Task 4.2

Install the requirements

In [1]:
```
import numpy as np
!wget --show-progress 'https://d3js.org/d3.v3.js'
!pip install --upgrade jinja2==3.0.3
from IPython.core.display import HTML
HTML('<script src="d3.v3.js"></script>')
```

```
--2025-05-31 18:51:12--  https://d3js.org/d3.v3.js
Resolving d3js.org (d3js.org)... 172.67.73.126, 104.26.7.30, 104.26.
6.30, ...
Connecting to d3js.org (d3js.org)|172.67.73.126|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [application/javascript]
Saving to: ‘d3.v3.js.2’

d3.v3.js.2             [ <=>                  ] 330.02K  --.-KB/s
in 0.02s

2025-05-31 18:51:12 (15.0 MB/s) - ‘d3.v3.js.2’ saved [337945]

Requirement already satisfied: jinja2==3.0.3 in /opt/conda/lib/pytho
n3.11/site-packages (3.0.3)
Requirement already satisfied: MarkupSafe>=2.0 in /opt/conda/lib/pyt
hon3.11/site-packages (from jinja2==3.0.3) (2.1.5)
```

Out[1]:

In [2]:
```
!pip install lxml
!pip install python-igraph
!pip install cairocffi
```

```
Requirement already satisfied: lxml in /opt/conda/lib/python3.11/sit
e-packages (5.4.0)
Requirement already satisfied: python-igraph in /opt/conda/lib/pytho
n3.11/site-packages (0.11.8)
Requirement already satisfied: igraph==0.11.8 in /opt/conda/lib/pyth
on3.11/site-packages (from python-igraph) (0.11.8)
Requirement already satisfied: texttable>=1.6.2 in /opt/conda/lib/py
thon3.11/site-packages (from igraph==0.11.8->python-igraph) (1.7.0)
Requirement already satisfied: cairocffi in /opt/conda/lib/python3.1
1/site-packages (1.7.1)
Requirement already satisfied: cffi>=1.1.0 in /opt/conda/lib/python
3.11/site-packages (from cairocffi) (1.16.0)
Requirement already satisfied: pycparser in /opt/conda/lib/python3.1
1/site-packages (from cffi>=1.1.0->cairocffi) (2.22)
```

If necessary, restart kernel now

In [3]:
```
import os
#download graphframes package
!wget -q --show-progress http://repos.spark-packages.org/graphframe
#tell to load graphframes and dependencies to the spark cluster for
os.environ["PYSPARK_SUBMIT_ARGS"] = '--repositories "http://repos.s
```

In [4]:
```
import pyspark
from pyspark.sql import *
try:
    sc = pyspark.SparkContext('local[*]',environment = {})
except:
    sc = sc
#create sqlcontext on the spark, enables the use of the SQL queries
sqlContext = SQLContext(sc)
sqlContext = pyspark.sql.SparkSession(sc)
sc.setLogLevel("ERROR")
```

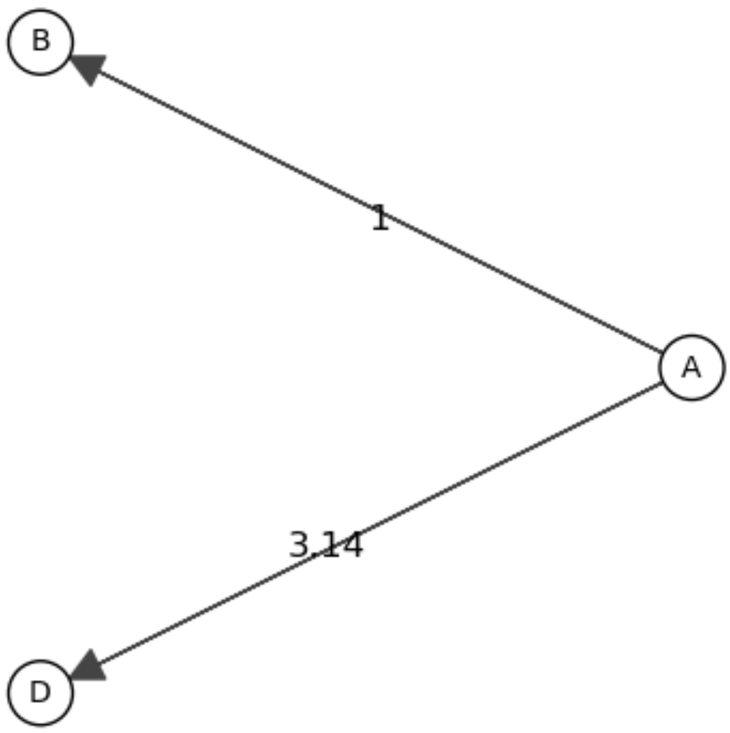
```
/usr/local/spark/python/pyspark/sql/context.py:113: FutureWarning: D
eprecated in 3.0.0. Use SparkSession.builder.getOrCreate() instead.
  warnings.warn(
```

Do not mind the warning about deprecation

Below is a tiny example on how to create a graph

In [5]:
```
import igraph as ig
import matplotlib.pyplot as plt
# Create a Vertex DataFrame with unique ID column "id"
#edges=([(from0,to0),(from1,to1),(to1,from0)]), etc... first is zer
edges=([(0, 1), (0, 2)])
g = ig.Graph(3,edges,directed=True)
g.vs["name"] = ["A", "B", "D"]
g.es["weights"]=[1,3.14]
display(g)

fig, ax = plt.subplots(figsize=(5,5))
ig.plot(g,target=ax,layout="circle",vertex_size=30,vertex_color="wh
plt.show()
```

```
<igraph.Graph at 0x7f90a92a7450>
```



Task 4.2.1 (3 p)

Modify the code below using the above example to form the graph that is shown in the Mining Massive Datasets book in figure 5.1 (figure below). Additionally, give random numbers to the edges (add random numbers in g.es["weights"]).

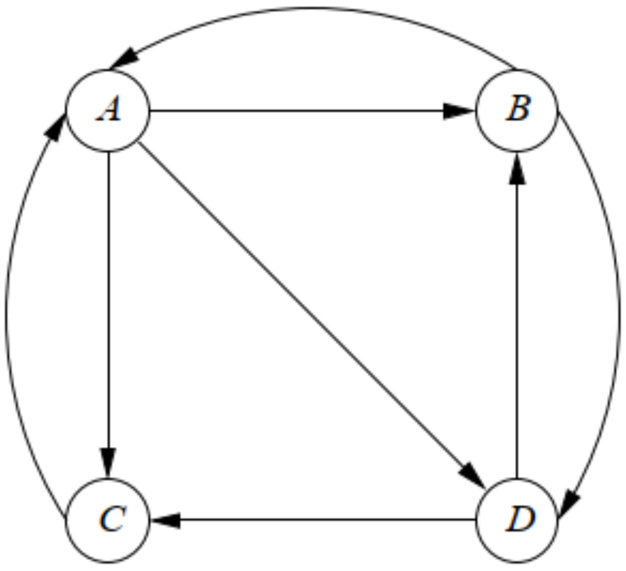
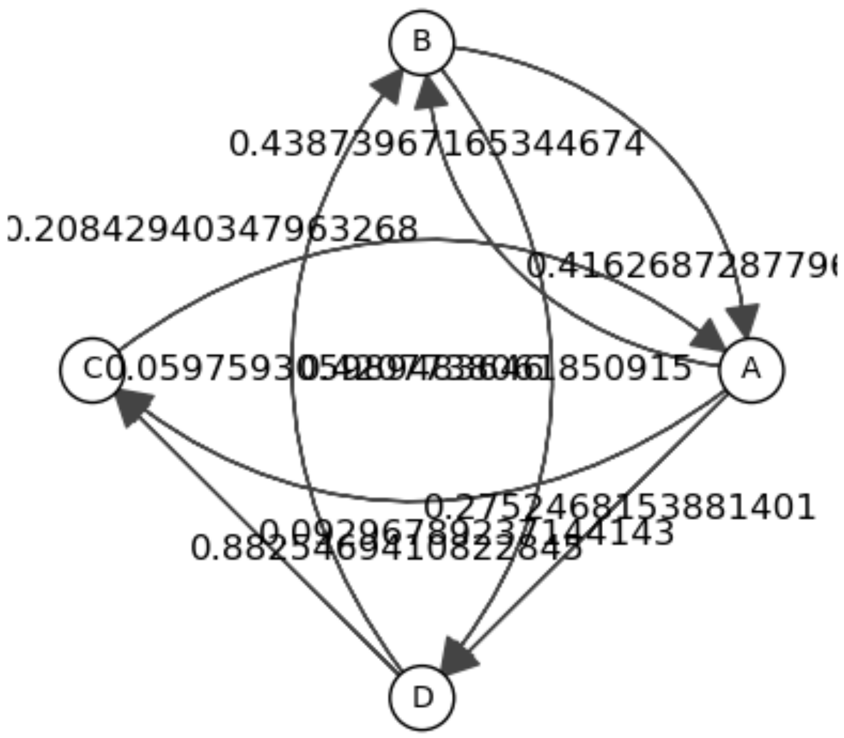


Figure 5.1: A hypothetical example of the Web

In [6]:
```
# Create a GraphFrame
```

```
import igraph as ig
import matplotlib.pyplot as plt
import numpy as np

edges = [(0, 1), (0, 2), (0, 3),(1, 0), (1, 3),(2, 0),(3, 1),(3, 2)

g = ig.Graph(4,edges,directed=True)
g.vs["name"] = ["A", "B", "C", "D"]
g.es["weights"]=np.random.rand(len(edges))  #Note: length of the "g
display(g)

fig, ax = plt.subplots(figsize=(5,5))
ig.plot(g,target=ax,layout="circle",vertex_size=30,vertex_color="wh
plt.show()
```

```
<igraph.Graph at 0x7f90a9311d50>
```



Below is the code for applying the pagerank algorithm to the graph. What the pagerank algorithm computes?

Calculate PageRank for the matrix in figure 5.3 of the course book change damping from 0.85, to 0.5 and 0.9. Explain what the 'damping' parameter does? How did the pageRank change in this case?

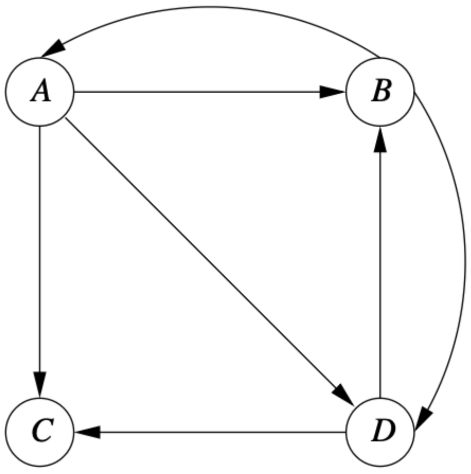
```
In [7]: # Run PageRank algorithm, and show results.
        results=ig.Graph.pagerank(g, implementation='power', directed=True,
        print(results)
```

```
[0.32456140350877194, 0.22514619883040934, 0.22514619883040934, 0.22
514619883040932]
```

Answers 4.2.1

What the pagerank algorithm computes?

PageRank computes the probability that a random surfer will be on each node, so pages with higher values are visited more often and therefore considered more important



Now rewriting graph's data and running pagerank with damping = 0.85, 0.50, 0.90

```
In [8]: import igraph as ig, numpy as np

        edges = [
            (0, 1), (0, 2), (0, 3),
            (1, 0), (1, 3),
            (3, 1), (3, 2)
        ]

        g53 = ig.Graph(4, edges, directed=True)
        g53.vs['name'] = ["A", "B", "C", "D"]

        for d in [0.85, 0.50, 0.90]:
            pr = g53.pagerank(implementation='power', directed=True, dampin
            print(f"damping = {d:>4}: {pr}")
```

```
damping = 0.85: [0.20618556701030927, 0.2646048109965636, 0.26460481
09965636, 0.2646048109965636]
damping =  0.5: [0.2222222222222222, 0.25925925925925924, 0.25925925
925925924, 0.25925925925925924]
damping =  0.9: [0.2040816326530612, 0.2653061224489796, 0.265306122
4489796, 0.2653061224489796]
```

Explain what the 'damping' parameter does?

The damping parameter manages how likely the random surfer is to go by an existing out-link instead of jumping to a random node; lower values mean more teleporting, and in this case the effect of the link structure is lower. It also helps to avoid dead ends and spider traps. Higher values go vice versa.

How did the PageRank change in this case?

With low damping (0.5), the PageRank was more equal because of higher teleporting, and with higher damping (0.9) we got bigger spread out since the real structure played a bigger role.

Task 4.2.2 (4 p)

In the code below, the Moodle zip-package data is used. Describe the data, what information do the files contain (you can have a look at the readme-Ego.txt file available on the web page https://snap.stanford.edu/data/egonets-Facebook.html)? We do not use the feature files in this exercise.

What is the degree of a node?

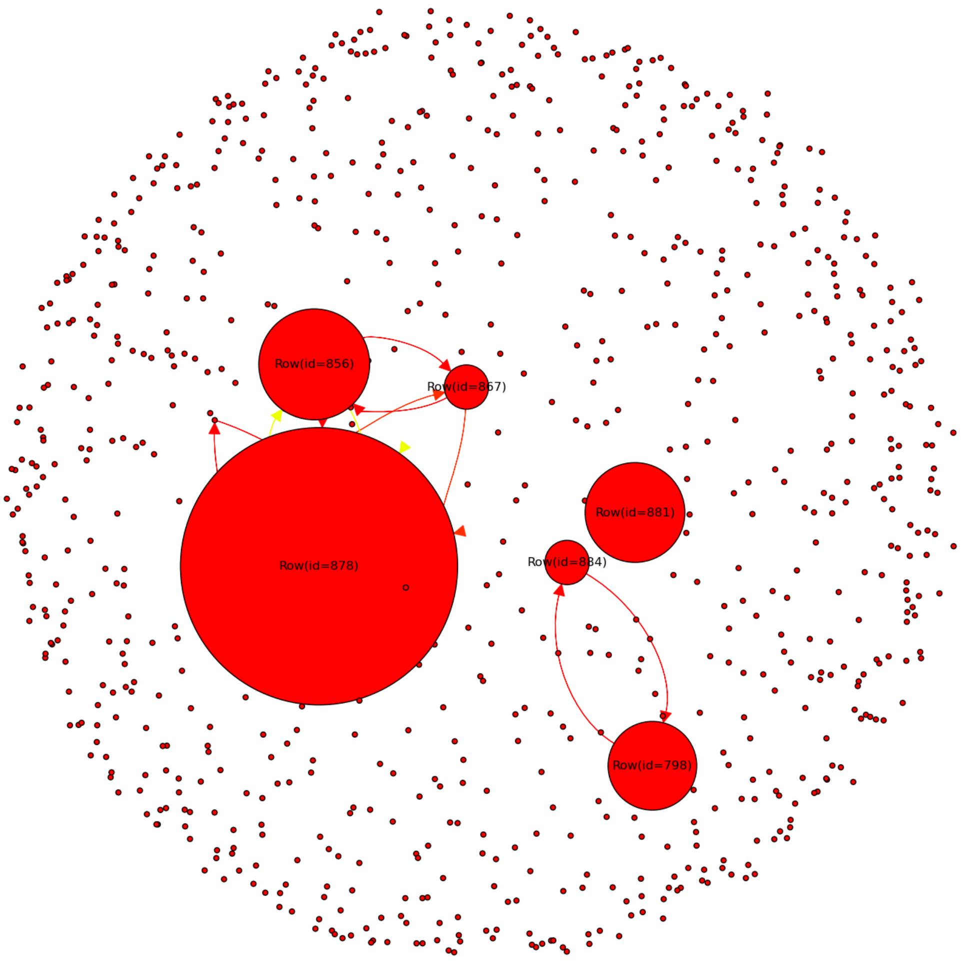
Is the PageRank algorithm any better than just the degree of a node to find the most important vertices of the network? Give some examples which user ids you think are important by visual inspection (see image), and in pagerank or degree. (For some analyses a vertex can be important if it connects network parts that would be isolated without the connecting vertex.)

In [9]:
```
import igraph as ig
from pyspark.sql.functions import col
import matplotlib.pyplot as plt
import pyspark.sql.functions as F
from pyspark.sql.window import Window as W
import numpy as np
#read graph edges(or arcs, lines) (multiple edges per vertex can ex
lines = sc.textFile("698.edges")
edges = lines.map(lambda l: l.split(" ")). \
            map(lambda p: Row( src=int(p[0]), dst=int(p[1])) )
edges = sqlContext.createDataFrame(edges)

#read graph vertices(or nodes, points) (these are unique)
lines = sc.textFile("698.feat")
vertices = lines.map(lambda l: l.split(" ")). \
            map(lambda p: Row(id=int(p[0]), name="userid_"+p[0]) )
vertices = sqlContext.createDataFrame(vertices)

idx_array = vertices.select("id").rdd.flatMap(lambda x: x).collect(
#print(idx_array)

#edges.show()
#vertices.show()
len_vertices=vertices.count()
print(len_vertices)
print(edges.count())
```

```
list_edges=[]
for i in edges.collect():
   list_edges.append(tuple(i))
#print(list_edges)
#print(len(list_edges))
#print(len_vertices)

#g = ig.Graph.DataFrame(edges, directed=True)
g = ig.Graph(len_vertices,edges=list_edges,directed=True) #,vertex_
g.vs["name"] = list(vertices.select("name").collect())
g.vs["id"] = list(vertices.select("id").collect())
degrees=g.degree()
degree_list=[]

for i in idx_array:
    degree_list.append(degrees[i])
#print(degree_list)
names = g.vs["name"]


pagerank=g.pagerank(vertices=idx_array,directed=True, damping=0.85)
print(pagerank)

a = vertices.persist()
#result = result.drop("id")

b = sqlContext.createDataFrame([(l,) for l in pagerank], ['pagerank
c = sqlContext.createDataFrame([(l,) for l in degree_list], ['degre
a = a.withColumn("idx2", F.monotonically_increasing_id())
b = b.withColumn("idx", F.monotonically_increasing_id())
c = c.withColumn("idx", F.monotonically_increasing_id())

windowSpec = W.orderBy("idx")
windowSpec2 = W.orderBy("idx2")
a = a.withColumn("idx2", F.row_number().over(windowSpec2))
b = b.withColumn("idx", F.row_number().over(windowSpec))
c = c.withColumn("idx", F.row_number().over(windowSpec))

d = a.join(b, a.idx2 == b.idx).drop("idx2")
d.show()

result=d.join(c,d.idx == c.idx).drop("idx")
result.orderBy(result.pagerank.desc())
result.show()


### NOTE: Image below does not scale to degree, but rather than sum

#g = ig.Graph.DataFrame(edges, directed=True)
g = ig.Graph(len_vertices,edges=list_edges,directed=True) #,vertex_
g.vs["name"] = list(vertices.select("name").collect())
g.vs["id"] = list(vertices.select("id").collect())
communities = g.community_edge_betweenness()
communities = communities.as_clustering()
num_communities = len(communities)
```

```
layout = g.layout_kamada_kawai()
g.vs["x"], g.vs["y"] = list(zip(*layout))
g.vs["size"] = 15
g.es["size"] = 15

cluster_graph = communities.cluster_graph(
    combine_vertices={
        "x": "mean",
        "y": "mean",
        "color": "first",
        "size": "sum",
    },
    combine_edges={
        "size": "sum",
    },
)
palette1 = ig.RainbowPalette(n=num_communities)
#select the circle radius based on the sum of edges connecting to i
g.es["color"] = [palette1.get(int(i)) for i in ig.rescale(cluster_g

for i in range(0,len(cluster_graph.vs["size"])):
    if cluster_graph.vs["size"][i]<20:
          g.vs[i]["id"]=None
          # set a minimum size on vertex_size, otherwise vertices ar
          cluster_graph.vs[i]["size"]=7
#Igraph node significance vs others
fig2, ax2 = plt.subplots(figsize=(20,20))
ig.plot(
    cluster_graph,
    target=ax2,
    palette=palette1,

    #vertex_size=[max(7, size) for size in cluster_graph.vs["size"]
    vertex_size=cluster_graph.vs["size"],
    vertex_label=g.vs["id"],
    edge_color=g.es["color"],
    edge_width=0.8,
)

plt.show()
```

```
66
540
[0.004487921923939519, 0.0057499210038887, 0.0037048036631419953, 0.
004623449286823166, 0.0016720765027717645, 0.0008053691275180309, 0.
006277732125535889, 0.0048124855945266025, 0.005781745862581488, 0.0
0467821694040376, 0.00536745386034648, 0.0057876505832815195, 0.0035
92232241906875, 0.004678216940402426, 0.004739284660697229, 0.006779
4672138418235, 0.007340304800119276, 0.00467821694040098, 0.00331750
83812565917, 0.007474031543113972, 0.006779467213843214, 0.006689835
111174023, 0.006483197761824865, 0.006833417484908045, 0.00583571501
8580414, 0.0038236574999233218, 0.004971971129013624, 0.006911697771
533001, 0.005835715018577381, 0.008056330923175151, 0.00080536912751
80309, 0.005974057177077773, 0.005367453860345792, 0.006363281256286
476, 0.004820719731696256, 0.009056580002896839, 0.00331851854041605
6, 0.004881685553549504, 0.005727614637276211, 0.003591008094333207
6, 0.006645306340729748, 0.00396079604008498, 0.0008053691275180309,
0.0030776807877570837, 0.0012500418197428575, 0.008157245884758909,
0.0053212912807295565, 0.007612277764831954, 0.004009980475402268,
0.004971971129009874, 0.004389430447929129, 0.0033889659030213806,
0.006770450614607926, 0.006478410869962027, 0.0033217986970258706,
0.007348331407057229, 0.0033055067210998206, 0.0008053691275180309,
0.0008053691275180309, 0.0019860331971365748, 0.0028466192844759913,
0.005746235970727437, 0.003733888246996407, 0.006829654798084521, 0.
005652162607403938, 0.013848054378457477]
+---+----------+--------------------+---+
| id|      name|            pagerank|idx|
+---+----------+--------------------+---+
|810|userid_810|0.004487921923939519|  1|
|857|userid_857|  0.0057499210038887|  2|
|811|userid_811|0.003704803663141...|  3|
|858|userid_858|0.004623449286823166|  4|
|859|userid_859|0.001672076502771...|  5|
|860|userid_860|8.053691275180309E-4|  6|
|769|userid_769|0.006277732125535889|  7|
|861|userid_861|0.004812485594526...|  8|
|840|userid_840|0.005781745862581488|  9|
|862|userid_862| 0.00467821694040376| 10|
|863|userid_863| 0.00536745386034648| 11|
|729|userid_729|0.005787650583281...| 12|
|864|userid_864|0.003592232241906875| 13|
|865|userid_865|0.004678216940402426| 14|
|866|userid_866|0.004739284660697229| 15|
|867|userid_867|0.006779467213841...| 16|
|697|userid_697|0.007340304800119276| 17|
|868|userid_868| 0.00467821694040098| 18|
|869|userid_869|0.003317508381256...| 19|
|708|userid_708|0.007474031543113972| 20|
+---+----------+--------------------+---+
only showing top 20 rows

+---+----------+--------------------+------+
| id|      name|            pagerank|degree|
+---+----------+--------------------+------+
|810|userid_810|0.004487921923939519|    20|
|857|userid_857|  0.0057499210038887|     8|
|811|userid_811|0.003704803663141...|    14|
|858|userid_858|0.004623449286823166|     8|
|859|userid_859|0.001672076502771...|     2|
|860|userid_860|8.053691275180309E-4|     0|
|769|userid_769|0.006277732125535889|    24|
|861|userid_861|0.004812485594526...|    18|
|840|userid_840|0.005781745862581488|    26|
|862|userid_862| 0.00467821694040376|     6|
|863|userid_863| 0.00536745386034648|    20|
|729|userid_729|0.005787650583281...|    22|
|864|userid_864|0.003592232241906875|     8|
|865|userid_865|0.004678216940402426|     6|
|866|userid_866|0.004739284660697229|     8|
|867|userid_867|0.006779467213841...|    14|
|697|userid_697|0.007340304800119276|    34|
|868|userid_868| 0.00467821694040098|     6|
|869|userid_869|0.003317508381256...|    12|
|708|userid_708|0.007474031543113972|    34|
+---+----------+--------------------+------+
only showing top 20 rows
```



Answers 4.2.2

Dataset description

The data shows ego-networks where each user is connected to everyone in their .edges file, though the ego user is not shown there. So files describe who is connected to who and represent structure a friend network.

What is the degree of a node?

Degree of a node is the number of edges connected to it.

Is PageRank any better than degree? Yes, because degree counts only the number of links the node has, while PageRank also considers the connections importance, so it will indentify users who matter even if they have small number of direct connections.

Examples of important users

By visual size: 878, 856, 881, 884, 798, 867

By high PageRank values: 708, 697, 769, 867

Task 4.2.3 (4 p)

A graph can be split into clusters by the connectivity, contents or with both connectivity and contents of the graph. In social network graphs individuals may belong to many groups or communities so the clustering is not strictly defined for this kind of data. A graph can be partitioned in many ways and the graph partition problem is NP hard to find the best partition. A big computation cluster seems to be the only way to find good solution for large graphs quickly. As seen in the course book chapter Mining Social-Network Graphs, there are many different graph clustering/partitioning/grouping algorithms available. In the following code, Label Propagation Algorithm is used to cluster communities. The code uses the data files from the last example.

Run the code. Examine the resulting graph. How many clusters the LPA algorihtm generated?

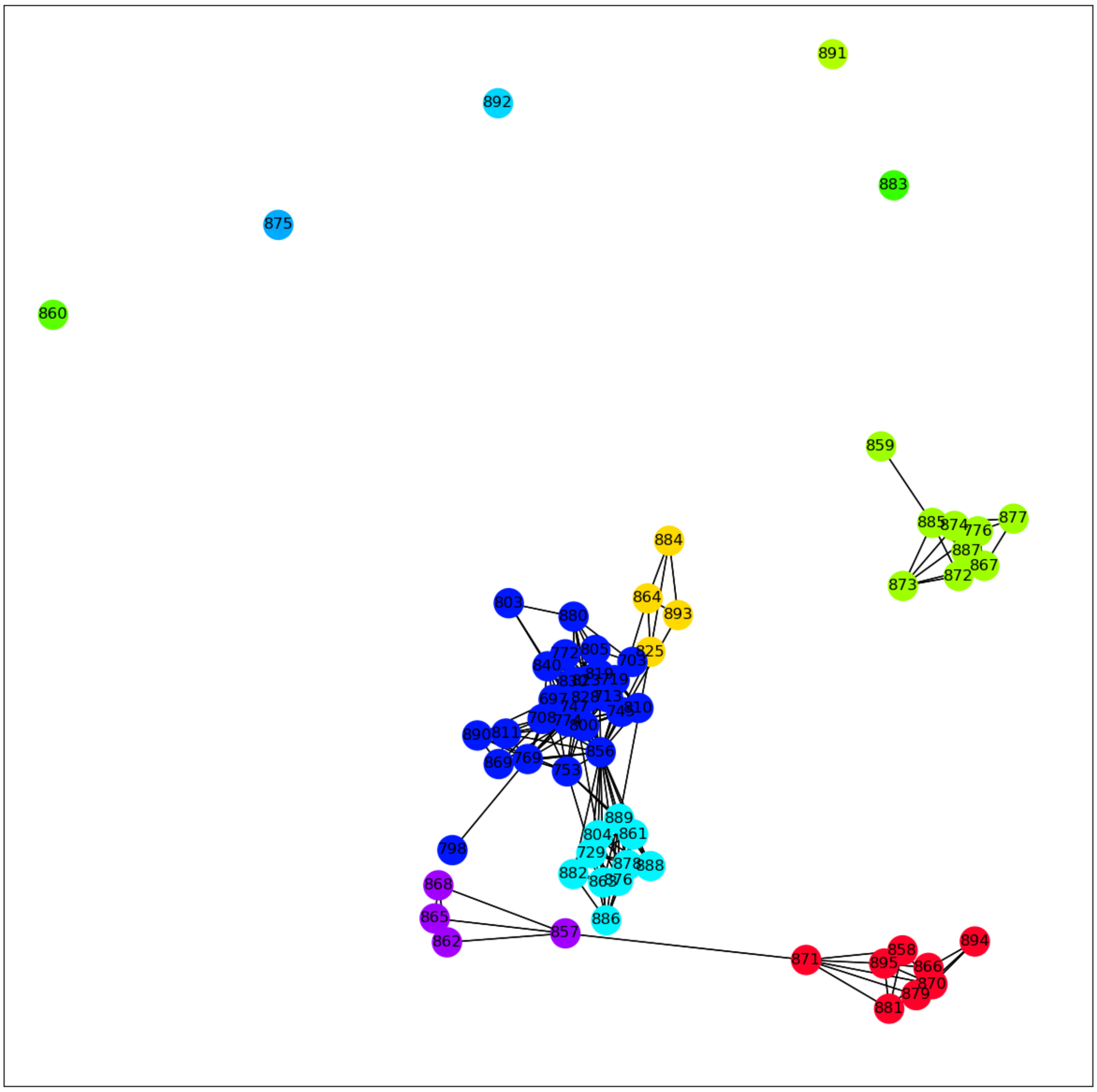
In [13]:
```
#res.show()
```

In [16]:
```
#%matplotlib notebook

import networkx as nx
import matplotlib.pyplot as plt
import numpy as nb
import igraph as ig

#readn graph edges(or arcs, lines) (multiple edges per vertex can e
lines = sc.textFile("698.edges")
edges = lines.map(lambda l: l.split(" ")). \
            map(lambda p: Row( src=int(p[0]), dst=int(p[1])) )
edges = sqlContext.createDataFrame(edges)

#read graph vertices(or nodes, points) (these are unique)
lines = sc.textFile("698.feat")
vertices = lines.map(lambda l: l.split(" ")). \
            map(lambda p: Row(id=int(p[0]), name="userid_"+p[0] ,fe
vertices = sqlContext.createDataFrame(vertices)

from graphframes import *
g = GraphFrame(vertices, edges)
#calculate simple clustering with the label propagation clustering 
lpa = g.labelPropagation(maxIter=5)
```

```
nodes = lpa.select("id","label")
lpa.show()
nodes.show()
fig, ax = plt.subplots(figsize=(15,15))
from igraph import *
ig = Graph.TupleList(g.edges.collect(), directed=True)
plot(ig)

#plotting, generate unique colors for each group ------------------
G = nx.DiGraph()
for x in g.edges.collect():
    G.add_edges_from([(x[0],x[1])], weight=1)
for x in lpa.select("id","label").rdd.map(lambda r: ( int(r[0]),int
    G.add_node(x[0],label=x[1])
grouplabels = [list(x[1].values())[0] for x in G.nodes(True)]
node_texts = {node:node for node in G.nodes()};
cmap = plt.get_cmap('gist_rainbow')
uniqlabels = nb.unique(grouplabels)
randvals = nb.random.random_sample((len(uniqlabels),1))
colorlut = dict(zip(uniqlabels,randvals))
gcolors = []
for x in grouplabels:
    gcolors.append(cmap(float(colorlut[x])))
positions=nx.spring_layout(G,k=0.1,scale=1.5,iterations=20)
nx.draw_networkx(G,positions, labels=node_texts, node_color = gcolo
#plotting end -----------------------------------------------------
```

```
+---+----------+--------------------+-----+
| id|      name|            features|label|
+---+----------+--------------------+-----+
|857|userid_857|[0, 0, 0, 0, 0, 0...|  857|
|858|userid_858|[0, 0, 0, 0, 0, 0...|  881|
|830|userid_830|[0, 0, 0, 0, 0, 0...|  828|
|876|userid_876|[0, 1, 0, 0, 0, 0...|  882|
|828|userid_828|[0, 0, 1, 1, 0, 0...|  828|
|703|userid_703|[1, 0, 0, 1, 0, 0...|  828|
|713|userid_713|[0, 0, 0, 0, 0, 0...|  828|
|861|userid_861|[1, 0, 0, 0, 0, 0...|  882|
|866|userid_866|[0, 0, 0, 0, 0, 0...|  881|
|769|userid_769|[0, 0, 0, 0, 0, 0...|  828|
|865|userid_865|[0, 0, 0, 0, 0, 0...|  857|
|859|userid_859|[0, 0, 0, 0, 1, 1...|  873|
|874|userid_874|[0, 0, 0, 0, 0, 0...|  873|
|870|userid_870|[0, 0, 0, 0, 1, 0...|  881|
|871|userid_871|[0, 0, 0, 0, 0, 0...|  881|
|697|userid_697|[0, 0, 0, 0, 0, 0...|  828|
|867|userid_867|[0, 0, 0, 0, 0, 0...|  873|
|800|userid_800|[0, 0, 0, 0, 0, 0...|  828|
|819|userid_819|[0, 0, 0, 0, 0, 0...|  828|
|863|userid_863|[0, 0, 0, 0, 0, 0...|  882|
+---+----------+--------------------+-----+
only showing top 20 rows

+---+-----+
| id|label|
+---+-----+
|857|  857|
|858|  881|
|830|  828|
|876|  882|
|828|  828|
|703|  828|
|713|  828|
|861|  882|
|866|  881|
|769|  828|
|865|  857|
|859|  873|
|874|  873|
|870|  881|
|871|  881|
|697|  828|
|867|  873|
|800|  828|
|819|  828|
|863|  882|
+---+-----+
only showing top 20 rows
```

```
/tmp/ipykernel_1262/3007408608.py:46: DeprecationWarning: Conversion
of an array with ndim > 0 to a scalar is deprecated, and will error
in future. Ensure you extract a single element from your array befor
e performing this operation. (Deprecated NumPy 1.25.)
  gcolors.append(cmap(float(colorlut[x])))
```



```
In [15]: lpa.select("label").distinct().count()
```

```
Out[15]: 11
```

Answer 4.2.3

Now we are running lpa.select("label").distinct().count() and as we can see, that the Label Propagation Algorithm generated 11 distinct clusters.

On the plot, we clearly see one main group with around 40 nodes in it, a few smaller clusters with up to 10 nodes each and 5 clusters that consist only of one node. The largest cluster likely represents the core of the network, smaller ones represent some specific groups of users, and the single node clusters probably are outliers with weak or no connection.