CS4423 - Networks

Prof. Götz Pfeiffer
School of Mathematics, Statistics and Applied Mathematics
NUI Galway

2. Centrality Measures

Lecture 7: Degree and Eigenvector Centrality

Key actors in a social network can be identified through centrality measures. The question of what it means to be central has a number of different answers. Accordingly, in the context of social network analysis, a variety of different centrality measures have been developed.

Here we introduce, in addition to the **degree centrality** we have already seen, three more further measures:

- **eigenvector centrality**, defined in terms of properties of the network's adjacency matrix,
- **closeness centrality**, defined in terms of a nodes distance to other nodes on the network,
- **betweenness centrality**, defined in terms of shortest paths.

Start by importing the necessary python libraries into this jupyter notebook. (Actually, networkx works with a number of useful python libraries, some of which are loaded automatically, while others have to be `import` ed explicitly, depending on the circumstances. In the following, we will also make explicit use of Pandas and Numpy.)

In [1]:
```
import networkx as nx
import matplotlib.pyplot as plt
```

Degree Centrality

The **degree** of a node is its number of neighbors in the graph. This number can serve as a simple measure of centrality.

Consider the following network of *florentine families*, linked by *marital ties*.

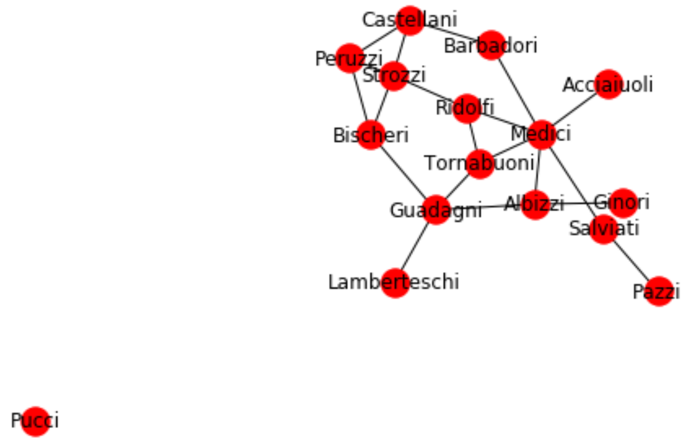
In [2]:
```
G = nx.generators.florentine_families_graph()
list(G.nodes())
```

Out[2]:
```
['Acciaiuoli',
 'Medici',
 'Castellani',
 'Peruzzi',
 'Strozzi',
 'Barbadori',
 'Ridolfi',
 'Tornabuoni',
 'Albizzi',
 'Salviati',
 'Pazzi',
 'Bischeri',
 'Guadagni',
 'Ginori',
 'Lamberteschi']
```

Unfortunately, this version of the graph misses the isolated node `'Pucci'` of the original graph. Let's just add it and draw the resulting graph.

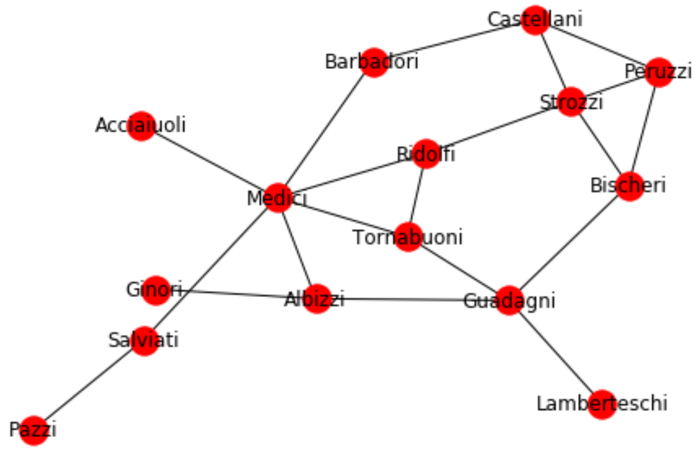
In [3]:
```
cc = list(G.nodes())
G.add_node('Pucci')
```

In [4]:
```
nx.draw(G, with_labels=True)
```



The large connected component can be drawn as a subgraph.

In [5]:
```
nx.draw(G.subgraph(cc), with_labels=True)
```



Known indicators of the importance of these families are their *wealth*, and the number of seats on the city council (*priorates*). These measures can be compared with the node degree in the graph 'G'.

In [6]:
```
wealth = {
  'Acciaiuoli': 10, 'Albizzi': 36, 'Barbadori': 55, 'Bischeri': 44,
  'Castellani': 20, 'Ginori': 32, 'Guadagni': 8, 'Lamberteschi': 42,
  'Medici': 103, 'Pazzi': 48, 'Peruzzi': 49, 'Pucci': 3,
  'Ridolfi': 27, 'Salviati': 10, 'Strozzi': 146, 'Tornabuoni': 48,
}

priorates = {
  'Acciaiuoli': 53, 'Albizzi': 65, 'Barbadori': 'n/a', 'Bischeri': 12,
  'Castellani': 22, 'Ginori': 'n/a', 'Guadagni': 21, 'Lamberteschi': 0,
  'Medici': 53, 'Pazzi': 'n/a', 'Peruzzi': 42, 'Pucci': 0,
  'Ridolfi': 38, 'Salviati': 35, 'Strozzi': 74, 'Tornabuoni': 'n/a',
}
```

In [7]:
```
nx.set_node_attributes(G, wealth, 'wealth')
nx.set_node_attributes(G, priorates, 'priorates')
nx.set_node_attributes(G, dict(G.degree()), 'degree')
```

In [8]:
```
dict(G.degree())
```

Out[8]:
```
{'Acciaiuoli': 1,
 'Medici': 6,
 'Castellani': 3,
 'Peruzzi': 3,
 'Strozzi': 4,
 'Barbadori': 2,
 'Ridolfi': 3,
 'Tornabuoni': 3,
 'Albizzi': 3,
 'Salviati': 2,
 'Pazzi': 1,
 'Bischeri': 3,
 'Guadagni': 4,
 'Ginori': 1,
 'Lamberteschi': 1,
 'Pucci': 0}
```

In [9]:
```
G.nodes['Pazzi']
```

Out[9]:
```
{'wealth': 48, 'priorates': 'n/a', 'degree': 1}
```

In [10]:
```
G.nodes(data=True)
```

Out[10]:
```
NodeDataView({'Acciaiuoli': {'wealth': 10, 'priorates': 53, 'degree': 1}, 'Me
dici': {'wealth': 103, 'priorates': 53, 'degree': 6}, 'Castellani': {'wealth'
: 20, 'priorates': 22, 'degree': 3}, 'Peruzzi': {'wealth': 49, 'priorates': 4
2, 'degree': 3}, 'Strozzi': {'wealth': 146, 'priorates': 74, 'degree': 4}, 'B
arbadori': {'wealth': 55, 'priorates': 'n/a', 'degree': 2}, 'Ridolfi': {'weal
th': 27, 'priorates': 38, 'degree': 3}, 'Tornabuoni': {'wealth': 48, 'priorat
es': 'n/a', 'degree': 3}, 'Albizzi': {'wealth': 36, 'priorates': 65, 'degree'
: 3}, 'Salviati': {'wealth': 10, 'priorates': 35, 'degree': 2}, 'Pazzi': {'we
alth': 48, 'priorates': 'n/a', 'degree': 1}, 'Bischeri': {'wealth': 44, 'prio
rates': 12, 'degree': 3}, 'Guadagni': {'wealth': 8, 'priorates': 21, 'degree'
: 4}, 'Ginori': {'wealth': 32, 'priorates': 'n/a', 'degree': 1}, 'Lamberteschi': {'wealth': 42, 'priorates': 0, 'degree': 1}, 'Pucci': {'wealth': 3, 'prio
rates': 0, 'degree': 0}})
```

In [11]:
```
import pandas as pd
```

In [12]:
```
pd.DataFrame.from_dict(dict(G.nodes(data=True)), orient='index').sort_values
('degree', ascending=False)
```

Out[12]:

	wealth	priorates	degree
Medici	103	53	6
Guadagni	8	21	4
Strozzi	146	74	4
Albizzi	36	65	3
Bischeri	44	12	3
Castellani	20	22	3
Peruzzi	49	42	3
Ridolfi	27	38	3
Tornabuoni	48	n/a	3
Barbadori	55	n/a	2
Salviati	10	35	2
Acciaiuoli	10	53	1
Ginori	32	n/a	1
Lamberteschi	42	0	1
Pazzi	48	n/a	1
Pucci	3	0	0

Definition (Degree Centrality). In a (simple) graph $G = (X, E)$, with $X = \{1, \dots, n\}$ and adjacency matrix $A = (a_{ij})$, the **degree centrality** c_i^D of node $i \in X$ is defined as

$$c_i^D = k_i = \sum_j a_{ij},$$

where k_i is the degree of node i.
The **normalized degree centrality** C_i^D of node $i \in X$ is defined as

$$C_i^D = \frac{k_i}{n-1} = \frac{c_i^D}{n-1},$$

the degree centrality of node i divided by its potential number of neighbors in the graph.

In a directed graph one distinguishes between the in-degree and the out-degree of a node and defines the in-degree centrality and the out-degree centrality accordingly.

In [13]:
```
nx.set_node_attributes(G, nx.degree_centrality(G), '$C_i^D$')
```

In [14]:
```
pd.DataFrame.from_dict(dict(G.nodes(data=True)), orient='index').sort_values
('degree', ascending=False)
```

Out[14]:

	wealth	priorates	degree	C_i^D
Medici	103	53	6	0.400000
Guadagni	8	21	4	0.266667
Strozzi	146	74	4	0.266667
Albizzi	36	65	3	0.200000
Bischeri	44	12	3	0.200000
Castellani	20	22	3	0.200000
Peruzzi	49	42	3	0.200000
Ridolfi	27	38	3	0.200000
Tornabuoni	48	n/a	3	0.200000
Barbadori	55	n/a	2	0.133333
Salviati	10	35	2	0.133333
Acciaiuoli	10	53	1	0.066667
Ginori	32	n/a	1	0.066667
Lamberteschi	42	0	1	0.066667
Pazzi	48	n/a	1	0.066667
Pucci	3	0	0	0.000000

Eigenvectors and Centrality

Recall that a (n-dimensional) vector v is called an **eigenvector** of a square ($n \times n$-)matrix A, if

$$Av = \lambda v$$

for some scalar (number) λ (which is then called an **eigenvalue** of the matrix A)

The basic idea of eigenvector centrality is that a node's ranking in a network should relate to the rankings of the nodes it is connected to. More specifically, up to some scalar λ, the centrality c_i^E of node i should be equal to the sum if the centralities c_j^E of its neighbor nodes j. In terms of the adjacency matrix $A = (a_{ij})$, this relationship is expressed as

$$\lambda c_i^E = \sum_j a_{ij} c_j^E,$$

which in turn, in matrix language is

$$\lambda c^E = A c_E,$$

for the vector $c^E = (c_i^E)$, which then is an eigenvector of A.

How to find c^E? Or λ? Linear Algebra:

1. Find the *characteristic polynomial* $p_A(x)$ of A (as *determinant* of the matrix $xI - A$, where I is the $n \times n$-identity matrix);
2. Find the *roots* λ of $p_A(x)$ (i.e. scalars λ such that $p_A(\lambda) = 0$;
3. Find a *nontrivial solution* of the linear system $(\lambda I - A)c = 0$ (where 0 stands for an all-0 column vector, and $c = (c_1, \ldots, c_n)$ is a column of *unknowns*).

In [15]:
```
A = nx.adjacency_matrix(G)
```

In [16]:
```
import numpy as np
B = np.array([[2,2],[3,1]])
poly = np.poly(B)
l, v = np.linalg.eig(B)
vv = v.transpose()
print(poly)
print(l); print (vv); print(vv[0])
print(B*vv[0], l[0]*vv[0])
```

```
[ 1. -3. -4.]
[ 4. -1.]
[[ 0.70710678  0.70710678]
 [-0.5547002   0.83205029]]
[0.70710678 0.70710678]
[[1.41421356 1.41421356]
 [2.12132034 0.70710678]] [2.82842712 2.82842712]
```

In [17]:
```
print(np.matmul(B, vv[0]))
```

```
[2.82842712 2.82842712]
```

In [18]:
```
print(A.todense())
```

```
[[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0]
 [0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 1 1 0 0 1 0 0 0 0 1 0 0 0 0]
 [0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0]
 [0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 1 1 0 0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0]
 [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

In [19]:
```
np.poly(A.todense())
```

Out[19]:
```
array([ 1.00000000e+00,  7.43849426e-15, -2.00000000e+01, -6.00000000e+00,
        1.39000000e+02,  6.80000000e+01, -4.17000000e+02, -2.42000000e+02,
        5.65000000e+02,  3.44000000e+02, -3.44000000e+02, -2.08000000e+02,
        8.20000000e+01,  4.60000000e+01, -5.00000000e+00, -2.00000000e+00,
        0.00000000e+00])
```

Numerical Linerar Algebra: forget algebraic precision, use the **Power method**:

1. start with $u = (1, 1, \ldots, 1)$, say;
2. keep replacing $u \leftarrow Au$ until $u/\|u\|$ becomes stable ...

If A has a *dominant* eigenvalue λ_0 then u will *converge* to an eigenvector for the eigenvalue λ_0.

In [20]:
```
u = [1 for x in A]
print(u)
print(u/np.linalg.norm(u))
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25
 0.25 0.25]
```

In [21]:
```
v = A*u
print(v)
print(v/np.linalg.norm(v))
```

```
[1 6 3 3 4 2 3 3 3 2 1 3 4 1 1 0]
[0.08638684 0.51832106 0.25916053 0.25916053 0.34554737 0.17277369
 0.25916053 0.25916053 0.25916053 0.17277369 0.08638684 0.25916053
 0.34554737 0.08638684 0.08638684 0.        ]
```

In [22]:
```
for i in range(40):
    u = A * u
    u = u/np.linalg.norm(u)
```

In [23]:
```
v = A *u
l = v[2]/u[2]
v = v/np.linalg.norm(v)
print(u/np.linalg.norm(u))
print(v/np.linalg.norm(v))
print("||v - u|| = ", np.linalg.norm(v - u))
print("l = ", l)
```

```
[0.13217021 0.43026554 0.25902292 0.27572882 0.355973   0.21172226
 0.34156271 0.32586538 0.24398716 0.14593566 0.04480665 0.282814
 0.2890864  0.07491127 0.08880275 0.        ]
[0.13214123 0.43034359 0.25902863 0.27573136 0.35598634 0.21169119
 0.34154438 0.32582335 0.24393056 0.14590205 0.04481911 0.28278835
 0.28913983 0.07493225 0.08878292 0.        ]
||v - u|| =  0.00013802994032227356
l =  3.256175481394629
```

In [24]:
```
l, w = np.linalg.eig(A.todense())
print (l)
print (w.transpose()[0])
```

```
[ 3.25610375  2.42381396 -2.69583872  1.70899078 -2.06786891 -1.87072241
  1.05403772  0.93839893  0.60199089  0.25781512 -0.20243481 -1.19329397
 -0.57626063 -0.76538496 -0.86934672  0.        ]
[[0.13215429 0.43030809 0.25902617 0.27573037 0.35598045 0.21170525
  0.34155264 0.3258423  0.24395611 0.1459172  0.04481344 0.28280009
  0.2891156  0.07492271 0.08879189 0.        ]]
```

In [25]:
```
eigen_cen = nx.eigenvector_centrality(G.subgraph(cc))
eigen_cen
```

Out[25]:
```
{'Acciaiuoli': 0.1321573195285342,
 'Medici': 0.4303154258349923,
 'Castellani': 0.2590200378423514,
 'Peruzzi': 0.2757224374104833,
 'Strozzi': 0.3559730326460451,
 'Barbadori': 0.2117057470647985,
 'Ridolfi': 0.3415544259074365,
 'Tornabuoni': 0.325846704169574,
 'Albizzi': 0.2439605296754477,
 'Salviati': 0.14592084164171834,
 'Pazzi': 0.044814939703863084,
 'Bischeri': 0.2827943958713356,
 'Guadagni': 0.2891171573226501,
 'Ginori': 0.0749245316027793,
 'Lamberteschi': 0.08879253113499548}
```

Time's up. Save the graph for future use.

In [26]:
```
nx.write_yaml(G, "florentine.yml")
```

The theoretical foundation for this approach is provided by the following Linear Algebra [theorem (https://en.wikipedia.org/wiki/Perron%E2%80%93Frobenius_theorem)](https://en.wikipedia.org/wiki/Perron%E2%80%93Frobenius_theorem) from 1907/1912.

Theorem. (Perron-Frobenius for irreducible matrices.) Suppose that A is a square, nonnegative, irreducible matrix. Then: * A has a real eigenvalue $\lambda > 0$ with $\lambda \geq |\lambda'|$ for all eigenvalues λ' of A; * λ is a simple root of the characteristic polynomial of A; * there is a λ-eigenvector v with $v > 0$.

Here, a matrix A is called **reducible** if, for some simultaneous permutation of its rows and columns, it has the block form

$$A = \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{21} \end{pmatrix}.$$

And A is **irreducible** if it is not reducible.

The incidence matrix of a simple graph G is irreducible if and only if G is connected.

Definition (Eigenvector centrality). In a simple, connected graph G, the **eigenvector centrality** c_i^E of node i is defined as

$$c_i^E = u_i,$$

where $u = (u_1, \dots, u_n)$ is the (unique) normalized eigenvector of the adjacency matrix A of G with eigenvalue λ, and where $\lambda > |\lambda'|$ for all eigenvalues λ' of A. The **normalised eigenvector centrality** of node i is defined as

$$C_i^E = \frac{c_i^E}{C^E},$$

where $C^E = \sum_j c_j^E$.

In [ ]: