Data Mining - Handin 2 - Graph mining

This handin corresponds to the topics in Week 11-15 in the course.

The handin is

- done in 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

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 9 points from the first three Parts (Part 1,2,3) and 3 points in the last part (Part 4). Good luck!

The exercise types: There are four 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 1-5 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. **[Prove]** means to provide a formal argument and NO code.
4. **[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!

Q&A

Q: If the task is to implement a mean function, may I just call `np.mean()`?
A: No.

Q: If the task is to compare the mean of X and Y, may I use `np.mean()` to calculate the mean?
A: Yes.

Q: If I have implemented a mean function in a previous task, but I am unsure of its correctness, may I use `np.mean()` in following task where mean is used as a helper function?
A: Yes.

Q: May I use `np.mean()` to debug my implementation of mean?
A: Yes.

Q: Do I get 0 points for a task if I skip it?
A: Yes.

Q: Can I get partial points for a task I did partially correct?
A: Yes.

Q: Is it OK to skip a task if I do not need the points from it?

A: Yes.

Q: Should I inform a TA if I find an error?

A: Yes.

Q: Should I ask questions if I am confused?

A: Yes.

Good luck!

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
# !{sys.executable} -m pip install torch
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
import pickle


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\sofie\anaconda3\envs\dm24\lib\site-packages\torchvision\io\image.py:13: UserWarning: Failed to load image Python extension: '[WinError 127] The specified procedure could not be found'If you don't plan on using image functionality from `torchvision.io`, you can ignore this warning. Otherwise, there might be something wrong with your environment. Did you have `libjpeg` or `libpng` installed before building `torchvision` from source?
  warn(
c:\Users\sofie\OneDrive\Skrivebord\Uni\F24\Data Mining\dm2024-exercises-main\handins
```

Task 1.1 Random walks and PageRank (10 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.

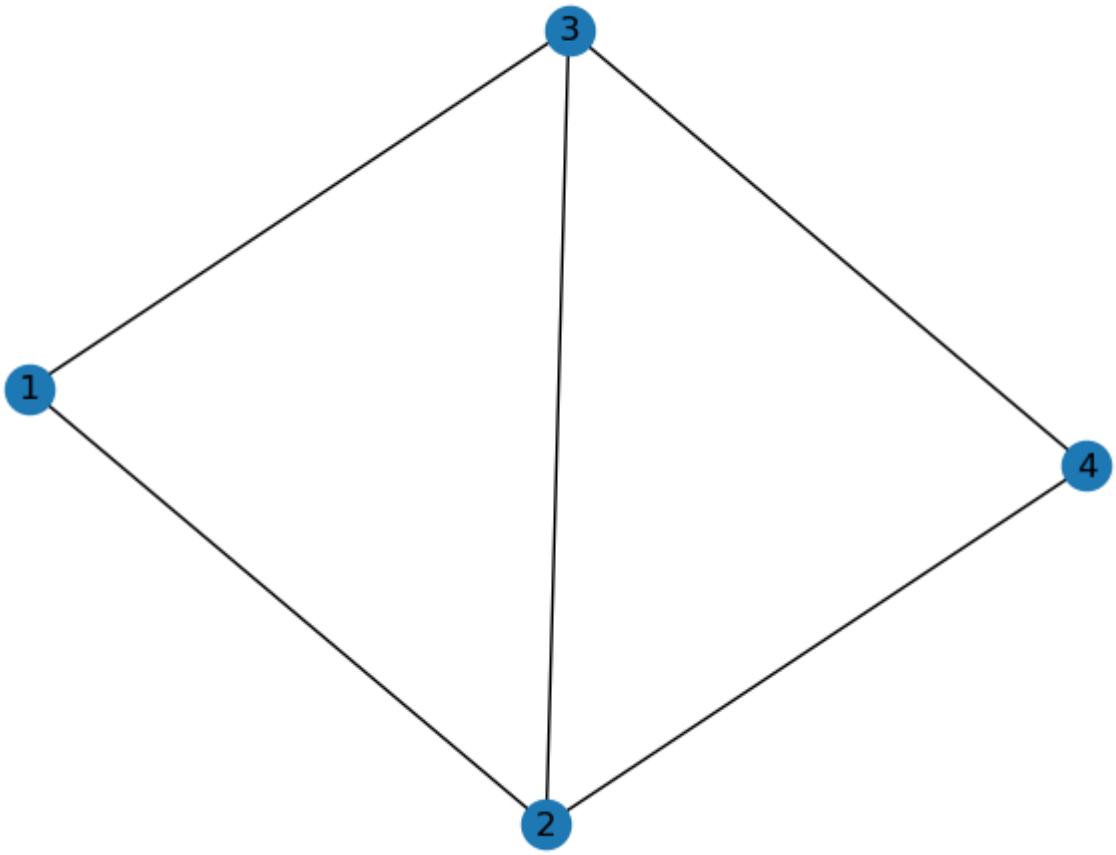
With an α value of 1, the page Rank is simply given by $\mathbf{r} = \mathbf{M}\mathbf{r}$. As per the definition of PageRank, the PageRank of a page is the sum of the PageRank of the pages that links to it. Since it is a d-regular graph, all nodes would have an equal amount of connections/links, and therefore would end up with the same PageRank score. The PageRank becomes evenly distributed for all nodes with a score of $\frac{1}{n}$

Task 1.1.2 (4 points)

Look at the graph below (run the code) and try make a guess about the PageRank values of each node by only looking the graph.

$$\mathbf{r} = \begin{pmatrix} \frac{1}{6} \\ \frac{2}{6} \\ \frac{2}{6} \\ \frac{1}{6} \end{pmatrix}$$

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 two iterations.

C) [Motivate] Do you observe a constanct decrease of the norm square difference as iterations are increasing, and is it 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 eigenvalues match with your first guess? Can you see a pattern between the eigenvalues of each node and its edges?

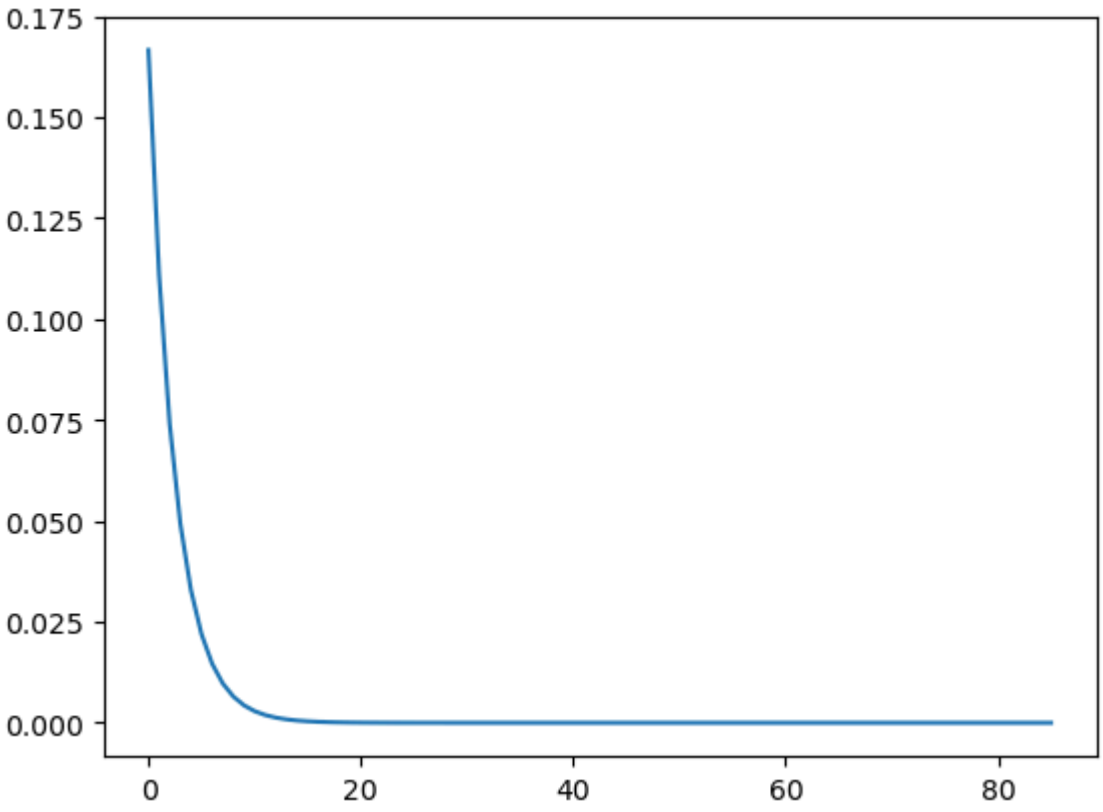
In [ ]:
```
#A)

M = np.array([[0, 1/3, 1/3, 0],
              [1/2, 0, 1/3, 1/2],
              [1/2, 1/3, 0, 1/2],
              [0, 1/3, 1/3, 0]])


def PageRankPowerIteration(M, tol=1e-16):
    n, _ = M.shape
    r0 = np.full(n, 1/n)

    dist = []

    while True:
        r = np.einsum('ij,j', M, r0)
        dist += [np.linalg.norm(r0-r)]
```

```
        if np.allclose(r0, r, atol=tol, rtol=tol):
            return r, dist
        r0 = r


pageRank, L = PageRankPowerIteration(M)
pageRank
```

Out[ ]: array([0.2, 0.3, 0.3, 0.2])

In [ ]:
```
#B)
plt.plot(range(len(L)), L)
```

Out[ ]: [<matplotlib.lines.Line2D at 0x27be4ae7310>]



C)

As seen on the graph above, the decrease is not constant, as expected.

In [ ]:
```
#D) YOUR CODE HERE
def PageRankEigen(M):
    val, vec = np.linalg.eig(M)
    rank = vec[:,0].T # select dominant eigenvector
    return rank/np.sum(rank) # normalize


PageRankEigen(M)
```

Out[ ]: array([0.2, 0.3, 0.3, 0.2])

E)

The achieved PageRank from both methods is the same. However, for very large graphs it can be very computationally expensive to find the eigenvectors for the large matrices. And as seen on the previous graph, power iteration starts toconverges after relativlely few iterations, and is more scaleble. Moreover, power iteration can be updated incrementally, whereas the eigenvector approach requires re-computing the eigenvectors once again.

F)

The PageRank does not match the initial guess. However, it can be seen that the PageRank is proportional to how many edges each node is connected to out of the total amount of 'connections' (ie, how many connections between node and edge - meaning two for each edge and ten in total for the given graph). For example node 3 has three connections out of ten in total, and hence has a PageRank of 0.3.

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 $d = 1$?

- ☐ 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 havee the same embedding and 2,3 will have very close embeddings.
- ☐ Nodes 3,4 will be very far apart.

In [ ]:
```
# Laplacian matrix
L = np.array([[ 2, -1, -1,  0],
              [-1,  3, -1, -1],
              [-1, -1,  3, -1],
              [ 0, -1, -1,  2]])
print(L)
```

```
[[ 2 -1 -1  0]
 [-1  3 -1 -1]
 [-1 -1  3 -1]
 [ 0 -1 -1  2]]
```

In [ ]:
```
np.linalg.eig(L)[1][:,1]
```

Out[ ]:
```
array([ 7.07106781e-01, -5.85177798e-17,  1.53094029e-16, -7.07106781e-01])
```

The entries of the (un)normalized laplacian represents the strength of connection betweeen two nodes; since 2 and 3 are connected and have the same 'community' (ie., connected to the same nodes), these would be expected to have the same embedding, whereas 1 and 4 aren't connected, and would be far apart.

Task 1.2: Spectral Properties of the Graph Laplacian (10 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 noted as:
$\lambda_0 \leq \lambda_1 \leq \ldots \leq \lambda_{n-1}$

Task 1.2.1 (2 points)

For all graphs $\lambda_0 = 0$

Let $v = (1, 1, \ldots, 1)^T$ be vector of length n consiting of all the ones (or just same value k).

In this case Dv is a vector where each entry is the degree of the corresponding node (times k), and Av is a vector with the sum of neighbors for each node (times k). I.e $Dv - Av = (0, 0, \ldots, 0)^T$.

Thus, it follows that $Lv = Dv - Av = (0, 0, \ldots, 0)^T = 0 \cdot v$.

Meaning that 0 is always an eigenvalue of L. Futhermore since L is symetric, all eigenvalues are ≥ 0 (c.f l.6 s.26), and thus $\lambda_0 = 0$ for all graphs.

Task 1.2.2 (2 points)

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

For the complete graph every node is connected to every other node, meaning that every value in the laplacian is -1 exept for the diagonal, where it is $n - 1$.

Therefore, for an arbitray $v = (v_0, \ldots, v_{n-1})^T$, the i'th index of Lv is

$$(Lv)_i = (n-1)v_i - \sum_{j \neq i} v_j = nv_i - v_i - \sum_{j \neq i} v_j = nv_i - \sum_j v_j$$

If v is an eigenvector of L with the corresponding eigenvalue λ then we have:

$$Lv = n \cdot v - \sum_j v_j \cdot (1, 1, \ldots, 1)^T = \lambda \cdot v$$

$$\Updownarrow$$

$$nv - \lambda v = (n - \lambda)v = \sum_j v_j \cdot (1, 1, \ldots, 1)^T$$

We see that this only holds in two cases:

1. if all indexes of v are equal, corresponding to an eigenvalue of 0 (as seen in 1.2.1),
2. If $\lambda = n$ and $\sum_j v_j = 0$

If $\sum_j v_j = 0$ it is also simple to see that

$$Lv = n \cdot v - \sum_j v_j \cdot (1, 1, \ldots, 1)^T = n \cdot v$$

(disregarding scaling) there can only be one eigenvector, in this setup, with eigenvalue 0, the rest $n-1$ orthogonal eigenvectors must therefore be zero-sum and have corresponding eigenvalues $\lambda_1, \ldots, \lambda_{n-1} = n$

Task 1.2.3 (2 points)

For all the graphs with k connected components $\lambda_0 = \lambda_1 = \ldots = \lambda_{k-1} = 0$

Each of the k connected components form a subgraph where, (as shown in 1.2.1), the vector of all ones has a corrsponding eigen value of 0.

In relation to the full graph, this corresponds to a vector $v^{(i)}$, for each of the k connected components, with ones in the indices corresponding to the nodes of the i'th component and 0 elsewhere.

$v^i \cdot v^j = 0$ for $i \neq j$. Therefore, there are k orthogonal vectors (one for each connected component) with a corresponding eigenvalue of 0

I.e. for a graph with k connected components, we have $\lambda_0 = \lambda_1 = \ldots = \lambda_{k-1} = 0$

Task 1.2.4 (2 points)

Given a graph G with eigenvalues of the laplacian $\lambda_0, \lambda_1, \ldots, \lambda_{n-1}$.
We remove a single 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 can not have $\lambda'_i > \lambda_i$ by removing an edge. In stead, cf. lecture 6 slide 28, we have $\lambda_i \geq \lambda'_i$.

Task 1.2.5 (2 points)

Suppose that the graph G is consisted of two connected componentes of equal size named as G_1 and G_2.
The Laplacian of G_1 has eigenvalues $\lambda^1_0, \lambda^1_1, \ldots, \lambda^1_{n/2-1}$.
The Laplacian of G_2 has eigenvalues $\lambda^2_0, \lambda^2_1, \ldots, \lambda^2_{n/2-1}$.
Prove that the Laplacian of G is consisted of the eigenvalues of the Laplacians of G_1 and G_2 in ascending order.

This can be easily shown, by considering that the rows of the laplacian, each representing a node, are not ordered in any certain way. Thus, we can arrange it in such a way that

$$L = \begin{pmatrix} L_1 & 0 \\ 0 & L_2 \end{pmatrix}$$

where L_i is the laplacian coreesponding to G_i.

It is now easy to see that the eigenvalues of L are exactly those of G_1 and G_2

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 biological data https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1003268.

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 (L2) distance.

Note: Be sure that your constructed graphs does not contain loop edges (edges from i to i for some node i)

In [ ]:
```
#YOUR CODE HERE
# Be sure that your constructed graphs does not
# contain loop edges (edges from i to i for some node i)
from sklearn.neighbors import NearestNeighbors
def dist(x_i, x_j):
    return np.sqrt(np.sum((x_i-x_j)**2))

def nn_graph(data, eps, remove_self=True, directed=False):
    n = len(data)
    G = nx.Graph()
    if directed:
        G = nx.DiGraph()
    ### YOUR CODE HERE
    for i, x_i in enumerate(data):
        G.add_node(i) # add node
        for j, x_j in enumerate(data):
            if (j != i) & (dist(x_i,x_j) < eps): # no loop edges
                G.add_edge(i, j) # add edge
    ### YOUR CODE HERE
    return G
```

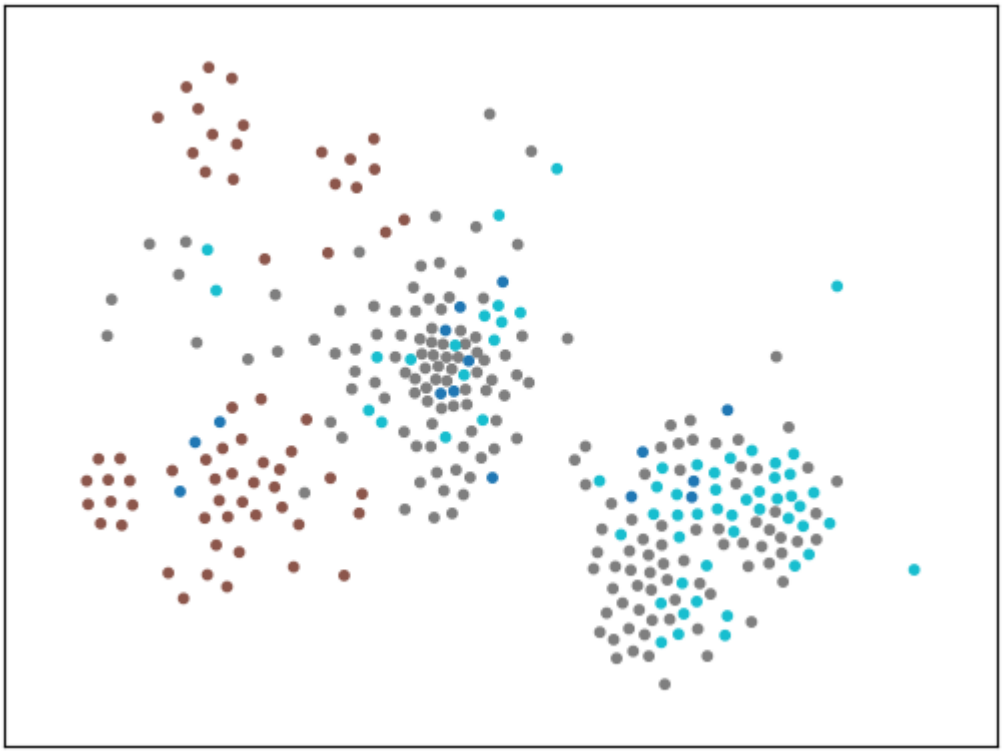
Task 2.1.2 (2 points)

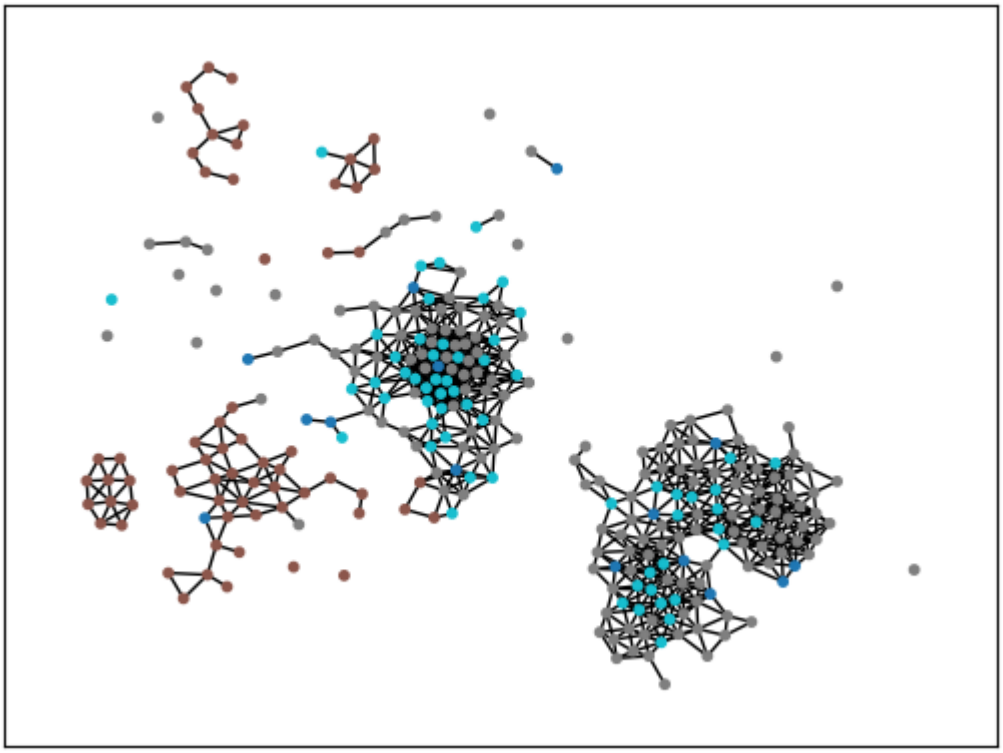
Try with different epsilons (select a small set of epsilons, e.g., 0.01-0.5 values) and plot the graphs.

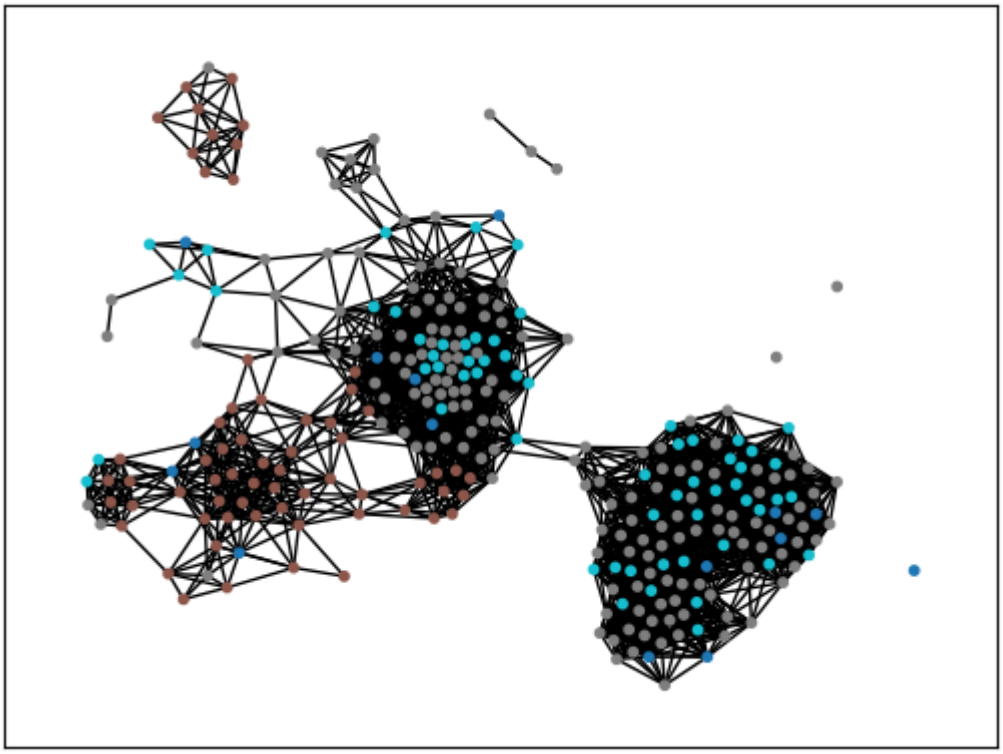
[Motivate] what you observe as epsilon increases.

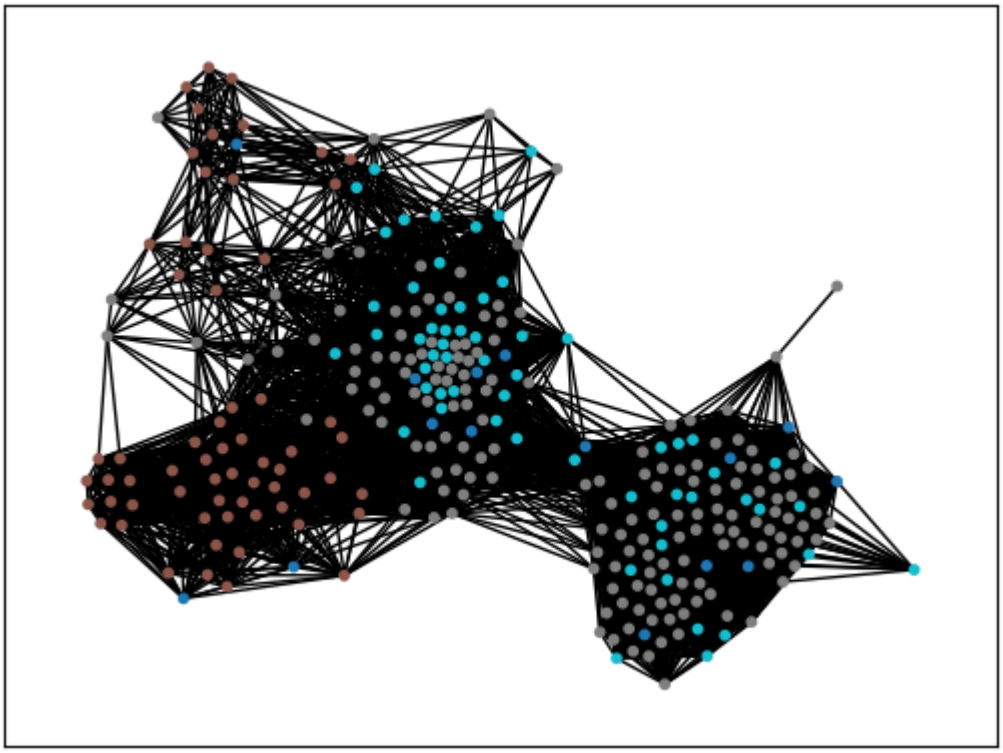
In [ ]:
```
### Run the code below
eps_values = [0.01, 0.05, 0.1, 0.2, 0.4]

for eps in eps_values:
    ax=plt.subplot()
```

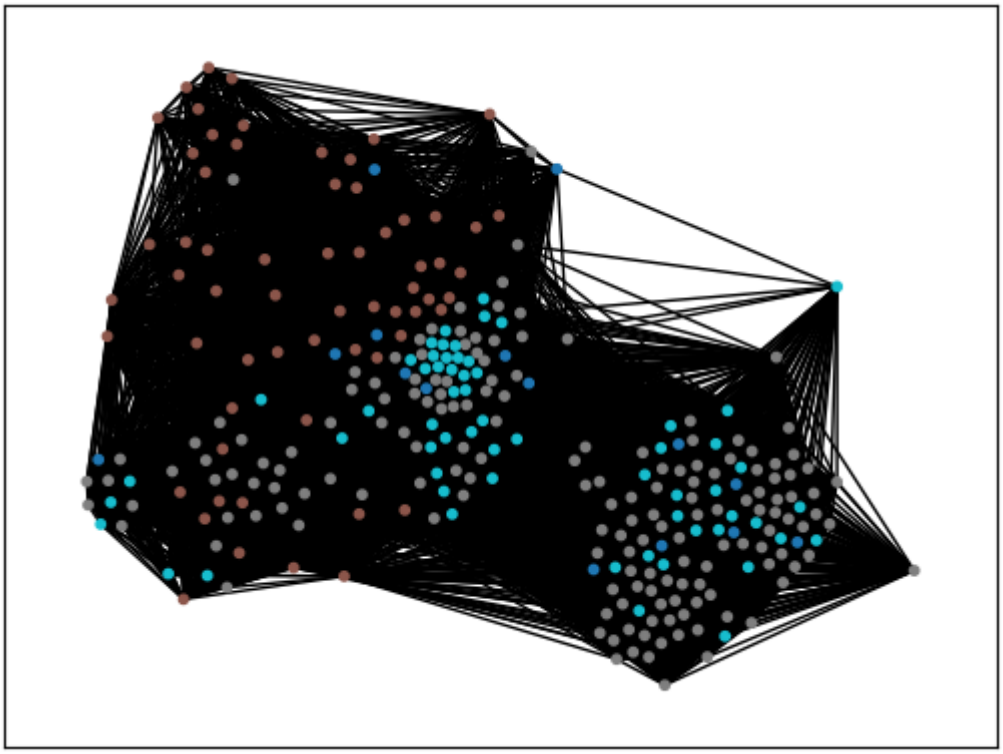
```
    ax1=plt.subplot()
    G = nn_graph(X, eps)
    #G1=nx.numpyto
    pos=nx.spring_layout(G)
    nx.draw_networkx_edges(G,pos=X)
    nx.draw_networkx_nodes(G, pos=X, node_color=Y, node_size=10, cmap=plt.get_cmap('tab10'))
    ax.set_xlim(-0.1, 1.1)
    ax.set_ylim(-0.1, 1.1)
    plt.show()
```











The ε-neighborhood graph is constructed by adding an edge between any node within radius ϵ of another node. So it makes sense what we see in the above plots: that as we increase the radius, more edges start to appear.

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.

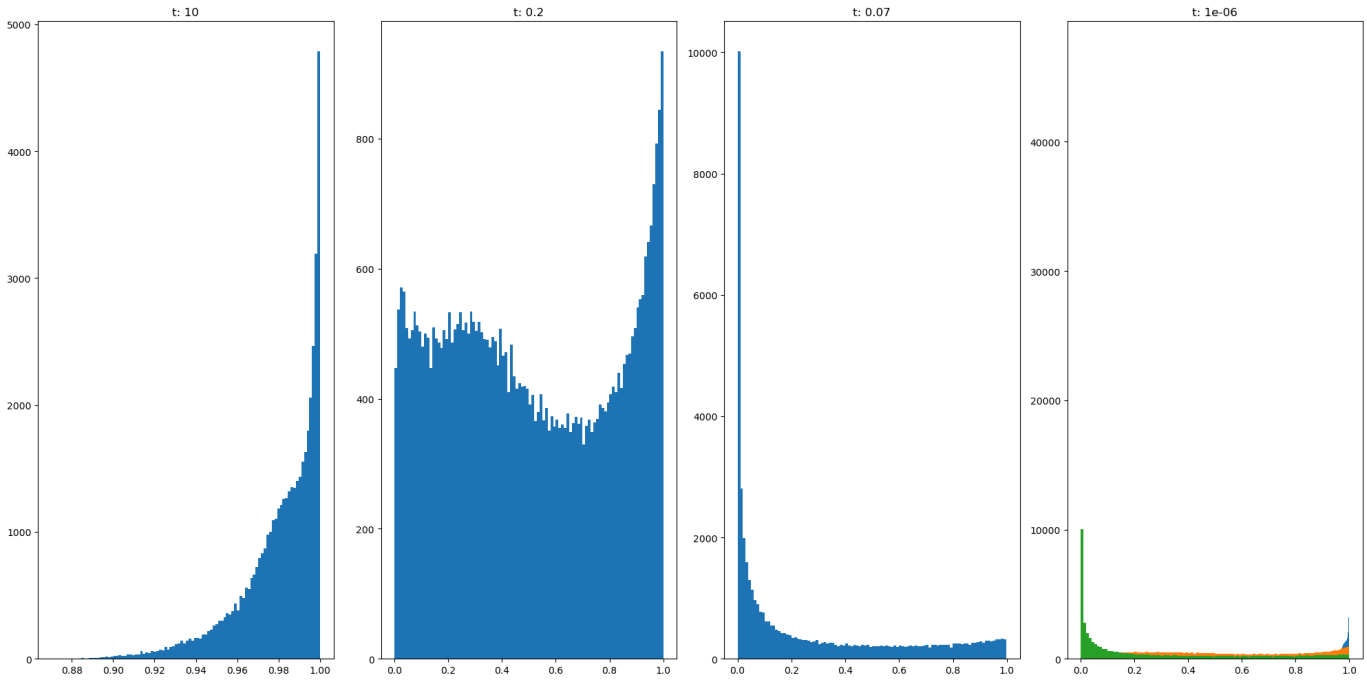
In [ ]:
```python
def weighted_nn_graph(data, eps=20, t=0.1):
    n = len(data)
    G = nx.Graph()
    ### YOUR CODE HERE
    def w(x_i, x_j, t): # weight function
        distance_squared = dist(x_i,x_j)**2
        return np.exp(-distance_squared/t)

    for i, x_i in enumerate(data):
        G.add_node(i) # add node
        for j, x_j in enumerate(data):
            if (j != i) & (dist(x_i,x_j) < eps): # no loop edges
                G.add_edge(i, j, weight= w(x_i,x_j,t)) # add weighted edge
    ### YOUR CODE HERE
    return G
```

Task 2.1.4 (2 points)

We now vary $t \in \{10, 0.1, 0.000001\}$. Plot the weights as a histogram 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()
```



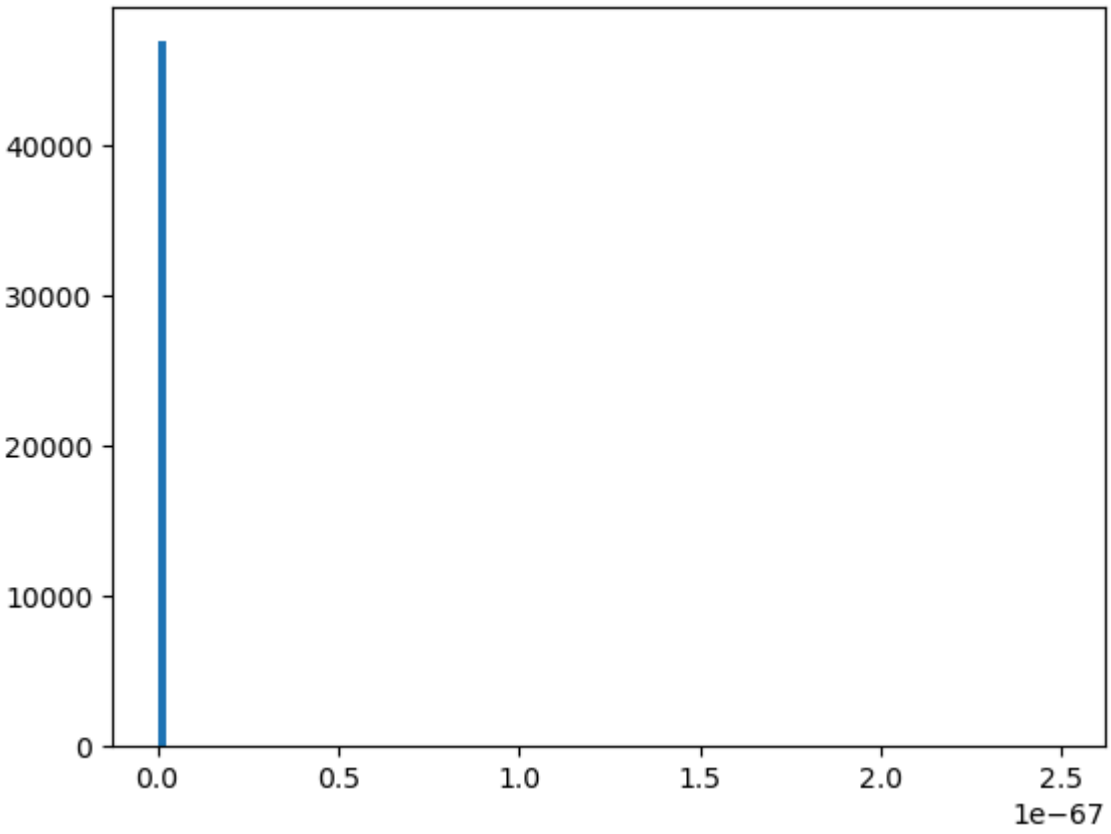
Taking a look at the weight function we see that:

1. when t is large compared to the squared distance the exponent becomes bigger, making the weight approach $e^0 = 1$ for $t \to \infty$
2. when t is small compared to the squared distance the exponent becomes smaller, making the weight approach $e^{-\infty} = 0$ for $t \to 0$
3. $t = 0$ is undefined, but all weights for $t = 0.000001$ should be (and are) very close to zero, so I don't know why the last histogram looks like it does, but I have some theories: It could just be due to numerical errors, rounding, under-/overflow, etc. On closer inspection it looks like the ealier three graphs have somehow been superimposed on the last histogram so I'm guessing it is a problem with the provided code.

Last plot should be like this:

In [ ]:
```
G = weighted_nn_graph(X, eps=60, t=0.000001)
ys = []
for i, d in enumerate(G.edges.data()):
    ys.append(d[2]['weight'])
len(ys)
```

```
plt.hist(ys, bins=100)
plt.show()
```



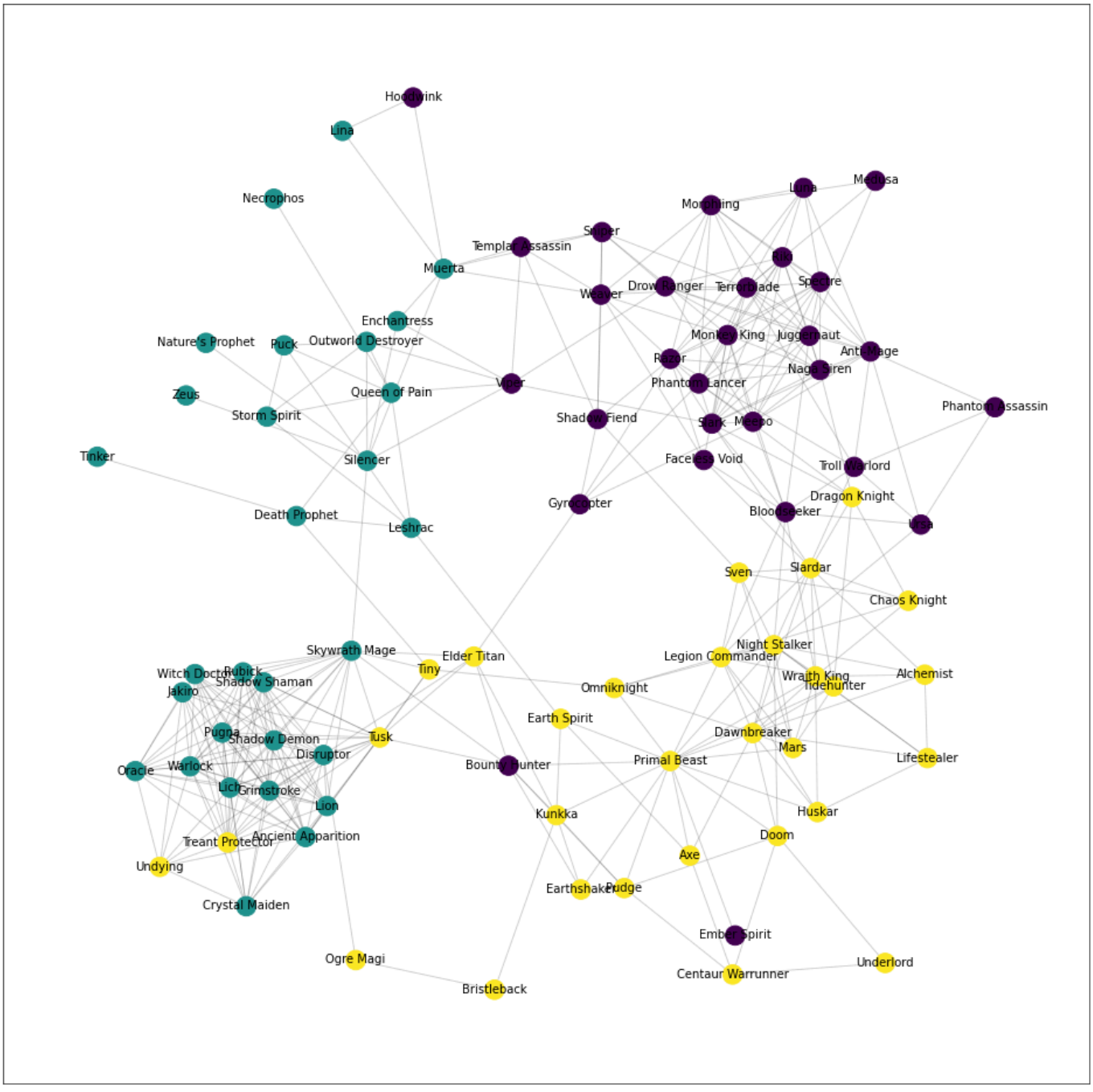
Task 2.2: Spectral clustering (20 points)

In [ ]:
```
G = pickle.load(open('./data/dota2graph.pickle', 'rb'))
groundTruth = pickle.load(open('./data/groundtruth.txt','rb'))
fig = plt.figure(1, figsize=(20, 20), dpi=60)
#nx.draw(G, pos=layout,with_labels = True, node_size=50, node_color=your_clusters,font_size=8,
#layout = nx.spring_layout(H, k=5.15, iterations=20)
layout = nx.kamada_kawai_layout(G)
nx.draw_networkx_edges(G, layout, alpha = 0.2)
nx.draw_networkx_nodes(G, layout, node_color=groundTruth, node_size=400)
nx.draw_networkx_labels(G, layout)
```

Out[ ]:

```
{'Anti-Mage': Text(0.6823282264647765, 0.46083622203208624, 'Anti-Mage'),
 'Axe': Text(0.2927103549777464, -0.6091356781088342, 'Axe'),
 'Bloodseeker': Text(0.49932034614311965, 0.12005720260751625, 'Bloodseeker'),
 'Crystal Maiden': Text(-0.6644615474044729, -0.7165678934839822, 'Crystal Maiden'),
 'Drow Ranger': Text(0.23947685649531614, 0.5992418150222546, 'Drow Ranger'),
 'Earthshaker': Text(0.05779092482210867, -0.6810280295347144, 'Earthshaker'),
 'Juggernaut': Text(0.550705650301141, 0.4942244317739438, 'Juggernaut'),
 'Morphling': Text(0.33854241950397274, 0.7713815704594219, 'Morphling'),
 'Shadow Fiend': Text(0.09351558038731475, 0.31744085022496743, 'Shadow Fiend'),
 'Phantom Lancer': Text(0.31193674179620856, 0.3931750665206623, 'Phantom Lancer'),
 'Puck': Text(-0.5827152135135053, 0.47388309149142555, 'Puck'),
 'Pudge': Text(0.15125862210620977, -0.6780388952423565, 'Pudge'),
 'Razor': Text(0.25072379085117436, 0.44523421621492715, 'Razor'),
 'Storm Spirit': Text(-0.6200370625488645, 0.3226659779655371, 'Storm Spirit'),
 'Sven': Text(0.39888256088656593, -0.008730950289357484, 'Sven'),
 'Tiny': Text(-0.2697078655718352, -0.21401790322695283, 'Tiny'),
 'Zeus': Text(-0.7942656516194662, 0.36757737285991243, 'Zeus'),
 'Kunkka': Text(0.0048188657801527075, -0.5232847048208314, 'Kunkka'),
 'Lina': Text(-0.4566102297385862, 0.9290698415078328, 'Lina'),
 'Lion': Text(-0.4901989313786442, -0.5037785532869871, 'Lion'),
 'Shadow Shaman': Text(-0.6268862247136514, -0.24129714104605615, 'Shadow Shaman'),
 'Slardar': Text(0.5537287097535732, 0.0011062491682894268, 'Slardar'),
 'Tidehunter': Text(0.6029064168758913, -0.249964235097955, 'Tidehunter'),
 'Witch Doctor': Text(-0.7750932005770703, -0.22376109710309408, 'Witch Doctor'),
 'Lich': Text(-0.7000077170527105, -0.46482332615085653, 'Lich'),
 'Riki': Text(0.4924920884978045, 0.6608833592086243, 'Riki'),
 'Tinker': Text(-0.9859811973969697, 0.23751042767090727, 'Tinker'),
 'Sniper': Text(0.10321630551841804, 0.7142358207341257, 'Sniper'),
 'Necrophos': Text(-0.6048526441540861, 0.7850324873150408, 'Necrophos'),
 'Warlock': Text(-0.7855592833930991, -0.41964856887310537, 'Warlock'),
 'Queen of Pain': Text(-0.35157521913059664, 0.3732891621300602, 'Queen of Pain'),
 'Faceless Void': Text(0.3228439976541231, 0.23054593159256562, 'Faceless Void'),
 'Wraith King': Text(0.563886848238809, -0.228475215772054, 'Wraith King'),
 'Death Prophet': Text(-0.5561612348954967, 0.11151971327196326, 'Death Prophet'),
 'Phantom Assassin': Text(0.9511848247164538, 0.34259526904878784, 'Phantom Assassin'),
 'Pugna': Text(-0.7162438766503973, -0.3481096643830949, 'Pugna'),
 'Templar Assassin': Text(-0.07162137662204554, 0.6831192194936277, 'Templar Assassin'),
 'Viper': Text(-0.09209976052659628, 0.3928923384724385, 'Viper'),
 'Luna': Text(0.5382209325812205, 0.8081925792656057, 'Luna'),
 'Dragon Knight': Text(0.6429463492445658, 0.151963447621353, 'Dragon Knight'),
 'Leshrac': Text(-0.3084263122961215, 0.08662792159743617, 'Leshrac'),
 "Nature's Prophet": Text(-0.7514227093307716, 0.4790905332056738, "Nature's Prophet"),
 'Lifestealer': Text(0.8054746549426566, -0.40210571957654967, 'Lifestealer'),
 'Omniknight': Text(0.1298128768027094, -0.25341134367461404, 'Omniknight'),
 'Enchantress': Text(-0.33834325064297865, 0.5253520790838143, 'Enchantress'),
 'Huskar': Text(0.5680775226838581, -0.5167810222060433, 'Huskar'),
 'Night Stalker': Text(0.4742021868464781, -0.16265421466085372, 'Night Stalker'),
 'Bounty Hunter': Text(-0.0983073825990013, -0.41825333152854965, 'Bounty Hunter'),
 'Weaver': Text(0.10109035500033046, 0.5814069200898612, 'Weaver'),
 'Jakiro': Text(-0.8017625353859138, -0.26248734932610346, 'Jakiro'),
 'Spectre': Text(0.5741371218608142, 0.6084540071051717, 'Spectre'),
 'Ancient Apparition': Text(-0.5364609963441151, -0.5696618658947576, 'Ancient Apparition'),
 'Doom': Text(0.4820604051232607, -0.5673653640623038, 'Doom'),
 'Ursa': Text(0.7913302947394518, 0.09354901169087493, 'Ursa'),
 'Gyrocopter': Text(0.05513685914224105, 0.1366834573899549, 'Gyrocopter'),
 'Alchemist': Text(0.7997492058250497, -0.22507094730250063, 'Alchemist'),
 'Silencer': Text(-0.4034190216798212, 0.22896600717181065, 'Silencer'),
 'Outworld Destroyer': Text(-0.40482104318575574, 0.4814085063450222, 'Outworld Destroyer'),
 'Shadow Demon': Text(-0.6040970846717493, -0.36459246210478485, 'Shadow Demon'),
 'Chaos Knight': Text(0.7642118991736924, -0.06823474018556314, 'Chaos Knight'),
 'Meepo': Text(0.4290564953216321, 0.3116798085198158, 'Meepo'),
 'Treant Protector': Text(-0.704886374859329, -0.5813259273390134, 'Treant Protector'),
 'Ogre Magi': Text(-0.42802684090853427, -0.8307482842824071, 'Ogre Magi'),
 'Undying': Text(-0.8511228010941942, -0.632994711582996, 'Undying'),
 'Rubick': Text(-0.671311055468428, -0.22029261788226623, 'Rubick'),
```

```
'Disruptor': Text(-0.49782980089773915, -0.39556022281238656, 'Disruptor'),
'Naga Siren': Text(0.5734836103428671, 0.42166346489451284, 'Naga Siren'),
'Slark': Text(0.340436839825971, 0.3077929128026792, 'Slark'),
'Medusa': Text(0.6934300883610984, 0.8231696445091995, 'Medusa'),
'Troll Warlord': Text(0.6473361509591867, 0.21587877296857902, 'Troll Warlord'),
'Centaur Warrunner': Text(0.3851779957823325, -0.8614306622475566, 'Centaur Warrunner'),
'Bristleback': Text(-0.12869392300186078, -0.8936800232104188, 'Bristleback'),
'Tusk': Text(-0.37644404643540563, -0.3584095585435012, 'Tusk'),
'Skywrath Mage': Text(-0.4375278324761213, -0.17432867275869085, 'Skywrath Mage'),
'Elder Titan': Text(-0.17313773141326616, -0.18663034288615926, 'Elder Titan'),
'Legion Commander': Text(0.36008325155262616, -0.18787335104488884, 'Legion Commander'),
'Ember Spirit': Text(0.3906285353530485, -0.7786100388788979, 'Ember Spirit'),
'Earth Spirit': Text(0.014167401719616633, -0.3187635128259376, 'Earth Spirit'),
'Underlord': Text(0.7143168180524232, -0.8369631990404595, 'Underlord'),
'Terrorblade': Text(0.4156679060988005, 0.5965204363075353, 'Terrorblade'),
'Oracle': Text(-0.9037068848341862, -0.429719585903633, 'Oracle'),
'Monkey King': Text(0.3742075008681622, 0.4957181756090103, 'Monkey King'),
'Grimstroke': Text(-0.6146620965175489, -0.4713512527154127, 'Grimstroke'),
'Hoodwink': Text(-0.30432723405732215, 0.9999999999999999, 'Hoodwink'),
'Mars': Text(0.515490101588171, -0.3807158701227094, 'Mars'),
'Dawnbreaker': Text(0.42851285173216924, -0.3489873054827882, 'Dawnbreaker'),
'Primal Beast': Text(0.2504345774155522, -0.4083319426438591, 'Primal Beast'),
'Muerta': Text(-0.2383357257226088, 0.6363619761820117, 'Muerta')}
```



Task 2.2.1 (3 points)

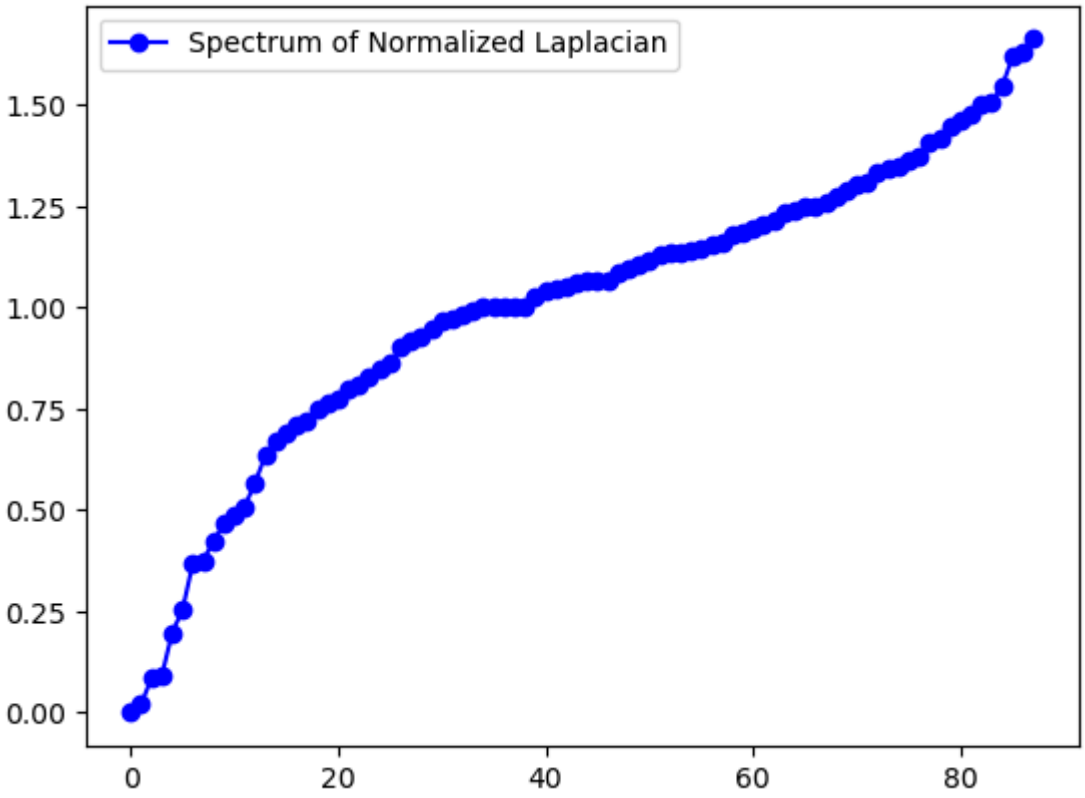
Compute the eigenvectors and eigenvalues (using the provided function) of the Normalized Laplacian and the Random Walk Laplacian of the graph G.
Plot the spectrum (eigenvalues).

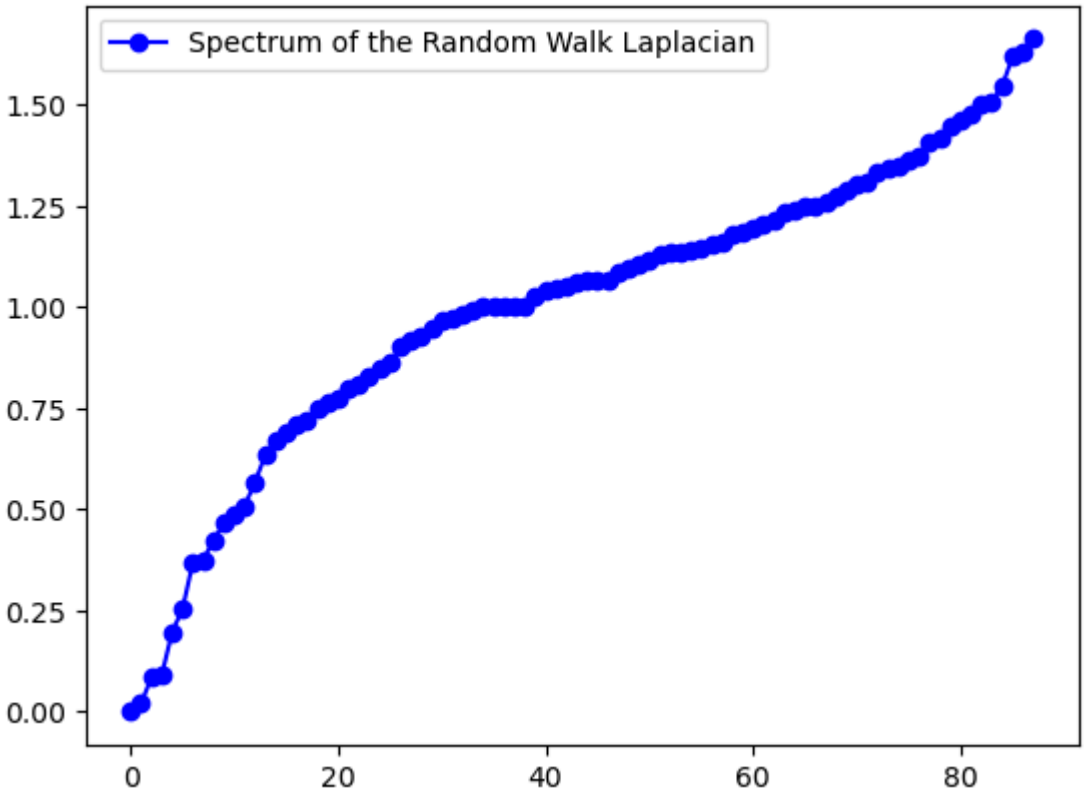
[Implement] the code to compute the different Laplacians.

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
nodeID = pickle.load(open('./data/nodeID.pickle', 'rb'))


### YOUR CODE HERE
def compute_L_norm(G):
    A = nx.to_numpy_array(G) # get adjencency matrix
    D = np.diag([val for (node, val) in G.degree]) # get degree matrix
    I = np.eye(G.number_of_nodes()) # n sized identity matrix
    D_sqrt_inv = np.linalg.inv(np.sqrt(D))

    return I - D_sqrt_inv @ A @ D_sqrt_inv

def compute_L_rw(G):
    A = nx.to_numpy_array(G) # get adjencency matrix
    D = np.diag([val for (node, val) in G.degree]) # get degree matrix
    I = np.eye(G.number_of_nodes()) # n sized identity matrix

    return I - np.linalg.inv(D) @ A
### YOUR CODE HERE

L_norm = compute_L_norm(G)
L_rw = compute_L_rw(G)

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 0x27beb19b610>
```





Task 2.2.3 (5 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.

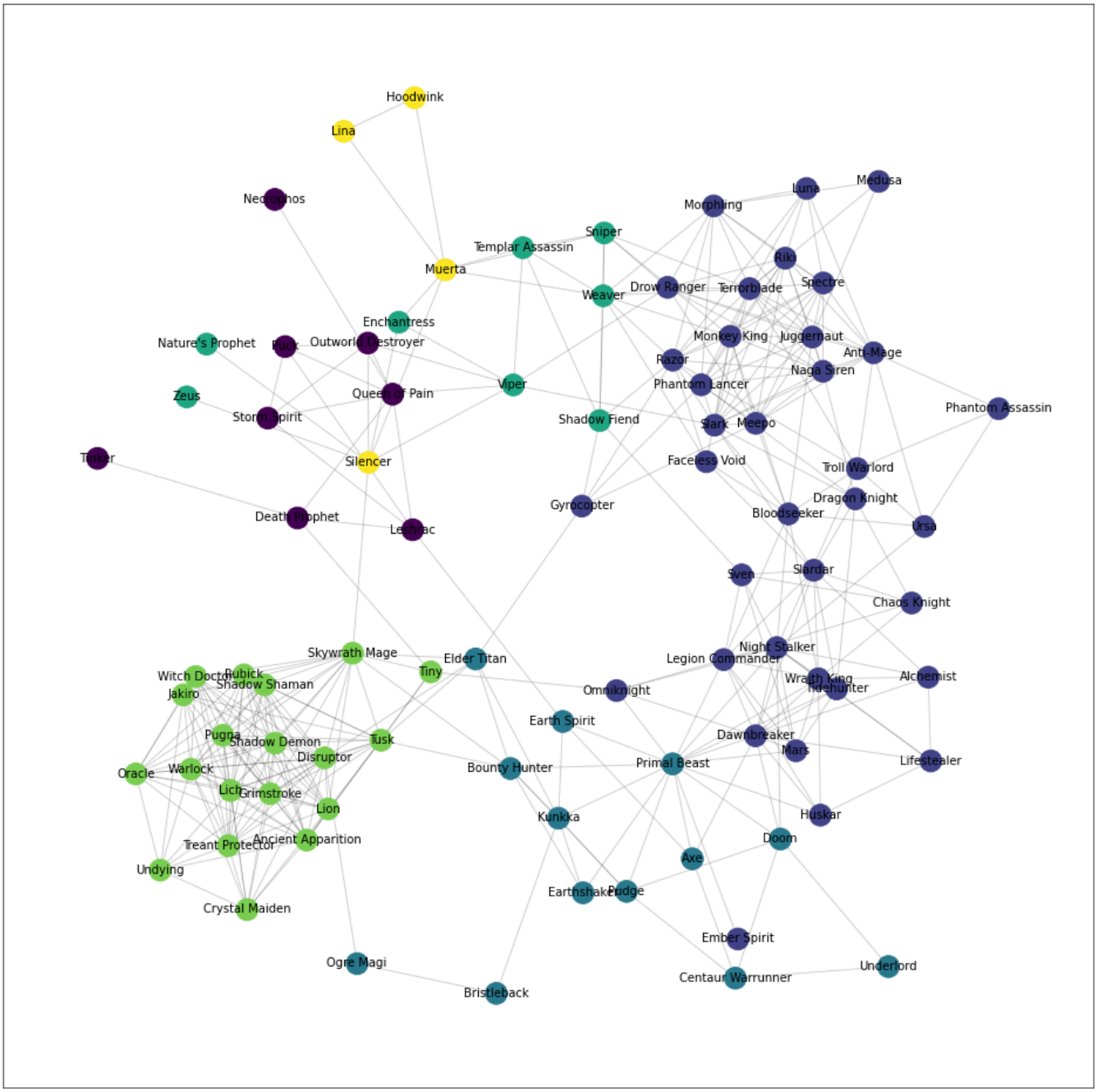
In [ ]:
```
from sklearn.cluster import k_means
import warnings

def spect_cluster(G, eig_type="normal", k=5, d=5):
    if eig_type == "normal":
        L = compute_L_norm(G)
```

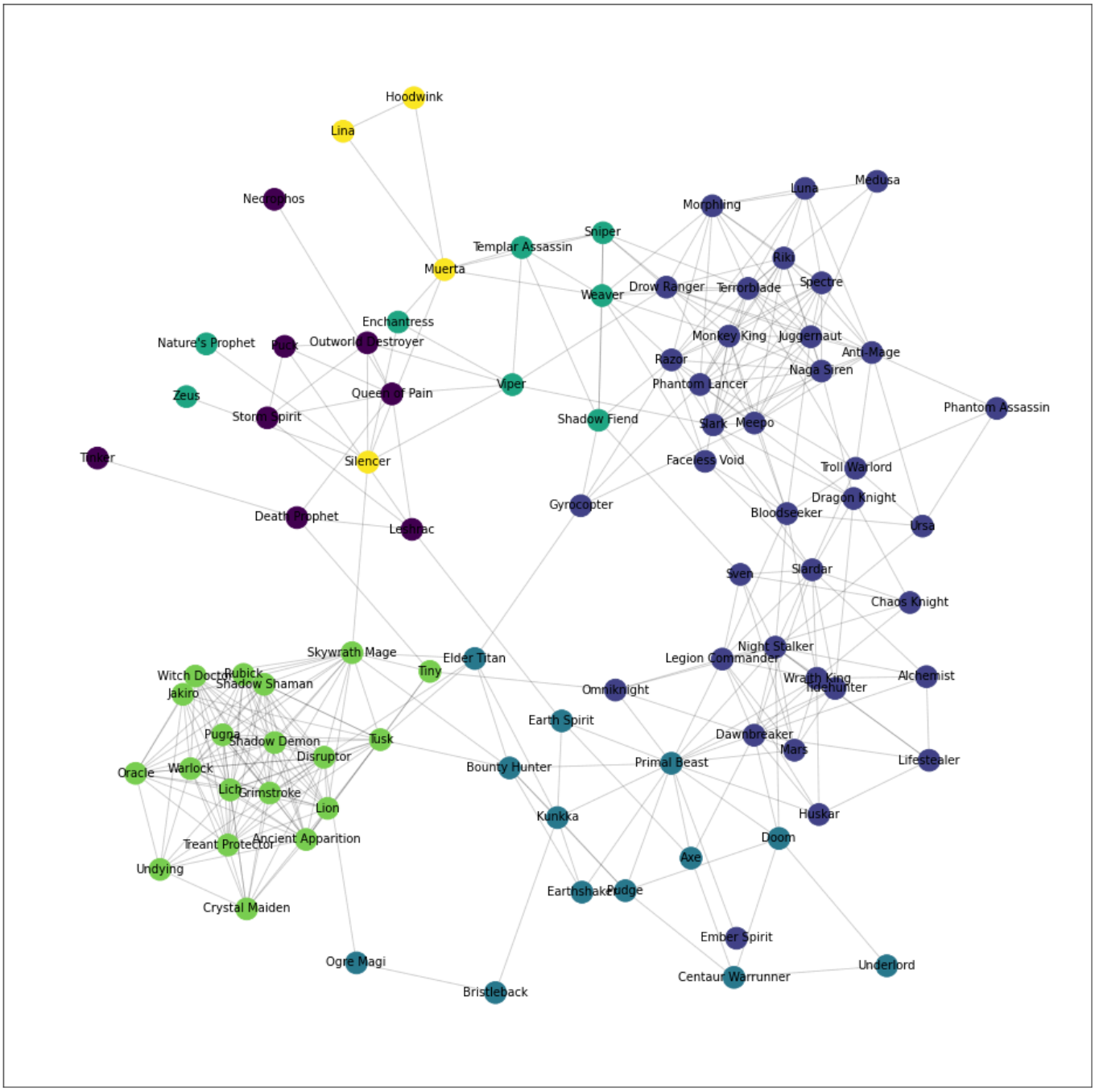
```
    else:
        L = compute_L_rw(G)

    eigval, eigvec = graph_eig(L)
    T = eigvec[:,0:k]

    warnings.filterwarnings('ignore')
    _, y_clust, _ = k_means(T, n_clusters=k, n_init="auto")
    warnings.filterwarnings("default")
    return y_clust
```

In [ ]:

```
def plot_graph(G, clusters, node_size = 500, labels=True):

    fig = plt.figure(1, figsize=(20, 20), dpi=60)
    #nx.draw(G, pos=layout,with_labels = True, node_size=50, node_color=your_clusters,font_si
    #layout = nx.spring_layout(H, k=5.15, iterations=20)
    layout = nx.kamada_kawai_layout(G)
    nx.draw_networkx_edges(G, layout, alpha = 0.2)
    nx.draw_networkx_nodes(G, layout, node_color=clusters, node_size=node_size)
    if labels:
        nx.draw_networkx_labels(G, layout)


    return fig

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

Out[ ]:


Hoodwink
Lina
Necrophos
Medusa
Luna
Morphling
Sniper
Templar Assassin
Riki
Muerta
Drow Ranger
Terrorblade
Spectre
Weaver
Enchantress
Nature's Prophet
Puck
Outworld Destroyer
Monkey King
Juggernaut
Anti-Mage
Razor
Naga Siren
Phantom Lancer
Viper
Zeus
Queen of Pain
Phantom Assassin
Storm Spirit
Shadow Fiend
Slark
Meepo
Tinker
Silencer
Faceless Void
Troll Warlord
Dragon Knight
Gyrocopter
Bloodseeker
Death Prophet
Leshrac
Ursa
Sven
Slardar
Chaos Knight
Skywrath Mage
Elder Titan
Night Stalker
Legion Commander
Witch Doctor
Rubick
Tiny
Wraith King
Alchemist
Shadow Shaman
Omniknight
Nightfall
Jakiro
Earth Spirit
Pugna
Shadow Demon
Tusk
Dawnbreaker
Mars
Disruptor
Oracle
Warlock
Bounty Hunter
Primal Beast
Lifestealer
Lich
Grimstroke
Lion
Kunkka
Huskar
Treant Protector
Ancient Apparition
Doom
Axe
Undying
Earthshaker
Pudge
Crystal Maiden
Ember Spirit
Ogre Magi
Underlord
Centaur Warrunner
Bristleback



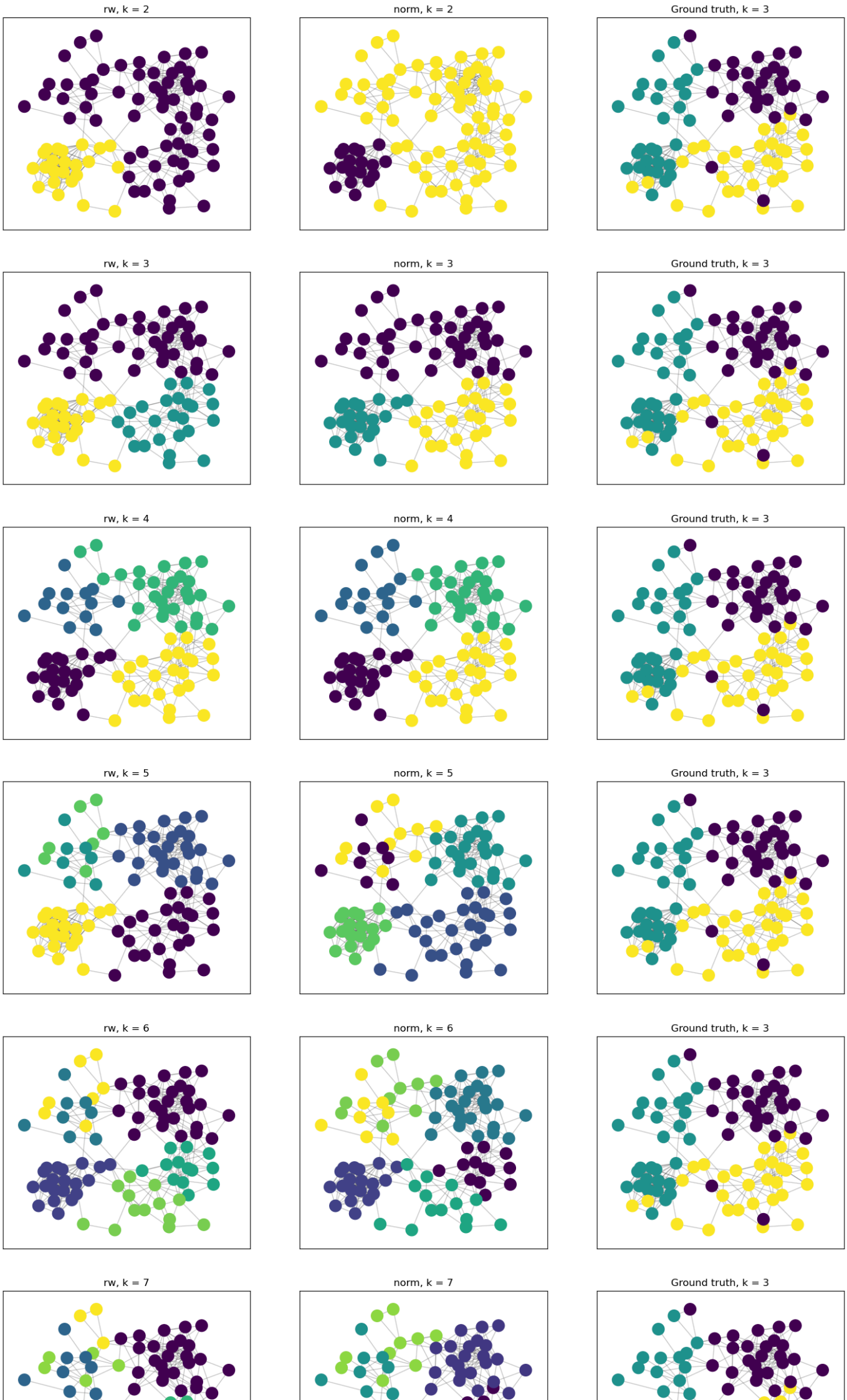
Task 2.2.5 (2 points)

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`.

[Motivate] 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? Can you detect easily all the ground truth communities? Are some communities not detected? Why do you think that happens?

In [ ]:
```
fig, axs = plt.subplots(ncols=3, nrows=6, figsize=(18, 32))

for i in range(6):

    clusters_norm = spect_cluster(G, k=i+2)
    clusters_rw = spect_cluster(G, k=i+2, eig_type=None)
    plt.subplot(6, 3, 3*i+2)
    plot_graph(G, clusters_norm, node_size=200, labels=False)
    plt.title(f'norm, k = {i + 2}')

    plt.subplot(6, 3, 3*i + 1)
    plot_graph(G, clusters_rw, node_size=200, labels=False)
    plt.title(f'rw, k = {i + 2}')
```

```
    plt.subplot(6, 3, 3*i + 3)
    plot_graph(G, groundTruth, node_size=200, labels=False)
    plt.title(f'Ground truth, k = 3')
```


rw, k = 2
norm, k = 2
Ground truth, k = 3
rw, k = 3
norm, k = 3
Ground truth, k = 3
rw, k = 4
norm, k = 4
Ground truth, k = 3
rw, k = 5
norm, k = 5
Ground truth, k = 3
rw, k = 6
norm, k = 6
Ground truth, k = 3
rw, k = 7
norm, k = 7
Ground truth, k = 3

It can be seen that Random Walk and normalized laplacian does give slighly different clusterings, which is more visible when the number of clusters is low. It seems random walk captures highly connected areas/nodes more, compared to using normalized laplacian, which doesn't appear to emphasize highly connected regions as much.

This could be due to the nature of random walk, which will emphasize nodes that are easily reachable from each other, and result in these nodes obtaining similar values in the laplacian. Whereas the normalized laplacian emphasizes relative connections between nodes instead of absolute connectivity, meaning it doesn't only consider whether two nodes are connected, but also weighs the connection compared to the overall connectivity of the nodes involved, hence it isn't as sensitive to densely connected regions.

It seems normalized laplacian better captures the true clustering, but neither method is able to capture the overlapping between communities. Moreover, when choosing a higher number of k, clustering on areas of the network that are particularly sparse becomes odd.

Task 2.2.6 (5 points)

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

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

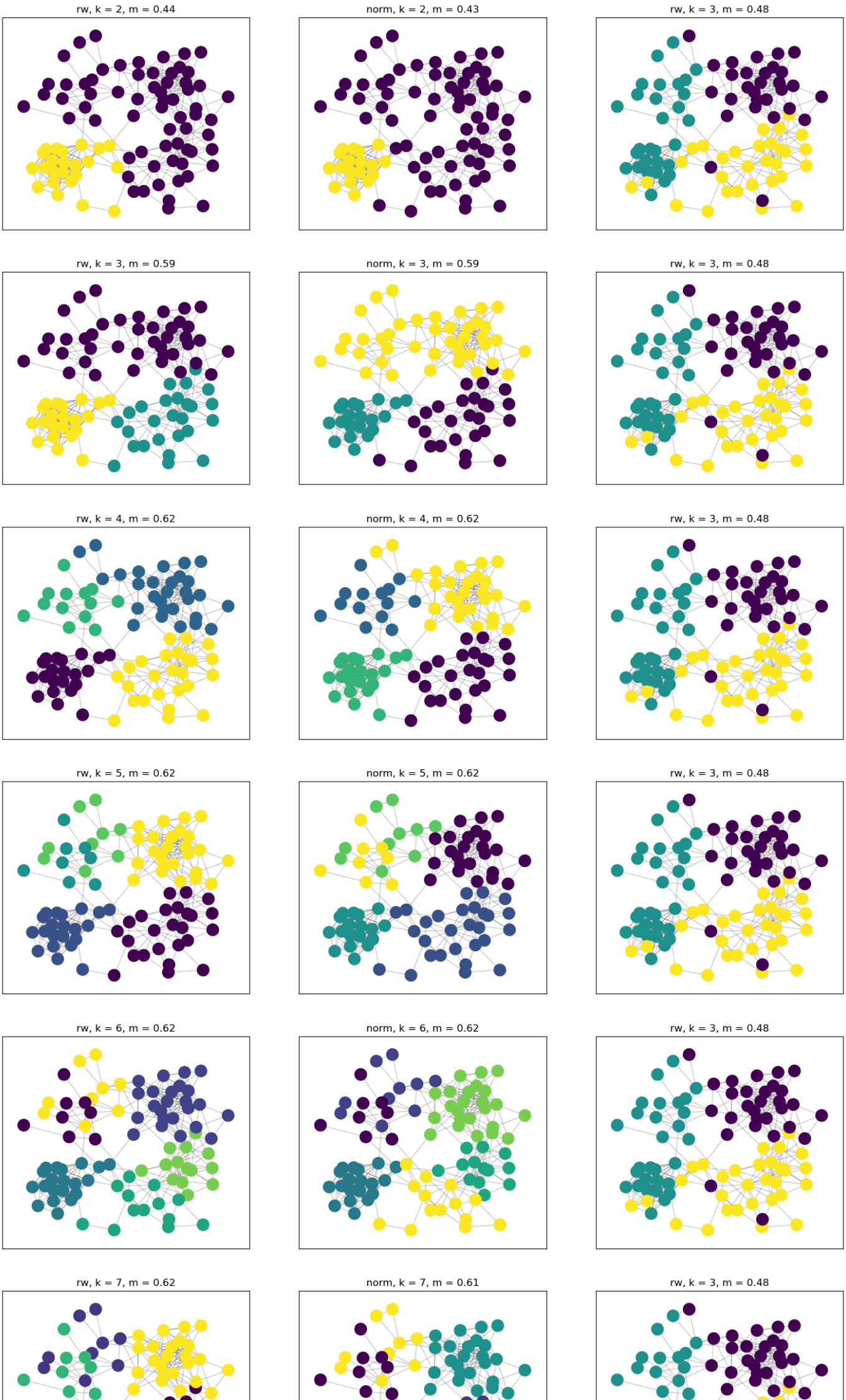
where A is the adjacency matrix, and d_i is the degree of node i

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, clustering):
    #IDnode = pickle.load(open('./data/IDnode.pickle', 'rb'))

    m = G.number_of_edges()
    A = nx.to_numpy_array(G) # get adjencency matrix
    d = np.array([val for (node, val) in G.degree]) # get degrees matrix
    clustering = np.array(clustering)

    dd = np.outer(d, d) /(2*m)
    match = (clustering[:, None] == clustering[None, :]).astype(int)

    return np.sum((A-dd)*match)/(2*m)
```

In [ ]:
```
fig, axs = plt.subplots(ncols=3, nrows=6, figsize=(18, 32))

for i in range(6):

    clusters_norm = spect_cluster(G, k=i+2)
    clusters_rw = spect_cluster(G, k=i+2, eig_type=None)
    plt.subplot(6, 3, 3*i+2)
    plot_graph(G, clusters_norm, node_size=200, labels=False)
```

```
    m_norm = modularity(G, clusters_norm)
    plt.title(f'norm, k = {i + 2}, m = {round(m_norm, ndigits=2)}')

    plt.subplot(6, 3, 3*i + 1)
    m_rw = modularity(G, clusters_rw)
    plot_graph(G, clusters_rw, node_size=200, labels=False)
    plt.title(f'rw, k = {i + 2}, m = {round(m_rw, ndigits=2)}')

    plt.subplot(6, 3, 3*i + 3)
    m_gt = modularity(G, groundTruth)
    plot_graph(G, groundTruth, node_size=200, labels=False)
    plt.title(f'rw, k = 3, m = {round(m_gt, ndigits=2)}')
```


rw, k = 2, m = 0.44
norm, k = 2, m = 0.43
rw, k = 3, m = 0.48
rw, k = 3, m = 0.59
norm, k = 3, m = 0.59
rw, k = 3, m = 0.48
rw, k = 4, m = 0.62
norm, k = 4, m = 0.62
rw, k = 3, m = 0.48
rw, k = 5, m = 0.62
norm, k = 5, m = 0.62
rw, k = 3, m = 0.48
rw, k = 6, m = 0.62
norm, k = 6, m = 0.62
rw, k = 3, m = 0.48
rw, k = 7, m = 0.62
norm, k = 7, m = 0.61
rw, k = 3, m = 0.48

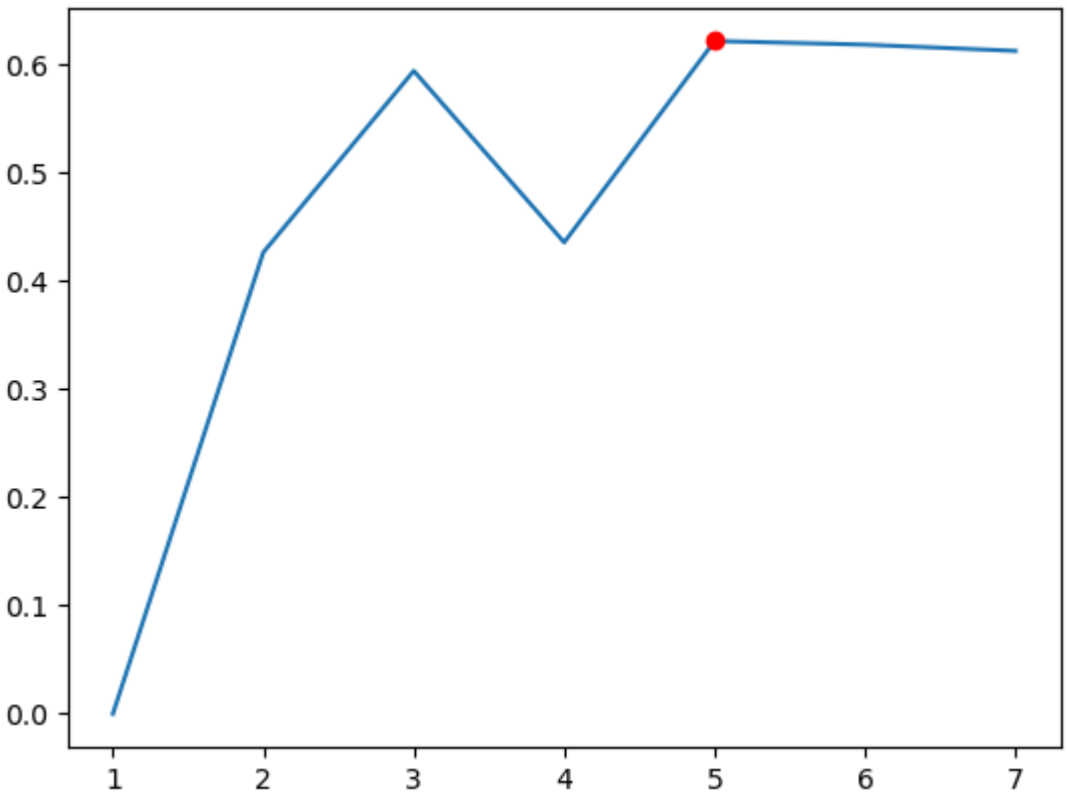
Task 2.2.5 (3 points)

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

[Motivate] which value maximises the modularity. Is $k = 6$ maximizing the modularity? If yes, is this consistent with the ground-truth? If not, is it because of an issue with modularity or with spectral clustering?

In [ ]:
```
mods = []
ks = [1, 2, 3, 4, 5, 6, 7]
for k in ks:
    clusters = spect_cluster(G, k=k) ### NOTE: If you do not use your implementation substitu
    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)
plt.plot(np.argmax(mods) + 1, max(mods), 'ro')
print(mods)
```

```
[3.8629677843121887e-17, 0.42636141865265526, 0.5940739350722463, 0.4356051792081066, 0.621643
8356164383, 0.6182923625445674, 0.6125351848376807]
```



The modullarity is maximized by choosing k with a value between 4 - 6. This is not consistent with the ground truth, which has a k value of 3. This could be due to a combination of both the clustering and the modularity measure. Modularity does not take ground truth into consideration as a clustering evaluation measure, but evaluates how well a network is partitioned into densely connected groups, meaning if there exists densely connected subgroups within a true cluster, it will favor including the densely connected supgrouop as it its own cluster, ie favor a larger number of k if this is the case.

Moreover, spectral clustering does not appear to create as clear a clustering for k = 3 compared to having 5 - 6 clusters, where there is smaller more densely connected regions, which is favored by the modularity measure.

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.

- [x] 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.

In linear embedding minimizing the L_2 distance between the dot product of the embedded vectors $z_i^T z_j$ and the similarity measure of the laplacian L_{ij} results in an equation resembling an eigenvalue problem, where the embedding matrix is equivalent to the eigenvectors of the laplacian.

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 (15 points)

Task 3.1.1 (8 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, node, alpha = 0.85, iterations = 1000):
    ppr = np.zeros(G.number_of_nodes())
    t_node = node

    for _ in range(iterations):
```

```
        if np.random.random() < 1 - alpha:
            t_node = node
        else:
            neighbors = list(G[t_node])
            if neighbors:  # Ensure there are neighbors
                t_node = random.choice(neighbors)

        idx = list(G.nodes()).index(t_node)
        ppr[idx] += 1

    return ppr/iterations


approx_personalized_pagerank(G, 'Anti-Mage', alpha = 0.85, iterations = 1000)
```

Out[ ]:
```
array([0.196, 0.001, 0.013, 0.001, 0.038, 0.002, 0.05 , 0.019, 0.007,
       0.038, 0.006, 0.001, 0.027, 0.005, 0.002, 0.   , 0.   , 0.002,
       0.   , 0.001, 0.001, 0.008, 0.001, 0.001, 0.002, 0.05 , 0.001,
       0.01 , 0.001, 0.002, 0.006, 0.007, 0.007, 0.001, 0.022, 0.001,
       0.001, 0.009, 0.027, 0.004, 0.003, 0.   , 0.004, 0.   , 0.002,
       0.004, 0.012, 0.002, 0.015, 0.   , 0.05 , 0.   , 0.002, 0.022,
       0.01 , 0.004, 0.004, 0.012, 0.   , 0.003, 0.033, 0.003, 0.   ,
       0.001, 0.002, 0.002, 0.037, 0.036, 0.009, 0.03 , 0.   , 0.   ,
       0.   , 0.001, 0.005, 0.007, 0.001, 0.001, 0.   , 0.062, 0.001,
       0.028, 0.001, 0.   , 0.003, 0.006, 0.007, 0.004])
```

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.number_of_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_values[:10]}\

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=[ 41  37  40  44  36  18  15 107 102 103]
 top-10 values=[0.2 0.2 0.1 0.1 0.1 0.1 0.1 0.  0.  0. ]

Iteration 307: top-10 r=[39 41 36 40 42 18 15 17 44 37]
 top-10 values=[0.09446254 0.08143322 0.07491857 0.07166124 0.06188925 0.06188925
 0.05863192 0.03908795 0.02931596 0.02931596]

Iteration 614: top-10 r=[18 41 39 40 15 57 42 36 44 37]
 top-10 values=[0.08469055 0.08306189 0.07980456 0.07166124 0.06677524 0.05863192
 0.0504886  0.04723127 0.04560261 0.04071661]

Iteration 1228: top-10 r=[40 41 39 18 15 42 36 44 57 37]
 top-10 values=[0.0757329  0.07003257 0.06188925 0.06026059 0.05863192 0.05456026
 0.0529316  0.04315961 0.03257329 0.03013029]

Iteration 30700: top-10 r=[39 40 41 18 42 15 36 57 44 17]
 top-10 values=[0.07234528 0.06785016 0.06693811 0.06234528 0.05775244 0.05641694
 0.0523127  0.04345277 0.04172638 0.03345277]

Iteration 307000: top-10 r=[39 40 41 18 42 15 36 44 57 17]
 top-10 values=[0.07338111 0.06761564 0.06733876 0.06411726 0.05826384 0.05779479
 0.05089902 0.04439414 0.04367427 0.03312052]

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]
```

For iterations smaller than the number of nodes, the random walk algorithm would not have the opportunity to visit all nodes within reach, and hence which nodes that have visitied the most is highly dependable on the startnode and the nodes that can be reached from here. Same goes for a number of iterations that are not many times bigger than the number of nodes in the graph, since the probability of nodes being visited many times is small, and therefore the ranking is still quite dependent on randomness. The more iterations, the more clear and stable the pattern becomes.

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.

[Motivate] the differences. Do the 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_nodes()*4, G.n
    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, alpha=0.8
    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 of inters
```

```
[18 41 40 39  0] [42 40 18 39  0]
Number of iterations 10
Approximate PPR: [0.2 0.  0.  0.  0.  0.  0.  0.  0.  0. ]
Real PPR: [0.20102640550640022, 0.0004761511237216732, 0.001421044532107968, 0.005179280113522
94, 0.0010162923813859888, 0.0006924636204434976, 0.000586579792176299, 0.000586579792176299,
0.0003234474438257967, 0.00019090839324436927]
Topk of nx.pagerank: [18 41 40 39  0], Topk of our estimation [37 41 39 42  0], Size of inters
ection: 3
Number of iterations 307
Approximate PPR: [0.17263844 0.         0.00325733 0.00977199 0.         0.
 0.         0.00325733 0.         0.         ]
Real PPR: [0.20102640550640022, 0.0004761511237216732, 0.001421044532107968, 0.005179280113522
94, 0.0010162923813859888, 0.0006924636204434976, 0.000586579792176299, 0.000586579792176299,
0.0003234474438257967, 0.00019090839324436927]
Topk of nx.pagerank: [18 41 40 39  0], Topk of our estimation [41 18 40 42  0], Size of inters
ection: 4
Number of iterations 614
Approximate PPR: [0.21661238 0.         0.         0.00162866 0.         0.00162866
 0.00162866 0.00162866 0.         0.         ]
Real PPR: [0.20102640550640022, 0.0004761511237216732, 0.001421044532107968, 0.005179280113522
94, 0.0010162923813859888, 0.0006924636204434976, 0.000586579792176299, 0.000586579792176299,
0.0003234474438257967, 0.00019090839324436927]
Topk of nx.pagerank: [18 41 40 39  0], Topk of our estimation [18 41 39 40  0], Size of inters
ection: 5
Number of iterations 1228
Approximate PPR: [0.21091205 0.         0.00162866 0.00488599 0.00162866 0.00081433
 0.         0.         0.         0.         ]
Real PPR: [0.20102640550640022, 0.0004761511237216732, 0.001421044532107968, 0.005179280113522
94, 0.0010162923813859888, 0.0006924636204434976, 0.000586579792176299, 0.000586579792176299,
0.0003234474438257967, 0.00019090839324436927]
Topk of nx.pagerank: [18 41 40 39  0], Topk of our estimation [18 40 41 39  0], Size of inters
ection: 5
Number of iterations 30700
Approximate PPR: [2.02866450e-01 1.62866450e-04 1.20521173e-03 4.78827362e-03
 8.79478827e-04 5.53745928e-04 2.93159609e-04 3.58306189e-04
 4.56026059e-04 1.30293160e-04]
Real PPR: [0.20102640550640022, 0.0004761511237216732, 0.001421044532107968, 0.005179280113522
94, 0.0010162923813859888, 0.0006924636204434976, 0.000586579792176299, 0.000586579792176299,
0.0003234474438257967, 0.00019090839324436927]
Topk of nx.pagerank: [18 41 40 39  0], Topk of our estimation [18 40 41 39  0], Size of inters
ection: 5
Number of iterations 307000
Approximate PPR: [2.00693811e-01 4.65798046e-04 1.46905537e-03 5.34201954e-03
 9.41368078e-04 7.19869707e-04 6.74267101e-04 5.27687296e-04
 3.55048860e-04 1.40065147e-04]
Real PPR: [0.20102640550640022, 0.0004761511237216732, 0.001421044532107968, 0.005179280113522
94, 0.0010162923813859888, 0.0006924636204434976, 0.000586579792176299, 0.000586579792176299,
0.0003234474438257967, 0.00019090839324436927]
Topk of nx.pagerank: [18 41 40 39  0], Topk of our estimation [18 41 40 39  0], Size of inters
ection: 5
```

As the number of iterations increases, the 5 highest ranking nodes from the approximation becomes more similar to the real 5 highest ranking nodes, as well does their PageRanking value. As described in the task above, the more iterations used in the approximation, the less the ranking is determined by randomness and begin to show a more stable pattern/ranking.

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_nodes()*4, G.n
    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, alpha=0.1

    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 of inters
```

```
Topk of nx.pagerank: [40 41 42 39  0], Topk of our estimation [ 99  98  97 104   0], Size of i
ntersection: 1
Topk of nx.pagerank: [40 41 42 39  0], Topk of our estimation [44 57 15 18  0], Size of inters
ection: 1
Topk of nx.pagerank: [40 41 42 39  0], Topk of our estimation [42 15 18 57  0], Size of inters
ection: 2
Topk of nx.pagerank: [40 41 42 39  0], Topk of our estimation [42 18 41 15  0], Size of inters
ection: 3
Topk of nx.pagerank: [40 41 42 39  0], Topk of our estimation [44 42 18 41  0], Size of inters
ection: 3
```

When decreasing the parameter α, the approximation requires more iterations to converge. This is due to the random walk having a large probability (1 - α) of randomly teleporting back to the start node. This means the random walk has a lower probability of visiting nodes further from the start node and a lower probablility of re-visiting the same nodes multiple times. Hence it takes more iterations for the ranking to converge.

Task 3.2 Spam and link farms (19 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 (4 points)

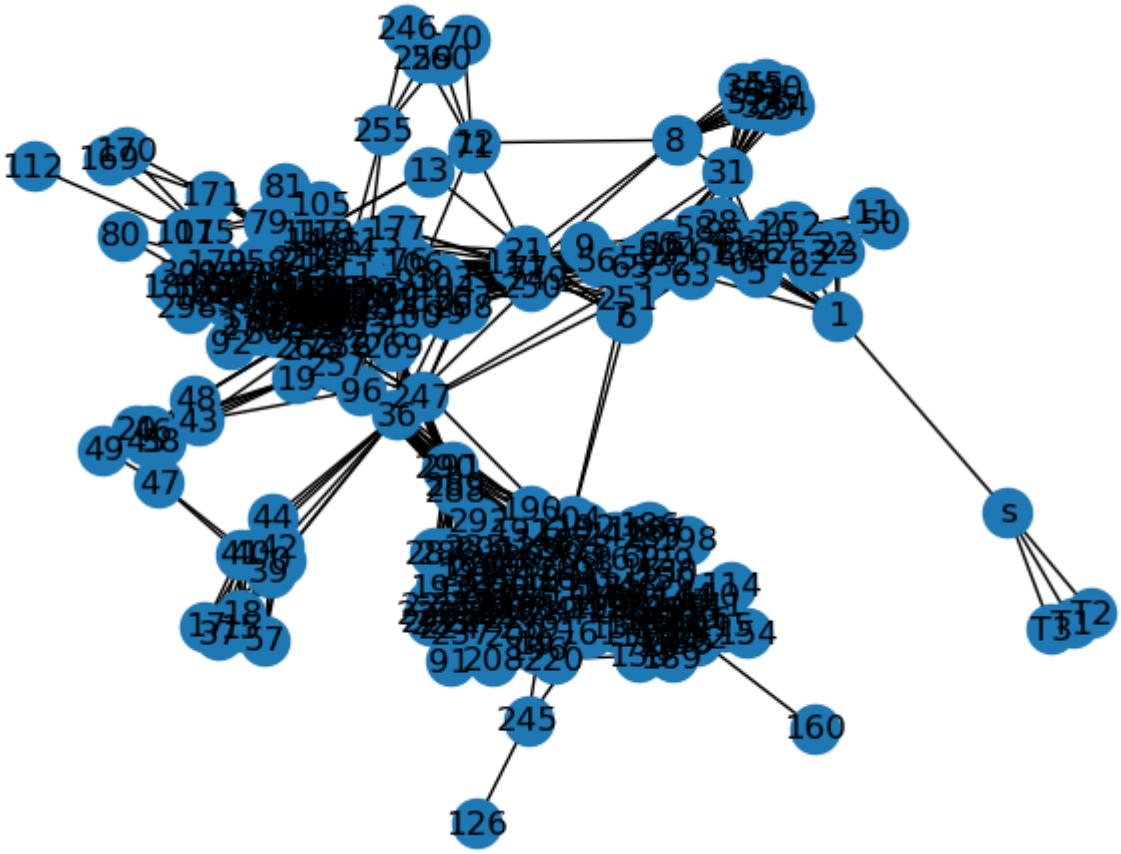
Based on the analysis in the slides, construct a spam farm s on the graph G with T fake nodes. Assume that s manages to get links from node 1. With $\alpha = 0.5$,

[Motivate] 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 [ ]:
```
spam_G = G.copy()
T = 1

# Add spam node
spam_G.add_node("s")
spam_G.add_edge(1, "s")

while True:
    spam_G.add_node(f"T{T}")
    spam_G.add_edge("s", f"T{T}")

    ppr_nx = nx.pagerank(spam_G, alpha=0.5)
    max_ranking = max(ppr_nx, key = ppr_nx.get)

    if max_ranking == "s":
        break
    T += 1

print(f"Max ranking node {max_ranking} with number of pages T = {T}")
```

```
Max ranking node s with number of pages T = 3
```

In [ ]:
```
nx.draw(spam_G, with_labels=True)
```



As computed using the code above, the minimum number of pages needed to get s being assigned the highest pageRank is 3.

Task 3.2.2 (4 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.

C.f the lecture notes from week 8 we can find the PageRank $\mathbf{r}_s$ of the link farm s as

$$r_s = \frac{r_X}{1-\alpha^2} + \frac{\alpha}{1+\alpha}\frac{T}{n}$$

We insert the known values and get

$$\begin{aligned} r_s &= \frac{r_X}{1-\alpha^2} + \frac{\alpha}{1+\alpha}\frac{\frac{n}{5}}{n} \\ &= \frac{r_X}{1-\alpha^2} + \frac{\alpha}{5+5\alpha} \end{aligned}$$

We note that both terms grow as $\alpha \to 1$. The second term approaches $\frac{1}{10}$, while the first term will grow to infinity (as long as $r_X \neq 0$)

As $\alpha = 1$ is undefined we choose $\alpha = 1 - \varepsilon$ as the value that maximizes r_s, for $\varepsilon > 0$

Task 3.2.3 (8 points)

Now we fix both $\alpha = 0.85$ 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=0.85):
    r = None

    ### YOUR CODE HERE
    # cf. 8.2.3 of lecture notes
    n = G.number_of_nodes()
    m = len(trusted_indices)

    r_x = nx.pagerank(G, alpha=alpha, max_iter=iterations)

    tp_dict = dict()
    for u in range(n):
        if u in trusted_indices:
            tp_dict[u] = 1/m
        else:
            tp_dict[u] = 0

    r_x_plus = nx.pagerank(G, alpha=alpha, max_iter=iterations, personalization=tp_dict)
    r_x_minus = [r_x[u] - r_x_plus[u] for u in range(n)]
    x = [r_x_minus[u] / r_x[u] for u in range(n)]
    r = x
    ### YOUR CODE HERE
    return r
```

Task 3.2.4 (3 points)

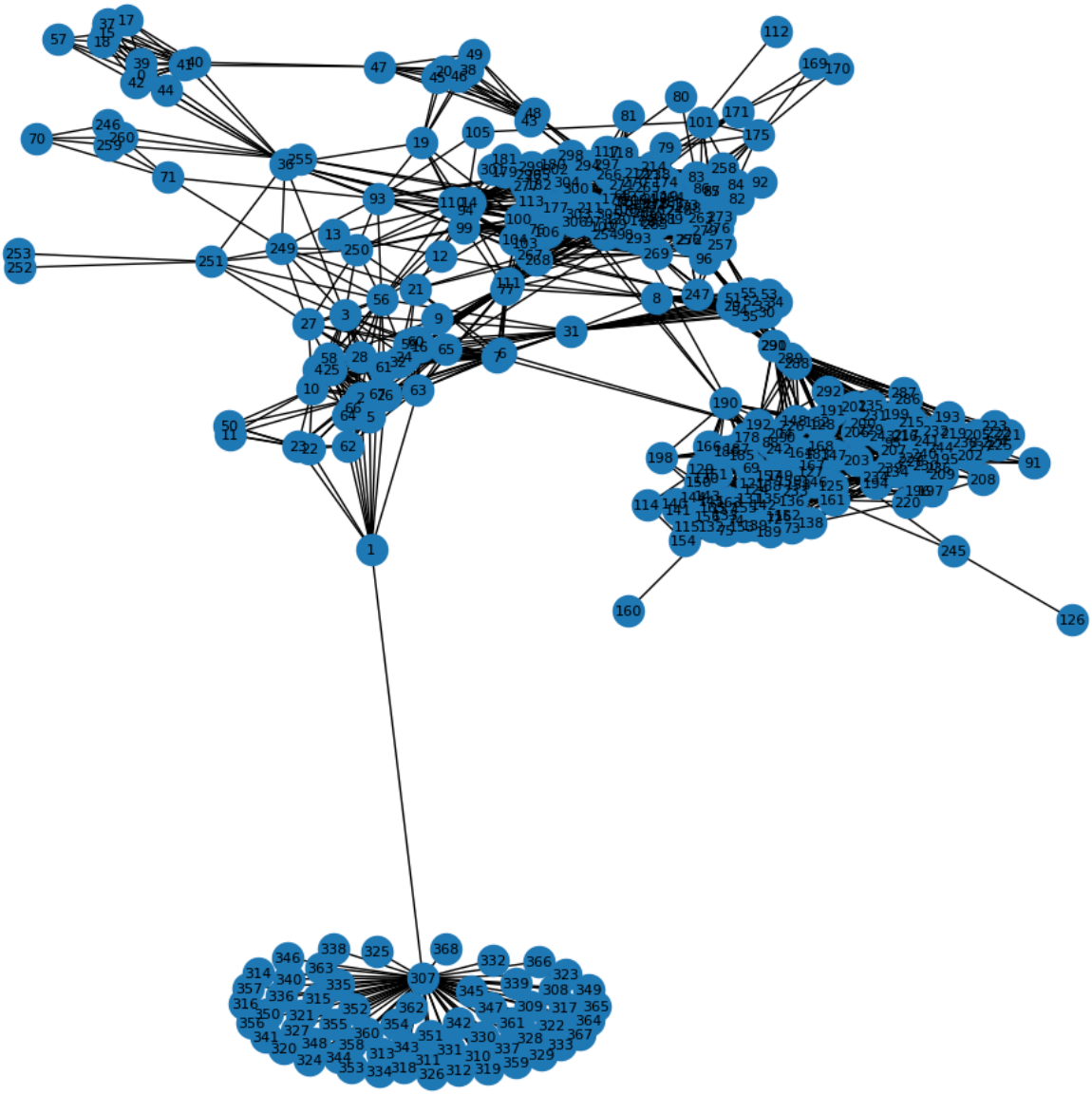
[Discuss] whether we are able to detect the node s, if the trusted set of nodes is a random sample 10% of the nodes in the original graph. If not, what could be a viable solution? Which nodes would you rather choose as trusted?

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

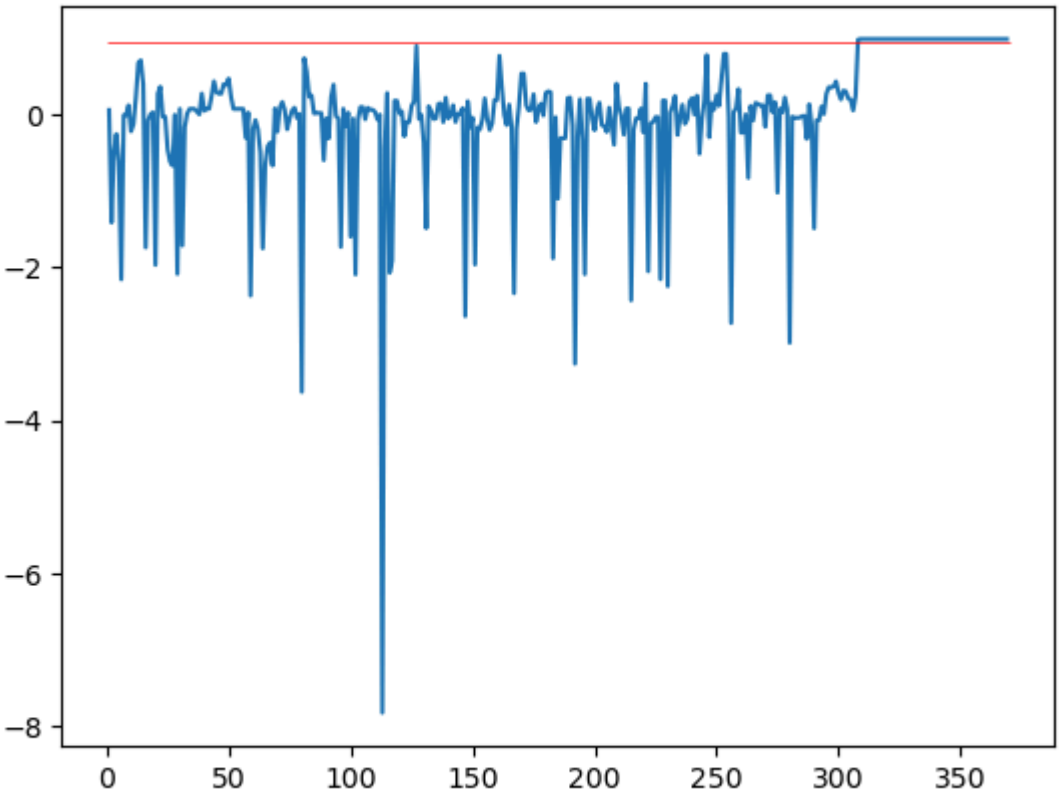
In [ ]:
```
# selecting trusted indices
from random import sample
n = G.number_of_nodes()
trusted_indices = sample(range(n), n//10 + 1)

spam_G = G.copy()
S = n
# Add spam node
spam_G.add_node(S)
spam_G.add_edge(1, S)

for T in range(n+1, n + (n//5)+1):
    spam_G.add_node(T)
    spam_G.add_edge(S, T)

max_spam_mass = trusted_pagerank(spam_G, trusted_indices)
print([i for i, x in enumerate(max_spam_mass) if x >= 0.95])

plt.figure(figsize=(10,10))
nx.draw(spam_G, with_labels=True,font_size=8)
```

```
[307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 32
5, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 3
44, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362,
363, 364, 365, 366, 367, 368]
```



In [ ]:
```
plt.plot(range(1, spam_G.number_of_nodes()+1), trusted_pagerank(spam_G, trusted_indices))
plt.hlines(y=0.95, xmin=0, xmax=370, linewidth=0.5, color='r')
```

Out[ ]:
```
<matplotlib.collections.LineCollection at 0x27be73978e0>
```



Yes, it is possible with 10% trusted nodes to capture the majority (if not all) of the spam nodes with a sufficiently high threshold, as illustrated in the figure above. However, it should be noted that the chosen threshold may result in 'real' nodes being marked as spam if not chosen appropriately, which can be difficult in real world cases.

10% is sufficient, however, with even a higher number of trusted pages it becomes easier to detect spam nodes.

Part 4: Graph embeddings (16 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. *Hint1:* it might be a help to look in the original article [1] above.

Hint2: Line 14-15 in the pseudo code from the paper contains a typo:
Line 14 should be W_u ← W_u+(g*W_v)

Line 15 should be W_v ← W_v+(g*W_u)

In [ ]:

```
def sigmoid(x):
    ''' Return the sigmoid function of x
        x: the input vector
    '''
    try: # it says "imput vector" but that is not always true
        # trying to avoid overflow:
        return [1/(1 + np.exp(-val)) if val > 0\
                else np.exp(val)/(np.exp(val)+1) for val in x]
    except TypeError:
        if x > 0:
            return 1/(1 + np.exp(-x))
        return np.exp(x)/(np.exp(x)+1)



def pagerank_matrix(G, 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
    '''
    ### YOUR CODE HERE
    n = G.number_of_nodes()
    P = np.zeros((n, n))

    # We use ealier implemented algorithm:
    for u in range(n):
        P[u] = approx_personalized_pagerank(G, u, alpha = alpha)
    ### YOUR CODE HERE
    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 indicating wh
            step_size: step size for gradient descent


       :return nothing, just update rows Z[v,:] and and Z[u,:]
    '''
    ### YOUR CODE HERE
    g = (C - sigmoid(Z[u,:] @ Z[v,:])) * step_size
    Z[u,:] = Z[u,:] + (g * Z[v,:])
    Z[v,:] = Z[v,:] + (g * Z[u,:])
    ### YOUR CODE HERE



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 HERE
    # in stead of repeating until converged we use the specified steps:
    for _ in range(steps):
        u = np.random.randint(n) # > Sample a node
        v = np.random.choice(n, p = S[u,:]) # > Sample positive example
        update(u, v, Z, 1, step_size)
        for i in range(k):
            v_tilde = np.random.randint(n) # > Sample negative example
            update(u, v_tilde, Z, 0, step_size)
    ### YOUR CODE HERE
    return Z
```

In [ ]:
```
# This code runs the `verse` algorithm above on G and stores the embeddings to 'verse.npy'
P   = pagerank_matrix(G)
emb = verse(G, P, 128, step_size=0.0025, steps=200_000)
np.save('verse.npy', emb)
```

Task 4.1.2 (3 points)

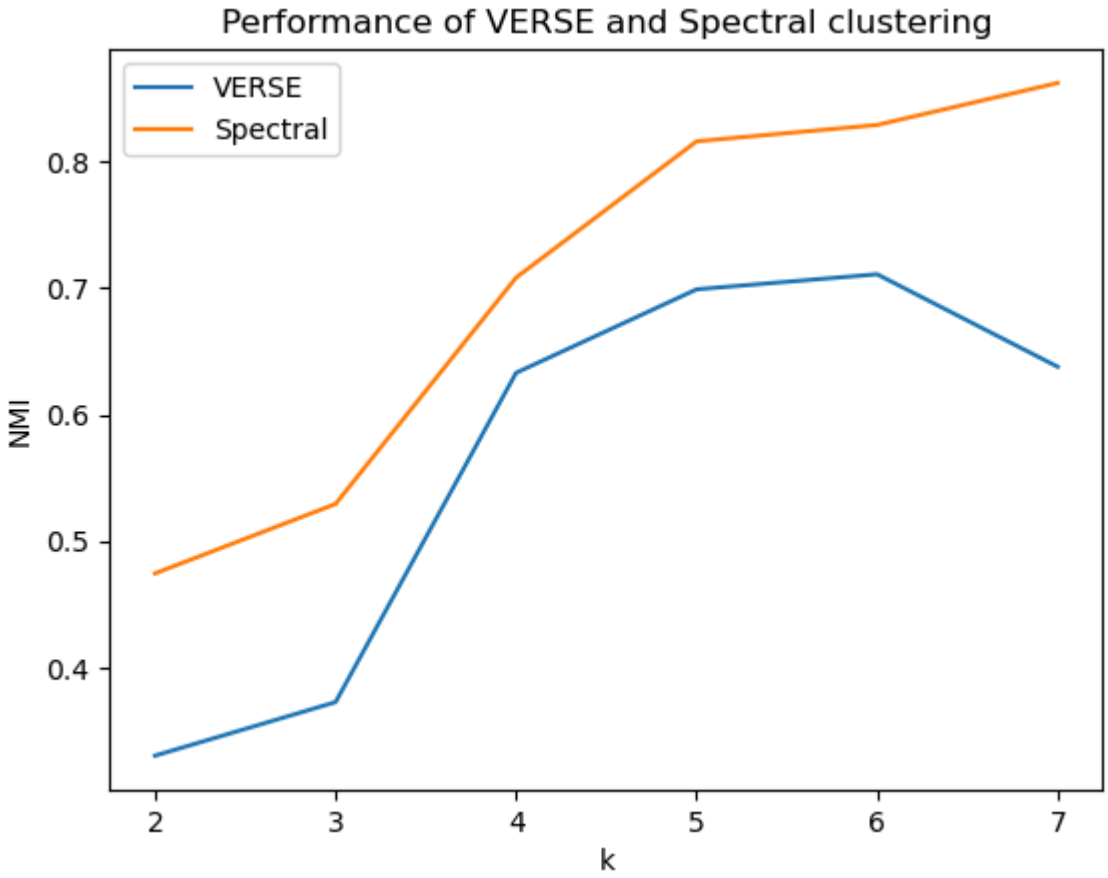
[Implement] a small piece of code that runs K-means on the embeddings with $k \in [2, 7]$ to evaluate the performance compared to Spectral clustering using the NMI as measure. You can use `sklearn.metrics.normalized_mutual_info_score` for the NMI and `sklearn.cluster.KMeans` for kmeans. In both cases,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

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

In [ ]:
```
### YOUR CODE HERE
from sklearn.metrics import normalized_mutual_info_score as NMI
from sklearn.cluster import KMeans

VERSE_NMI = []
Spectral_NMI =[]
ground_truth = email.communities


for k in range(2, 8):
    warnings.filterwarnings('ignore')
    print(f"k = {k}:")
    VERSE_labels = KMeans(k).fit(emb).labels_
    VERSE_NMI.append(NMI(ground_truth, VERSE_labels))

    Spectral_labels = spect_cluster(G, k=k)
    Spectral_NMI.append(NMI(ground_truth, Spectral_labels))
    print(f"For VERSE NMI = {VERSE_NMI[-1]:.3}",\
          f"For Spectral NMI = {Spectral_NMI[-1]:.3}", sep="\t")
    warnings.filterwarnings("default")

plt.plot(range(2, 8), VERSE_NMI, label="VERSE")
plt.plot(range(2, 8), Spectral_NMI, label="Spectral")
plt.title("Performance of VERSE and Spectral clustering")
plt.xlabel("k")
```

```
plt.ylabel("NMI")
plt.legend()
plt.show()
### YOUR CODE HERE
```

```
k = 2:
For VERSE NMI = 0.33    For Spectral NMI = 0.474
k = 3:
For VERSE NMI = 0.373   For Spectral NMI = 0.529
k = 4:
For VERSE NMI = 0.633   For Spectral NMI = 0.708
k = 5:
For VERSE NMI = 0.699   For Spectral NMI = 0.816
k = 6:
For VERSE NMI = 0.711   For Spectral NMI = 0.829
k = 7:
For VERSE NMI = 0.638   For Spectral NMI = 0.862
```



With the original 10_000 steps for VERSE the method performed abysmally, but after uping the amount of steps a lot to ensure convergence we observed that the performance of VERSE became very similar to that of spectral clustering, although the latter tends to gernerally outperform the former a tiny bit.

Overall the performance of the to methods seem VERY similar across different runs which could be a testement to similarities between the two methods.

Task 4.1.3 (1 points)

[Motivate] how you would conceptually expand the graph embeddings, 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.

Perhaps transform the multi-label-graph into a single-label problem, where multiple approaches to do this are possible. For example combining multiple lables into is own new label, which may however create numerous new labels/groups with few nodes in them. Another apporach could be to run classification on the graph, and letting a node keep the label to which is most strongly belongs to, before creating a graph embedding.

Task 4.2 (6 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 [ ]:
```
# 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
dii   = np.sum(A, axis=1, keepdims=False)
D     = np.diag(dii)

# Normalized Laplacian
D_inv  = np.diag(dii**(-0.5))
L      = D_inv @ A @ D_inv

# Create input vectors
X     = np.eye(G.order())
### TODO your code here

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

In [ ]:
```
# 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.input_features = input_features
        self.output_features = output_features
        self.activation = activation
        self.fc = nn.Linear(input_features, output_features)

        ### TODO Your code here

    def forward(self, X):
        ### TODO Your code here
        X = self.L @ X
        X = self.fc(X)
        if self.activation: X = self.activation(X)
        ### TODO Your code here
        return X
```

Define the modularity matrix and the modularity loss

In [ ]:
```
m = G.number_of_edges()

def modularity_matrix(A):
    B = None
    ### YOUR CODE HERE
    dii   = np.sum(A, axis=1, keepdims=False)
    dd = np.outer(dii, dii) / (2 * m)
    B = A - dd
    ### 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.linalg.norm(C)
    ### YOUR CODE HERE
    return loss
```

Compute labels from communities

In [ ]:
```
### Compute labels from communities
labels = None
### YOUR CODE HERE
labels = email.communities
### 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 the number
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 the forward
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(
    (fc): Linear(in_features=156, out_features=16, bias=True)
  )
  (1): GCNLayer(
    (fc): 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=539.64935
Epoch 100, loss=509.51981
Epoch 200, loss=47.82678
Epoch 300, loss=32.45679
Epoch 400, loss=26.83179
Epoch 500, loss=23.91821
Epoch 600, loss=22.65137
Epoch 700, loss=21.90430
Epoch 800, loss=21.28650
Epoch 900, loss=20.90869
Epoch 1000, loss=20.66821
Epoch 1100, loss=20.49084
Epoch 1200, loss=20.35266
Epoch 1300, loss=20.23743
Epoch 1400, loss=20.13367
Epoch 1500, loss=20.03015
Epoch 1600, loss=19.92578
Epoch 1700, loss=19.83618
Epoch 1800, loss=19.76343
Epoch 1900, loss=19.70020
Epoch 2000, loss=19.65283
```

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.

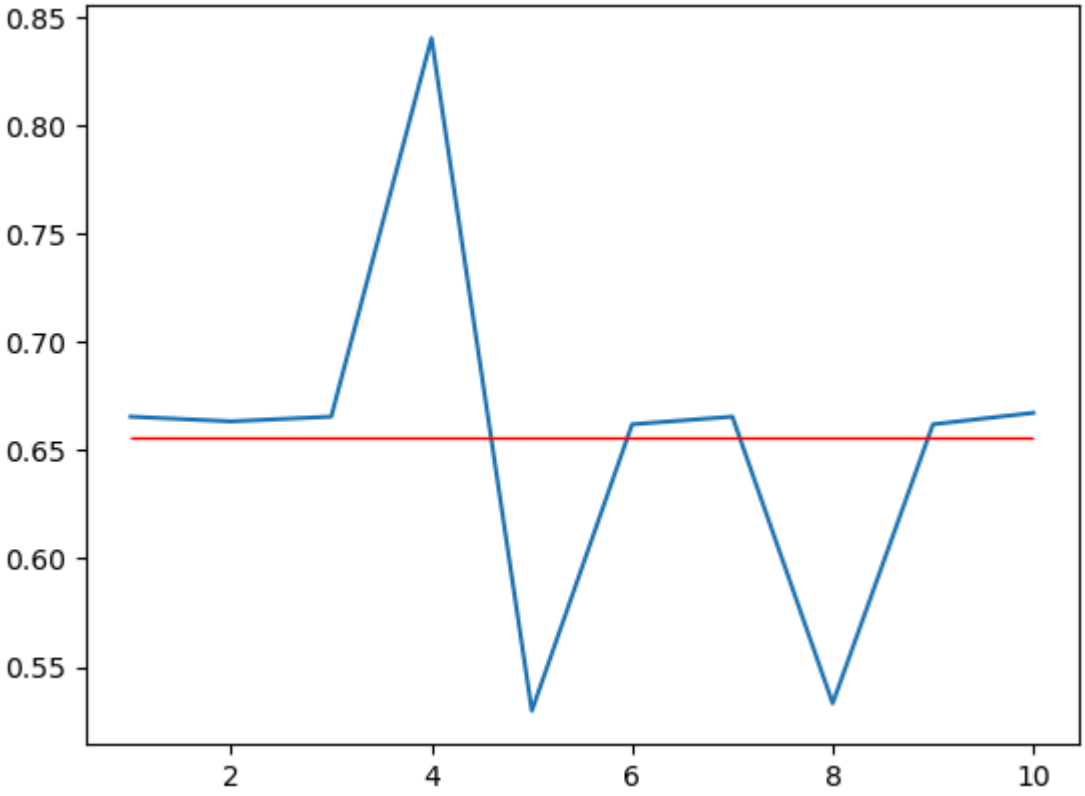
Note: You have to create the model 5 times otherwise you are keeping training the same model's parameters!

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

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_size=100,
    plt.axis('off')
    plt.show()

### YOUR CODE HERE
NMI_scores = []
for _ 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)

    B = modularity_matrix(A)
    optimizer = torch.optim.Adam(gcn.parameters(), lr=0.01)
    train_model(gcn, optimizer, X, B, epochs=epochs, print_every=1000)

    with torch.no_grad():
        output = gcn(X)
    _, y_pred = torch.max(output, 1)

    NMI_scores += [NMI(labels, y_pred)]

    print(f" NMI = {NMI(labels, y_pred):.3}")
### YOUR CODE HERE
```

```
plt.plot(range(1,11), NMI_scores)
plt.hlines(y=np.mean(NMI_scores), xmin=1, xmax=10, linewidth=1, color='r')
```

```
Epoch  0, loss=542.08374
Epoch 1000, loss=21.74829
Epoch 2000, loss=20.15698
 NMI = 0.665
Epoch  0, loss=541.11407
Epoch 1000, loss=22.02490
Epoch 2000, loss=19.83484
 NMI = 0.663
Epoch  0, loss=543.18909
Epoch 1000, loss=20.53845
Epoch 2000, loss=19.65393
 NMI = 0.665
Epoch  0, loss=540.30975
Epoch 1000, loss=-207.73767
Epoch 2000, loss=-212.52673
 NMI = 0.84
Epoch  0, loss=542.28247
Epoch 1000, loss=204.22339
Epoch 2000, loss=202.75580
 NMI = 0.53
Epoch  0, loss=538.95801
Epoch 1000, loss=20.92017
Epoch 2000, loss=19.76562
 NMI = 0.662
Epoch  0, loss=542.65790
Epoch 1000, loss=20.76746
Epoch 2000, loss=19.58911
 NMI = 0.665
Epoch  0, loss=546.82458
Epoch 1000, loss=204.35663
Epoch 2000, loss=202.78094
 NMI = 0.533
Epoch  0, loss=539.91339
Epoch 1000, loss=20.71631
Epoch 2000, loss=19.75647
 NMI = 0.662
Epoch  0, loss=543.34760
Epoch 1000, loss=22.28674
Epoch 2000, loss=20.15295
 NMI = 0.667
```

Out[ ]: <matplotlib.collections.LineCollection at 0x27bf5fd5f40>



In [ ]:
```
plot_graph(G, y_pred)
```