CS4423 - Networks

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

2. Centrality Measures

Lecture 8: Closeness and Betweenness Centrality

... continuing from last time. First, load the libraries.

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

Next, recover the graph `G` of marital ties between Florentine families, together with the node attributes we have already determined.

In [2]:
```
G = nx.read_yaml("florentine.yml")
print(G.number_of_nodes())
G.nodes['Medici']
```

```
16
```

Out[2]:
```
{'$C_i^D$': 0.4, 'degree': 6, 'priorates': 53, 'wealth': 103}
```

In [3]:
```
G.number_of_nodes()
```

Out[3]:
```
16
```

In [4]:
```
cc = list(nx.connected_components(G))[0]
GG = G.subgraph(cc)
```

Eigenvectors and Centrality (cont'd.)

Recall that

$$\begin{pmatrix} 3 & 1 \\ 2 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 4 \end{pmatrix} = 4 \begin{pmatrix} 1 \\ 1 \end{pmatrix},$$

making the vector $\binom{1}{1}$ an **eigenvector** for the **eigenvalue** $\lambda = 4$ of the matrix A.

In this example

- all entries a_{ij} of the matrix $A = (a_{ij})$ are positive;
- the eigenvalue 4 is strictly larger than the magnitude $|\lambda'|$ of all the other (complex or real) eigenvalues of A (here, $\lambda' = -1$);
- and the eigenvalue $\lambda = 4$ has an eigenvector with all its entries positive.

The theoretical foundation for eigenvector centrality 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, which basically states that the above observations are no coincidence.

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

Let's attach the eigenvector centralities as node attributes and display the resulting table.

In [5]:
```
eigen_cen = nx.eigenvector_centrality(GG)
nx.set_node_attributes(G, eigen_cen, '$c_i^E$')
```

In [6]:
```
pd.DataFrame.from_dict(dict(GG.nodes(data=True)), orient='index').sort_values('
```

Out[6]:

	C_i^D	degree	priorates	wealth	c_i^E
Medici	0.400000	6	53	103	0.430315
Guadagni	0.266667	4	21	8	0.289117
Strozzi	0.266667	4	74	146	0.355973
Albizzi	0.200000	3	65	36	0.243961
Bischeri	0.200000	3	12	44	0.282794
Castellani	0.200000	3	22	20	0.259020
Peruzzi	0.200000	3	42	49	0.275722
Ridolfi	0.200000	3	38	27	0.341554
Tornabuoni	0.200000	3	n/a	48	0.325847
Barbadori	0.133333	2	n/a	55	0.211706
Salviati	0.133333	2	35	10	0.145921
Acciaiuoli	0.066667	1	53	10	0.132157
Ginori	0.066667	1	n/a	32	0.074925
Lamberteschi	0.066667	1	0	42	0.088793
Pazzi	0.066667	1	n/a	48	0.044815

Closeness centrality

A node x in a network can be regarded as being central, if it is **close** to (many) other nodes, as it can then quickly interact with them. A simple way to measure closeness in this sense is based on the sum of all the distances to the other nodes, as follows.

Definition (Closeness Centrality). In a simple, connected graph G, the **closeness centrality** c_i^C of node i is defined as

$$c_i^C = \Big(\sum_j d_{ij}\Big)^{-1}.$$

The **normalized closeness centrality** of node i, defined as

$$C_i^C = (n-1)c_i^C$$

takes values in the interval $[0, 1]$.

Why is $0 \leq C_i^D \leq 1$? When is $C_i^D = 1$?

BFS again. This time as a python function, which takes a graph $G = (X, E)$ and a vertex $x \in X$ as its arguments. It returns a **dictionary**, which for each node as key has the distance to x as its value.

In [7]:
```
from queue import Queue

def dists(graph, node):

    # 1. init: set up the dictionary and queue
    dists = { x : None for x in graph.nodes() }
    dists[node] = 0
    q = Queue()
    q.put(node)

    # 2. loop
    while not q.empty():
        x = q.get()
        for y in G.neighbors(x):
            if dists[y] == None:
                dists[y] = dists[x] + 1
                q.put(y)

    # 3. stop here
    return dists
```

In [8]:
```
d = dists(G, 'Medici')
```

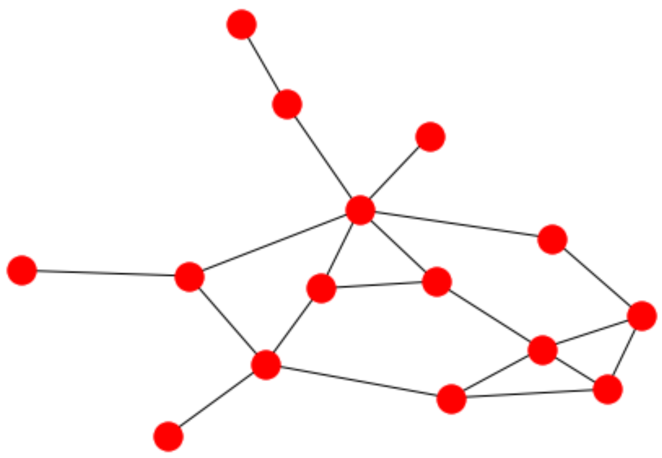
In [9]:
```
d
```

Out[9]:
```
{'Acciaiuoli': 1,
 'Albizzi': 1,
 'Barbadori': 1,
 'Bischeri': 3,
 'Castellani': 2,
 'Ginori': 2,
 'Guadagni': 2,
 'Lamberteschi': 3,
 'Medici': 0,
 'Pazzi': 2,
 'Peruzzi': 3,
 'Pucci': None,
 'Ridolfi': 1,
 'Salviati': 1,
 'Strozzi': 2,
 'Tornabuoni': 1}
```

In [10]:
```
cc = list(nx.connected_components(G))[0]
GG = G.subgraph(cc)
```

In [11]:
```
nx.draw(GG)
```

```
/home/goetz/anaconda3/lib/python3.7/site-packages/networkx/drawing/nx_pylab.py
:611: MatplotlibDeprecationWarning: isinstance(..., numbers.Number)
  if cb.is_numlike(alpha):
```



In [12]:
```
d = dists(GG, 'Medici')
```

In [13]:
```
sum(d.values())
```

Out[13]: 25

In [14]:
```
n = GG.number_of_nodes()
close_cen = { x : (n-1)/sum(dists(GG, x).values()) for x in GG.nodes() }
```

In [15]:
```
close_cen
```

Out[15]:
```
{'Acciaiuoli': 0.3684210526315789,
 'Albizzi': 0.4827586206896552,
 'Barbadori': 0.4375,
 'Bischeri': 0.4,
 'Castellani': 0.3888888888888889,
 'Ginori': 0.3333333333333333,
 'Guadagni': 0.4666666666666667,
 'Lamberteschi': 0.32558139534883723,
 'Medici': 0.56,
 'Pazzi': 0.2857142857142857,
 'Peruzzi': 0.3684210526315789,
 'Ridolfi': 0.5,
 'Salviati': 0.3888888888888889,
 'Strozzi': 0.4375,
 'Tornabuoni': 0.4827586206896552}
```

In [16]:
```
nx.closeness_centrality(GG)
```

Out[16]:
```
{'Acciaiuoli': 0.3684210526315789,
 'Albizzi': 0.4827586206896552,
 'Barbadori': 0.4375,
 'Bischeri': 0.4,
 'Castellani': 0.3888888888888889,
 'Ginori': 0.3333333333333333,
 'Guadagni': 0.4666666666666667,
 'Lamberteschi': 0.32558139534883723,
 'Medici': 0.56,
 'Pazzi': 0.2857142857142857,
 'Peruzzi': 0.3684210526315789,
 'Ridolfi': 0.5,
 'Salviati': 0.3888888888888889,
 'Strozzi': 0.4375,
 'Tornabuoni': 0.4827586206896552}
```

In [17]:
```
nx.set_node_attributes(G, close_cen, '$C_i^C$')
```

In [18]:
```
pd.DataFrame.from_dict(dict(G.nodes(data=True)), orient='index').sort_values('c
```

Out[18]:

	C_i^D	degree	priorates	wealth	c_i^E	C_i^C
Medici	0.400000	6	53	103	0.430315	0.560000
Guadagni	0.266667	4	21	8	0.289117	0.466667
Strozzi	0.266667	4	74	146	0.355973	0.437500
Albizzi	0.200000	3	65	36	0.243961	0.482759
Bischeri	0.200000	3	12	44	0.282794	0.400000
Castellani	0.200000	3	22	20	0.259020	0.388889
Peruzzi	0.200000	3	42	49	0.275722	0.368421
Ridolfi	0.200000	3	38	27	0.341554	0.500000
Tornabuoni	0.200000	3	n/a	48	0.325847	0.482759
Barbadori	0.133333	2	n/a	55	0.211706	0.437500
Salviati	0.133333	2	35	10	0.145921	0.388889
Acciaiuoli	0.066667	1	53	10	0.132157	0.368421
Ginori	0.066667	1	n/a	32	0.074925	0.333333
Lamberteschi	0.066667	1	0	42	0.088793	0.325581
Pazzi	0.066667	1	n/a	48	0.044815	0.285714
Pucci	0.000000	0	0	3	NaN	NaN

Betweenness Centrality

BFS once more. This time as a python function, which returns a **dictionaries**, which contains, for each node y, a list of **immediate predecessors** of y in a shortest path from x to y. Yes, that's another piece of information, BFS can determine on the fly. From this, recursively, one can reconstruct **all shortest paths** from x to y. We still need to compute the shortest path lengths as part of the algorithm.

In [19]:
```
from queue import Queue

def preds(graph, node):

    # 1. init: set up the two dictionaries and queue
    dists = { x : None for x in graph.nodes() }
    preds = { x : [] for x in graph.nodes() }
    dists[node] = 0
    q = Queue()
    q.put(node)

    # 2. loop
    while not q.empty():
        x = q.get()
        for y in G.neighbors(x):
            if dists[y] == None:
                dists[y] = dists[x] + 1
                q.put(y)
                preds[y].append(x)
            elif dists[y] == dists[x] + 1:
                preds[y].append(x)

    # 3. stop here
    return preds
```

In [20]:
```
p = preds(GG, 'Medici')
p
```

Out[20]:
```
{'Acciaiuoli': ['Medici'],
 'Albizzi': ['Medici'],
 'Barbadori': ['Medici'],
 'Bischeri': ['Guadagni', 'Strozzi'],
 'Castellani': ['Barbadori'],
 'Ginori': ['Albizzi'],
 'Guadagni': ['Albizzi', 'Tornabuoni'],
 'Lamberteschi': ['Guadagni'],
 'Medici': [],
 'Pazzi': ['Salviati'],
 'Peruzzi': ['Castellani', 'Strozzi'],
 'Ridolfi': ['Medici'],
 'Salviati': ['Medici'],
 'Strozzi': ['Ridolfi'],
 'Tornabuoni': ['Medici']}
```

When interactions between non-adjacent agents in a network depend on middle men (on shortest paths between these agents, power comes to those in the middle. Betweennness centrality measures centrality in terms of the number of shortest paths a node lies on.

Defintion (Betweenness Centrality). In a simple, connected graph G, the **betweenness centrality** c_i^B of node i is defined as

$$c_i^B = \sum_{j \neq i} \sum_{k \neq i} \frac{n_{jk}(i)}{n_{jk}},$$

where n_{jk} denotes the **number** of shortest paths from node i to node j, and where $n_{jk}(i)$ denotes the number of those shortest paths **passing through** node i.

The **normalized betweenness centrality** of node i, defined as

$$C_i^B = \frac{c_i^B}{(n-1)(n-2)}$$

takes values in the interval $[0, 1]$.

Using the predcessor lists, shortest paths can be enumerated recursively: the shortest path from x to itself is the empty path starting an ending at x. Else, if $y \neq x$ then each shortest path from y to x travels through exactly one of y's predecessors ... and ends in y.

In [21]:
```
def shortest_paths(x, y, pre):
    if x == y:
        return [[x]]
    else:
        paths = []
        for p in pre[y]:
            for path in shortest_paths(x, p, pre):
                paths.append(path + [y])

        return paths
```

In [22]:
```
shortest_paths('Medici', 'Pazzi', p)
```

Out[22]:
```
[['Medici', 'Salviati', 'Pazzi']]
```

In [23]:
```
between = { x : 0 for x in GG.nodes() }
```

In [24]:
```
n = GG.number_of_nodes()
for x in GG.nodes():
    pre = preds(GG, x)
    for y in GG.nodes():
        paths = shortest_paths(x, y, pre)
        njk = len(paths)*(n-1)*(n-2)
        for p in shortest_paths(x, y, pre):
            for z in p[1:-1]:  # exclude endpoints
                between[z] += 1/njk
```

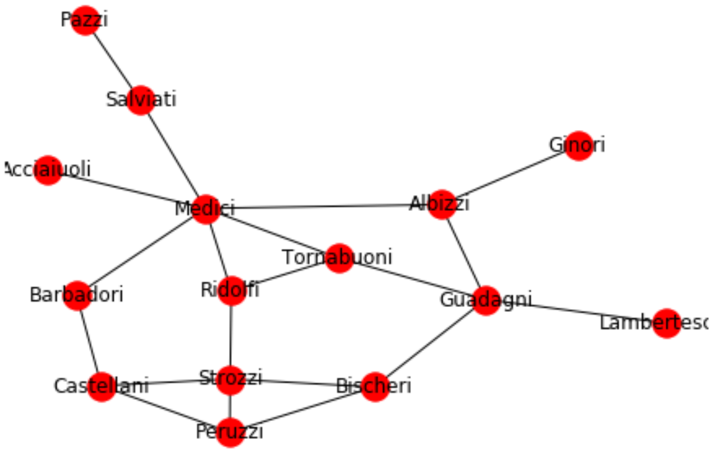
In [25]:
```
between
```

Out[25]:
```
{'Acciaiuoli': 0,
 'Albizzi': 0.2124542124542123,
 'Barbadori': 0.09340659340659344,
 'Bischeri': 0.10439560439560439,
 'Castellani': 0.054945054945054944,
 'Ginori': 0,
 'Guadagni': 0.25457875457875445,
 'Lamberteschi': 0,
 'Medici': 0.521978021978021,
 'Pazzi': 0,
 'Peruzzi': 0.02197802197802198,
 'Ridolfi': 0.11355311355311352,
 'Salviati': 0.14285714285714285,
 'Strozzi': 0.10256410256410255,
 'Tornabuoni': 0.09157509157509156}
```

In [26]:
```
nx.betweenness_centrality(GG)
```

Out[26]:
```
{'Acciaiuoli': 0.0,
 'Albizzi': 0.21245421245421245,
 'Barbadori': 0.09340659340659341,
 'Bischeri': 0.1043956043956044,
 'Castellani': 0.05494505494505495,
 'Ginori': 0.0,
 'Guadagni': 0.25457875457875456,
 'Lamberteschi': 0.0,
 'Medici': 0.521978021978022,
 'Pazzi': 0.0,
 'Peruzzi': 0.02197802197802198,
 'Ridolfi': 0.11355311355311355,
 'Salviati': 0.14285714285714288,
 'Strozzi': 0.10256410256410257,
 'Tornabuoni': 0.09157509157509158}
```

In [27]:
```
nx.draw(GG, with_labels=True)
```



Finally, let's add the normalized betweenness centralities as attributes to the nodes of the graph, and display the resulting table.

In [28]:
```
nx.set_node_attributes(G, between, '$C_i^B$')
```

In [29]:
```
pd.DataFrame.from_dict(dict(G.nodes(data=True)), orient='index').sort_values('c
```

Out[29]:

	C_i^D	degree	priorates	wealth	c_i^E	C_i^C	C_i^B
Medici	0.400000	6	53	103	0.430315	0.560000	0.521978
Guadagni	0.266667	4	21	8	0.289117	0.466667	0.254579
Strozzi	0.266667	4	74	146	0.355973	0.437500	0.102564
Albizzi	0.200000	3	65	36	0.243961	0.482759	0.212454
Bischeri	0.200000	3	12	44	0.282794	0.400000	0.104396
Castellani	0.200000	3	22	20	0.259020	0.388889	0.054945
Peruzzi	0.200000	3	42	49	0.275722	0.368421	0.021978
Ridolfi	0.200000	3	38	27	0.341554	0.500000	0.113553
Tornabuoni	0.200000	3	n/a	48	0.325847	0.482759	0.091575
Barbadori	0.133333	2	n/a	55	0.211706	0.437500	0.093407
Salviati	0.133333	2	35	10	0.145921	0.388889	0.142857
Acciaiuoli	0.066667	1	53	10	0.132157	0.368421	0.000000
Ginori	0.066667	1	n/a	32	0.074925	0.333333	0.000000
Lamberteschi	0.066667	1	0	42	0.088793	0.325581	0.000000
Pazzi	0.066667	1	n/a	48	0.044815	0.285714	0.000000
Pucci	0.000000	0	0	3	NaN	NaN	NaN

In [ ]: