Data Mining - Handin 2 - Graph mining

This handin corresponds to the topics in Week 10-15 in the course.

The handin is

- done in the chosen handin groups
- worth 10% of the grade

For the handin, you will prepare a report in PDF format, by exporting the Jupyter notebook. Please submit

1. The jupyter notebook file with your answers
2. The PDF obtained by exporting the jupyter notebook

Submit both files on Brightspace no later than **April 21 kl. 11.59PM**.

The grading system: Tasks are assigned a number of points based on the difficulty and time to solve it. The sum of the number of points is **100**. For the maximum grade you need to get at least *80 points*. The minimum grade (02 in the Danish scale) requires **at least** 30 points, with at least 8 points on of the first three Parts (Part 1,2,3) and 6 points in the last part (Part 4). Good luck!

The exercise types: There are three different types of exercises

1. **[Compute by hand]** means that you should provide **NO code**, but show the main steps to reach the result (not all).
2. **[Motivate]** means to provide a short answer of 1-2 lines indicating the main reasoning, e.g., the PageRank of a complete graph is 1/n in all nodes as all nodes are symmetric and are connected one another.
3. **[Describe]** means to provide a potentially longer answer of 1-5 lines indicating the analysis of the data and the results.
4. **[Prove]** means to provide a formal argument and NO code.
5. **[Implement]** means to provide an implementation. Unless otherwise specified, you are allowed to use helper functions (e.g., `np.mean`, `itertools.combinations`, and so on). However, if the task is to implement an algorithm, by no means a call to a library that implements the same algorithm will be deemed as sufficient!

In [ ]:
```
### BEGIN IMPORTS - DO NOT TOUCH!
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
import sys
sys.path.append('..')
#!{sys.executable} -m pip install matplotlib
#!{sys.executable} -m pip install networkx
```

```
#!{sys.executable} -m pip install torchvision
import random
import scipy.io as sio
import time

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

import csv
from itertools import count


import torch
import torch.optim
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms


from utilities.load_data import load_mnist
import utilities.email as email
from utilities.mnist import *

from utilities.make_graphs import read_edge_list, read_list, load_data

### END IMPORTS - DO NOT TOUCH!
```

```
c:\Users\Joachim Brendborg\Documents\Universitet\8. Semester\Data Mining\dm2023-exercises\handins
```

Task 1.1 Random walks and PageRank (12 points)

In this exercise recall that the PageRank is defined as

$$\mathbf{r} = \alpha \mathbf{M} \mathbf{r} + (1 - \alpha)\mathbf{p}$$

where $\mathbf{r} \in \mathbb{R}^n$ is the PageRank vector, α is the restart probability, $\mathbf{M} = A\Delta^{-1}$, and $\mathbf{p}$ is the restart (or personalization) vector.

Task 1.1.1 (4 points)

What is the PageRank of a d-**regular** graph with n nodes and $\alpha = 1$?

[Motivate] your answer without showing the exact computation.

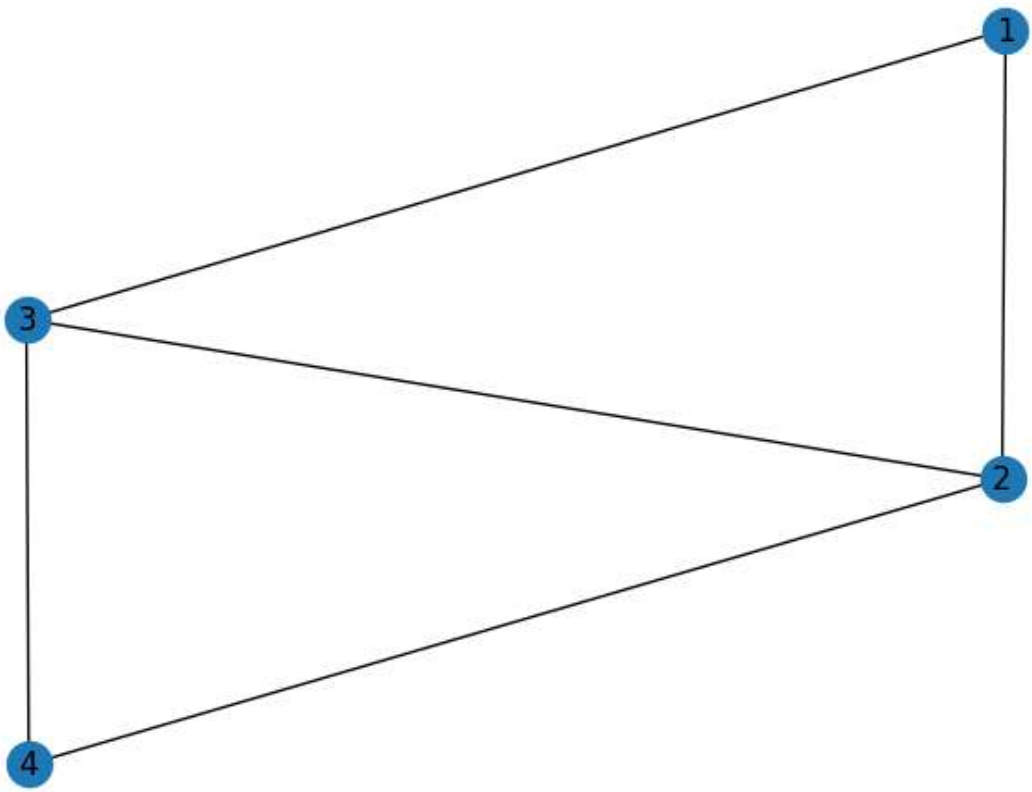
When $\alpha = 1$ we can ignore the restart vector completely.
In a d-regular graph all nodes will have the same degree, namely d. So between neighbours i and j, $m_{ij} = \frac{1}{d}$ and every node i has d neighbours, which cancels out. That means all nodes will have PageRank $\frac{1}{n}$.

Task 1.1.2 (6 points)

Look at the graph below (run the code) and try make a guess about the PageRank values of each node only by reasoning on the graph's connections.

Since the pagerank has to sum to 1 I would guess something like (0.2,0.3,0.3,0.2), since nodes 1 and 4 seems equally important as well as nodes 2 and 3 being equally important due to having the same degree.

In [ ]:
```
G = nx.Graph()
G.add_edges_from([(1,2),(2,3), (2,4), (3,4), (1,3)])
nx.draw(G, with_labels=True,)
```



A) [Implement] the PageRank for $\alpha = 1$ for the graph using the Power Iteration method (use $\epsilon = 1e - 16$ to stop the iteration).

B) [Implement] Plot the norm square difference of the r vector (between any two consecutive iterations) for each iteration.

C) [Motivate] Do you observe a constant decrease of the norm square difference as iterations are increasing, and is this decrease expected or not?

D) [Implement] the PageRank for $\alpha = 1$ using the eigenvector method.

E) [Motivate] Are solutions of both methods the same? Why don't we only use the eigenvector method that optimally solves the problem?

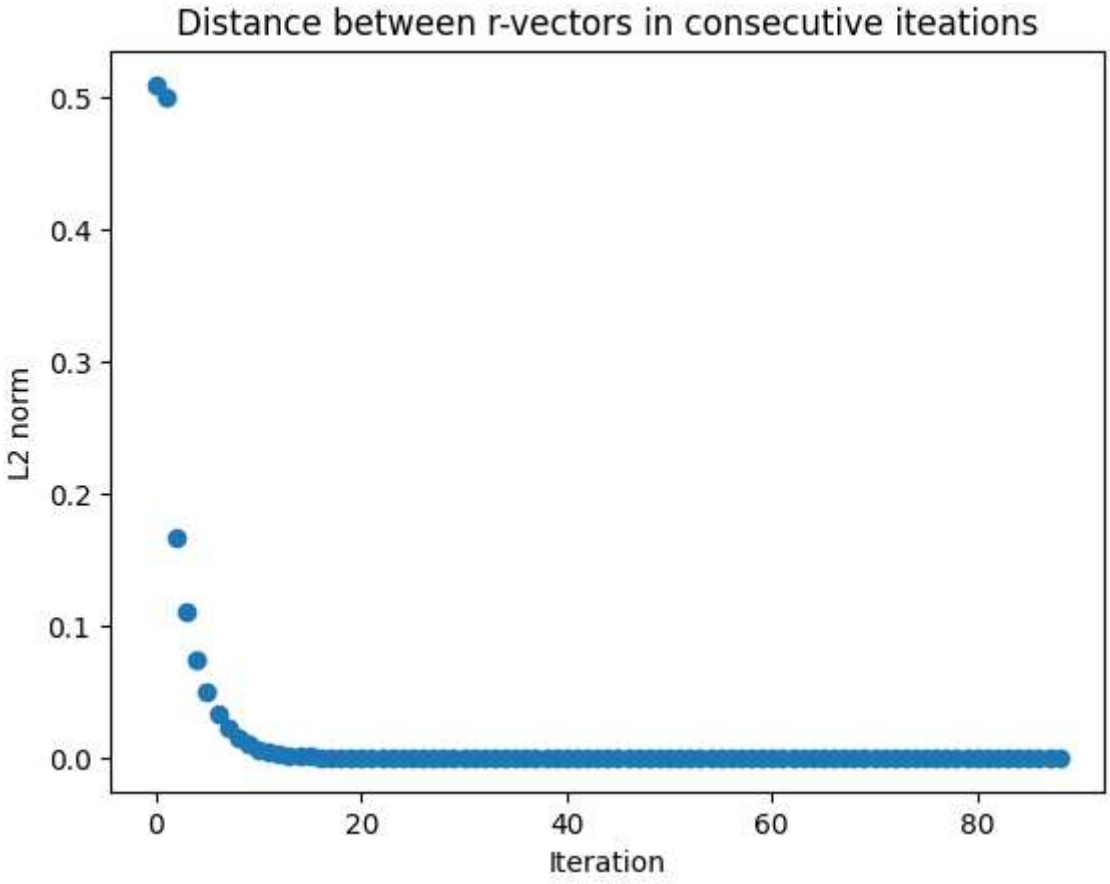
F) [Motivate] Do the real vector match with your first guess? Can you see a pattern between the pagerank score of each node and its edges?

In [ ]:
```
def L2_norm(x, y):
    return np.sqrt(np.sum((x-y)**2))
```

In [ ]:
```
#A) YOUR CODE HERE
def power_iteration(M,n,eps=10**-16,dist=L2_norm):
    prev_r = np.zeros(n)
    r = np.ones(n)/n
    rs = [prev_r,r]
    while (dist(r,prev_r)>=eps):
        prev_r = r
        r = M @ prev_r
        rs.append(r)
    return r,rs

degrees = [val for (node, val) in G.degree()]
M = nx.to_numpy_array(G).T/np.array(degrees)
r,rs =power_iteration(M,len(G.nodes))
print(r)
```

```
[0.2 0.3 0.3 0.2]
```

In [ ]:
```
#B) YOUR CODE HERE
dists = [L2_norm(rs[i],rs[i-1]) for i,x in enumerate(rs[1:])]
iteration = [x for x in range(len(dists))]
plt.scatter(iteration, dists)
plt.title("Distance between r-vectors in consecutive iteations")
plt.xlabel("Iteration")
plt.ylabel("L2 norm")
plt.show()
```



C)

For the power-iteration method we expect to see convergence of the r-vector.
This would also mean that the difference between r-vectors in consecutive iterations would become smaller and smaller until, hopefully, it becomes lower than ϵ

In [ ]:
```
#D) YOUR CODE HERE
degrees = [val for (node, val) in G.degree()]
M = nx.to_numpy_array(G).T/np.array(degrees)
w,v =  np.linalg.eig(M)
i = np.argmax(w)
vector = v[:,i]
norm = np.linalg.norm(vector,ord=1)
print(np.abs(vector/norm))
```

```
[0.2 0.3 0.3 0.2]
```

E)

Both solutions are the same. I guess we don't use the eigenvector method since for large graphs solving this system of linear equations would be very slow and a simulation based approach is better suited for the task.

F)

It does match what I thought it would be. And it makes sense, since in an undirected graph

there would be a direct correlation between the degree and pagerank of a node.

Task 1.1.3 (2 points)

[Motivate]

Assume you have embedded the graph in **1.1.2** with a **Linear Embedding** using unormalized Laplacian matrix of the graph as the similarity matrix. How do you expect the embeddings to be if the embedding dimension is $d = 1$? (1) Check the correct box below and (2) motivate your answer.

- ☐ Nodes 1, 2, 3, 4 will be placed in the corners of a hypercube
- ☐ Nodes 2,3 will have the same embedding while 1,4 will be far from each other.
- ☑ Nodes 1,4 and 2,3 will have very close embeddings.
- ☐ Nodes 3,4 will be very far apart.

IMPORTANT: Do NOT just choose one answer. Please clarify WHY this is the correct answer.

I would imagine nodes 1 and 4 to have the same embedding, since they are connected to the same nodes and have the same degree. Whereas nodes 2 and 3 have the same degree and share 2 neighbours, they are also connected to each other, but not to themselves, which would mean that their entries in the laplacian would differ slightly. I would think they should have a similar embedding, but probably not completely the same.

Task 1.2: Spectral Properties of the Graph Laplacian (17 points)

[Prove] the following properties: You will be given points for each of the properties that you prove, rather than points for the exercise as a whole.

Note that all question correspond to the eigenvalues of the LAPLACIAN (NOT THE NORMALIZED)

For a graph with n nodes the eigenvalues of the LAPLACIAN ($L = D - A$) is sorted in ascending order, i.e.,
$\lambda_0 \leq \lambda_1 \leq \ldots \leq \lambda_{n-1}$

Task 1.2.1 (1 points)

For all graphs $\lambda_0 = 0$

For the graph Laplacian $L = D - A$, we know that the vector of all ones ($v_0 = \mathbf{1}$) is an eigenvector with eigenvalue 0. This follows directly from the fact that $Lv_0 = L\mathbf{1} = 0$. Since every row sum and column sum of L is zero and furthermore, $\mathcal{L}$ is PSD, i.e., all eigenvalues are non-negative, 0 must be the smallest eigenvalue.

Task 1.2.2 (2 points)

For the complete graph, $\lambda_1, \ldots, \lambda_{n-1} = n$

The Laplacian matrix $L = D - A$ for a complete graph has $n - 1$ on the diagonal and -1 everywhere else. Intuitively this means that $Lv_0 = 0v_0$ as we saw above.
Consider any vector x which is orthogonal to the first eigenvector v_0.
Since they are orthogonal i holds that $x \cdot v_0 = 0$ and since v_0 is the vector of all ones then $\sum_{i=1}^{n} x_i = 0$.
Consider now $Lx = (n-1)x_i + \sum_{i \neq j}^{n} -x_j = n \cdot x_i - \sum_{j=1}^{n} x_j = n \cdot x$
So actually $Lx = n \cdot x$, which means that any vector orthorgonal to v_0 is an eigenvector to L with eigenvalue n

Task 1.2.3 (3 points)

For all the graphs with k connected components $\lambda_0 = \lambda_1 = \ldots = \lambda_k = 0$

A graph with k connected components would be able to be represented as k seperate graphs with their own adjacency matrices, and therefor also k different laplacians. From task 1.2.1 we know that every such graph will have an eigenvector of all ones with an eigenvalue of 0. These can trivially be extended to be eigenvectors of the entire graph, and still have eigenvalue 0, by just padding with 0's. This along with the original λ_0 will result in at least $k + 1$ eigenvalues that are equeal to 0.

Task 1.2.4 (5 points)

Given a graph G with eigenvalues of the laplacian $\lambda_0, \lambda_1, \ldots, \lambda_{n-1}$.
We randomly remove an edge from G and we re-calculate the eigenvalues as $\lambda'_0, \lambda'_1, \ldots, \lambda'_{n-1}$.
Can we have $\lambda'_i > \lambda_i$ for some $0 \leq i \leq n - 1$? Why? Why not?

We cannot have that $\lambda'_i > \lambda_i$ for any $0 \leq i \leq n - 1$, since removing an edge can only decrease the first and largest eigenvalue, which means that since $\lambda_0 \geq \lambda_1 \geq \ldots \geq \lambda_{n-1}$, the monotonicity of the eigenvalues is maintained.

Task 1.2.5 (6 points)

Suppose that the graph G consists of two connected componentes of equal size named G_1 and G_2. For simplicity assume that n is even.
The Laplacian of G_1 has eigenvalues $\lambda_0^1, \lambda_1^1, \ldots, \lambda_{n/2-1}^1$.
The Laplacian of G_2 has eigenvalues $\lambda_0^2, \lambda_1^2, \ldots, \lambda_{n/2-1}^2$.
Prove that the Laplacian of G is consisted of the eigenvalues of the Laplacians of G_1 and G_2 in ascending order.

YOUR ANSWER HERE

Part 2: Graphs and Spectral clustering

In this part, you will experiment and reflect on spectral clustering as a technique for partitioning a graph.

Task 2.1: ε-neighbourhood graph (10 points)

In this subsection you will experiment with some biological data https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1003268.

!IMPORTANT! First run the following code to load the data.

In [ ]:
```
#Load Data
from utilities.make_graphs import read_edge_list, read_list, load_data
import numpy as np
X, Y = load_data()
```

Task 2.1.1 (4 points)

[Implement] the ε-neighborhood graph, using Eucledian (L_2) distance.

Note: Be sure that your constructed graph does not contain self-loop edges (edges from i to i for each i)

In [ ]:
```
# Be sure that your constructed graphs does not
# contain loop edges (edges from i to i for some node i)

def L2_norm(x, y):
    return np.sqrt(np.sum((x-y)**2))
```

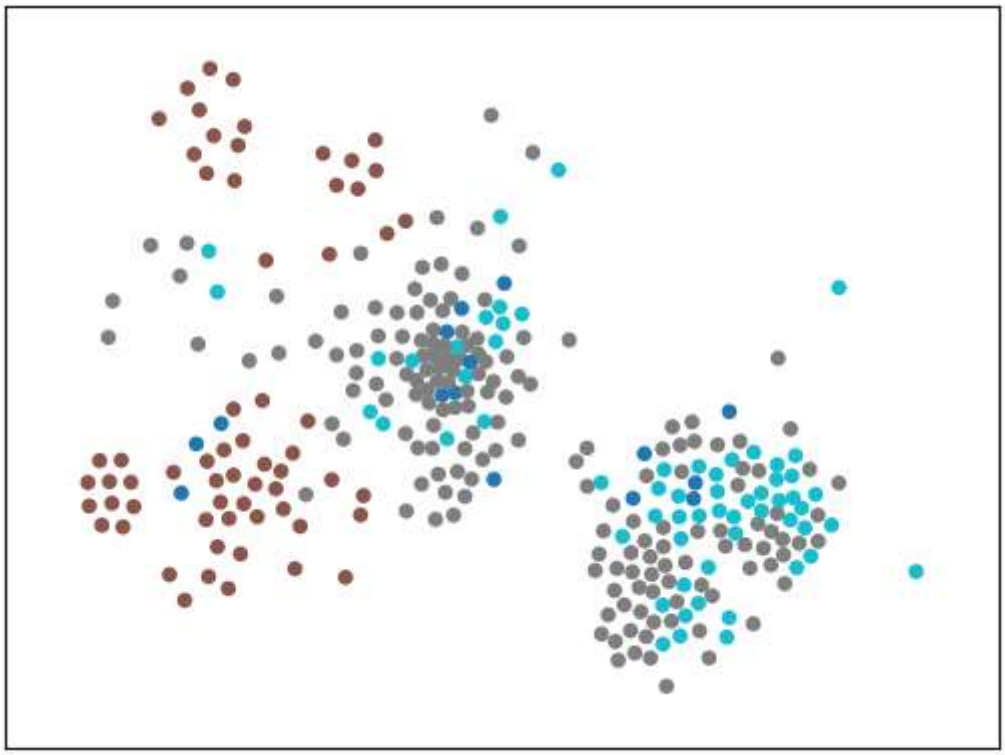
```
def nn_graph(data, eps, remove_self=True, directed=False):
    n = len(X)
    G = nx.Graph()
    if directed:
        G = nx.DiGraph()
    for idi, i in enumerate(data):
        G.add_node(idi)
    for idx, x in enumerate(data):
        for idy, y in enumerate(data):
            if L2_norm(x,y) <= eps:
                if idx != idy:
                    G.add_edge(idx,idy)
                elif not remove_self:
                    G.add_edge(idx,idy)

    return G
```

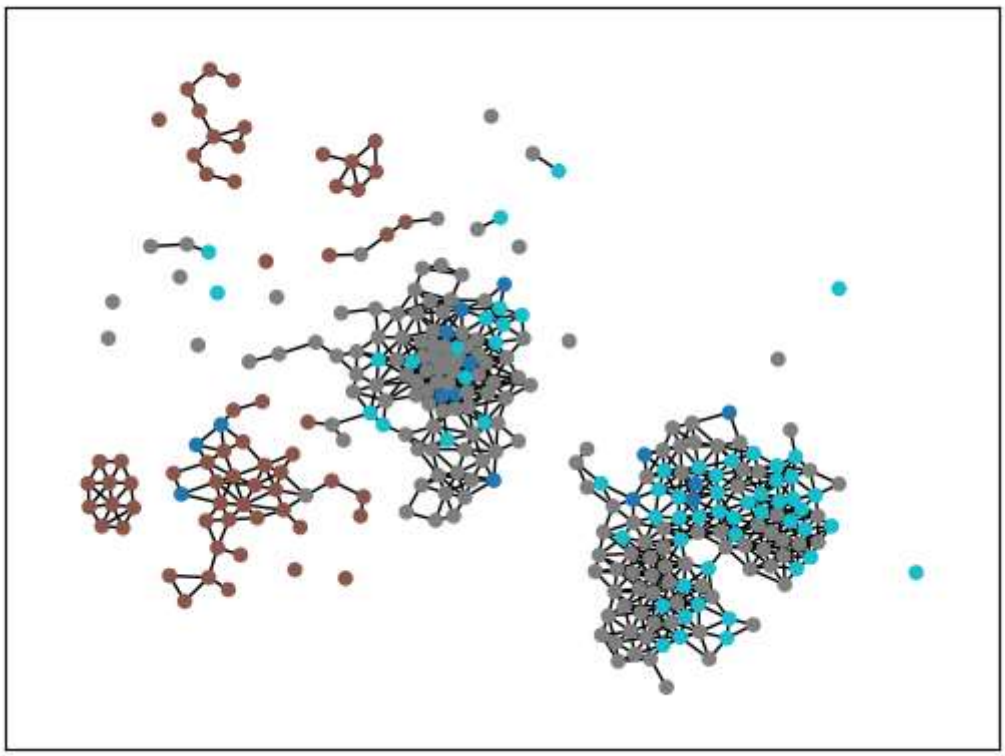
Task 2.1.2 (2 points)

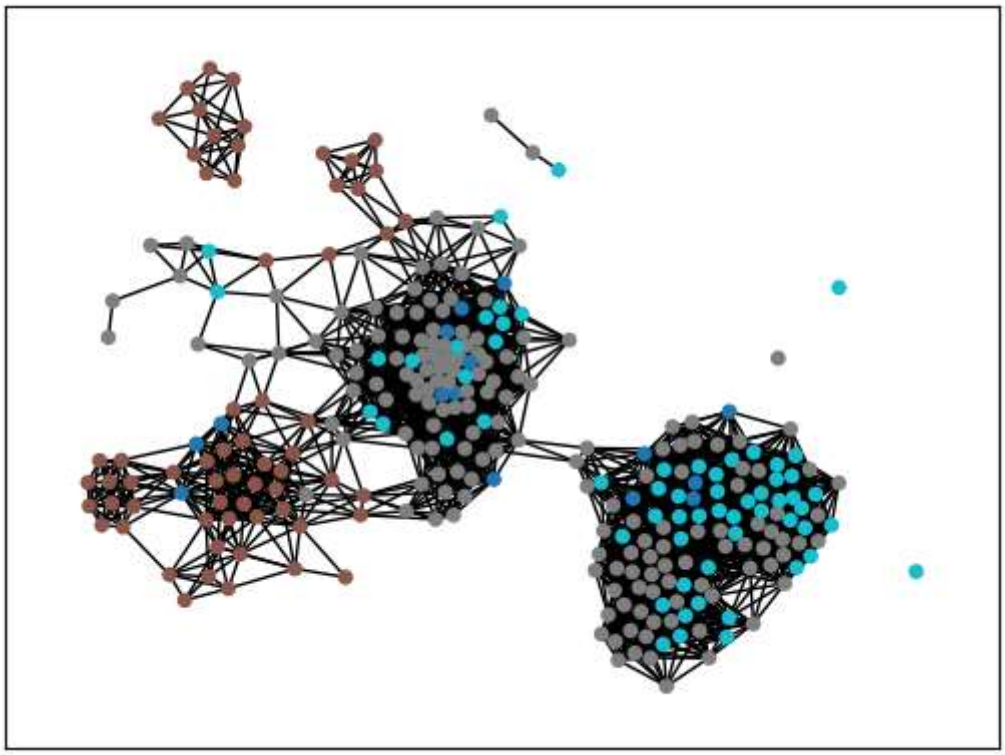
Try with different ε values (select a small set of ε, e.g., 0.01-0.5 values) and plot the graphs.

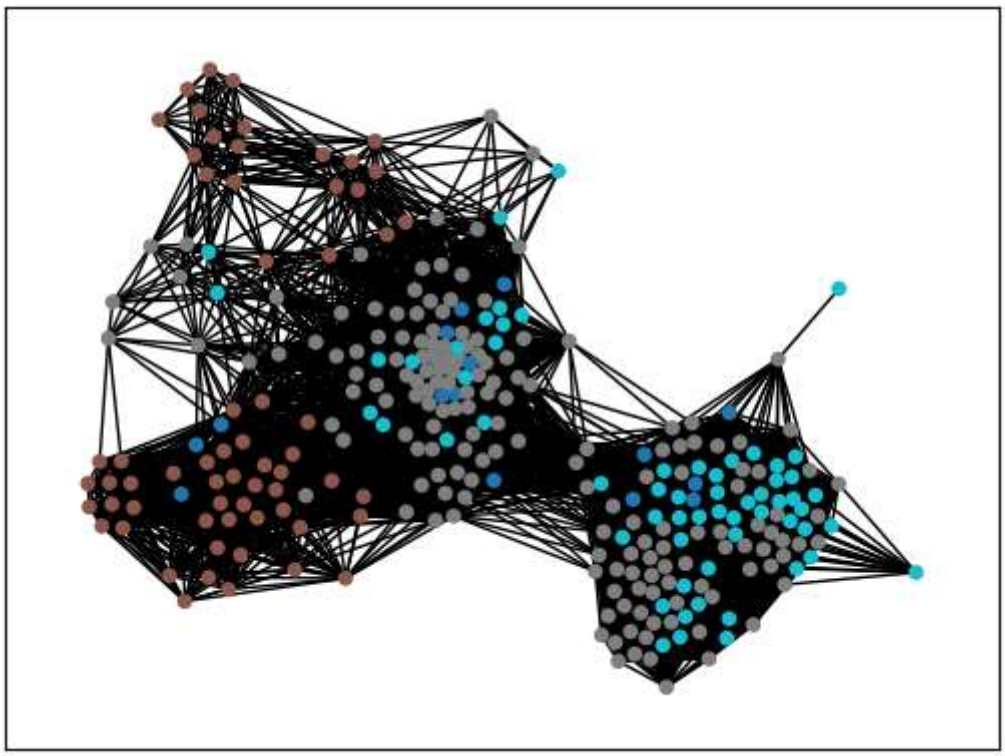
[Motivate] what you observe as epsilon increases.

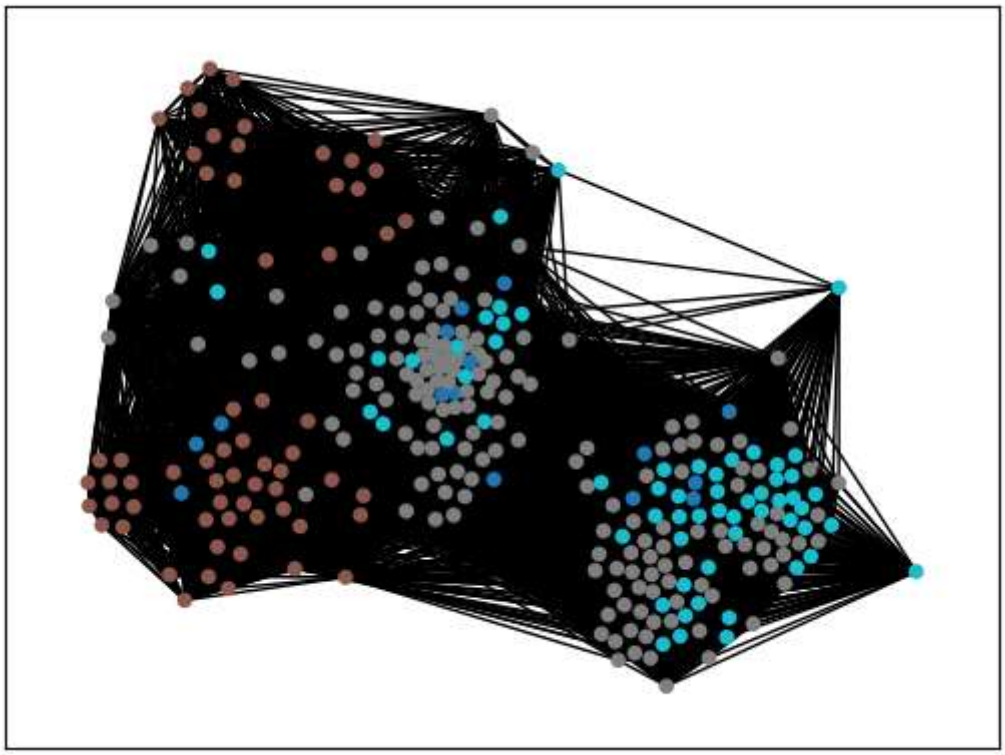
In [ ]:
```
### Run the code below
eps_values = [0.01, 0.05, 0.1, 0.2, 0.4]

for eps in eps_values:
    ax=plt.subplot()
    ax1=plt.subplot()
    G = nn_graph(X, eps)
    pos=nx.spring_layout(G)
    nx.draw_networkx_edges(G,pos=X)
    nx.draw_networkx_nodes(G, pos=X, node_color=Y, node_size=20, cmap=plt.get_cmap(
    ax.set_xlim(-0.1, 1.1)
    ax.set_ylim(-0.1, 1.1)
    plt.show()
```











Well, I observe that the more we increase the ϵ values, the more edges are created. This happens because the ϵ-neighborhood graph create edges between neighbors at distance at most ϵ. Thereby, increasing the ϵ value would also lead to an increase in the number of edges.

Task 2.1.3 (2 points)

Assign to each edge in the ε-neighborhood graph a weight

$$W_{ij} = e^{-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{t}}$$

[Implement] the function `weighted_nn_graph` below that returns the weighted graph given the data matrix in input and the values eps and t, where t is the parameter of the equation above.

In [ ]:
```
def L2_norm(x, y):
    return np.sqrt(np.sum((x-y)**2))

def weighted_nn_graph(data, eps=20, t=0.1):
    n = len(data)
    G = nx.Graph()
    for idi, i in enumerate(data):
        G.add_node(idi)
    for idx, x in enumerate(data):
        for idy, y in enumerate(data):
```

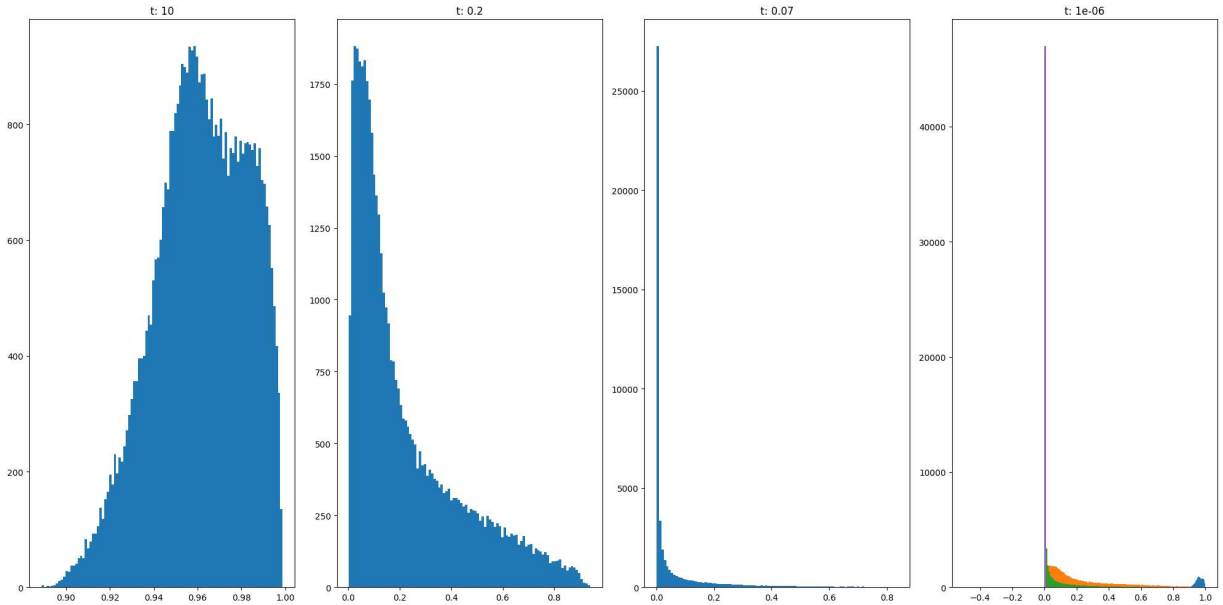
```
            if L2_norm(x,y) <= eps:
                if idx != idy:
                    eps_weight = np.exp(-(L2_norm(x,y) / t))
                    G.add_edge(idx,idy, weight=eps_weight)

    return G
```

Task 2.1.4 (2 points)

Vary $t \in \{10, 0.1, 0.000001\}$. Plot the weights as a histogram using the code below in order to analyse the results using the provided code.
What happens when t is very small, close to 0, i.e., $t \to 0$?
What happens when t is very large?
Is the behaviour with $t = 0$ expected?

[Motivate] your answer reasoning on the formula.

In [ ]:
```
ts = [10, 0.2, 0.07, 0.000001]
fig, ax = plt.subplots(1,4, figsize=(20, 10))
row = 0

for i, t in enumerate(ts):
    G = weighted_nn_graph(X, eps=60, t=t)
    ys = []


    col = i
    for i, d in enumerate(G.edges.data()):
        ys.append(d[2]['weight'])
    plt.hist(ys, bins=100)
    ax[col].hist(ys, bins=100)
    ax[col].set_title("t: "+str(t))

plt.tight_layout()
```



When $t \to 0$ then $-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{t} \to -\infty$, this making the weight $W_{ij} \to 0$. This is also visable in the above.
When t is large then the fraction in the weight formula goes towards 0, thus making $W_{ij} \to 1$, which is clear from the above.

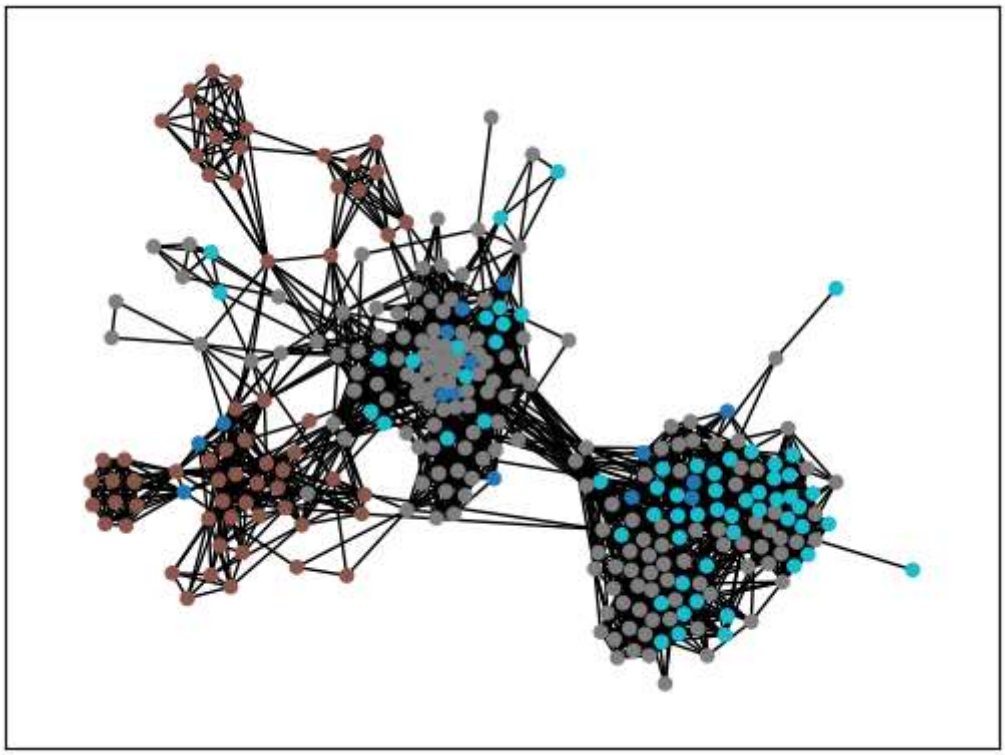
Task 2.2: Spectral clustering (20 points)

We will now look at spectral clustering and its properties.
For this Task we will use a subgraph from malaria_genes.
Note that this dataset is the biological network of the data used in Task 2.1.

Task 2.2.1 (5 point)

Run the code below to load and visualize the network.
By only observing the below plot and the nn-plots (nearest-neighbor plots)of task 2.1.2, which ε values seems to better approximate the real network? (just think of the answer you don't have to write something)

In [ ]:
```
edgelist = read_edge_list('./data/edges.txt')
n = np.max(edgelist)+1
G = nx.Graph()
for i in range(n):
    G.add_node(i)
for edge in edgelist:
    G.add_edge(edge[0], edge[1])
pos=nx.spring_layout(G)
nx.draw_networkx_edges(G,pos=X)
nx.draw_networkx_nodes(G, pos=X, node_color=Y, node_size=20, cmap=plt.get_cmap('tab
plt.show()
```



Now you are having the real network, lets check how good nn-graph (and for which ϵ value) is a good "approximates"the real graph.

A) **[Implement]** a function that calculates the absolute edge difference between the real network G and the one ϵ-neighborhood graph. Note that in order to do that you have to follow two steps:

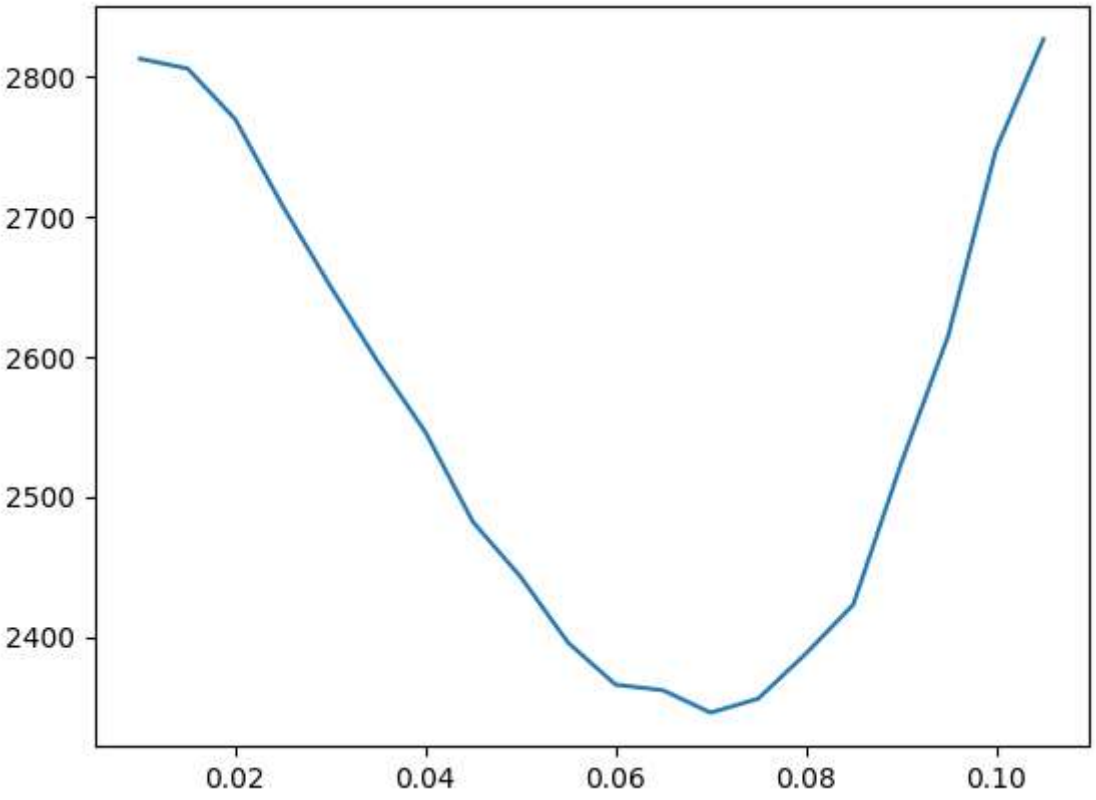
1. In the first step you have to check if an edge in the real graph is also presented in the nn-graph, if not you increase the counter
2. In the second step you follow the opposite direction, that is you check if for every edge of the nn-graph if is also presented in the original one, if not you increase the counter. (Faster way just use the adjacency matrices)

B) **[Implement]** Plot the edge-difference plot for the range of epsilon values in the range[0.01, 0.11] with step = 0.005.
C) **[Motivate]** By observing the plot it seems that there exists only one global minimum and no local minimum. Try to prove/disprove this intuition.

In [ ]:
```python
#A)
def edge_difference(G: nx.Graph, nn: nx.Graph):
    counter = 0
    for i,j in G.edges():
        if not nn.has_edge(i,j):
            counter += 1
    for i,j in nn.edges():
        if not G.has_edge(i,j):
            counter += 1
    return counter
```

In [ ]:
```python
#B)
first = np.arange(0.01,0.11,0.005)
second = []
for x in np.arange(0.01,0.11,0.005):
    nn = nn_graph(X, x)
    diff = edge_difference(G,nn)
    second.append(diff)
plt.plot(first,second)
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x18453d76a10>]
```



The difference is the sum of *wrong* edges and when ϵ increases we only add edges. Thereby, if the difference start increasing (perhaps after reaching a minimum) then the curve will never decrease again. The reason for this is that the *wrong* edges that lead to an increase in the curve will not be removed if we increase ϵ. On the contrary, increasing the ϵ value further might just cause an even large amount of *wrong* edges to be added to the network.

Task 2.2.2 (2 points)

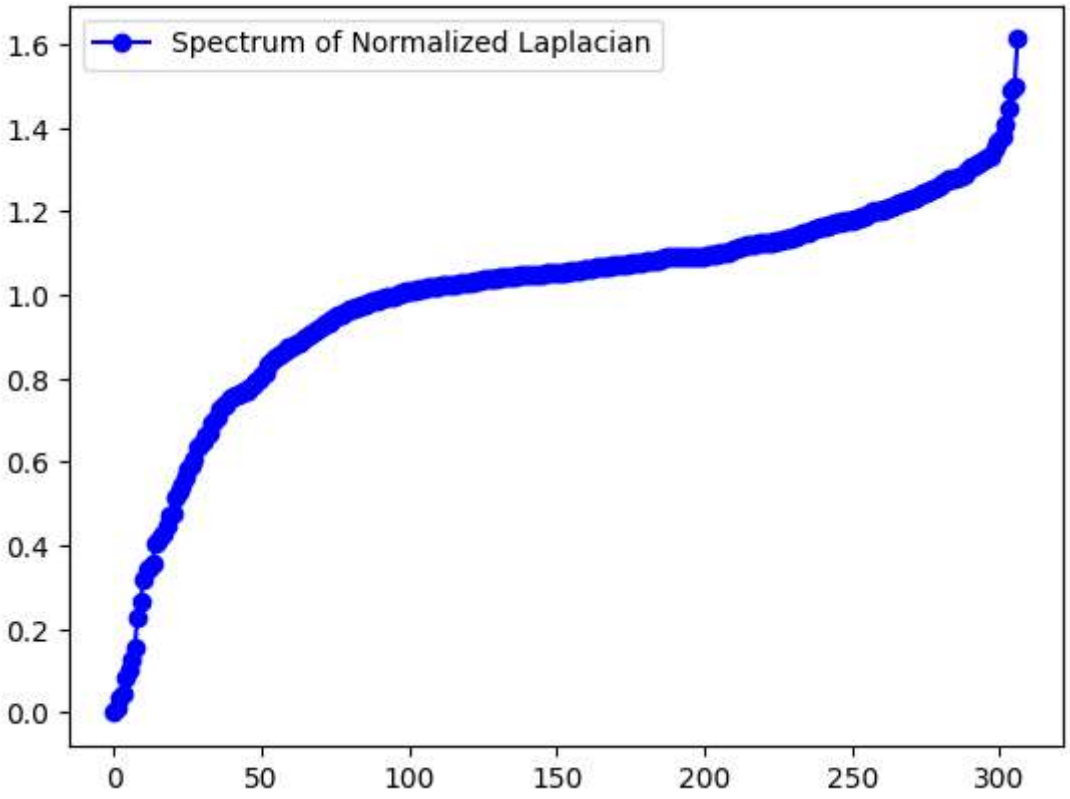
Compute the eigenvectors and eigenvalues (using the provided function) of the Normalized Laplacian and the Random Walk Laplacian of the graph G.
Plot the spectrum (eigenvalues).

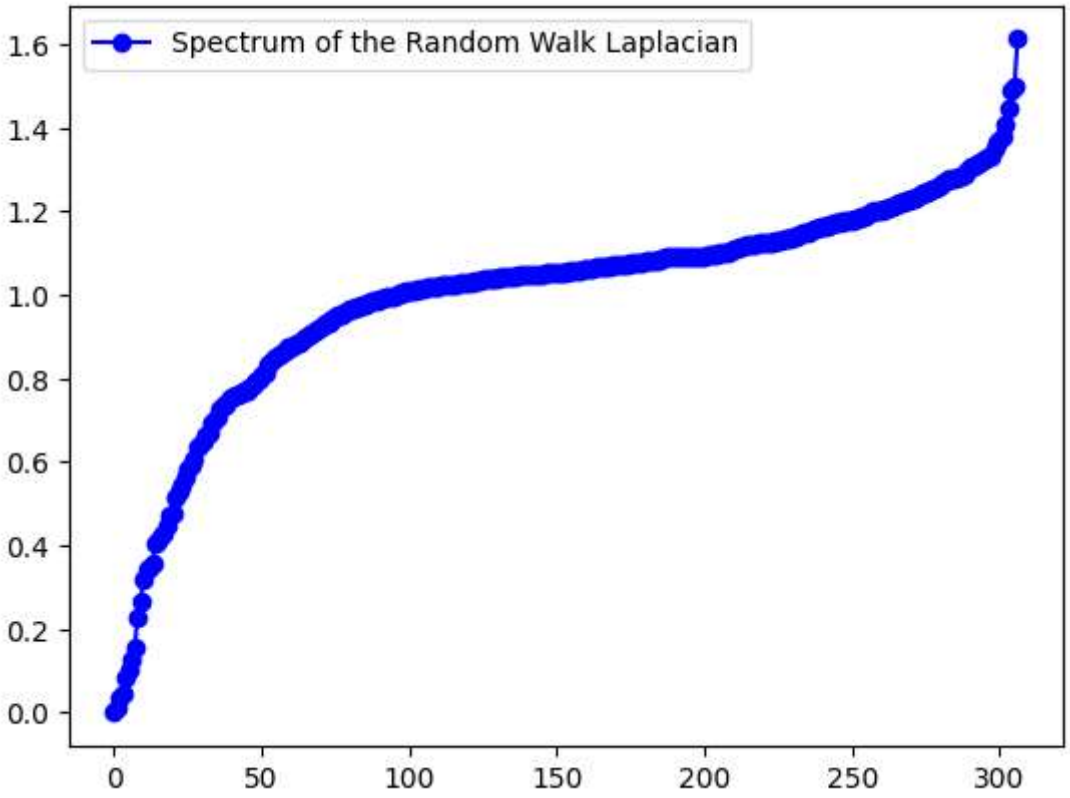
[Implement] the code to compute the different Laplacians.

In [ ]:
```
def graph_eig(L):
    """
        Takes a graph Laplacian and returns sorted the eigenvalues and vectors.
    """
    lambdas, eigenvectors = np.linalg.eig(L)
    lambdas = np.real(lambdas)
    eigenvectors = np.real(eigenvectors)

    order = np.argsort(lambdas)
    lambdas = lambdas[order]
    eigenvectors = eigenvectors[:, order]

    return lambdas, eigenvectors
```

In [ ]:
```
L_norm = None
L_rw = None

adj = nx.to_numpy_array(G)
degree_v = [val for (node, val) in G.degree()]
degree_m = np.zeros((G.number_of_nodes(),G.number_of_nodes()))
np.fill_diagonal(degree_m, degree_v)
L = np.subtract(degree_m, adj)

from scipy.linalg import fractional_matrix_power
L_norm = fractional_matrix_power(degree_m, -1/2) @ L @ fractional_matrix_power(degr
L_rw =np.subtract(np.identity(G.number_of_nodes()), (np.linalg.inv(degree_m) @ adj)


eigval_norm, eigvec_norm = graph_eig(L_norm)
eigval_rw, eigvec_rw = graph_eig(L_rw)

plt.figure(0)
plt.plot(eigval_norm, 'b-o', label='Spectrum of Normalized Laplacian', )
plt.legend()
plt.figure(1)
plt.plot(eigval_rw, 'b-o', label='Spectrum of the Random Walk Laplacian')
plt.legend()
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x18455200d90>
```





Task 2.2.3 (4 points)

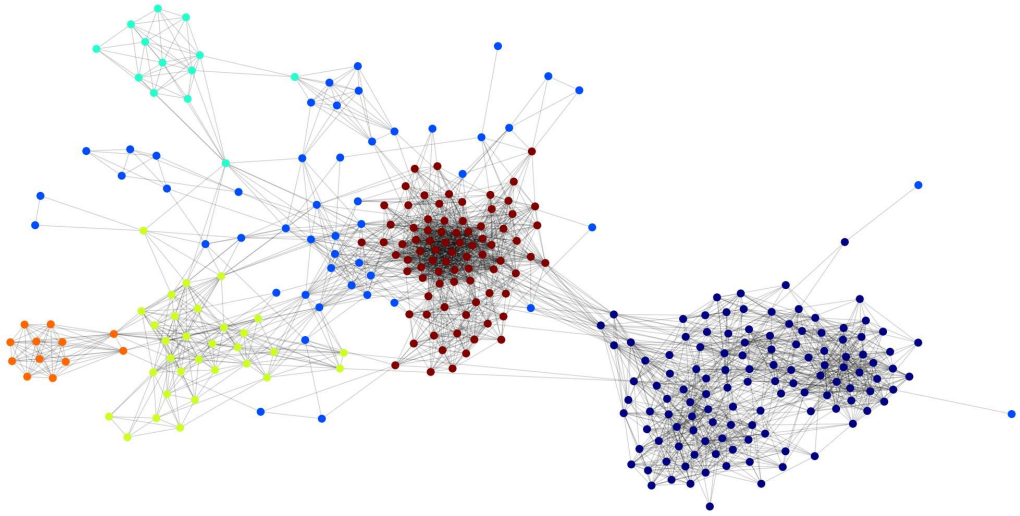
[Implement] the function `spect_cluster` that returns a vector `y_clust` in which each entry y_clust[i] represents the community assigned to node i. The method should be able to handle both the Normalized Laplacian, and the Random Walk Laplacian. You are allowed to

use your implementation from the weekly exercises and `sklearn.cluster.k_means` for k-means clustering.

In [ ]:
```
from sklearn.cluster import k_means

def spect_cluster(G, eig_type="normal", k=5, d=5):
    adj = nx.to_numpy_array(G)
    degree_v = [val for (node, val) in G.degree()]
    degree_m = np.zeros((G.number_of_nodes(),G.number_of_nodes()))
    np.fill_diagonal(degree_m, degree_v)
    L = np.subtract(degree_m, adj)

    if eig_type == "normal":
        L = fractional_matrix_power(degree_m, -1/2) @ L @ fractional_matrix_power(d
    elif eig_type == "random":
        L = np.subtract(np.identity(G.number_of_nodes()), (np.linalg.inv(degree_m)

    l, e = np.linalg.eig(L)
    index = np.argsort(l)
    l = l[index]

    eigen_vec = np.arange(len(l))[:d]
    data = e[:, eigen_vec]

    best_inertia = float('inf')
    y_clust = None
    for i in range(10):
        _, label, inertia = k_means(data, k)
        if inertia < best_inertia:
            y_clust = label
            best_inertia = inertia

    return y_clust
```

In [ ]:
```
def plot_graph(G, clusters):
    plt.figure(1,figsize=(30,15))
    nodes = G.nodes()
    ec = nx.draw_networkx_edges(G, X, alpha=0.2)
    nc = nx.draw_networkx_nodes(G, X, nodelist=nodes, node_color=clusters, node_siz

    plt.axis('off')
    plt.show()
```

In [ ]:
```
import warnings
warnings.filterwarnings("ignore")

your_clusters = spect_cluster(G, k=6)
plot_graph(G, your_clusters)
```

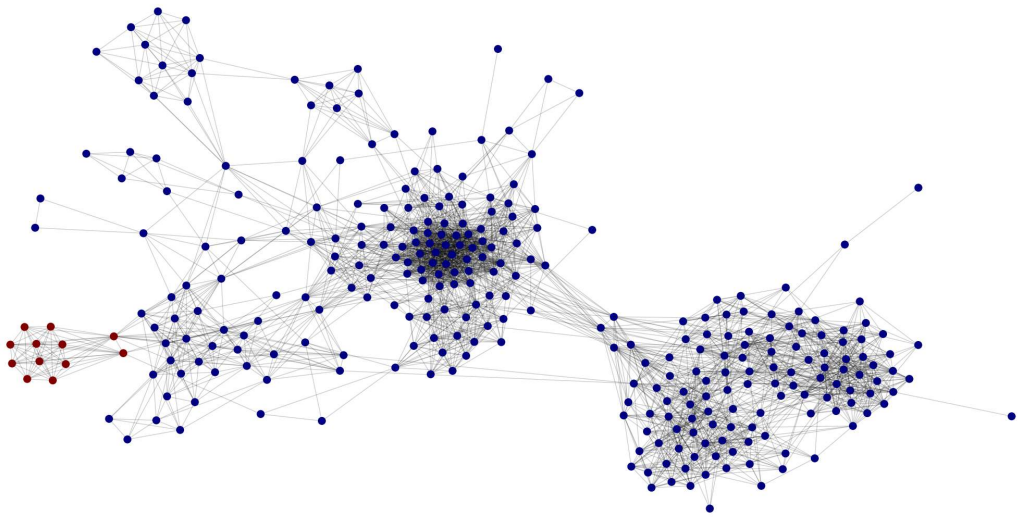


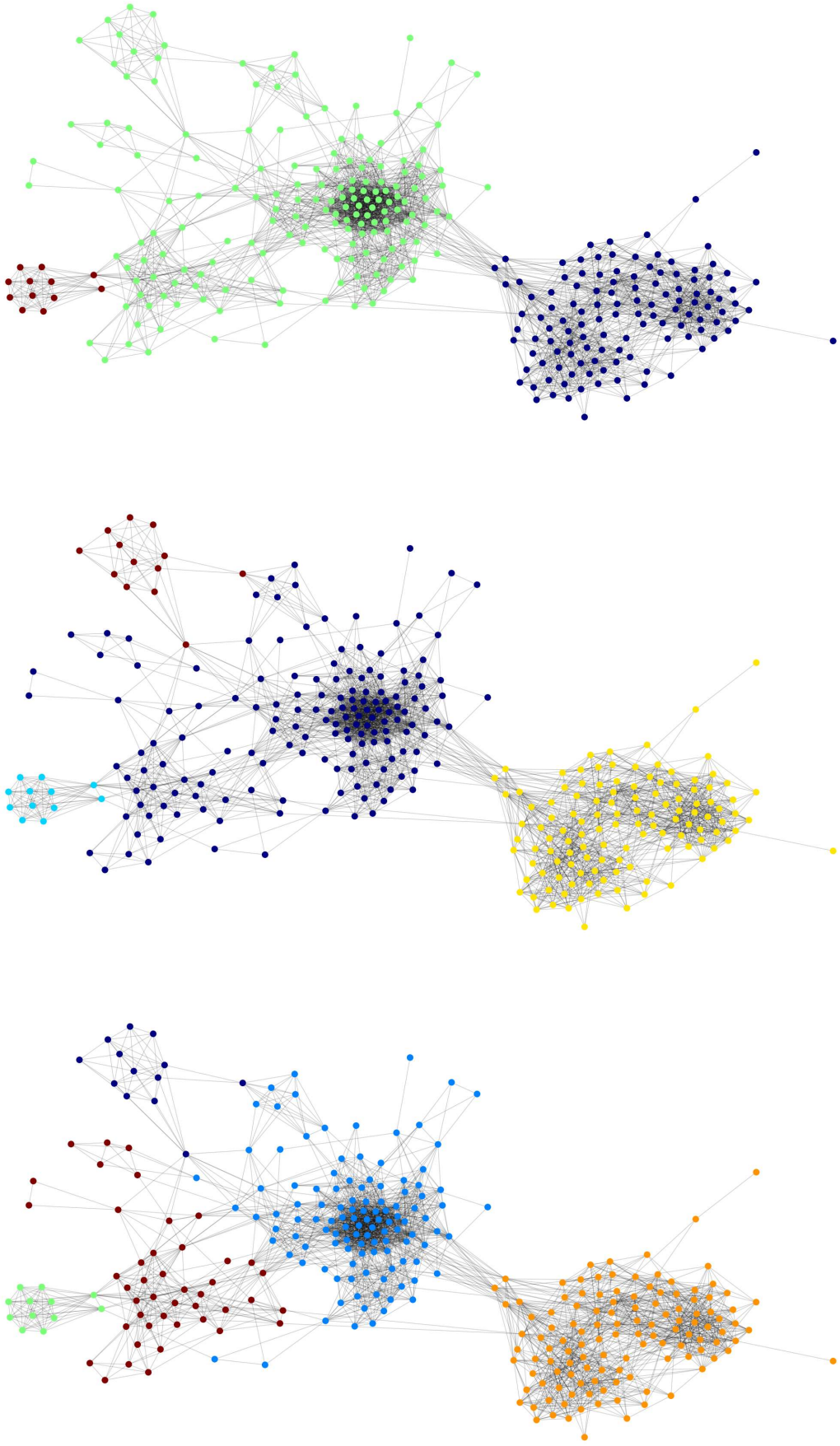
Task 2.2.5 (1 points)

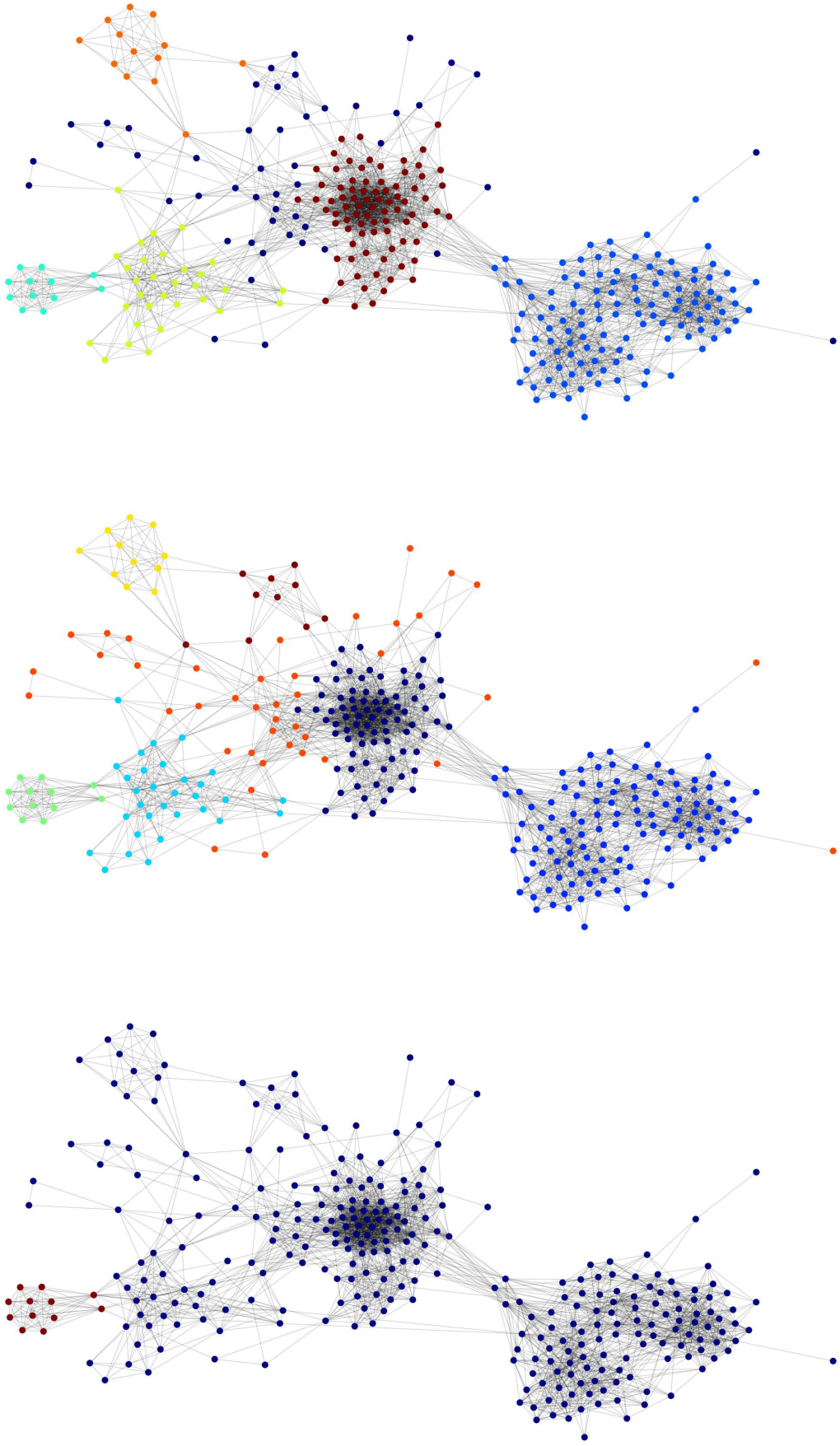
Finally, use your implementation of spectral clustering with different Laplacians and different values of $k \in [2, 7]$ and plot the results using the helper function `plot_graph`.

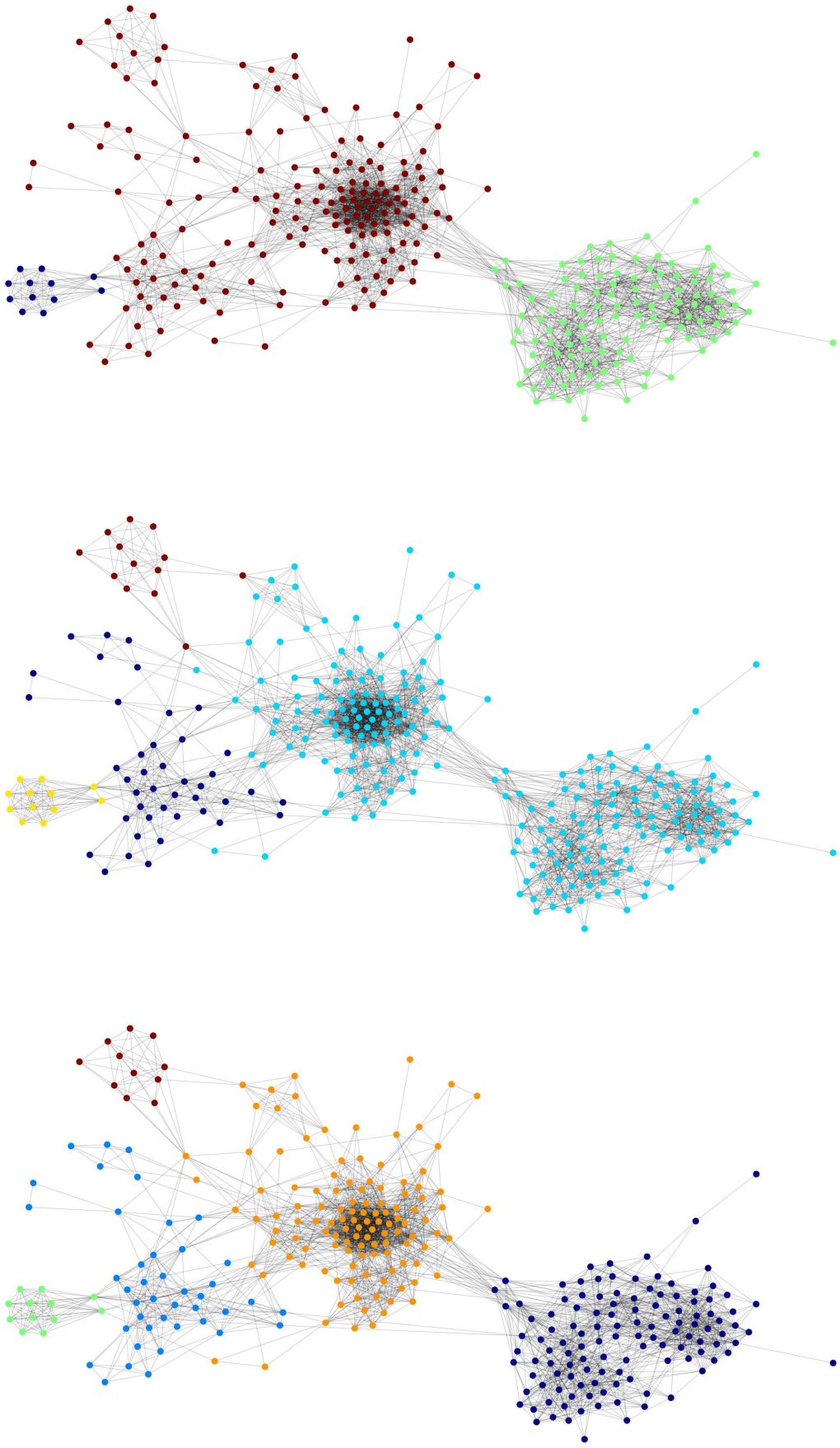
[Describe] the results you obtain. Especially, what is the difference between the Random Walk and the Normalized Laplacians, if any? How do you explain such differences?

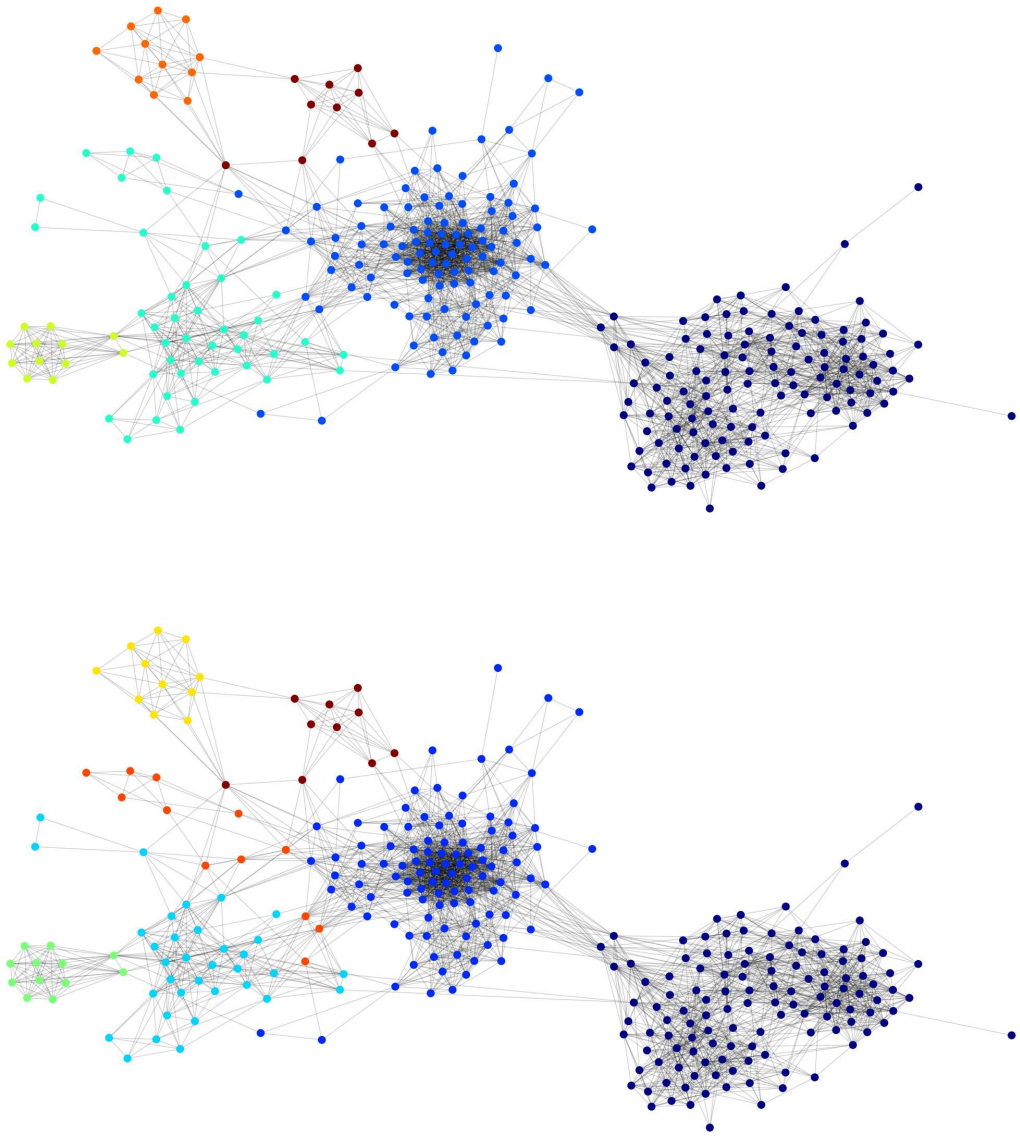
In [ ]:
```
for method in ['normal', 'random']:
    for k in np.arange(2,8):
        your_clusters = spect_cluster(G,eig_type=method, k=k)
        plot_graph(G, your_clusters)
```











When $k = 2, 3$ the resulting clustering are almost identical. However, when $k = 4$ we observe a clear difference. The Normalized Laplacian seems to "add" new clusters to the results of previous clustering, so to speak, whereas the Random Walk Laplacian considers every iteration as an entirely new clustering. It seems that the Normalized Laplacian is more deterministic in its results since one could see the k=4 clustering as an extenstion of the k=3, but for the random walk that is not the case.
Likely this is due to the randomness of the Random Walk Laplacian.

Task 2.2.6 (4 points)

[Implement] the modularity. Recall that the definition of modularity for a set of communities C is

$$Q = \frac{1}{2m} \sum_{c \in C} \sum_{i \in c} \sum_{j \in c} \left(A_{ij} - \frac{d_i d_j}{2m} \right) \qquad (1)$$

where A is the adjacency matrix, and d_i is the degree of node i

Note: Use `plot_graph` function in order to see for yourself if maximising modularity leads a better clustering. If you did not succeed with the previous Task, you are allowed to use Scikit Learn Spectral Clustering

In [ ]:
```
def modularity(G: nx.Graph, clustering):
    modularity = 0
    ### YOUR CODE STARTS HERE
    A = nx.adjacency_matrix(G)
    num_clusters = np.max(clustering)
    clusters = [[] for i in range(num_clusters + 1)]
    for idx,x in enumerate(clustering):
        clusters[x].append(idx)

    for c in clusters:
        for i in c:
            for j in c:
                modularity += (A[i,j] - ((G.degree[i]*G.degree[j]) / (2*G.number_of

    modularity = modularity / (2*G.number_of_edges())

    ### YOUR CODE ENDS HERE
    return modularity
```

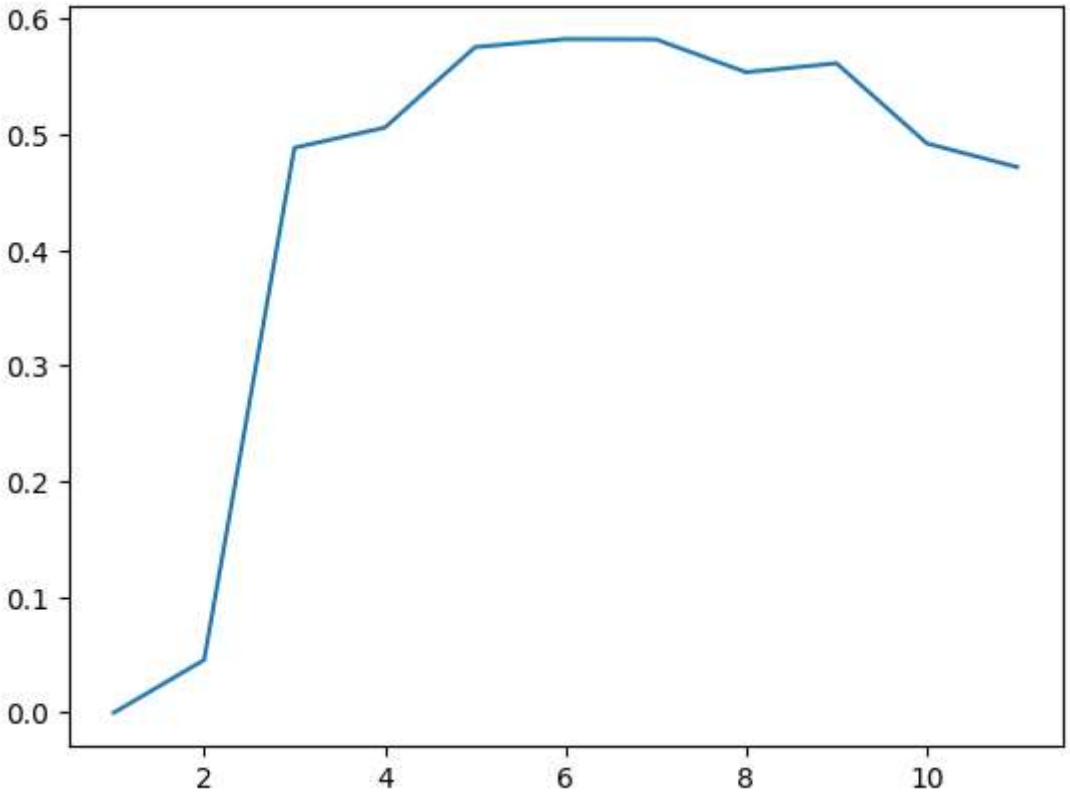
Task 2.2.5 (2 points)

Compute the modularity of your Spectral Clustering Implementation for different values of k.

[Motivate] Which k value maximizes the modularity? From your perspective, does spectral clustering forms "clear" clusters for the best k found by modularity? Using the spectral graph theory, why do you think this is/isn't the case?

In [ ]:
```
mods = []
ks = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
for k in ks:
    clusters = spect_cluster(G, k=k) ### NOTE: If you do not use your implementatio
    mods.append(modularity(G, clusters))

# You may want to use plt.plot to plot the modularity for different values of k
plt.plot(ks, mods)
print(mods)
```

```
[1.0664971355863102e-17, 0.04561221821133083, 0.4885412219324053, 0.505961736330164
7, 0.5755745544900962, 0.5825167085180474, 0.5822726316194062, 0.5538490547521433,
0.5615787045561618, 0.49225100109468073, 0.47179925395935035]
```



When looking at the above given modularity plot, we see that the $k = 6$ maximizes the modularity. When comparing with the spectual clustering for the Normalized Laplacian for $k = 6$, we also observe "clear" / nicely separated clusterings.

Task 2.2.6 (2 points)

[Motivate] There seems to be a relationship between graph embeddings and spectral clustering, can you guess that? *Hint*: Think to the eigenvectors of the graph's Laplacians. (1) Check the correct box below and (2) motivate your answer.

- ☐ If the embeddings are linear and the similarity is the Laplacian, the embeddings we obtain minimizing the L_2 norm are equivalent to the eigenvectors of the Laplacian.
- ☑ If the embeddings are random-walk-based embeddings, the eigenvectors of the Random Walk Laplacian are related to the embeddings obtained by such methods.
- ☐ The relationship is just apparent.
- ☐ If the embeddings are linear and the similarity is the Adjacency matrix, the eigenvectors of the Laplacian are equivalent to the embeddings.

IMPORTANT: Do NOT just choose one answer. Please clarify WHY this is the correct answer.

It makes sense that there is some relationship, since spectral clustering is essentially just a clustering on an embedding based on the eigenvectors. But if the embeddings are based on random walks, then the laplacian should be too, since the type of laplacian is the core of the this relationship.

Part 3: Link analysis

In this exercise, we will work with PageRank, Random Walks and their relationships with graph properties. We will use the most generic definition

$$\mathbf{r} = \alpha \mathbf{M} \mathbf{r} + (1 - \alpha)\mathbf{p}$$

with $\mathbf{r}$ the PageRank vector, $\mathbf{M}$ the weighted transition matrix, and $\mathbf{p}$ the personalization vector. Additionally, let $n = |V|$, where V is the nodes in the graph above. Remember that in the case of PageRank the entries of the personalization vector are $p_i = 1/n$ for all i.

Task 3.1 Approximate PageRank (10 points)

Task 3.1.1 (3 points)

[Implement] a different algorithm for computing Personalized PageRank. This algorithm runs a fixed number of iterations and uses the definition of random walks. At each step, the algorithm either selects a random neighbor with probability α or returns to the starting node with probability $1 - \alpha$. Every time a node is visited a counter on the node is incremented by one. Initially, each counter is 0. The final ppr value is the values in the nodes divided by the number of iterations.

In [ ]:
```
import random
def approx_personalized_pagerank(G: nx.Graph, starting_node, alpha = 0.85, iteratio
    ppr = np.zeros(G.number_of_nodes())
    node = starting_node
    for i in range(iterations):
        choice = random.uniform(0, 1)
        if choice < alpha:
            n = [n for n in G.neighbors(node)]
            edge = random.randint(0, len(n) - 1)
            node = n[edge]
        else:
            node = starting_node
        ppr[node] += 1

    return ppr / iterations
```

Task 3.1.2 (3 points)

Run the `approx_personalized_pagerank` with default α and iterations $\{10, n, 2n, 4n, 100n, 1000n\}$ where n is the number of nodes in the graph and starting node the node with the highest PageRank computed in Task 3.1.2.

[Motivate] what you notice as the number of iterations increase. Why are the values and the top-10 nodes ranked by PPR changing so much?

In [ ]:
```
edgelist = read_edge_list('./data/edges.txt')
n = np.max(edgelist)+1
G = nx.Graph()
for i in range(n):
    G.add_node(i)
for edge in edgelist:
    G.add_edge(edge[0], edge[1])
starting_node = np.argmax(nx.pagerank(G))
for i, iterations in enumerate([10, G.number_of_nodes(), G.number_of_nodes()*2, G.n
    r = approx_personalized_pagerank(G, starting_node, iterations = iterations)
    r[starting_node] = 0
    r_sorted = np.argsort(r)[::-1]
    r_values = np.sort(r)[::-1]
    print(f'Iteration {iterations}: top-10 r={r_sorted[:10]}\n top-10 values={r_val

import operator
rr = nx.pagerank(G, alpha=0.85, personalization = {starting_node: 1})
rr[starting_node] = 0
r=np.zeros(len(rr))
for k in rr: r[k] = rr[k]
r_sorted = np.argsort(r)[::-1]
r_values = np.sort(r)[::-1]
print(f'top-10 r={r_sorted[:10]}\n top-10 values={r_values[:10]}\n')
```

```
Iteration 10: top-10 r=[ 39  17  15  40  36  42  18  41  88 111]
 top-10 values=[0.2 0.2 0.1 0.1 0.1 0.1 0.1 0.1 0.  0. ]

Iteration 307: top-10 r=[40 36 42 41 39 15 57 44 18 37]
 top-10 values=[0.09771987 0.06514658 0.05863192 0.05537459 0.05537459 0.03908795
 0.03583062 0.03583062 0.03583062 0.02605863]

Iteration 614: top-10 r=[41 18 39 15 40 42 36 57 37 17]
 top-10 values=[0.0732899  0.07166124 0.06677524 0.06514658 0.05863192 0.05537459
 0.04723127 0.03745928 0.03745928 0.03583062]

Iteration 1228: top-10 r=[18 42 39 40 15 41 36 44 57 17]
 top-10 values=[0.0781759  0.07084691 0.06840391 0.06840391 0.05944625 0.05781759
 0.0529316  0.0504886  0.04153094 0.04071661]

Iteration 30700: top-10 r=[39 41 40 18 15 42 36 44 57 37]
 top-10 values=[0.07286645 0.06807818 0.06638436 0.06267101 0.06026059 0.05609121
 0.05071661 0.04400651 0.04241042 0.03390879]

Iteration 307000: top-10 r=[39 40 41 18 15 42 36 44 57 37]
 top-10 values=[0.07405863 0.06890554 0.06848208 0.06456678 0.05877199 0.05786319
 0.0507101  0.04462215 0.04313355 0.03340391]

top-10 r=[39 41 40 18 15 42 36 44 57 37]
 top-10 values=[0.07370303 0.0681049  0.0681049  0.0641951  0.05796678 0.05772109
 0.05078277 0.04421339 0.04303241 0.03313206]
```

We notice that whe increasing the number of iterations convergence seems be become more stable. We also converges towards the actual result obtained from the nx.pagerank. There might be some deviations in the convergence as we do random walks.

Task 3.1.3 (2 points)

Compare the 5 nodes with the highest PPR obtained from `nx.pagerank(G, alpha=0.85, personalization={node_highest_pagerank: 1})` and the one obtained by the approximation.

[Describe] the differences. Does the number of iterations affect the results? Is there a relationship between the number of iterations and the results? Is there a relationship between the approximated value of PageRank and the real value? Do you notice anything as the number of iteration increases?

In [ ]:
```
k = 5
ppr_nx = nx.pagerank(G, alpha=0.85, personalization = {starting_node: 1})
r_nx = [0 for _ in range(G.number_of_nodes())]
for k, v in ppr_nx.items():
    r_nx[k] = v
r_est = approx_personalized_pagerank(G, starting_node, alpha=0.85)
```

```
topk_nx = np.argsort(r_nx)[-5:]
topk_est = np.argsort(r_est)[-5:]

print(topk_nx, topk_est)

for iterations in [10, G.number_of_nodes(), G.number_of_nodes()*2, G.number_of_node
    print(f'Number of iterations {iterations}')
    ppr_nx = nx.pagerank(G, alpha=0.85, personalization = {starting_node: 1})
    r_nx = [0 for _ in range(G.number_of_nodes())]
    for k, v in ppr_nx.items():
        r_nx[k] = v
    r_est = approx_personalized_pagerank(G, starting_node, iterations = iterations,
    print(f'Approximate PPR: {r_est[:10]}')
    print(f'Real PPR: {r_nx[:10]}')


    topk_nx = np.argsort(r_nx)[-5:]
    topk_est = np.argsort(r_est)[-5:]

    print(f"Topk of nx.pagerank: {topk_nx}, Topk of our estimation {topk_est}, Size
```

```
[18 41 40 39  0] [18 41 40 39  0]
Number of iterations 10
Approximate PPR: [0.1 0.  0.  0.  0.  0.  0.  0.  0.  0. ]
Real PPR: [0.20102640550640022, 0.0004761511237216732, 0.001421044532107968, 0.00517
928011352294, 0.0010162923813859888, 0.0006924636204434976, 0.00058657979217629 9, 0.
00058657979217629 9, 0.0003234474438257967, 0.00019090839324436927]
Topk of nx.pagerank: [18 41 40 39  0], Topk of our estimation [44 37 42  0 15], Size
of intersection: 1
Number of iterations 307
Approximate PPR: [0.22801303 0.         0.         0.00977199 0.         0.
 0.         0.00325733 0.         0.         ]
Real PPR: [0.20102640550640022, 0.0004761511237216732, 0.001421044532107968, 0.00517
928011352294, 0.0010162923813859888, 0.0006924636204434976, 0.00058657979217629 9, 0.
00058657979217629 9, 0.0003234474438257967, 0.00019090839324436927]
Topk of nx.pagerank: [18 41 40 39  0], Topk of our estimation [15 41 42 39  0], Size
of intersection: 3
Number of iterations 614
Approximate PPR: [0.17589577 0.00162866 0.         0.00814332 0.         0.00162866
 0.         0.         0.00325733 0.00162866]
Real PPR: [0.20102640550640022, 0.0004761511237216732, 0.001421044532107968, 0.00517
928011352294, 0.0010162923813859888, 0.0006924636204434976, 0.00058657979217629 9, 0.
00058657979217629 9, 0.0003234474438257967, 0.00019090839324436927]
Topk of nx.pagerank: [18 41 40 39  0], Topk of our estimation [41 36 40 39  0], Size
of intersection: 4
Number of iterations 1228
Approximate PPR: [0.1970684  0.         0.00081433 0.00651466 0.00081433 0.
 0.         0.002443   0.         0.         ]
Real PPR: [0.20102640550640022, 0.0004761511237216732, 0.001421044532107968, 0.00517
928011352294, 0.0010162923813859888, 0.0006924636204434976, 0.00058657979217629 9, 0.
00058657979217629 9, 0.0003234474438257967, 0.00019090839324436927]
Topk of nx.pagerank: [18 41 40 39  0], Topk of our estimation [40 15 41 39  0], Size
of intersection: 4
Number of iterations 30700
Approximate PPR: [2.02573290e-01 4.56026059e-04 1.59609121e-03 5.63517915e-03
 1.07491857e-03 7.16612378e-04 4.88599349e-04 5.21172638e-04
 1.62866450e-04 2.60586319e-04]
Real PPR: [0.20102640550640022, 0.0004761511237216732, 0.001421044532107968, 0.00517
928011352294, 0.0010162923813859888, 0.0006924636204434976, 0.00058657979217629 9, 0.
00058657979217629 9, 0.0003234474438257967, 0.00019090839324436927]
Topk of nx.pagerank: [18 41 40 39  0], Topk of our estimation [18 40 41 39  0], Size
of intersection: 5
Number of iterations 307000
Approximate PPR: [0.20097068 0.000443   0.00136808 0.00534528 0.00095765 0.00062866
 0.00054397 0.00046906 0.00033876 0.00020847]
Real PPR: [0.20102640550640022, 0.0004761511237216732, 0.001421044532107968, 0.00517
928011352294, 0.0010162923813859888, 0.0006924636204434976, 0.00058657979217629 9, 0.
00058657979217629 9, 0.0003234474438257967, 0.00019090839324436927]
Topk of nx.pagerank: [18 41 40 39  0], Topk of our estimation [18 40 41 39  0], Size
of intersection: 5
```

It seems like the differences in results become smaller as the number of iterations increases. Our approximated values oscilate over the actual ppr values and the larger the number of iterations the smaller the oscillation, which makes sense due to the law of large numbers.

Initially though, in the round with 10 iterations, many of the values are 0 since the random walk will not have had iterations enough to visit all nodes.

Task 3.1.4 (2 points)

Run again the same experiment but this time use $\alpha = 0.1$.

[Motivate] Motivate whether and why you need more or less iterations to predict the 5 nodes with the highest PPR.

In [ ]:
```
for iterations in [10, G.number_of_nodes(), G.number_of_nodes()*2, G.number_of_node
    ppr_nx = nx.pagerank(G, alpha=0.1, personalization = {starting_node: 1})
    r_nx = [0 for _ in range(G.number_of_nodes())]
    for k, v in ppr_nx.items():
        r_nx[k] = v
    r_est = approx_personalized_pagerank(G, starting_node, iterations = iterations,

    topk_nx = np.argsort(r_nx)[-5:]
    topk_est = np.argsort(r_est)[-5:]

    print(f"Topk of nx.pagerank: {topk_nx}, Topk of our estimation {topk_est}, Size
```

```
Topk of nx.pagerank: [40 41 42 39  0], Topk of our estimation [ 99  98  97 104   0],
Size of intersection: 1
Topk of nx.pagerank: [40 41 42 39  0], Topk of our estimation [15 36 44 41  0], Size
of intersection: 2
Topk of nx.pagerank: [40 41 42 39  0], Topk of our estimation [40 42 18 15  0], Size
of intersection: 3
Topk of nx.pagerank: [40 41 42 39  0], Topk of our estimation [40 44 36 42  0], Size
of intersection: 3
Topk of nx.pagerank: [40 41 42 39  0], Topk of our estimation [40 39 42 41  0], Size
of intersection: 5
```

You need a lot more iterations as we travel back to the starting node with a very high probability. Therefore, in order for the pagerank values for the node to converge, we need a lot more iterations as the probability of reaching the nodes is smaller.

Task 3.2 Spam and link farms (12 points)

We will now study the effect of spam in the network and construct a link farm. In this part, if you want to modify the graph, use a copy of the original graph every time you run your code, so that you do not keep adding modifications.

In [ ]:
```
edgelist = read_edge_list('./data/edges.txt')
n = np.max(edgelist)+1
G2 = nx.Graph()
```

```
for i in range(n):
    G2.add_node(i)
for edge in edgelist:
    G2.add_edge(edge[0], edge[1])
G = G2.copy()
```

Task 3.2.1 (3 points)

Based on the analysis in the slides, construct a spam farm s on the graph G with T fake nodes. Assume that s manages to get links from node 1. With $\alpha = 0.5$,

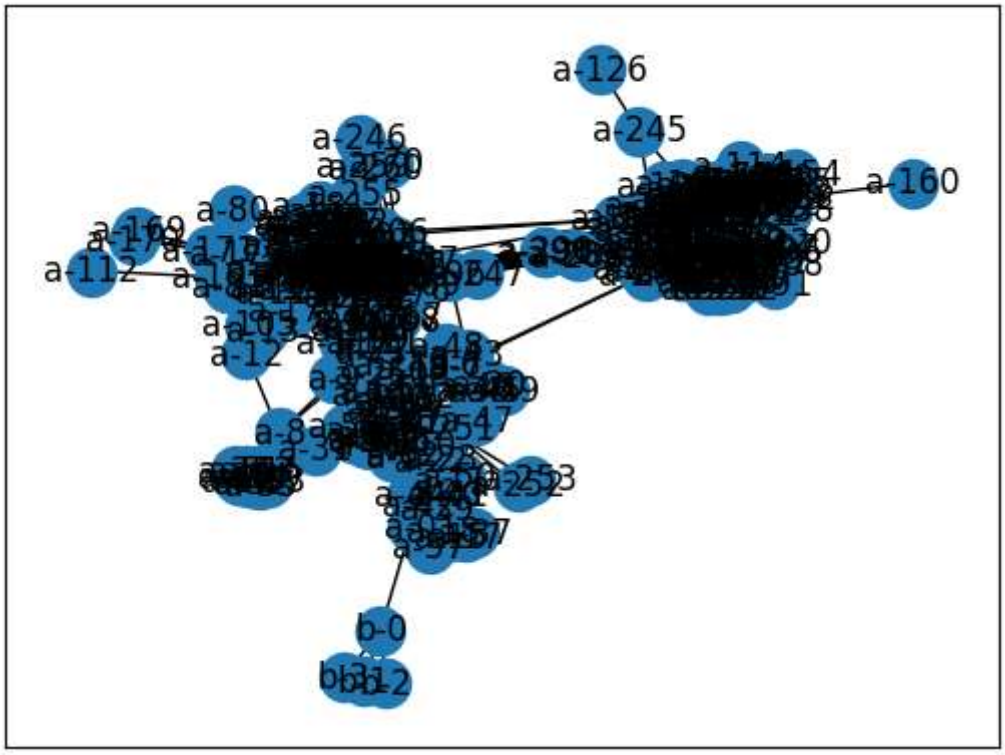
[Describe] which is the minimum number of pages T that we you need to add in order to get s being assigned the highest PageRank?

In [ ]:
```
graph = nx.Graph()
graph.add_node(0)

for i in range(1, 4):
    graph.add_node(i)
    graph.add_edge(0, i)

c = nx.union(G,graph, rename=('a-', 'b-'))
c.add_edge('a-0', 'b-0')
ppr_nx = nx.pagerank(c, alpha=0.5)
ppr_nx_dict = {k: v for k, v in sorted(ppr_nx.items(), key=lambda item: item[1], re
print(ppr_nx_dict)
print(next(iter(ppr_nx_dict)))

nx.draw_networkx(c)
```

{'b-0': 0.005200703607779701, 'a-256': 0.004765976734794501, 'a-272': 0.004765976734794501, 'a-155': 0.0046639421101888805, 'a-24': 0.004656596738025832, 'a-2': 0.004635600408489595, 'a-262': 0.004578932972411321, 'a-119': 0.0045067290466968175, 'a-206': 0.004447691036749183, 'a-149': 0.004441847434729985, 'a-102': 0.004426979905089441, 'a-163': 0.004395765118600438, 'a-240': 0.0042819994719029714, 'a-215': 0.004280790932568061, 'a-142': 0.004254509412846875, 'a-31': 0.004241638024260357, 'a-164': 0.004222983058870922, 'a-265': 0.004179743825024913, 'a-269': 0.004169967546499799, 'a-172': 0.004164066931766172, 'a-293': 0.0041483712124804825, 'a-211': 0.004136754643644903, 'a-176': 0.0041359207997778025, 'a-68': 0.004090559625366447, 'a-178': 0.004076360901500 2904, 'a-137': 0.004057383561201815, 'a-288': 0.0039929953426237645, 'a-261': 0.003989791351448655, 'a-216': 0.003965294552798513, 'a-217': 0.003965294552798513, 'a-0': 0.003957774960651786, 'a-184': 0.003937314430479552, 'a-56': 0.003934058597875684, 'a-264': 0.003930082691829618, 'a-280': 0.003929590114051714, 'a-173': 0.003912791193084509, 'a-278': 0.003912791193084509, 'a-36': 0.00390409498946359 73, 'a-274': 0.003902819965818661, 'a-275': 0.003902819965818661, 'a-77': 0.003898153501500282, 'a-233': 0.003889003665003107, 'a-3': 0.003884623847789213, 'a-248': 0.003882194645460422, 'a-285': 0.0038765432750340824, 'a-305': 0.0038704233689526462, 'a-151': 0.0038575348890022895, 'a-130': 0.003847403970608416, 'a-132': 0.003818271691 5566043, 'a-78': 0.003814550646315167, 'a-289': 0.0037968244672857625, 'a-39': 0.003793062045642522, 'a-183': 0.0037807212340141907, 'a-284': 0.0037781578209865807, 'a-120': 0.00373787435664402, 'a-239': 0.0037294737770708496, 'a-40': 0.003719501223068727, 'a-41': 0.003719501223068727, 'a-109': 0.0037183221572608836, 'a-111': 0.003705475541578, 'a-100': 0.00368928259422387, 'a-204': 0.0036881912671176773, 'a-202': 0.0036719330034839888, 'a-8': 0.003666205821967843, 'a-67': 0.00365810466372819, 'a-26': 0.0036581046637281897, 'a-165': 0.003656088739064209, 'a-283': 0.003631740434814442, 'a-281': 0.0036261247935276525, 'a-282': 0.0036261247935276525, 'a-251': 0.003625584486099713, 'a-63': 0.0036141489007525795, 'a-304': 0.0036084964846754414, 'a-10': 0.0036047185134460333, 'a-124': 0.003603734713257883, 'a-303': 0.003595184466553677, 'a-122': 0.0035913232517934992, 'a-123': 0.0035913232517934992, 'a-83': 0.003581357395926 7856, 'a-94': 0.0035703655123730206, 'a-210': 0.003563780887190909, 'a-97': 0.0035558505054935187, 'a-131': 0.0035426226611810887, 'a-257': 0.003515566892787 9815, 'a-227': 0.003508892820963484, 'a-228': 0.003508892820963484, 'a-93': 0.003500906307607760678, 'a-153': 0.0034864936950750585, 'a-203': 0.003484744509139186, 'a-89': 0.003477905208575946, 'a-99': 0.0034769239352018732, 'a-95': 0.003454886452689965 57, 'a-135': 0.003450495331757706, 'a-238': 0.003437146233797426, 'a-190': 0.003435290522510276, 'a-175': 0.0034340212328874916, 'a-14': 0.003433405264196978, 'a-18': 0.0034250211756911243, 'a-157': 0.003421646981599669, 'a-300': 0.003415348101851362, 'a-306': 0.0034035188435079067, 'a-226': 0.0033964567147462986, 'a-98': 0.0033922580354160663, 'a-128': 0.0033913341907872404, 'a-162': 0.0033913341907872404, 'a-107': 0.0033874512234462863, 'a-108': 0.0033874512234462863, 'a-143': 0.003381069190879166 6, 'a-144': 0.0033810691908791666, 'a-145': 0.00337141333051837, 'a-85': 0.003356109 2830519798, 'a-86': 0.0033561092830519798, 'a-87': 0.0033561092830519798, 'a-244': 0.0033481130576275726, 'a-205': 0.0033383894827286047, 'a-101': 0.003331905644375098 4, 'a-133': 0.0033310389953418867, 'a-207': 0.0033293497456089397, 'a-158': 0.003327018535310446, 'a-159': 0.003327018535310446, 'a-224': 0.0033268240602021912, 'a-225': 0.0033268240602021912, 'a-200': 0.0033207844021136525, 'a-150': 0.00331992369921 75413, 'a-75': 0.0033143496604497247, 'a-88': 0.003311458338193342, 'a-231': 0.003310382149045829 2, 'a-180': 0.003309700620051751, 'a-302': 0.003309700620051751, 'a-16': 0.0033029176498179883, 'a-60': 0.0033029176498179883, 'a-129': 0.003261539811201605, 'a-286': 0.0032572999653917344, 'a-194': 0.00324732228861312, 'a-134': 0.003245028613288 6573, 'a-90': 0.003236618678302194, 'a-234': 0.0032323264789538625, 'a-47': 0.003229771827309386, 'a-259': 0.0032231313908001576, 'a-260': 0.003222313139080015 76, 'a-15': 0.003219521640755238, 'a-212': 0.003217221081171625, 'a-213': 0.003217221081171625, 'a-218': 0.003217221081171625, 'a-174': 0.003213304328917055, 'a-147': 0.0032069094572801265, 'a-5': 0.0032064265412954625, 'a-139': 0.0031932654934607773, 'a-61': 0.0031879582479429932, 'a-66': 0.0031873719279990435, 'a-74': 0.003155238068

17333, 'a-266': 0.0031462136240860868, 'a-161': 0.0031422559116617518, 'a-299': 0.00
31347954624150994, 'a-295': 0.003120895348455282, 'a-296': 0.003120895348455282, 'a-
290': 0.0031200894835278223, 'a-291': 0.0031200894835278223, 'a-125': 0.003117374742
2600867, 'a-59': 0.0031159102104692245, 'a-140': 0.0031148068704365253, 'a-29': 0.00
3113384311334661, 'a-30': 0.003113384311334661, 'a-33': 0.003113384311334661, 'a-3
4': 0.003113384311334661, 'a-35': 0.003113384311334661, 'a-51': 0.00311338431133466
1, 'a-52': 0.003113384311334661, 'a-53': 0.003113384311334661, 'a-54': 0.00311338431
1334661, 'a-55': 0.003113384311334661, 'a-42': 0.0031133721261284277, 'a-28': 0.0031
113349681689646, 'a-64': 0.0031087568454950905, 'a-229': 0.0031014699908784322, 'a-3
2': 0.003094360537152544, 'a-166': 0.0030834043692182797, 'a-156': 0.003081150222974
9817, 'a-116': 0.0030802337503480857, 'a-127': 0.0030764701513320966, 'a-152': 0.003
0757016575511855, 'a-27': 0.0030729601935053793, 'a-185': 0.0030721394549973425, 'a-
186': 0.0030721394549973425, 'a-187': 0.0030721394549973425, 'a-294': 0.003071761300
51068, 'a-223': 0.003069430821812303, 'a-199': 0.0030644146630898366, 'a-138': 0.003
0612278835409276, 'a-219': 0.0030581230810857265, 'a-48': 0.003055165622008816, 'a-2
49': 0.0030515949462486254, 'a-110': 0.003046403651999638, 'a-20': 0.003032325650525
412, 'a-38': 0.003032325650525412, 'a-45': 0.003032325650525412, 'a-46': 0.003032325
650525412, 'a-43': 0.0030265049327246243, 'a-84': 0.0030219780074418633, 'a-254': 0.
0030202465257502803, 'a-19': 0.0030141016235055874, 'a-136': 0.003004049524288406, 'a
-4': 0.003001163248411341, 'a-25': 0.003001163248411341, 'a-58': 0.00299711172929976
66, 'a-243': 0.0029949682691565766, 'a-179': 0.0029852252718585714, 'a-188': 0.00298
31678535306894, 'a-221': 0.002981859505394961, 'a-222': 0.002981859505394961, 'a-2
1': 0.0029776187660669577, 'a-237': 0.0029758054052342717, 'a-195': 0.00297100664972
19507, 'a-72': 0.002950725891359224, 'a-171': 0.0029449745374410816, 'a-6': 0.002943
651120355605, 'a-7': 0.002943651120355605, 'a-96': 0.002941034640806621, 'a-22': 0.0
02939197513350296, 'a-23': 0.002939197513350296, 'a-1': 0.0029361268902320945, 'a-6
2': 0.0029361268902320945, 'a-301': 0.0029274155902062602, 'a-71': 0.002920215239628
7906, 'a-182': 0.002906704255649134, 'a-277': 0.002906704255649134, 'a-70': 0.002906
279600157263, 'a-115': 0.0029010242195302563, 'a-246': 0.002891842509478284, 'a-26
7': 0.002875102028526626, 'a-268': 0.002875102028526626, 'a-250': 0.0028731439196597
77, 'a-292': 0.0028719521986828784, 'a-235': 0.0028695656471003092, 'a-49': 0.002866
368204248174, 'a-192': 0.0028639302959102666, 'a-241': 0.0028632394990117313, 'a-24
5': 0.0028567972669628185, 'a-69': 0.0028525624163120277, 'a-236': 0.002851569102656
869, 'a-113': 0.002837452731065378, 'a-167': 0.0028122760636906854, 'a-270': 0.00281
04041043911818, 'a-271': 0.0028104041043911818, 'a-117': 0.0027636941268501325, 'a-1
18': 0.0027636941268501325, 'a-255': 0.002757977871695101, 'a-103': 0.00275456929985
427, 'a-104': 0.00275456929985427, 'a-298': 0.002747020523824065, 'a-73': 0.00274030
24332006524, 'a-193': 0.002739682104395075, 'a-17': 0.0027378050529023703, 'a-37':
0.0027378050529023703, 'a-242': 0.0027370069150413376, 'a-141': 0.00273080679878942
1, 'a-146': 0.0027250941799715394, 'a-230': 0.0027204852574673143, 'a-273': 0.002720
074490164964, 'a-201': 0.002714723351806451, 'a-297': 0.002709628671245256, 'a-263':
0.0026943738144603307, 'a-168': 0.0026916834582821547, 'a-209': 0.002688749059071026
5, 'a-82': 0.002678913165417353, 'a-76': 0.002676371641816478, 'a-106': 0.0026763716
41816478, 'a-121': 0.002659965806457361, 'a-189': 0.0026524589653431525, 'a-44': 0.0
02641255571417661, 'a-232': 0.0026309950195860013, 'a-196': 0.0026154851651921706,
'a-197': 0.0026154851651921706, 'a-214': 0.0026134914165707943, 'a-148': 0.002599075
1002528943, 'a-287': 0.0025646356842982565, 'a-57': 0.002564451439551879, 'a-181':
0.0025164639993537 48, 'a-191': 0.0025083598092661683, 'a-11': 0.0024548368675996616,
'a-50': 0.0024548368675996616, 'a-252': 0.002446040875242521, 'a-253': 0.00244604087
5242521, 'a-114': 0.0024330646976002098, 'a-208': 0.0024243885263546113, 'a-65': 0.0
02418056848909798, 'a-247': 0.002413270396001201, 'a-169': 0.0023692277027391098, 'a
-170': 0.0023692277027391098, 'a-220': 0.002333663066527961, 'a-258': 0.002308738312
903861, 'a-198': 0.002277342474652983, 'a-279': 0.002270130921914714, 'a-13': 0.0022
690796818480453, 'a-276': 0.0022681249840152263, 'a-79': 0.0022593360295328603, 'b-
1': 0.0022547176945891772, 'b-2': 0.0022547176945891772, 'b-3': 0.00225471769458917 7
2, 'a-154': 0.0022519721426883443, 'a-12': 0.0022395171077797685, 'a-177': 0.0021883

```
779781507297, 'a-126': 0.0020838198935831435, 'a-105': 0.0020795642341293868, 'a-9':
0.002024654300432063, 'a-91': 0.0019625991488931445, 'a-81': 0.001955248174743188,
'a-92': 0.001880747908248711, 'a-112': 0.0018457097275237872, 'a-80': 0.001791500191
6496232, 'a-160': 0.0017081940789719354}
b-0
```



Just above we have created a node *b-0* that connects to *b-1*, *b-2* and *b-3*. The node *b-0* is then also connected to *a-0* and running the *nx.pagerank* with $\alpha = 0.5$, we quickly notice that *b-0* is assigned the highest PageRank. If *b-0* had one less connection, the node *a-256* would have been assigned the highest PageRank.

Task 3.2.2 (3 points)

In the above scenario, assume that $T = \frac{1}{5}$ of the nodes in the original graph.

[Motivate] what value of α will maximize the PageRank $\mathbf{r}_s$ of the link farm s. Provide sufficient justification for your choice.

The value of α that maximizes the PageRank r_s values is $\alpha = 0.8$. The reasoning behind this is that the probability of jumping to a different node will be 20% which is equivalent to the percentage of the network that is connected to s. So with 20% probability we jump to a random node and the probability of that node being in our link farm is 20%.

Task 3.2.3 (3 points)

Now we fix both $\alpha = 0.5$ and $T = \frac{1}{5}n$.

[Implement] `trusted_pagerank` the method for spam mass estimation.

In [ ]:
```
def trusted_pagerank(G, trusted_indices, iterations=500, alpha=0.5):
    r = None
    ### YOUR CODE STARTS HERE
    p = np.zeros(G.number_of_nodes())
    p[trusted_indices] = 1/len(trusted_indices)
    personalization = {v: k for v, k in enumerate(p)}
    r = nx.pagerank(G,alpha=alpha,max_iter=iterations,personalization=personalizati
    ### YOUR CODE ENDS HERE
    return r
```

Task 3.2.4 (3 points)

[Motivate] whether we are able to detect the node s, if the trusted set of nodes is a random sample 10% of the nodes in the original graph. If not, what could be a viable solution? Which nodes would you rather choose as trusted?

You are not obliged to, but you can write some helper code to reach the answer faster.

Hint: Remember the spam mass formula in the Link Analysis lecture

In [ ]:
```
### YOUR CODE HERE
n = G.number_of_nodes()
trusted = random.sample(list(range(n)), n // 10)

r_x_plus = list(trusted_pagerank(G, trusted).values())
r_x = list(nx.pagerank(G,alpha=0.5,max_iter=500).values())

r_x_minus = np.subtract(r_x, r_x_plus)
x = np.divide(r_x_minus, r_x)
print(np.argmax(x))
#[g for g,i in enumerate(x) if i < 0]
```

126

We would detect the node s, as this node would have a large spam mass value. If we are not able to do so with the 10% sample we could selected central / trusted nodes from the original graph.

Part 4: Graph embeddings (19 points)

In this final part, we will try a different approach for clustering the data from above. The strategy is going to be the following:

1. Use VERSE [1] to produce embeddings of the nodes in the graph.
2. Use K-Means to cluster the embeddings. Measure and report NMI for the clustering.

[1] Tsitsulin, A., Mottin, D., Karras, P. and Müller, E., 2018, April. Verse: Versatile graph embeddings from similarity measures. In Proceedings of the 2018 World Wide Web Conference (pp. 539-548).

In [ ]:
```
G = email.S_dir.copy()
```

Task 4.1.1 (6 points)

[Implement] the methods below to compute sampling version of VERSE. *Hint:* it might be a help to look in the original article [1] above.

In [ ]:
```
from random import choice, choices
def sigmoid(x):
    ''' Return the sigmoid function of x
        x: the input vector
    '''
    return 1 / (1 + np.exp(-x))

def pagerank_matrix(G: nx.Graph, alpha = 0.85):
    ''' Return the Personalized PageRank matrix of a graph

        Args:
            G: the input graph
            alpha: the dumping factor of  PageRank

        :return The nxn PageRank matrix P
    '''
    P = np.empty((G.number_of_nodes(), G.number_of_nodes()), dtype=float)
    for i in G.nodes():
        page_rank = nx.pagerank(G, alpha=alpha, personalization = {i: 1})
        P[i] = list(page_rank.values())
    return P

def update(u, v, Z, C, step_size) :
    '''Update the matrix Z using row-wise gradients of the loss function

       Args:
            u : the first node
            v : the second node
            Z : the embedding matrix
            C : the classification variable used in Noise Contrastive estimation in
            step_size: step size for gradient descent


       :return nothing, just update rows Z[v,:] and and Z[u,:]
```

```
    '''
    ### YOUR CODE STARTS HERE

    prev_z_v = Z[v].copy()
    prev_z_u = Z[u].copy()

    gradient = (C - sigmoid(Z[u] @ Z[v].T)) * step_size

    Z[u,:] += (gradient * prev_z_v).reshape(-1)
    Z[v,:] += (gradient * prev_z_u).reshape(-1)

    ### YOUR CODE ENDS HERE

def simG(G, S, u):
    dist = S[u]
    return choices(list(G.nodes()), dist)[0]

def verse(G, S, d, k = 3, step_size = 0.0025, steps = 10000):
    ''' Return the sampled version of VERSE

        Args:
            G: the input Graph
            S: the PageRank similarity matrix
            d: dimension of the embedding space
            k: number of negative samples
            step_size: step size for gradient descent
            steps: number of iterations

        :return the embedding matrix nxd
    '''
    n = G.number_of_nodes()
    Z = 1/d*np.random.rand(n,d)

    ### YOUR CODE STARTS HERE
    for _ in range(steps):

        u = random.randrange(n)
        v = np.random.choice(n, 1, p=S[u])

        update(u, v, Z, 1, step_size)

        for _ in range(k):
            u_hat = random.randrange(n)
            update(u, u_hat, Z, 0, step_size)

    ### YOUR CODE ENDS HERE
    return Z
```

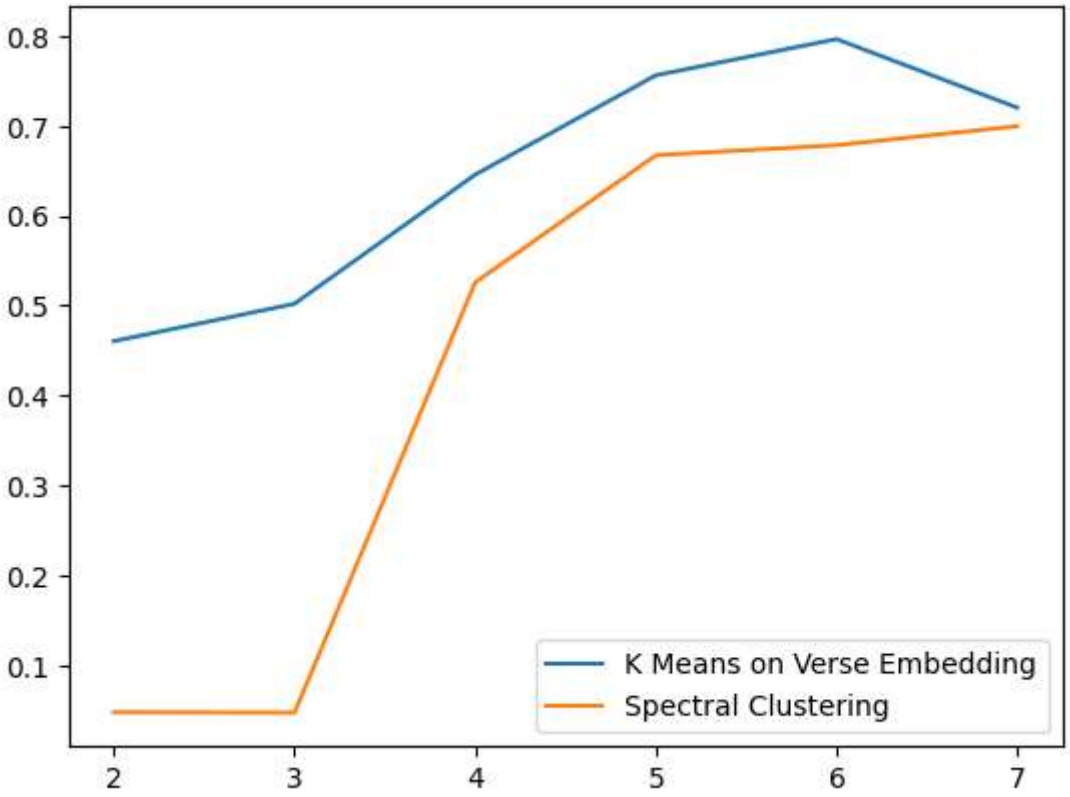
In [ ]:
```
# This code runs the `verse` algorithm above on G and stores the embeddings to 'ver
P   = pagerank_matrix(G)
emb = verse(G, P, 128, step_size=0.015, steps=50_000)
np.save('verse.npy', emb)
```

Task 4.1.2 (3 points)

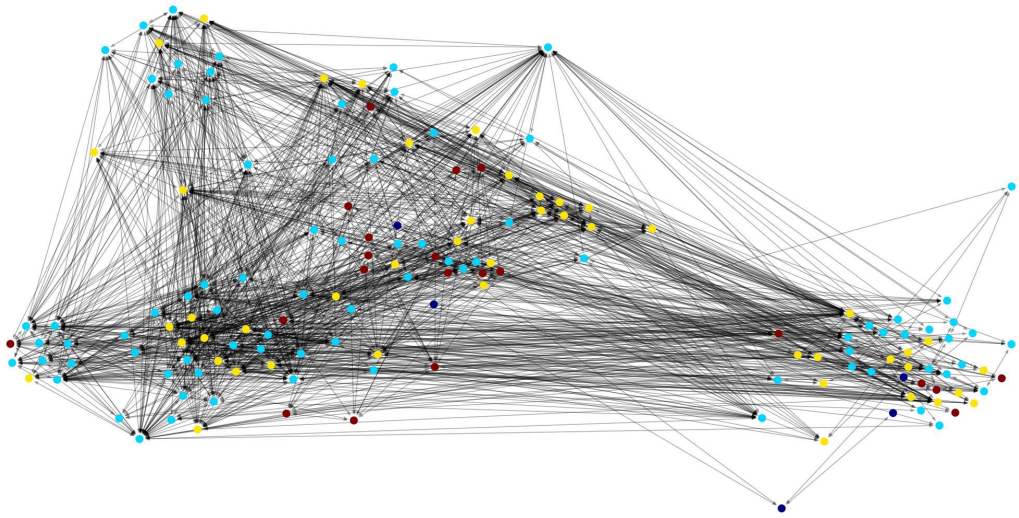
[Implement] a small piece of code that runs k-means on the embeddings with $k \in [2, 7]$ to evaluate the performance compared to Spectral clustering using the NMI as measure. You can use `sklearn.metrics.normalized_mutual_info_score` for the NMI and `sklearn.cluster.KMeans` for kmeans. In both cases, you can use your own implementation from Handin 1 or the exercises, but it will not give you extra points.

[Describe] which of the method performs the best and whether the results show similarities between the two methods

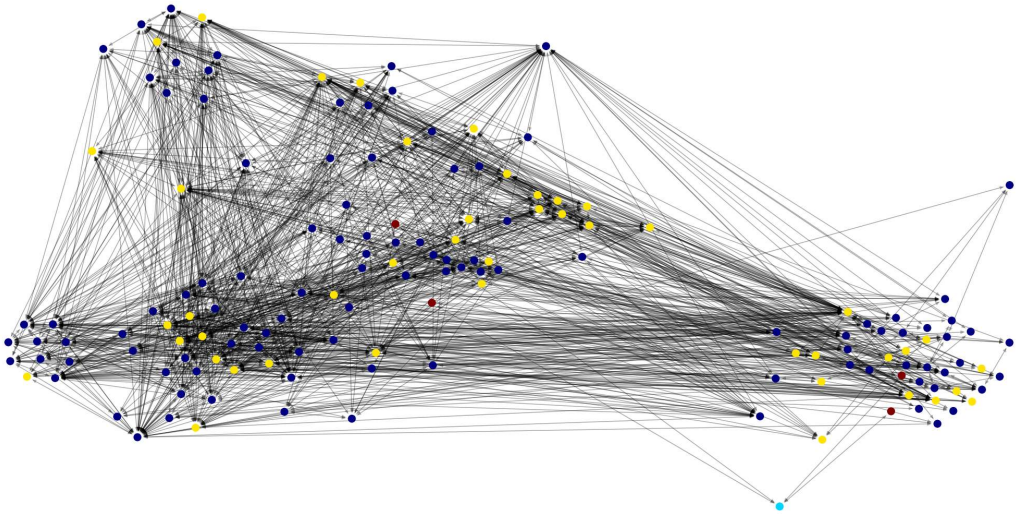
In [ ]:
```
from sklearn.cluster import KMeans, SpectralClustering
from sklearn.metrics.cluster import normalized_mutual_info_score
### YOUR CODE STARTS HERE
G = email.S_dir.copy()
k_means_nmi = []
spect_nmi = []
x = range(2,8)
for i in range(2,8):
    label = KMeans(n_clusters=i).fit_predict(emb)
    your_clusters = spect_cluster(G, k=i)

    k_nmi = normalized_mutual_info_score(email.communities, label)
    spec = normalized_mutual_info_score(email.communities, your_clusters)

    k_means_nmi.append(k_nmi)
    spect_nmi.append(spec)
### YOUR CODE ENDS HERE
plt.plot(x, k_means_nmi, label = "K Means on Verse Embedding")
plt.plot(x, spect_nmi, label = "Spectral Clustering")
plt.legend()
plt.show()
```



In [ ]:
```
label = KMeans(n_clusters=4).fit_predict(emb)
plot_graph(G, label)
```



In [ ]:
```
your_clusters = spect_cluster(G, k=4)
plot_graph(G, your_clusters)
```



In the above we observe that the k Means clustering based on the VERSE embedding result in the best clusterings.
In the two images just above we clearly see that the two clustering method provide clusters that are very similar.

Task 4.1.3 (2 points)

[Motivate] how you would conceptionally expand the way of embedding a graph, if you had a multi-label-graph. E.g. meaning you have multiple labels and each edge needs to have exacrly one of those. So you can also have multiple edges between the same nodes, as long as they have different labels.

If we have a multi-label graph where each edge can have one of multiple labels, we could expand the way of embedding the graph by considering each label as a separate dimension. We would simply have each edge have encoding of the labels and use that for embedding.

Task 4.2 (8 points)

This is a hard exercise. Do it for fun or only if you are done with easier questions.

[Implement] a new GCN that optimizes for modularity. The loss function takes in input a matrix $C \in \mathbb{R}^{n \times k}$ of embeddings for each of the nodes. C represents the community assignment matrix, i.e. each entry C_{ij} contains the probability that node i belong to community j.

The loss function is the following

$$loss = -Tr(C^\top BC) + l||C||_2$$

where B is the modularity matrix that you will also implement, and l is a regularization factor controlling the impact of the L_2 regularizer. We will implement a two-layer GCN similar to the one implemented in the exercises, but the last layer's activation fucntion is a Softmax.

In [ ]:
```
import pykeen

# Adjacency matrix
G     = email.S_undir.copy()
A     = np.array(nx.adjacency_matrix(G, weight=None).todense())
I     = np.eye(A.shape[0])
A     = A + I # Add self loop

# Degree matrix
### YOUR CODE HERE
degree_v = [val for (node, val) in G.degree()]
degree_m = np.zeros((G.number_of_nodes(),G.number_of_nodes()))
np.fill_diagonal(degree_m, degree_v)
# Normalized Laplacian
from scipy.linalg import fractional_matrix_power
L = fractional_matrix_power(degree_m, -1/2) @ A @ fractional_matrix_power(degree_m,

# Create input vectors
X = np.identity(G.number_of_nodes())
### TODO your code here

X = torch.tensor(X, dtype=torch.float, requires_grad=True) # Indicate to pytorch th
As = torch.tensor(A, dtype=torch.float)
L = torch.tensor(L, dtype=torch.float)  # We don't need to learn this so no grad re
```

In [ ]:
```
import torch
import torch.optim
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms

import networkx as nx
import numpy as np

%matplotlib inline
import matplotlib.pyplot as plt

# Define a GCN
class GCNLayer(nn.Module):
    def __init__(self, L, input_features, output_features, activation=F.relu):
        """
            Inputs:
                L:               The "Laplacian" of the graph, as defined above
                input_features:  The size of the input embedding
                output_features: The size of the output embedding
```

```
                activation:      Activation function sigma
        """
        super().__init__()

        ### TODO Your code here
        self.L = L
        self.layer = nn.Linear(input_features, output_features)
        self.activation = activation
        ### TODO Your code here

    def forward(self, X):
        ### TODO Your code here
        if (self.activation):

            X = self.activation(self.L @ self.layer(X))
        else:
            X = self.L @ self.layer(X)
        ### TODO Your code here
        return X
```

Define the modularity matrix and the modularity loss

In [ ]:
```
def modularity_matrix(A):
    n, _ = A.shape
    B = np.empty((n, n))
    ### YOUR CODE HERE
    for i in range(n):
        for j in range(n):
            B[i,j] = A[i,j] - ((G.degree[i]*G.degree[j]) / (2*G.number_of_edges()))
    ### YOUR CODE HERE
    return torch.tensor(B, dtype=torch.float)

def modularity_loss(C, B, l = 0.01):
    ''' Return the modularity loss

        Args:
            C: the node-community affinity matrix
            B: the modularity matrix
            l: the regularization factor

        :return the modularity loss as described at the beginning of the exercise
    '''
    loss = 0
    ### YOUR CODE HERE
    loss += - torch.trace(C.T @ B @ C) + l * torch.norm(C, p='fro')
    ### YOUR CODE HERE
    return loss
```

Compute labels from communities

In [ ]:
```
### Compute labels from communities
labels = None
### YOUR CODE HERE
labels = [d['community'] for i,d in G.nodes().data()]
```

```
### YOUR CODE HERE
```

Create the model

In [ ]:
```
from sklearn.preprocessing import LabelEncoder
import torch.nn.functional as F

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

### Encode the labels with one-hot encoding
def to_categorical(y):
    """ 1-hot encodes a tensor """
    num_classes = np.unique(y).size
    return np.eye(num_classes, dtype='uint8')[y]

def encode_label(labels):
    label_encoder = LabelEncoder()
    labels = label_encoder.fit_transform(labels)
    labels = to_categorical(labels)
    return labels, label_encoder.classes_

y, classes = encode_label(labels)
y = torch.tensor(y)

# Define convolutional network
in_features, out_features = X.shape[1], classes.size # output features as many as t
hidden_dim = 16

# Stack two GCN layers as our model
# nn.Sequential is an implicit nn.Module, which uses the layers in given order as t
gcn = nn.Sequential(
    GCNLayer(L, in_features, hidden_dim),
    GCNLayer(L, hidden_dim, out_features, None),
    nn.Softmax(dim=1)
)
gcn.to(device)
```

Out[ ]:
```
Sequential(
  (0): GCNLayer(
    (layer): Linear(in_features=156, out_features=16, bias=True)
  )
  (1): GCNLayer(
    (layer): Linear(in_features=16, out_features=6, bias=True)
  )
  (2): Softmax(dim=1)
)
```

Train the unsupervised model once

In [ ]:
```
l = 100
epochs = 2000

def train_model(model, optimizer, X, B, epochs=100, print_every=10, batch_size = 2)
    for epoch in range(epochs+1):
```

```
        y_pred = model(X)
        loss = modularity_loss(y_pred, B, l=l)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if epoch % print_every == 0:
            print(f'Epoch {epoch:2d}, loss={loss.item():.5f}')

B = modularity_matrix(A)
optimizer = torch.optim.Adam(gcn.parameters(), lr=0.01)
train_model(gcn, optimizer, X, B, epochs=epochs, print_every=100)
```

```
Epoch  0, loss=491.28857
Epoch 100, loss=72.33704
Epoch 200, loss=-72.52649
Epoch 300, loss=-83.15662
Epoch 400, loss=-85.74866
Epoch 500, loss=-86.76086
Epoch 600, loss=-87.33752
Epoch 700, loss=-87.86841
Epoch 800, loss=-88.57532
Epoch 900, loss=-88.84766
Epoch 1000, loss=-89.07080
Epoch 1100, loss=-89.26965
Epoch 1200, loss=-89.75439
Epoch 1300, loss=-89.93359
Epoch 1400, loss=-90.02722
Epoch 1500, loss=-90.07678
Epoch 1600, loss=-90.11475
Epoch 1700, loss=-90.14514
Epoch 1800, loss=-90.17029
Epoch 1900, loss=-90.19165
Epoch 2000, loss=-90.20996
```

Evaluate your model using NMI. Since the initialization is random train the model 10 times and take the average NMI. Assign each node to the community with the highest probability. You should obtain an Average $\text{NMI} \approx 0.5$.

Plot the last graph with the nodes colored by commnitieis communities using `plot_graph` below.

Note: You have to create the model 5 times otherwise you are keeping training the same model's parameters!

In [ ]:
```
from sklearn.metrics.cluster import normalized_mutual_info_score

def plot_graph(G, y_pred):
    plt.figure(1,figsize=(15,5))
    pos = nx.spring_layout(G)
    ec = nx.draw_networkx_edges(G, pos, alpha=0.2)
    nc = nx.draw_networkx_nodes(G, pos, nodelist=G.nodes(), node_color=y_pred, node
    plt.axis('off')
    plt.show()
```

```
### YOUR CODE STARTS HERE
nmi = 0
B = modularity_matrix(A)
y_pred_final = None
for i in range(10):
    gcn = nn.Sequential(
    GCNLayer(L, in_features, hidden_dim),
    GCNLayer(L, hidden_dim, out_features, None),
    nn.Softmax(dim=1)
    )
    gcn.to(device)


    optimizer = torch.optim.Adam(gcn.parameters(), lr=0.01)
    train_model(gcn, optimizer, X, B, epochs=epochs, print_every=100)
    y_pred = torch.argmax(gcn(X), dim=1).numpy()
    y_pred_final = y_pred
    nmi += normalized_mutual_info_score(y_pred, labels)
nmi = nmi / 10.

print("Average NMI ", nmi)
plot_graph(G, y_pred_final)

# y_pred = torch.argmax(gcn(X), dim=1).numpy()
# print(normalized_mutual_info_score(y_pred, labels))
# plot_graph(G, labels)
# plot_graph(G,y_pred)

### YOUR CODE ENDS HERE
```

```
Epoch  0, loss=489.87927
Epoch 100, loss=177.01642
Epoch 200, loss=-0.71143
Epoch 300, loss=-77.31445
Epoch 400, loss=-82.01514
Epoch 500, loss=-84.29333
Epoch 600, loss=-86.31470
Epoch 700, loss=-87.14661
Epoch 800, loss=-87.51587
Epoch 900, loss=-87.77966
Epoch 1000, loss=-87.96948
Epoch 1100, loss=-88.13635
Epoch 1200, loss=-88.53503
Epoch 1300, loss=-88.80859
Epoch 1400, loss=-89.39465
Epoch 1500, loss=-89.56995
Epoch 1600, loss=-89.71729
Epoch 1700, loss=-89.83374
Epoch 1800, loss=-89.90662
Epoch 1900, loss=-89.95081
Epoch 2000, loss=-89.98328
Epoch  0, loss=486.12244
Epoch 100, loss=191.46838
Epoch 200, loss=158.32129
Epoch 300, loss=148.73828
Epoch 400, loss=26.99774
Epoch 500, loss=-73.70728
Epoch 600, loss=-81.25598
Epoch 700, loss=-84.20630
Epoch 800, loss=-85.55237
Epoch 900, loss=-86.84705
Epoch 1000, loss=-87.47815
Epoch 1100, loss=-88.04578
Epoch 1200, loss=-88.60559
Epoch 1300, loss=-88.97351
Epoch 1400, loss=-89.26257
Epoch 1500, loss=-89.49084
Epoch 1600, loss=-89.66528
Epoch 1700, loss=-89.80090
Epoch 1800, loss=-89.90088
Epoch 1900, loss=-89.97656
Epoch 2000, loss=-90.03540
Epoch  0, loss=490.57452
Epoch 100, loss=103.85834
Epoch 200, loss=-74.92456
Epoch 300, loss=-79.31445
Epoch 400, loss=-81.99646
Epoch 500, loss=-84.72119
Epoch 600, loss=-86.24341
Epoch 700, loss=-86.99548
Epoch 800, loss=-87.46777
Epoch 900, loss=-87.74890
Epoch 1000, loss=-88.00159
Epoch 1100, loss=-88.43396
Epoch 1200, loss=-88.58960
Epoch 1300, loss=-88.74829
```

```
Epoch 1400, loss=-89.03149
Epoch 1500, loss=-89.46143
Epoch 1600, loss=-89.54736
Epoch 1700, loss=-89.64758
Epoch 1800, loss=-89.77820
Epoch 1900, loss=-89.86682
Epoch 2000, loss=-89.93018
Epoch  0, loss=485.17538
Epoch 100, loss=278.31543
Epoch 200, loss=-71.18616
Epoch 300, loss=-77.47314
Epoch 400, loss=-79.61780
Epoch 500, loss=-81.62549
Epoch 600, loss=-83.27161
Epoch 700, loss=-84.18347
Epoch 800, loss=-84.77502
Epoch 900, loss=-85.23560
Epoch 1000, loss=-85.60181
Epoch 1100, loss=-85.86475
Epoch 1200, loss=-86.09619
Epoch 1300, loss=-86.28735
Epoch 1400, loss=-87.23132
Epoch 1500, loss=-87.89844
Epoch 1600, loss=-88.06030
Epoch 1700, loss=-88.19482
Epoch 1800, loss=-88.35840
Epoch 1900, loss=-88.91907
Epoch 2000, loss=-89.03687
Epoch  0, loss=489.93597
Epoch 100, loss=470.47885
Epoch 200, loss=64.20239
Epoch 300, loss=-19.80371
Epoch 400, loss=-62.43506
Epoch 500, loss=-70.31580
Epoch 600, loss=-74.50378
Epoch 700, loss=-76.94946
Epoch 800, loss=-78.96960
Epoch 900, loss=-80.44202
Epoch 1000, loss=-81.56848
Epoch 1100, loss=-82.62048
Epoch 1200, loss=-83.38818
Epoch 1300, loss=-84.06580
Epoch 1400, loss=-84.53809
Epoch 1500, loss=-84.87976
Epoch 1600, loss=-85.16968
Epoch 1700, loss=-85.45752
Epoch 1800, loss=-85.72607
Epoch 1900, loss=-85.90979
Epoch 2000, loss=-86.06738
Epoch  0, loss=487.68817
Epoch 100, loss=24.06763
Epoch 200, loss=-79.11926
Epoch 300, loss=-82.33533
Epoch 400, loss=-84.20667
Epoch 500, loss=-85.21021
Epoch 600, loss=-86.45667
```

```
Epoch 700, loss=-87.20825
Epoch 800, loss=-87.84338
Epoch 900, loss=-88.18799
Epoch 1000, loss=-88.38916
Epoch 1100, loss=-88.57507
Epoch 1200, loss=-88.91821
Epoch 1300, loss=-89.34778
Epoch 1400, loss=-89.83545
Epoch 1500, loss=-89.92896
Epoch 1600, loss=-89.98767
Epoch 1700, loss=-90.03223
Epoch 1800, loss=-90.07190
Epoch 1900, loss=-90.12769
Epoch 2000, loss=-90.16797
Epoch  0, loss=487.33218
Epoch 100, loss=348.36667
Epoch 200, loss=315.69336
Epoch 300, loss=-148.36572
Epoch 400, loss=-223.09766
Epoch 500, loss=-224.46387
Epoch 600, loss=-225.02124
Epoch 700, loss=-228.13025
Epoch 800, loss=-232.36023
Epoch 900, loss=-233.01697
Epoch 1000, loss=-233.68921
Epoch 1100, loss=-234.21094
Epoch 1200, loss=-234.69702
Epoch 1300, loss=-235.04163
Epoch 1400, loss=-235.31238
Epoch 1500, loss=-235.47717
Epoch 1600, loss=-235.63086
Epoch 1700, loss=-235.77881
Epoch 1800, loss=-235.84631
Epoch 1900, loss=-235.93164
Epoch 2000, loss=-236.27283
Epoch  0, loss=488.12427
Epoch 100, loss=184.39209
Epoch 200, loss=-36.40369
Epoch 300, loss=-78.85193
Epoch 400, loss=-81.62488
Epoch 500, loss=-83.21790
Epoch 600, loss=-84.42737
Epoch 700, loss=-85.38171
Epoch 800, loss=-86.03723
Epoch 900, loss=-86.48352
Epoch 1000, loss=-86.86340
Epoch 1100, loss=-87.24109
Epoch 1200, loss=-87.50806
Epoch 1300, loss=-87.70996
Epoch 1400, loss=-87.88611
Epoch 1500, loss=-88.04443
Epoch 1600, loss=-88.19641
Epoch 1700, loss=-88.32446
Epoch 1800, loss=-88.41479
Epoch 1900, loss=-88.48279
Epoch 2000, loss=-88.53577
```

```
Epoch  0, loss=487.35614
Epoch 100, loss=153.31689
Epoch 200, loss=-223.01733
Epoch 300, loss=-228.56812
Epoch 400, loss=-230.15918
Epoch 500, loss=-230.96191
Epoch 600, loss=-231.56677
Epoch 700, loss=-232.11255
Epoch 800, loss=-232.74268
Epoch 900, loss=-233.09644
Epoch 1000, loss=-233.34961
Epoch 1100, loss=-233.62769
Epoch 1200, loss=-234.11780
Epoch 1300, loss=-234.46936
Epoch 1400, loss=-234.95105
Epoch 1500, loss=-235.15625
Epoch 1600, loss=-235.31885
Epoch 1700, loss=-235.46887
Epoch 1800, loss=-235.64160
Epoch 1900, loss=-235.86230
Epoch 2000, loss=-236.04150
Epoch  0, loss=487.14621
Epoch 100, loss=228.91681
Epoch 200, loss=156.57837
Epoch 300, loss=145.87720
Epoch 400, loss=16.49365
Epoch 500, loss=-74.71863
Epoch 600, loss=-82.56580
Epoch 700, loss=-85.22607
Epoch 800, loss=-86.54468
Epoch 900, loss=-87.44055
Epoch 1000, loss=-88.06287
Epoch 1100, loss=-88.56750
Epoch 1200, loss=-88.90283
Epoch 1300, loss=-89.15540
Epoch 1400, loss=-89.42200
Epoch 1500, loss=-89.81116
Epoch 1600, loss=-89.93457
Epoch 1700, loss=-90.01465
Epoch 1800, loss=-90.07263
Epoch 1900, loss=-90.12500
Epoch 2000, loss=-90.16101
Average NMI  0.706010883961978
```