G.neighbors(u):

            if v not in selected\_nodes:

                t[v] += 1

                dd[v] = degree[v] - 2 \* t[v] - (degree[v] - t[v]) \* t[v] \* p

    return selected\_nodes  # 返回已选节点集合

# 使用独立级联模型（ICM）模拟节点传播过程，返回所有被激活的节点集合

def icm(G, probabilities, seed\_nodes):

    dic = dict(zip(G.nodes(), range(len(G.nodes()))))  # 创建一个字典，将图中的节点映射到它们在矩阵中的索引

    newly\_activated = set(seed\_nodes)  # newly\_activated记录新增激活的节点集合，初始值为种子节点集合

    activated\_nodes = set(seed\_nodes)  # activated\_nodes记录所有已经被激活的节点集合

    while newly\_activated:

        next\_activated = set()

        for u in newly\_activated:

            for v in G.neighbors(u):

                if v not in activated\_nodes:

                    if random.random() < probabilities[dic.get(u)][dic.get(v)]:  # 根据概率模拟节点激活

                        next\_activated.add(v)

                        activated\_nodes.add(v)  # 将新激活的节点添加到已激活节点集合中

        newly\_activated = next\_activated

    return activated\_nodes  # 返回所有被激活的节点集合

# 计算度中心性种子节点

def degree\_centrality\_seeds(G, k):

    degree = [d[1] for d in G.degree()]

    degree = np.divide(degree , (len(G.nodes()) - 1))

    return np.argsort(degree)[-k:]

# 计算感染比例的平均值

def average\_infected\_ratio(G, probabilities, seeds, runs=1000):

    infected\_ratios = []

    for i in range(runs):

        activated\_nodes = icm(G, probabilities, seeds)

        infected\_ratios.append(len(activated\_nodes) / len(G.nodes()))

    return np.mean(infected\_ratios)

# 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),), create\_using=nx.Graph())

# 设置传播概率矩阵

probabilities = np.full(graph.shape, 0.1)

np.fill\_diagonal(probabilities, 0)

# 测试不同k值的情况

k\_values = [5, 10, 15, 20, 25]

degree\_centrality\_infected\_ratios = []

degree\_discount\_infected\_ratios = []

# 创建一个字典，将图中节点在矩阵中的索引映射到它们

dic = dict(zip(range(len(G.nodes())),G.nodes()))

for k in k\_values:

    degree\_seeds = degree\_centrality\_seeds(G, k)

    degree\_seeds = [dic.get(seed) for seed in degree\_seeds]

    degree\_infected\_ratio = average\_infected\_ratio(G, probabilities, degree\_seeds)

    degree\_centrality\_infected\_ratios.append(degree\_infected\_ratio)

    degree\_discount\_seeds = degree\_discount(G, k, 0.1)

    degree\_discount\_infected\_ratio = average\_infected\_ratio(G, probabilities, degree\_discount\_seeds)

    degree\_discount\_infected\_ratios.append(degree\_discount\_infected\_ratio)

# 创建一个新的图形

plt.figure()

# 在同一张图上绘制每个中心性度量

plt.plot(k\_values, degree\_centrality\_infected\_ratios, label="Degree Centrality")

plt.plot(k\_values, degree\_discount\_infected\_ratios, label="Degree Discount")

# 为图形添加标签和图例

plt.xlabel("k")

plt.xticks(k\_values)

plt.ylabel("Infected Ratio")

plt.title("ICM run 1000 times")

plt.legend()

# 显示图形

plt.show()

### 运行结果：

A picture containing text, diagram, line, screenshot

Description automatically generatedA picture containing text, screenshot, line, diagram

Description automatically generated