Complex Systems - Vingron part - Assignment 4

(2) (A) let $T, S \in \mathbb{R}^{n\times n}$ be stochastic and let $x \in \mathbb{R}^n_+$ be a distribution;
$\|x\|_1 = \sum x_i = 1$, let $\alpha \in [0,1] \Rightarrow \|Tx\|_1 = \|Sx\|_1 = 1$
then: $\|(\alpha T + (1-\alpha) S) x\|_1 \underset{\uparrow}{=} \alpha \|T\|_1 + (1-\alpha)\|S\|_1 = \alpha + (1-\alpha) = 1$
because $\alpha, (1-\alpha), x, T, S \geq 0$

(B) $Q \stackrel{\Delta}{=} \frac{1}{N} E$

$P_{n+1} = [\alpha Q + (1-\alpha) T] P_n = \ldots = [\alpha Q + (1-\alpha) T]^n P_0 \qquad q \stackrel{\Delta}{=} QP_0 = \frac{1}{N} \mathbb{1} \stackrel{\Delta}{=} P_0$

given that the limit exists: $\lim_{n\to\infty} P_n = P = \lim_{n\to\infty} [\alpha Q + (1-\alpha) T] P_n = [\alpha Q + (1-\alpha) T] P$

it must follow that $P = IP = [\alpha Q + (1-\alpha) T] P = \alpha QP + (1-\alpha) TP = \alpha q + (1-\alpha) TP$

$\Leftrightarrow \alpha q = (I - (1-\alpha) T) P$

$\Leftrightarrow P = \alpha (I - (1-\alpha) T)^{-1} q$

Remark for any $v \in \mathbb{R}^n_+$ s.t $\|v\|_1 = 1$ we have $Qv = q = (\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n})$
so P_0 doesn't matter as long as we pick a distribution.

hw(vingron)4_final

July 8, 2021

0.1 Problem 1

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import networkx as nx

from mpl_toolkits.axes_grid1 import make_axes_locatable

import matplotlib as mpl

from numpy.random import MT19937
from numpy.random import RandomState, SeedSequence

%pylab inline

def transitionMatrixG(G):
    """input G: a graph.
    output T: the transition matrix of G,
    column normalized.
    """
    A = nx.to_numpy_array(G)
    T = A.T / A.sum(axis=1)
    return T

def coreTransitionMatrixG(G):
    """Similar to transitionMatrixG but
    Returns the core normalized transition matrix of G.
    The cloumns are normalized.
    """
    A = nx.to_numpy_array(G)
    coreness = nx.core_number(G)
    coreness = np.array([coreness[k] for k in range(len(coreness))])
    A = A * coreness
    T = A.T / A.sum(axis=1)
    #T.sum(axis=0)
    return T
```

```python
def diffusionMatrix(T, alpha=0.2):
    """
    input T: a transition matrix (column normalized).
    input alpha: a the restart probability.
    Output K: the diffusion matrix, which is
    K = a [I - (1-a)T]^(-1)
    """
    n = T.shape[0]
    I = np.identity(n)
    K = I - (1 - alpha)*T
    K = alpha * np.linalg.inv(K)
    return K

def diffusionMatrixG(G, alpha=0.2, coreness=False):
    """
    input G: a networkz graph.
    input alpha: the restart parameter.
    input bool coreness: If True, the normalization uses core number rather
    than the standard adjacency matrix.
    Output K: the diffusion matrix, which is
    K = a [I - (1-a)T]^(-1)
    """
    #A = nx.to_numpy_array(G)
    #T = A.T / A.sum(axis=1)
    if coreness:
        T = coreTransitionMatrixG(G)
    else:
        T = transitionMatrixG(G)
    n = T.shape[0]
    I = np.identity(n)
    K = I - (1 - alpha)*T
    K = alpha * np.linalg.inv(K)
    return K


def RWR(T, alpha=0.2, q=1, epsilon=1e-6, maxiter=10**6):
    """Calculates the stationary distribution of a RWR process
    using the power method.
    input T: a transition matrix (column normalized).
    input alpha: restart probability.
    input q: restart distribution. If none is provided the uniform distribution
    is used (pageRank).
    input epsilon: the stop condition for the convergence.
    input maxiter: maximum number of iterations if convergence isn't reached.
    output p: the stationary distribution
    """
```

```python
    n = T.shape[0]
    if q==1:
        q = 1/n * np.ones(n)
    x = q
    y = alpha * q + (1 - alpha) * np.dot(T, x)
    #while np.linalg.norm((x-y)) > epsilon:
    for _ in range(maxiter):
        x = y
        y = alpha * q + (1 - alpha) * np.dot(T, x)
        if np.linalg.norm((x-y)) < epsilon:
            break
    return y

def RWRG(G, alpha=0.2, q=1, epsilon=1e-6, maxiter=10**6):
    """Calculates the stationary distribution of a RWR process
    using the power method.
    input G: a networkx graph.
    input alpha: restart probability.
    input q: restart distribution. If none is provided the uniform distribution
    is used (pageRank).
    input epsilon: the stop condition for the convergence.
    input maxiter: maximum number of iterations if convergence isn't reached.
    output p: the stationary distribution
    outut c: vector with the difference between iterations (convergence)
    """
    A = nx.to_numpy_array(G)
    #T = A.T / A.sum(axis=1)
    s =  A.sum(axis=1)
    s = s + (s == 0) # flip 0s
    T = A.T / s
    n = T.shape[0]
    c = np.zeros(maxiter)
    if q==1:
        q = 1/n * np.ones(n)
    x = q
    y = alpha * q + (1 - alpha) * np.dot(T, x)
    #while np.linalg.norm((x-y)) > epsilon:
    for i in range(maxiter):
        x = y
        y = alpha * q + (1 - alpha) * np.dot(T, x)
        c[i] = np.linalg.norm((x-y))
        if  c[i] < epsilon:
            break
    return y,c

#rs = RandomState(MT19937(SeedSequence(42)))
```

```
Populating the interactive namespace from numpy and matplotlib
```

0.2 (C) Create the following 5 random networks (remember to use a seed= 42) using networkX:

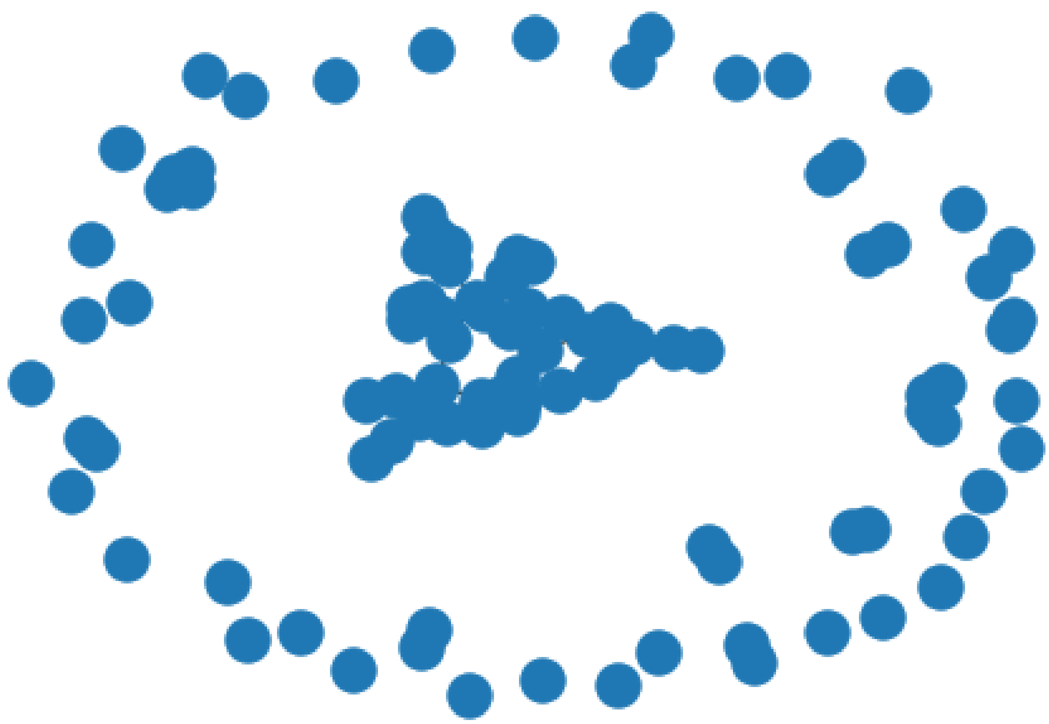
- Erdős-Rényi (for n = 100, p = {0.01, 0.08, 0.4}),
- Watts-Strogatz (for n = 50, k = 7, p = 0.3)
- Barabási-Albert (for n = 50 and m = 3).

0.2.1 Erdős-Rényi (for n = 100, p = 0.01)

[2]:
```
G1 = nx.erdos_renyi_graph(n=100, p=0.01, seed=42)
y1,c1 = RWRG(G1, alpha=0.15, q=1)
q1 = np.ones(100)/100
y1.sum()
```

[2]:
```
0.6939999999999998
```

[3]:
```
nx.draw_spring(G1)
```



[4]:
```
K1 = diffusionMatrixG(G1, alpha=0.15)
p1 = np.dot(K1,q1)
print("Direct Method states:", p1)
print("Difference between both RWR methods:", np.linalg.norm(y1-p1))
```

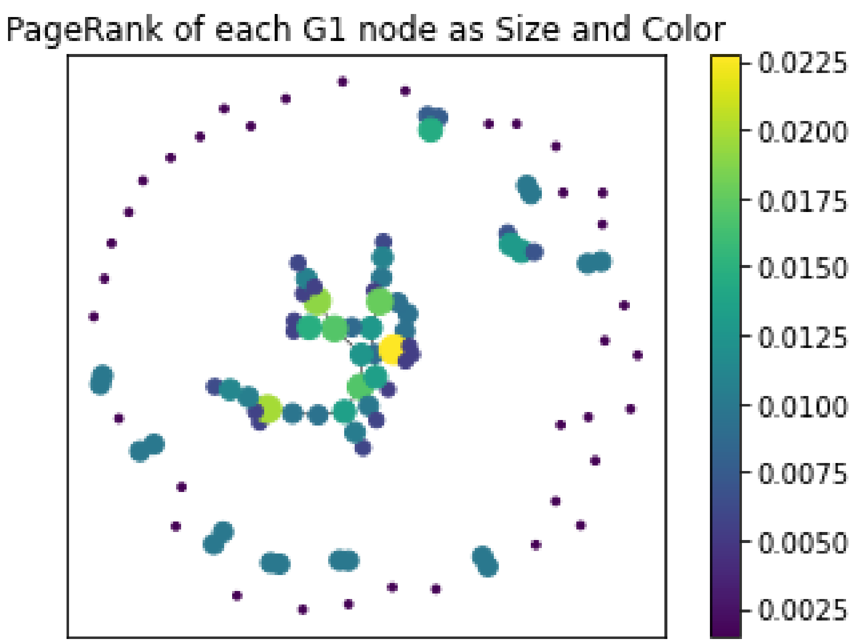
```
Direct Method states: [nan nan nan nan nan nan nan nan nan nan nan nan nan nan
```

```
nan nan nan nan
 nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan
 nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan
 nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan
 nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan
 nan nan nan nan nan nan nan nan nan nan]
Difference between both RWR methods: nan
```

```
D:\Anaconda3\lib\site-packages\ipykernel_launcher.py:22: RuntimeWarning: invalid
value encountered in true_divide
```

The graph is not connected and some of the vertices have 0 edges. Therefore this graph is not stochastic because a vertex without edges doesn't have a transition distribution. So we can't really talk about pageRank of this graph.

[5]:
```
plt.imshow(y1.reshape((10,10)))
plt.colorbar()
plt.cla()
pos = nx.layout.spring_layout(G1)
nodes = nx.draw_networkx_nodes(G1, pos, node_size=5000*y1, node_color=y1,␣
 ↪cmap=plt.cm.viridis)
edges = nx.draw_networkx_edges(G1,pos,width=0.5)
plt.title("PageRank of each G1 node as Size and Color")
plt.show()
```



[6]:
```
print("All pagerank values:")
for node, rank in zip(G1.nodes(), y1):
```

```python
    print(node, ":", rank)
```

```
All pagerank values:
0 : 0.00571489608786104
1 : 0.007702588999769012
2 : 0.009931270583889268
3 : 0.009468170029973142
4 : 0.0127794629255878
5 : 0.0015
6 : 0.0015
7 : 0.009738816514354999
8 : 0.0015
9 : 0.008912444439415692
10 : 0.01
11 : 0.0015
12 : 0.01257947536010542
13 : 0.0015
14 : 0.0015
15 : 0.007702588999769012
16 : 0.0015
17 : 0.0015
18 : 0.0015
19 : 0.0015
20 : 0.019835127785025056
21 : 0.0015
22 : 0.0052915755187352915
23 : 0.0015
24 : 0.016924176449673655
25 : 0.00701754385964912
26 : 0.0015
27 : 0.014594822000461973
28 : 0.01
29 : 0.01
30 : 0.01913516241885217
31 : 0.0015
32 : 0.0052765458491523184
33 : 0.009427148492464923
34 : 0.0015
35 : 0.005894086966860439
36 : 0.01
37 : 0.013381932593035559
38 : 0.00571489608786104
39 : 0.005566219402370979
40 : 0.012982456140350875
41 : 0.0015
42 : 0.008741561707967714
43 : 0.009283058291206799
44 : 0.0015
```

45 : 0.009974720805407398
46 : 0.0015
47 : 0.0015
48 : 0.0015
49 : 0.01
50 : 0.00936893723424102
51 : 0.01
52 : 0.00615472473473695
53 : 0.0015
54 : 0.0015
55 : 0.0015
56 : 0.01
57 : 0.005692982818646525
58 : 0.01
59 : 0.01703855737988888
60 : 0.0015
61 : 0.01
62 : 0.0015
63 : 0.0015
64 : 0.01071032238381748
65 : 0.022720272423282336
66 : 0.00536242479218654
67 : 0.010542435489445324
68 : 0.00701754385964912
69 : 0.005692982818646525
70 : 0.00536242479218654
71 : 0.01
72 : 0.012982456140350875
73 : 0.0015
74 : 0.00536242479218654
75 : 0.010952367796106026
76 : 0.0015
77 : 0.0015
78 : 0.013514365173985967
79 : 0.01
80 : 0.0015
81 : 0.01
82 : 0.014798765623081944
83 : 0.005566219402370979
84 : 0.0015
85 : 0.0015
86 : 0.01
87 : 0.01
88 : 0.0015
89 : 0.0015
90 : 0.01033912945962275
91 : 0.0015
92 : 0.01

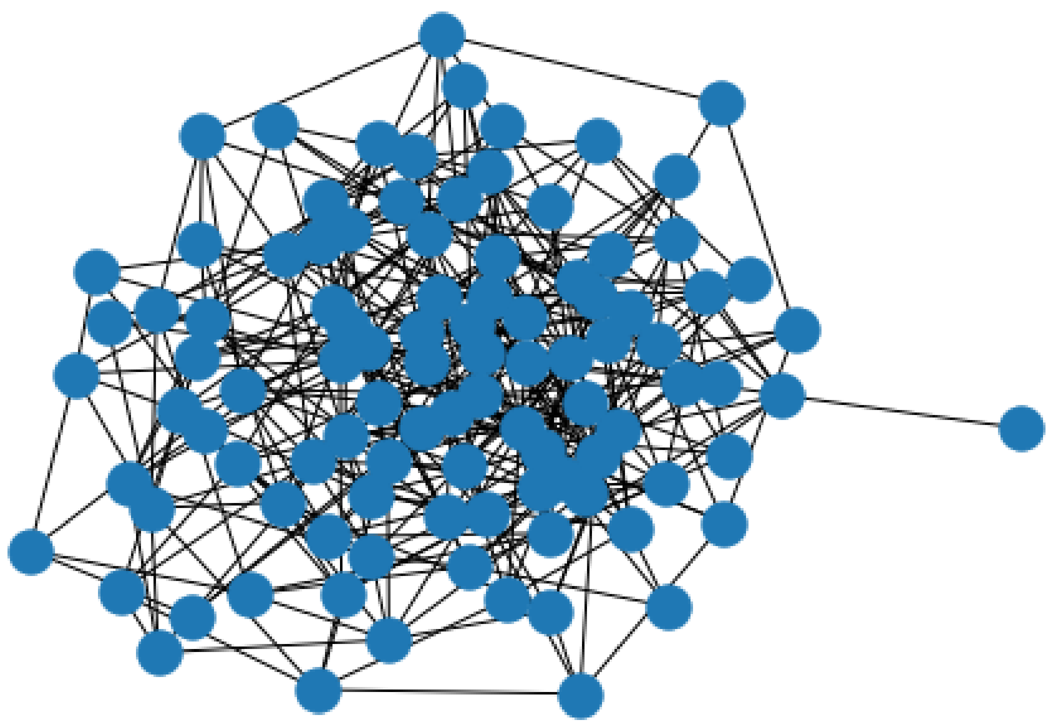
```
93 : 0.01
94 : 0.006327539401584285
95 : 0.006051890766069046
96 : 0.017772105969235286
97 : 0.011359091609947473
98 : 0.005739286828930811
99 : 0.0015
```

0.2.2 Erdős-Rényi (for n = 100, p = 0.08)

[7]:
```
G2 = nx.erdos_renyi_graph(n=100, p=0.08, seed=42)
y2,c2 = RWRG(G2, alpha=0.15, q=1)
q2 = np.ones(100)/100
y2.sum()
```

[7]:
```
0.9999999999999996
```

[8]:
```
nx.draw_spring(G2)
```

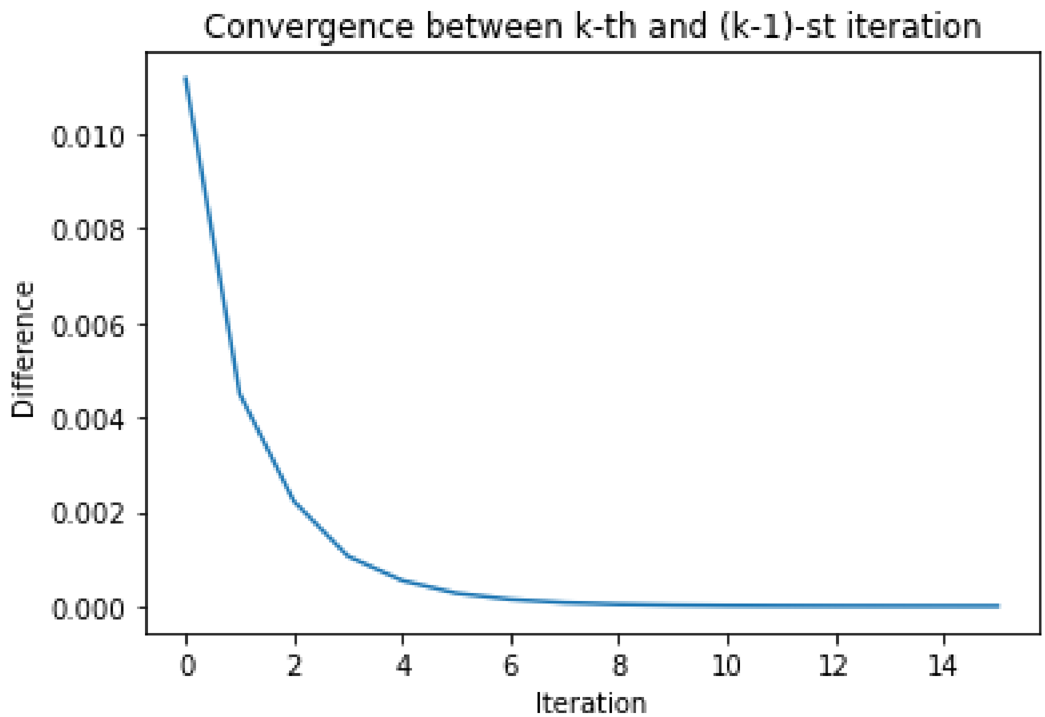


[9]:
```
K2 = diffusionMatrixG(G2, alpha=0.15)
p2 = np.dot(K2,q2)
print("Difference between both RWR methods:", np.linalg.norm(y2-p2)) # we see␣
↪that both methods give very similar result
```

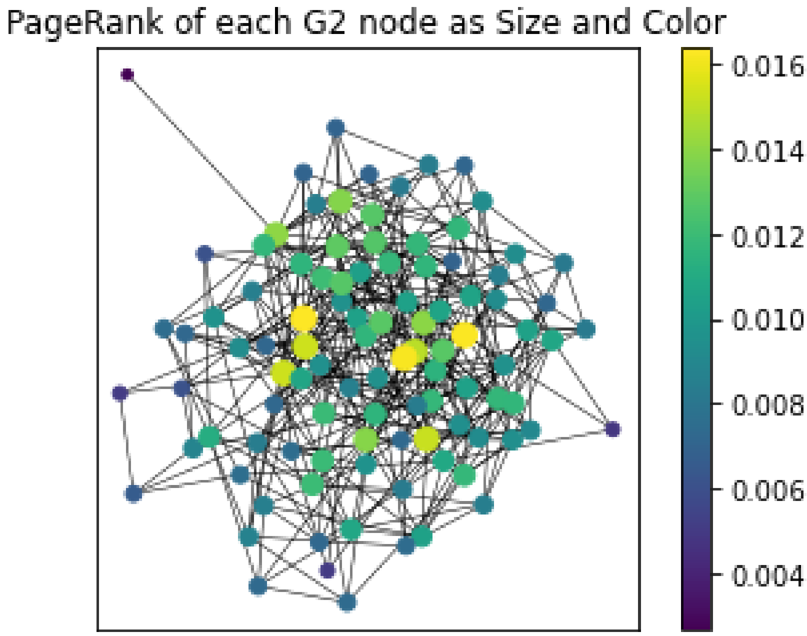
```
Difference between both RWR methods: 3.7494711607160316e-07
```

[10]:
```
numIters2 = (c2 > 0).sum()
print("Number of Iterations till convergence:", numIters2)

plt.plot(np.arange(numIters2+1),c2[:numIters2+1])
plt.ylabel("Difference")
plt.xlabel("Iteration")
plt.title("Convergence between k-th and (k-1)-st iteration")
plt.show()
```

```
Number of Iterations till convergence: 15
```



[11]:
```
plt.imshow(y2.reshape((10,10)))
plt.colorbar()
plt.cla()
pos = nx.layout.spring_layout(G2)
nodes = nx.draw_networkx_nodes(G2, pos, node_size=5000*y2, node_color=y2,␣
 ↪cmap=plt.cm.viridis)
edges = nx.draw_networkx_edges(G2,pos,width=0.5)
plt.title("PageRank of each G2 node as Size and Color")
plt.show()
```



[12]:
```
print("All pagerank values:")
for node, rank in zip(G2.nodes(), y2):
    print(node, ":", rank)
```

```
All pagerank values:
0 : 0.011749878209451637
1 : 0.009465248127316207
2 : 0.010514319992097298
3 : 0.010830585679058487
4 : 0.010460926095139563
5 : 0.010625321938303931
6 : 0.008606758920755157
7 : 0.008502462790574681
8 : 0.007378333661641805
9 : 0.007155245990874632
10 : 0.011693165073391774
11 : 0.009237139161359468
12 : 0.014963756279054077
13 : 0.010868164869989096
14 : 0.008495207671811283
15 : 0.012781847232808239
16 : 0.009093824970101923
17 : 0.011805817606789497
18 : 0.006611326057421513
19 : 0.008490508968972797
20 : 0.007454687064277814
```

21 : 0.011661695420044386
22 : 0.011753558277567685
23 : 0.0026934449972816966
24 : 0.01636064210204045
25 : 0.009549916946910339
26 : 0.005080451416852819
27 : 0.013995416993002366
28 : 0.009339344620407601
29 : 0.008377616699155661
30 : 0.015214186210240227
31 : 0.01208564653646586
32 : 0.011493077140092709
33 : 0.009561960909723506
34 : 0.015311283550313231
35 : 0.007270577408410804
36 : 0.007051503375088265
37 : 0.013848726969141937
38 : 0.007313552977426056
39 : 0.011667061763950897
40 : 0.009345476475834969
41 : 0.004865276221600505
42 : 0.01280513057839638
43 : 0.009525412726675564
44 : 0.010448374908361086
45 : 0.012798959375884527
46 : 0.006114452653296153
47 : 0.00505925130010101
48 : 0.008267191176701922
49 : 0.010521671691159964
50 : 0.014040263426018586
51 : 0.008338231490175487
52 : 0.012005211096472089
53 : 0.008878197049689273
54 : 0.010731203267833438
55 : 0.009668341658256743
56 : 0.007216248481540845
57 : 0.008252305050324874
58 : 0.011784631659736207
59 : 0.0162769369974545
60 : 0.007053475805135779
61 : 0.008230795575284477
62 : 0.008529792941692687
63 : 0.01069947791475565
64 : 0.009436576986596439
65 : 0.012726377677507303
66 : 0.015317392982404233
67 : 0.008725416734094436
68 : 0.00719377363582689

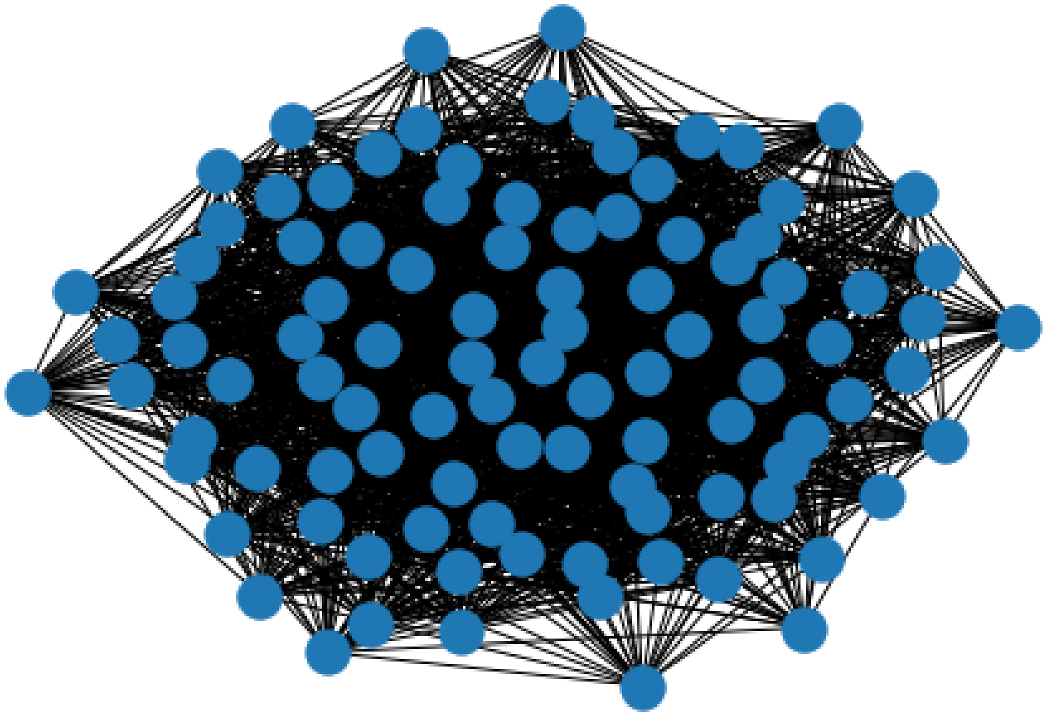
```
69 : 0.011958099752460424
70 : 0.01164310987037751
71 : 0.007394186739309664
72 : 0.011747404362599445
73 : 0.007009803821583052
74 : 0.01160774280333405
75 : 0.011936963980605274
76 : 0.009580858392950318
77 : 0.009342932518663949
78 : 0.011605966820517528
79 : 0.009540430648681727
80 : 0.010480283368862678
81 : 0.007260286559725087
82 : 0.010353285523510957
83 : 0.007549999081323366
84 : 0.012945711646350087
85 : 0.007592919672076178
86 : 0.008553103740043273
87 : 0.01276660808047725
88 : 0.00928086641994394
89 : 0.006111541900393492
90 : 0.0162236409424093
91 : 0.007234265609850844
92 : 0.007065223389912502
93 : 0.01171221510187361
94 : 0.007535692952087775
95 : 0.010788363958673938
96 : 0.013880592096264781
97 : 0.011195483163054219
98 : 0.008419138262259531
99 : 0.00841324060570454
```

0.2.3 Erdős-Rényi (for n = 100, p = 0.4)

[13]:
```
G3 = nx.erdos_renyi_graph(n=100, p=0.4, seed=42)
y3,c3 = RWRG(G3, alpha=0.15, q=1)
q3 = np.ones(100)/100
y3.sum()
```

[13]:
```
0.9999999999999996
```

[14]:
```
nx.draw_spring(G3)
```



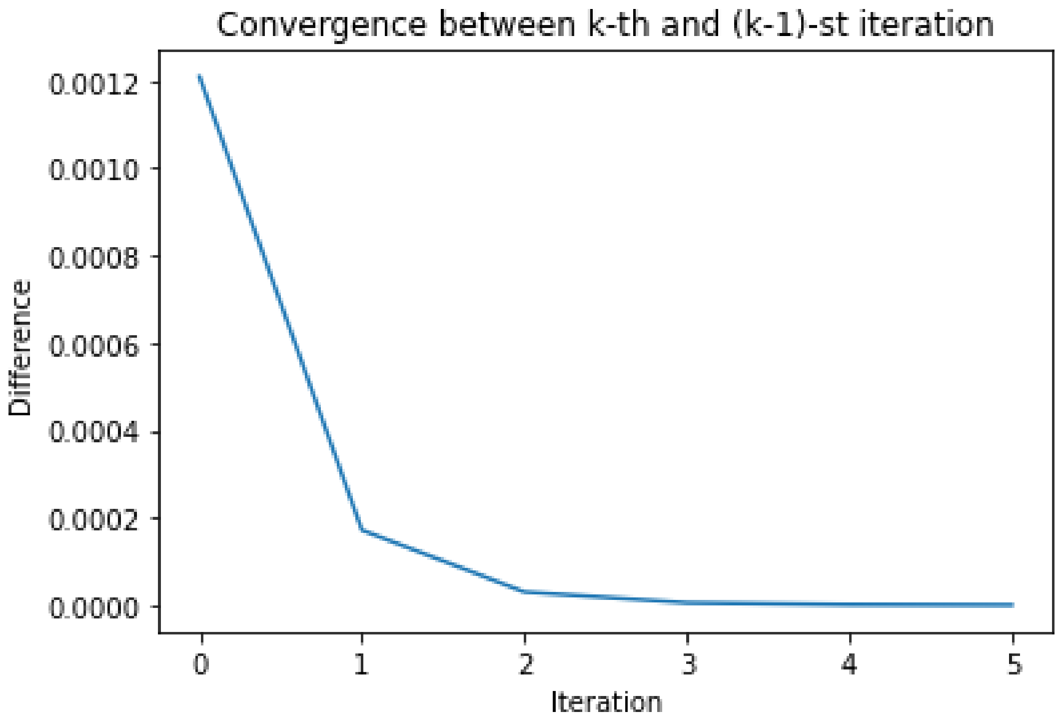
[15]:
```
K3 = diffusionMatrixG(G3, alpha=0.15)
p3 = np.dot(K3,q3)
print("Difference between both RWR methods:", np.linalg.norm(y3-p3)) # we see␣
↪that both methods give very similar result
```

```
Difference between both RWR methods: 1.6364350313944108e-07
```

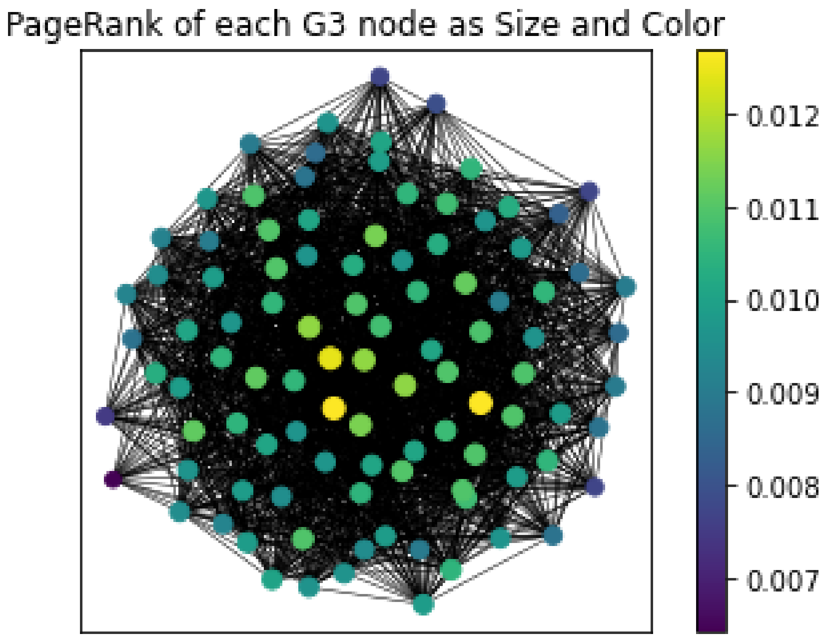
[16]:
```
numIters3 = (c3 > 0).sum()
print("Number of Iterations till convergence:", numIters3)
```

```
Number of Iterations till convergence: 5
```

[17]:
```
plt.plot(np.arange(numIters3+1),c3[:numIters3+1])
plt.ylabel("Difference")
plt.xlabel("Iteration")
plt.title("Convergence between k-th and (k-1)-st iteration")
plt.show()
```



[18]:
```
plt.imshow(y3.reshape((10,10)))
plt.colorbar()
plt.cla()
pos = nx.layout.spring_layout(G3)
nodes = nx.draw_networkx_nodes(G3, pos, node_size=5000*y3, node_color=y3,␣
 ↪cmap=plt.cm.viridis)
edges = nx.draw_networkx_edges(G3,pos,width=0.5)
plt.title("PageRank of each G3 node as Size and Color")
plt.show()
```



[19]:
```
print("All pagerank values:")
for node, rank in zip(G3.nodes(), y3):
    print(node, ":", rank)
```

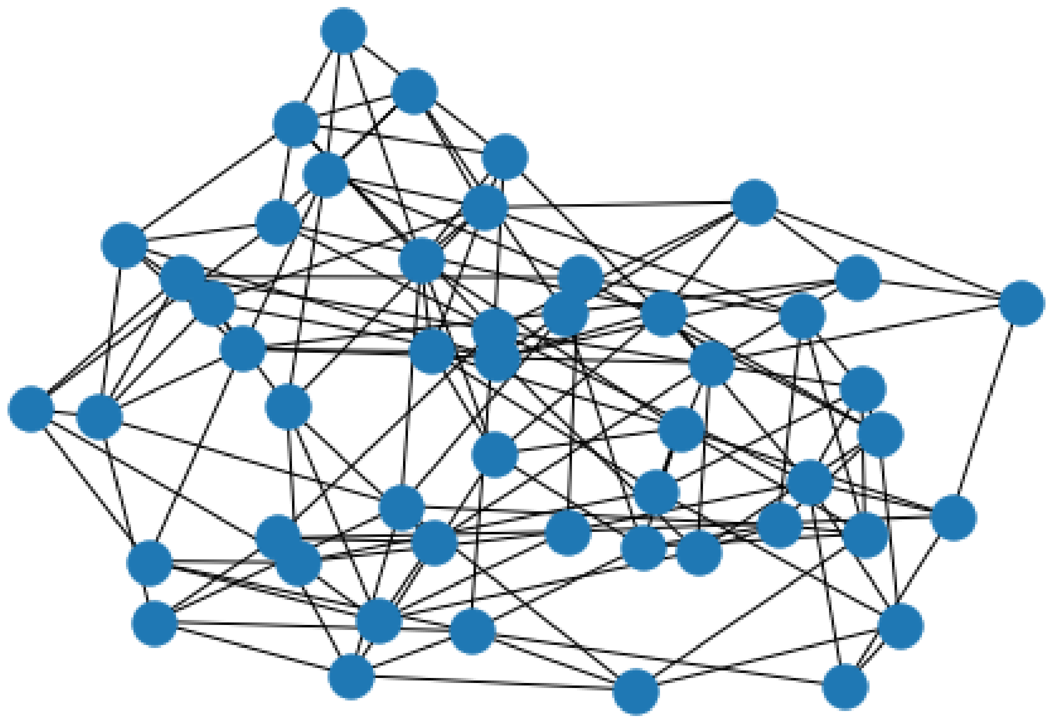
```
All pagerank values:
0 : 0.011639117850394565
1 : 0.00963237330709581
2 : 0.009857843972450276
3 : 0.010105639246459377
4 : 0.008806032119144921
5 : 0.009670513223449643
6 : 0.009663641725543865
7 : 0.009001737050334078
8 : 0.007519675464541473
9 : 0.008817448172728599
10 : 0.009869813794178244
11 : 0.009202749787044474
12 : 0.010954958295328408
13 : 0.009641122596384232
14 : 0.01053574384344565
15 : 0.009684013575482294
16 : 0.009871689610652669
17 : 0.009021349903312924
18 : 0.009718688345388353
19 : 0.009651115411778531
20 : 0.009500116557377243
```

```
21 : 0.008597458993464629
22 : 0.01098340166396368
23 : 0.009049624951346803
24 : 0.010104339931135334
25 : 0.010752669089095376
26 : 0.007715435265582642
27 : 0.009848706123604887
28 : 0.010106136073615416
29 : 0.009671512707003147
30 : 0.01119177624938665
31 : 0.008826253012127696
32 : 0.01054049272319639
33 : 0.009901595222109257
34 : 0.011147630855318882
35 : 0.008630576300548011
36 : 0.010116022084589215
37 : 0.012666562779214859
38 : 0.006434091034033808
39 : 0.010938187918818988
40 : 0.009873268056322496
41 : 0.009461893307345787
42 : 0.011617656576866385
43 : 0.010955584670949122
44 : 0.010693324231309784
45 : 0.01143484625245953
46 : 0.009460037391215492
47 : 0.009869261756154728
48 : 0.00902674099569965
49 : 0.010115575003062932
50 : 0.010547891575140507
51 : 0.010502633188870129
52 : 0.010527108574770343
53 : 0.007740935991982554
54 : 0.010104004681796713
55 : 0.00944399727499631
56 : 0.010963875182420896
57 : 0.010516924953372045
58 : 0.009244700256121233
59 : 0.009709026741152262
60 : 0.008780067189768965
61 : 0.01031504218993461
62 : 0.009664725264712744
63 : 0.012438567246145976
64 : 0.00988535578423685
65 : 0.010503496016261599
66 : 0.010537505634768362
67 : 0.010342297678116796
68 : 0.009017391596749767
```

```
69 : 0.010981047413026001
70 : 0.008352526476611105
71 : 0.010745331369971254
72 : 0.009676098992963554
73 : 0.010961226983804398
74 : 0.010065904434572384
75 : 0.009860792698586064
76 : 0.010516081305099966
77 : 0.010327468391855537
78 : 0.010933348837742241
79 : 0.01267993733854762
80 : 0.010779140097949549
81 : 0.009660243113005623
82 : 0.009244838754769552
83 : 0.010505443173179405
84 : 0.01030636230763719З
85 : 0.011632320530413362
86 : 0.010112878761101092
87 : 0.010992267600129635
88 : 0.00861821083666434
89 : 0.00988122019836635
90 : 0.010915565968414134
91 : 0.009003243539363935
92 : 0.009691352652393237
93 : 0.011158023977156638
94 : 0.01139883933194049
95 : 0.01096081886959504
96 : 0.010950578794280078
97 : 0.007749536673122198
98 : 0.007933206488667176
99 : 0.010524551995694624
```

0.2.4 Watts-Strogatz (for n = 50, k = 7, p = 0.3)

[20]:
```
G4 = nx.watts_strogatz_graph(n=50, k=7, p=0.3, seed=42)
q4 = np.ones(50)/50
nx.draw_spring(G4)
```



[21]:
```
y4,c4 = RWRG(G4, alpha=0.15, q=1)
y4.sum()
```

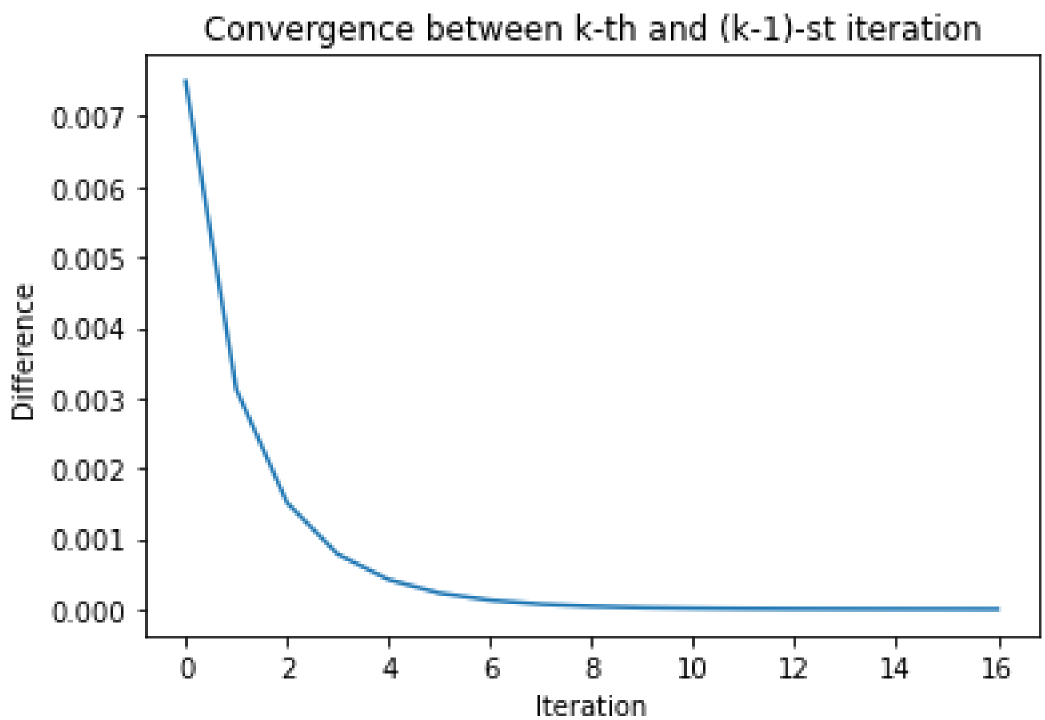
[21]:
```
0.9999999999999996
```

[22]:
```
K4 = diffusionMatrixG(G4, alpha=0.15)
p4 = np.dot(K4,q4)
print("Difference between both RWR methods:", np.linalg.norm(y4-p4)) # we see␣
↪that both methods give very similar result
```

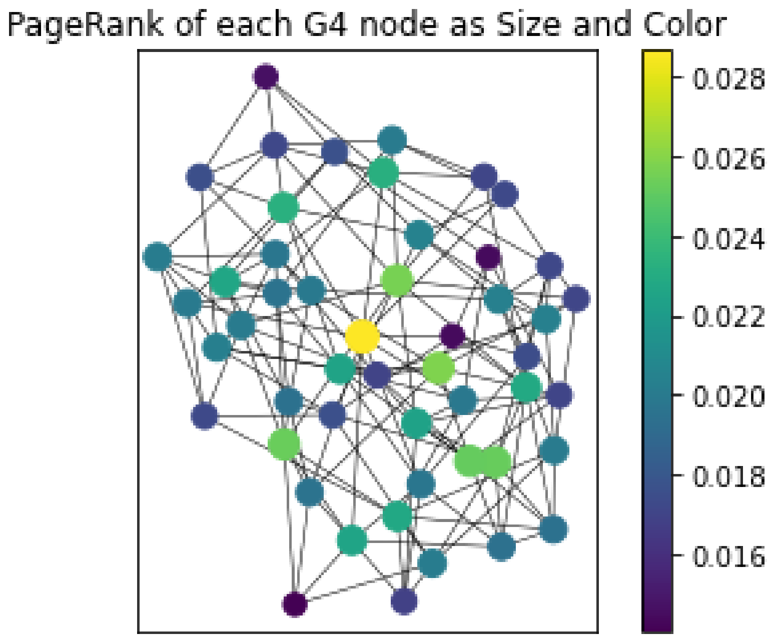
```
Difference between both RWR methods: 1.0868785762758303e-06
```

[23]:
```
numIters4 = (c4 > 0).sum()
print("Number of Iterations till convergence:", numIters4)
plt.plot(np.arange(numIters4+1),c4[:numIters4+1])
plt.ylabel("Difference")
plt.xlabel("Iteration")
plt.title("Convergence between k-th and (k-1)-st iteration")
plt.show()
```

```
Number of Iterations till convergence: 16
```



[24]:
```
plt.imshow(y4.reshape((10,5)))
plt.colorbar()
plt.cla()
pos = nx.layout.spring_layout(G4)
nodes = nx.draw_networkx_nodes(G4, pos, node_size=5000*y4, node_color=y4,␣
 ↪cmap=plt.cm.viridis)
edges = nx.draw_networkx_edges(G4,pos,width=0.5)
plt.title("PageRank of each G4 node as Size and Color")
plt.show()
```



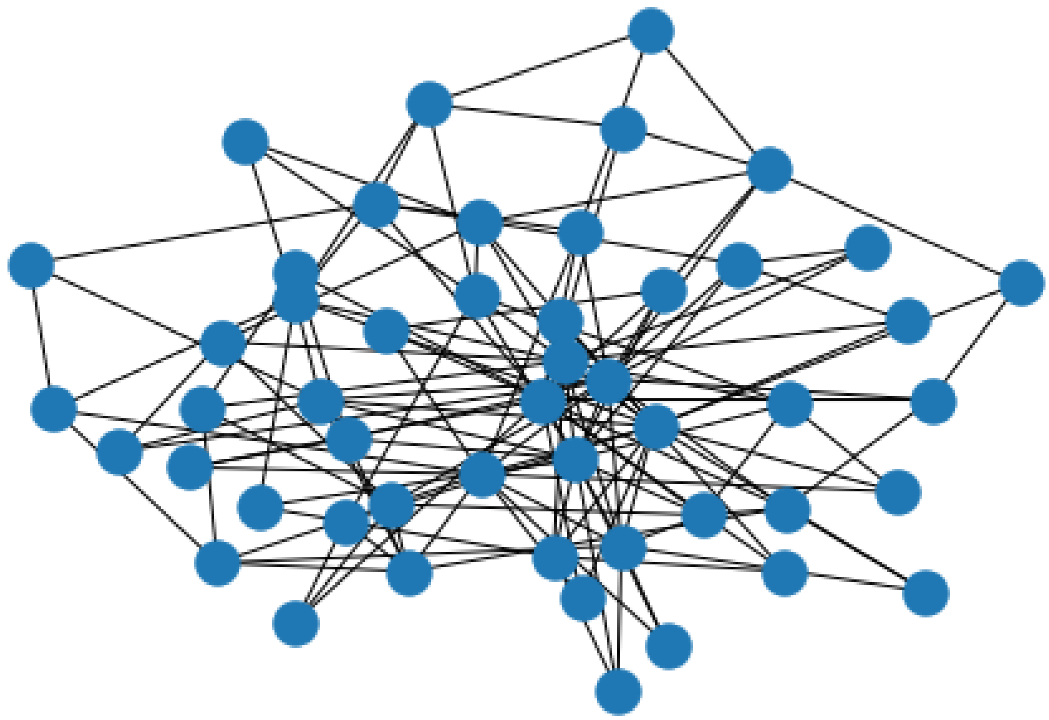
[25]:
```
print("All pagerank values:")
for node, rank in zip(G4.nodes(), y4):
    print(node, ":", rank)
```

```
All pagerank values:
0 : 0.023296177814411886
1 : 0.017576840897649916
2 : 0.023237885380037564
3 : 0.017231217245334832
4 : 0.020098957633082715
5 : 0.017109819591521042
6 : 0.025640500959879547
7 : 0.01728582882946785
8 : 0.017248993929739602
9 : 0.017135412637715303
10 : 0.020374716765371552
11 : 0.0203814608291317
12 : 0.020072387666177464
13 : 0.020549166254589402
14 : 0.02015783996532755
15 : 0.02278543396413318
16 : 0.017656191212874497
17 : 0.02030331491279442
18 : 0.017301899698200125
19 : 0.020089043962648327
20 : 0.0200319944377319
```

```
21 : 0.020115731787028197
22 : 0.01978194933938149
23 : 0.01997791783665201
24 : 0.022719163999173526
25 : 0.01959757857291608
26 : 0.01951912639598511
27 : 0.025306507524790904
28 : 0.014069830799798549
29 : 0.019634721024229508
30 : 0.022595956215818943
31 : 0.0286467441418698
32 : 0.019782088193145052
33 : 0.016898124721865446
34 : 0.014454942394228922
35 : 0.017470717663728767
36 : 0.022876347398072046
37 : 0.01950424137513895
38 : 0.02525618875073178
39 : 0.019448044405525645
40 : 0.02519981009793475
41 : 0.02247784860635125
42 : 0.017028696440761243
43 : 0.017042703800854365
44 : 0.014465530520219462
45 : 0.025803352358488692
46 : 0.019907930032962144
47 : 0.0176848277059894
48 : 0.022523228849419732
49 : 0.014645064459117445
```

0.2.5 Barabási-Albert (for n = 50 and m = 3)

[26]:
```
G5 = nx.barabasi_albert_graph(n=50, m=3, seed=42)
q5 = np.ones(50)/50
nx.draw_spring(G5)
```



[27]:
```
y5,c5 = RWRG(G5, alpha=0.15, q=1)
y5.sum()
```

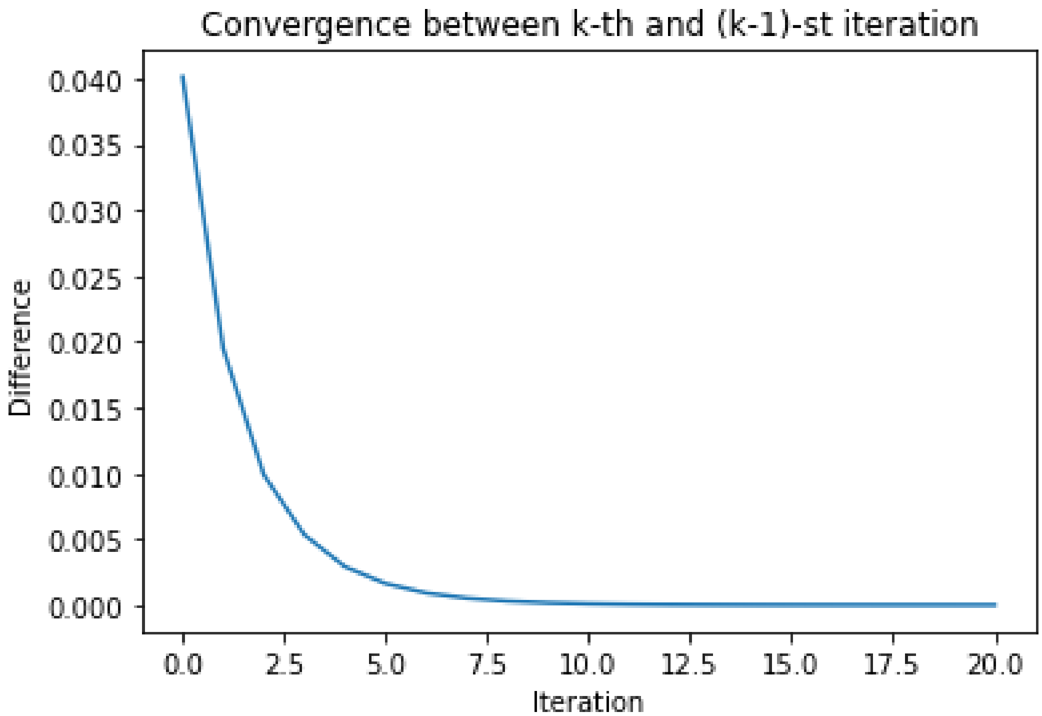
[27]: 0.9999999999999998

[28]:
```
K5 = diffusionMatrixG(G5, alpha=0.15)
p5 = np.dot(K5,q5)
print("Difference between both RWR methods:", np.linalg.norm(y5-p5)) # we see␣
↪that both methods give very similar result
```

```
Difference between both RWR methods: 2.6716341454496354e-07
```

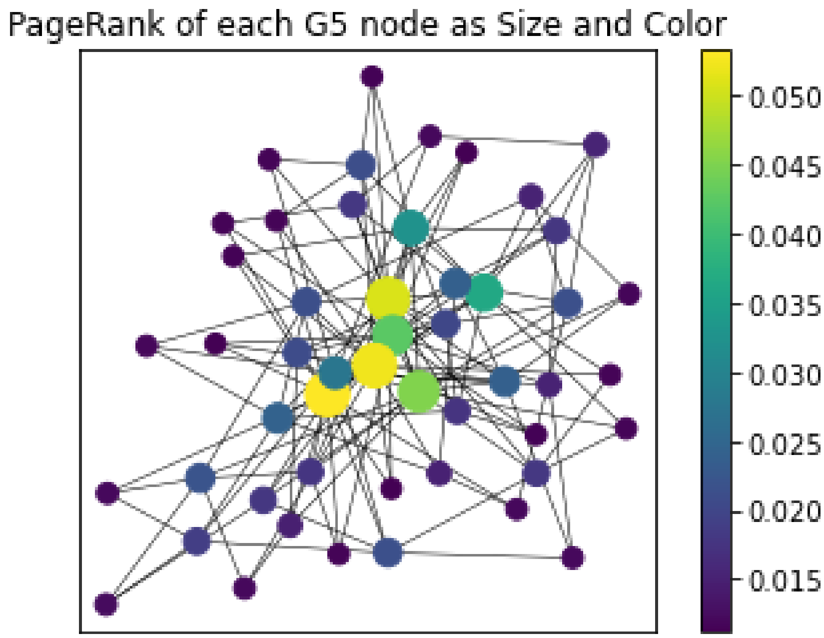
[29]:
```
numIters5 = (c5 > 0).sum()
print("Number of Iterations till convergence:", numIters5)
plt.plot(np.arange(numIters5+1),c5[:numIters5+1])
plt.ylabel("Difference")
plt.xlabel("Iteration")
plt.title("Convergence between k-th and (k-1)-st iteration")
plt.show()
```

```
Number of Iterations till convergence: 20
```



[30]:

```
plt.imshow(y5.reshape((10,5)))
plt.colorbar()
plt.cla()
pos = nx.layout.spring_layout(G5)
nodes = nx.draw_networkx_nodes(G5, pos, node_size=5000*y5, node_color=y5,␣
 ↪cmap=plt.cm.viridis)
edges = nx.draw_networkx_edges(G5,pos,width=0.5)
plt.title("PageRank of each G5 node as Size and Color")
plt.show()
```



[31]:
```
print("All pagerank values:")
for node, rank in zip(G5.nodes(), y5):
    print(node, ":", rank)
```

```
All pagerank values:
0 : 0.02393378123421
1 : 0.05331557345802402
2 : 0.03246354716956735
3 : 0.050978127368944894
4 : 0.045378394548766895
5 : 0.04256705172199071
6 : 0.011250827813334339
7 : 0.020286822638053877
8 : 0.05237246092960013
9 : 0.03644269117771701
10 : 0.02114347104385858
11 : 0.02081431728975516
12 : 0.024021302699037007
13 : 0.014966268506538545
14 : 0.02769054349692764
15 : 0.01762471864162042
16 : 0.01773452898711803
17 : 0.011462927849418478
18 : 0.024299419415420655
19 : 0.015071690051659151
20 : 0.017944420134685512
```

```
21 : 0.02158102174693858
22 : 0.018132078028503824
23 : 0.011645656798328598
24 : 0.011322864680116715
25 : 0.021472879585911462
26 : 0.017961778517871625
27 : 0.018921043441236105
28 : 0.021307724361656934
29 : 0.011490359098228237
30 : 0.015248755867886588
31 : 0.011653605965325057
32 : 0.011911383529386494
33 : 0.015235837602693671
34 : 0.022189612995755684
35 : 0.018147003098186815
36 : 0.0113978693607951
37 : 0.012201199280803147
38 : 0.011901739303523819
39 : 0.012077058687260698
40 : 0.011680755807134797
41 : 0.012106114795314073
42 : 0.015363093551318041
43 : 0.012410592787101833
44 : 0.012192421873984984
45 : 0.011561682513857887
46 : 0.011609969396879243
47 : 0.011630465629027156
48 : 0.011700556223329075
49 : 0.01218198929539524
```

0.2.6 (E)

Try different initial distributions. Does it change the end result?

p_0 doesn't change the result, becaue $E \cdot v = (1/n, \ldots, 1/n)$ for any distribution vector, where E is the matrix with all entries equal $1/n$.

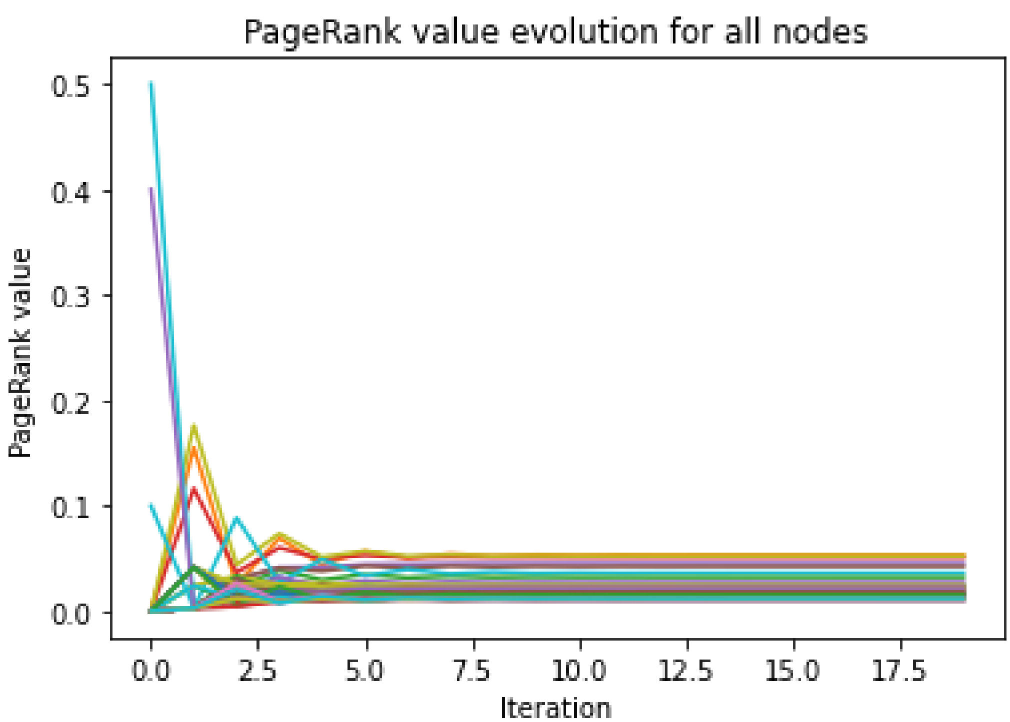
0.2.7 (G)

In the Barabási-Albert network randomly assign the probabilities $\{0.4, 0.1, 0.5\}$ to 3 nodes (the rest should have a 0 assigned) and propagate these scores in the network using the iterative approach in RWR. Plot how the PageRank value changes (X-axis - iteration, Y-axis - PageRank value) for all nodes (overlay the 50 lines in one figure).

[32]:
```python
G = nx.barabasi_albert_graph(n=50, m=3, seed=42)
T = transitionMatrixG(G)
q = np.zeros_like(T[0])
q[:3]=[0.5,0.4,0.1]
np.random.shuffle(q)
```

```
its = 20 #number of iterations
pranks1 = np.zeros((its, 50)) #here we'll store the results of each iteration
pranks1[0]=q
q = np.ones_like(q)/50
a = 0.15
for i in range(1,its,1):
    pranks1[i] = a*q + (1-a) * np.dot(T,pranks1[i-1])

# plot
for node in range(50):
    #plt.plot(np.arange(50),pranks[node])
    x = np.arange(its)
    y = pranks1[:,node]
    plt.plot(x,y)
#
plt.xlabel("Iteration")
plt.ylabel("PageRank value")
plt.title("PageRank value evolution for all nodes")
plt.plot()
```

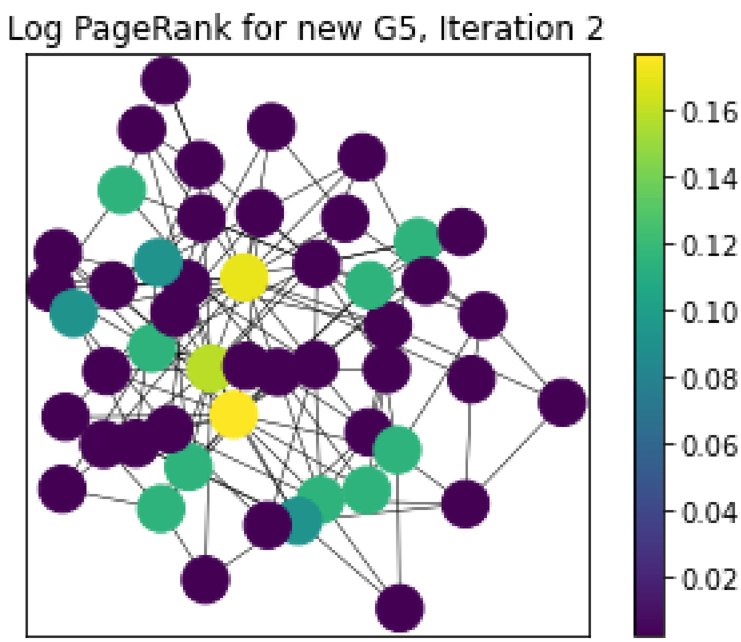
[32]: []

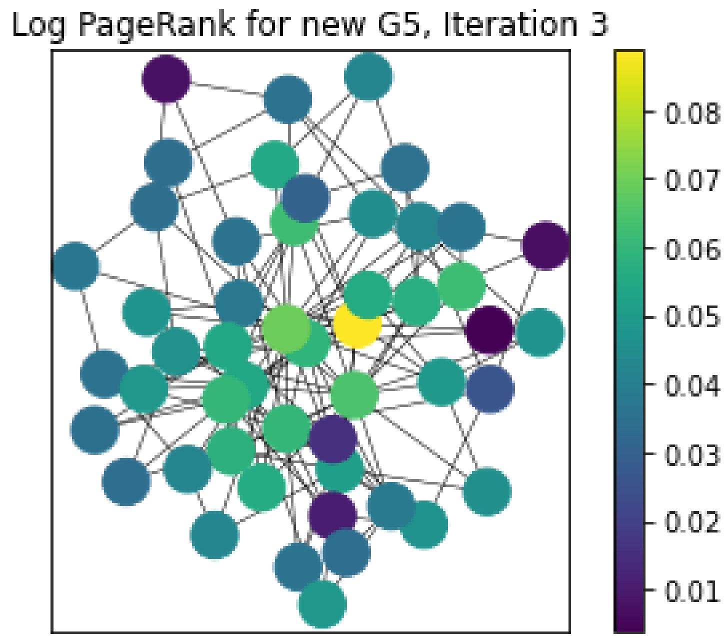


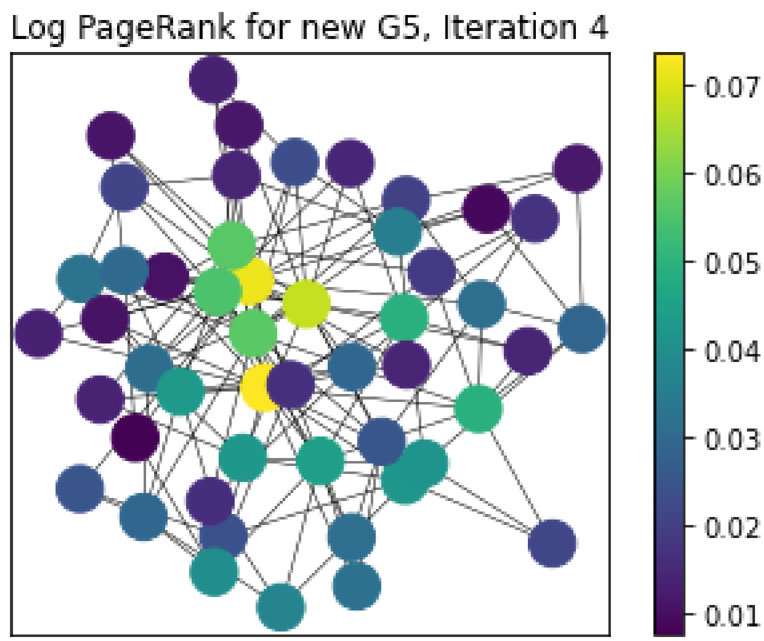
0.2.8 (H)

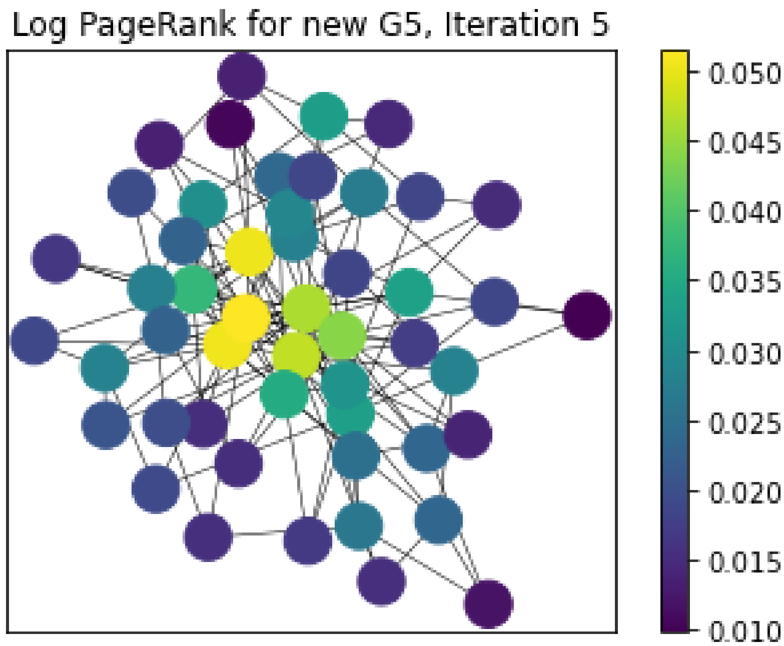
In which iterations is there the largest change in the scores? Create 4 plots - each in a different iteration to illustrate this change (the propagation). Let the color of the nodes represent the logarithmized PageRank value (to avoid errors add a pseudo-count of 0.0001) of the node after the respective iteration.

[33]:
```
for i in [1, 2, 3, 4]:
    plt.imshow(pranks1[i,:].reshape((10,5)))
    plt.colorbar()
    plt.cla()
    pos = nx.layout.spring_layout(G5)
    nodes = nx.draw_networkx_nodes(G5, pos, node_color=np.log(pranks1[i,:]),␣
 ↪cmap=plt.cm.viridis)
    edges = nx.draw_networkx_edges(G5,pos,width=0.5)
    plt.title(f"Log PageRank for new G5, Iteration {i+1}")
    plt.show()
```




Log PageRank for new G5, Iteration 3
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01


Log PageRank for new G5, Iteration 4
0.07
0.06
0.05
0.04
0.03
0.02
0.01



Result: The largest change is between Iteration 2 and 3, when instead of few nodes with high page ranks, the ranks get very homogenious. Then from Iteration 3 to 4 after the initial overshooting, a few more nodes return to a lower/higher page rank and then stabilize.

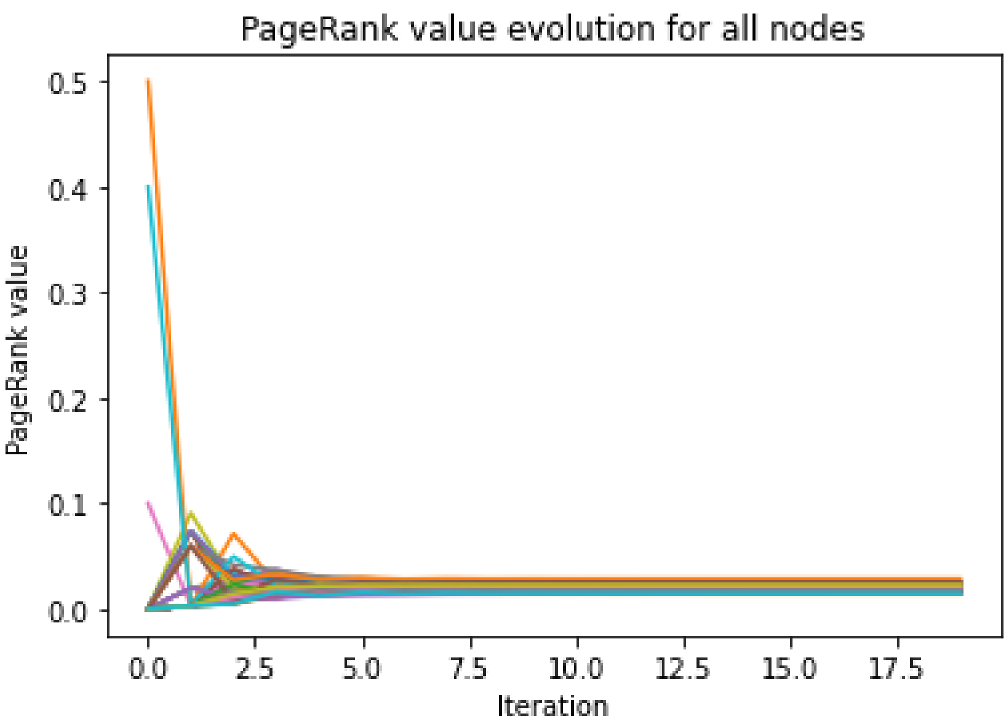
0.2.9 (I)

Repeat the above two steps for the Watts-Strogatz network.

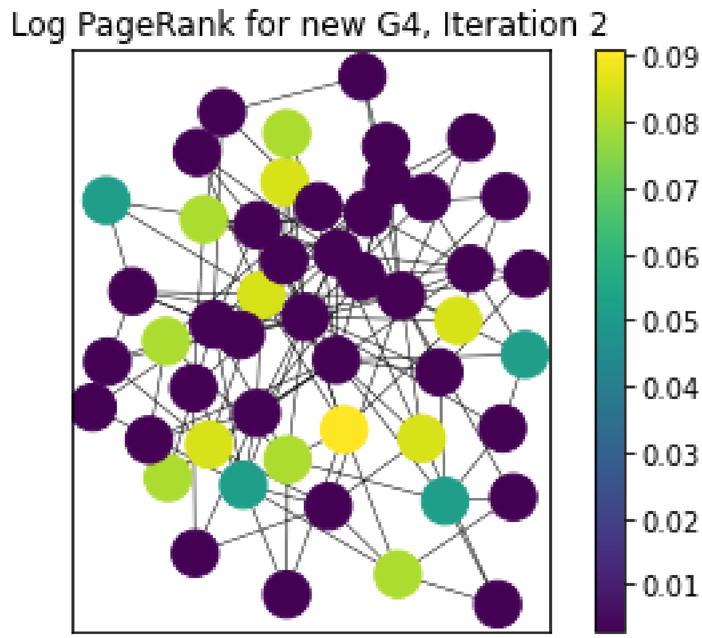
[34]:
```
T = transitionMatrixG(G4)
q = np.zeros_like(T[0])
q[:3]=[0.5,0.4,0.1]
np.random.shuffle(q)
its = 20 #number of iterations
pranks2 = np.zeros((its, 50)) #here we'll store the results of each iteration
pranks2[0]=q
q = np.ones_like(q)/50
a = 0.15
for i in range(1,its,1):
    pranks2[i] = a*q + (1-a) * np.dot(T,pranks2[i-1])

# plot
for node in range(50):
    #plt.plot(np.arange(50),pranks[node])
    x = np.arange(its)
    y = pranks2[:,node]
    plt.plot(x,y)
```

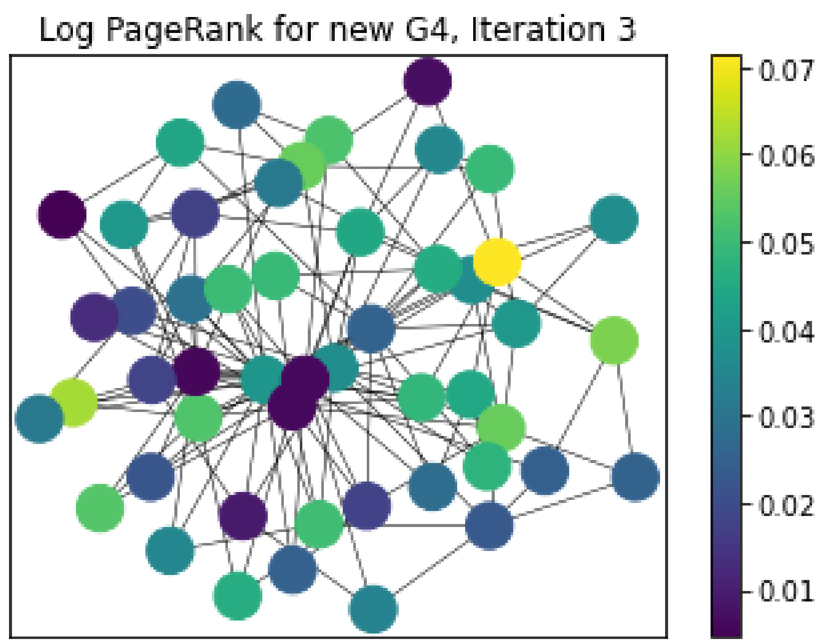
```
#
plt.xlabel("Iteration")
plt.ylabel("PageRank value")
plt.title("PageRank value evolution for all nodes")
plt.plot()
```

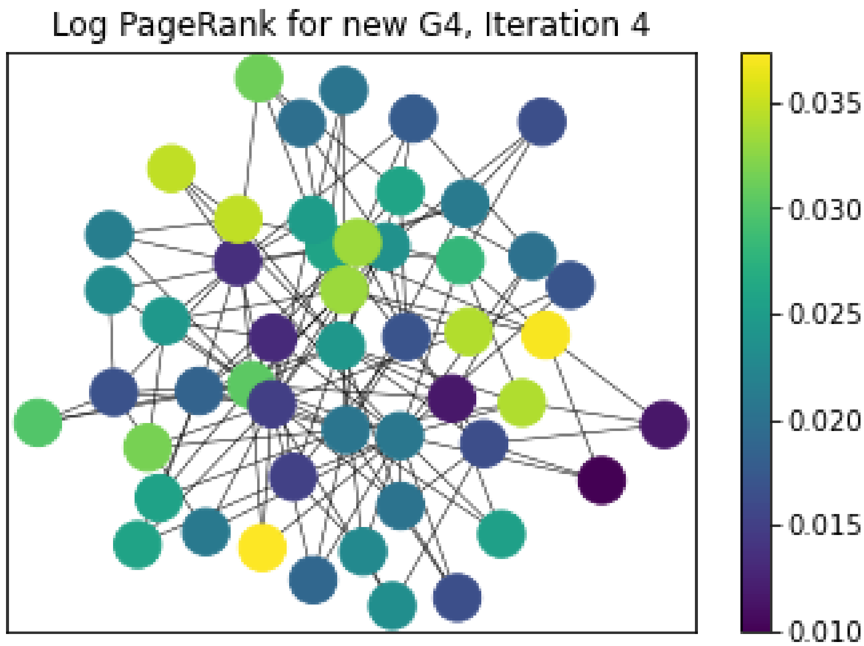
[34]: []

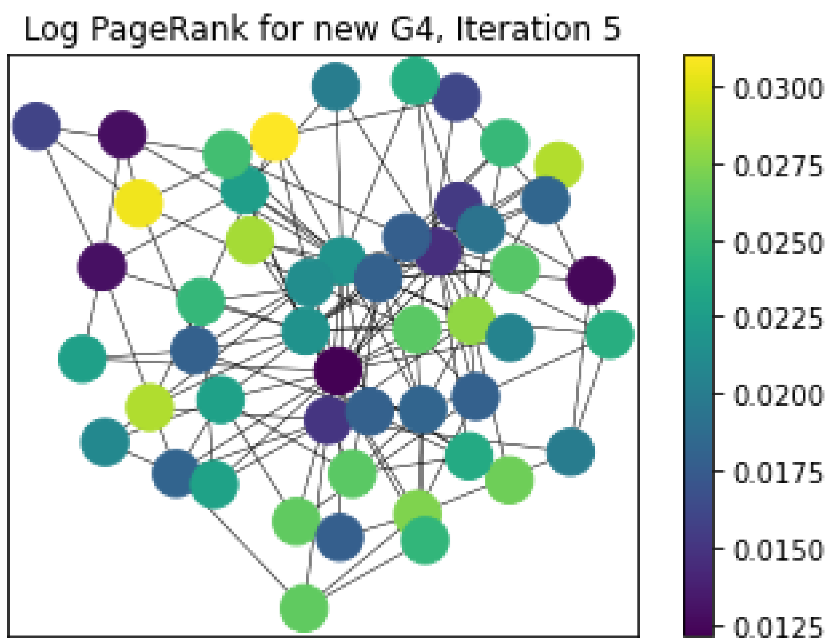


[35]:
```
for i in [1, 2, 3, 4]:
    plt.imshow(pranks2[i,:].reshape((10,5)))
    plt.colorbar()
    plt.cla()
    pos = nx.layout.spring_layout(G5)
    nodes = nx.draw_networkx_nodes(G5, pos, node_color=np.log(pranks2[i,:]),␣
 ↪cmap=plt.cm.viridis)
    edges = nx.draw_networkx_edges(G5,pos,width=0.5)
    plt.title(f"Log PageRank for new G4, Iteration {i+1}")
    plt.show()
```


Log PageRank for new G4, Iteration 2
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01


Log PageRank for new G4, Iteration 3
0.07
0.06
0.05
0.04
0.03
0.02
0.01





Result: The largest change is between Iteration 2 and 3, when instead of few nodes with high page ranks, the ranks get more homogenious. Here the high PageRank nodes have a lot less total connections and are more at the outside of the plotted graph.

0.2.10 (J)

Now calculate the propagation using the direct solution. Is it the same as the converged iterative solution?

[36]:
```
# for the Barabási-Albert network
T = transitionMatrixG(G5)
q6 = np.zeros_like(T[0])
q6[:3]=[0.5,0.4,0.1]
np.random.shuffle(q6)
q6 = np.ones_like(q6)/50
K6 = diffusionMatrixG(G5, alpha=0.15)
p6 = np.dot(K6,q6)
print("Difference between both RWR methods:", np.linalg.norm(pranks1[-1,:]-p6))
# we see that both methods give very similar result
```

```
Difference between both RWR methods: 2.076193150315235e-06
```

[37]:
```
# for the Watts-Strogatz network
T = transitionMatrixG(G4)
q7 = np.zeros_like(T[0])
q7[:3]=[0.5,0.4,0.1]
np.random.shuffle(q7)
q7 = np.ones_like(q7)/50
K7 = diffusionMatrixG(G4, alpha=0.15)
p7 = np.dot(K7,q7)
print("Difference between both RWR methods:", np.linalg.norm(pranks2[-1,:]-p7))
# we see that both methods give very similar result
```

```
Difference between both RWR methods: 1.1577328160264661e-05
```

Result: Both methods produce very similar results for both methods.