CS4423-W07-Part-2

February 27, 2025

Table of Contents

CS4423-Networks: Week 7 (26+27 Feb 2025)

1 Part 2: Computing Centrality Measures

Niall Madden, School of Mathematical and Statistical Sciences
University of Galway

This Jupyter notebook, and PDF and HTML versions, can be found at https://www.niallmadden.ie/2425-CS4423/#Week07

This notebook was written by Niall Madden, adapted from notebooks by Angela Carnevale.

1.0.1 Modules for this notebook

[1]:
```
import networkx as nx
import numpy as np
opts = { "with_labels": True,  "node_color": "gold"} # gold nodes today

np.set_printoptions(precision=3)    # just display arrays to 3 decimal places
np.set_printoptions(suppress=True)  # avoid scientific notation (better for␣
 ↪matrices)

from queue import Queue # Use this in computing distances
import yaml    # for saving and diplaying data, especially dictionaries
import pandas as pd  # summarising data
import matplotlib.pyplot as plt
```

1.1 Computing Degree Centrality

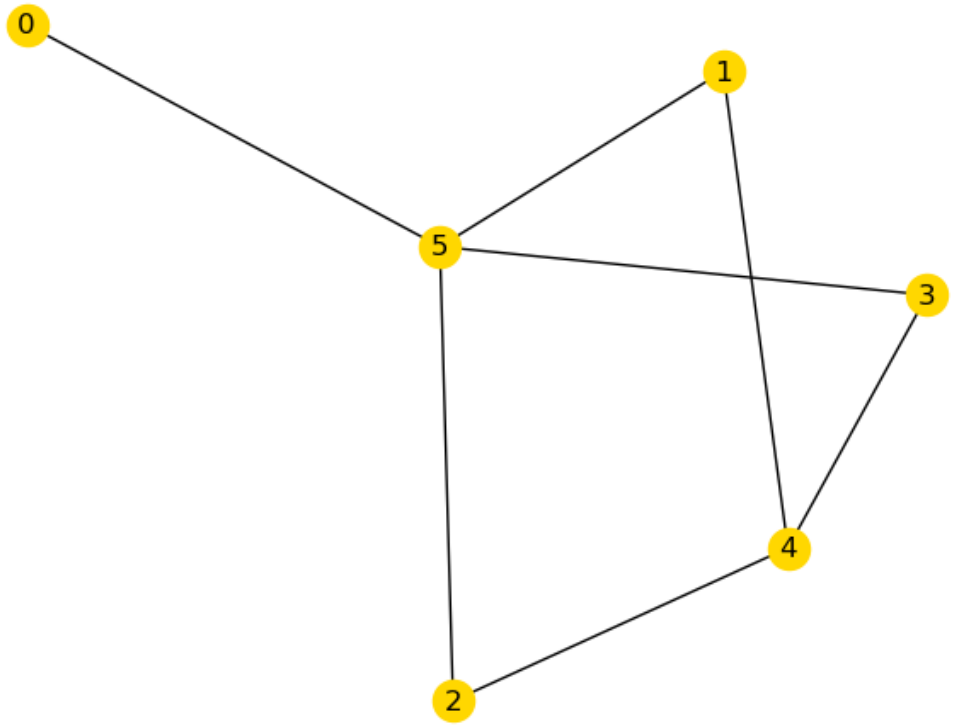
Computing the *Degree Centrality* of a graph is easy, and there are many ways to do it. Here we'll look at one way involving the adjancy matrix, since the core idea will be used again for Closeness Centraility

We'll start with an example: where $deg(5) = 4$, $deg(4) = 3$ and $deg(3) = 2$.

As an extra trick, we'll force `networkx` to order the nodes lexigraphically, by * Creating an empty graph * adding the nodes first (in the order I want) * then add the edges.

[2]:
```
G1 = nx.Graph()                 # empty graph
G1.add_nodes_from(range(6))     # add nodes 0,1,2,3,4,5 in that oder
EdgeList = [[5,0],[5,1],[5,2],[5,3],[4,3],[4,2],[4,1]]
G1.add_edges_from(EdgeList)
```

[3]:
```
nx.draw(G1,**opts)
```



Compute the adjacency matrix, A_1 (as as numpy array), and then multiply by a vector of ones, thus computing the row sums of A_1. Then we normalise:

[4]:
```
A1 = nx.adjacency_matrix(G1).toarray();
n = G1.order()
e = np.ones((n,1))     # vector of ones
n = G1.order()
degree_vector = A1@e/(n-1)    # normalised
for i in range(6):
    print(f"Node {i} has degree {degree_vector[i]}")
```

```
Node 0 has degree [0.2]
Node 1 has degree [0.4]
Node 2 has degree [0.4]
Node 3 has degree [0.4]
Node 4 has degree [0.6]
Node 5 has degree [0.8]
```

1.1.1 Computing it in `netwprkx`

Though it is hardly needed, one can compute the degree cenrality of this network using the `nx.degree_centrality()` method. Note that this returns a dictionary:

[5]:
```
CD = nx.degree_centrality(G1)
print(CD)
print(f"\nThe degree centrality of Node 3 is {CD[3]:.3f}")
```

```
{0: 0.2, 1: 0.4, 2: 0.4, 3: 0.4, 4: 0.6000000000000001, 5: 0.8}

The degree centrality of Node 3 is 0.400
```

1.2 Eigenvector Centrality

To compute the Eigenvector Centrality of nodes in network, G: * Compute the adjacency matrix, A. * Compute the largest, positive eigenvalue of A (since A is symmetric, this is unique) * It has a corresponding positive eigenvector, $\vec{v}$, which we can scale so that $v^v = 1$. * v_i is the Eigenvector Centrality node i.

1.2.1 Computing Eigenvalues with `eigh`

We can use `np.linalg.eig()` which computes the eigenvalues and eigenvectors of a matrix: `l, V = np.linalg.eig(A)` computes * `l` : an array of length n containing the eigenvalues of A. (Note: we can't call this array `lambda`, since that is a keyword in Python. * `V`: a $n \times n$ matrix; column i of V is the eigenvector corresponding to the eigenvalues λ_i.

(Note: since A s symmetric, it can be faster to use the `np.linalg.eigh()` function)

[6]:
```
l, V = np.linalg.eig(A1)
print(f"The eigenvalues of A are {l}")
```

```
The eigenvalues of A are [-2.558 -0.677  0.677  2.558  0.     0.   ]
```

We can see that there is an eigenvalue listed which is positive, amd larger than the rest. Let's look at the corresponding eigenvector. We make have to scale by -1, if the enties are negative:

[7]:
```
i_max = np.argmax(l)     # get index of largest eigenvalue
v = V[:,i_max]*np.sign(V[0,i_max])   # set v to be corresponding e'vec; ensure␣
 ↪it is positive
print("v=",v)
```

```
v= [0.211 0.39  0.39  0.39  0.457 0.54 ]
```

[8]:
```
for i in range(6):
    print(f"Node {i} has eigenvector centrality {v[i]:7.4}")
```

```
Node 0 has eigenvector centrality   0.211
Node 1 has eigenvector centrality  0.3897
Node 2 has eigenvector centrality  0.3897
Node 3 has eigenvector centrality  0.3897
Node 4 has eigenvector centrality  0.4571
Node 5 has eigenvector centrality  0.5395
```

1.2.2 The Power Method

There are subfields in the *Numerical Linerar Algebra* dedicated to computing estimates for eigenvalues and eigenvectors. When we only need one eigenvalue, and it is the largest, use the **Power method**:

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

Questions Does this work? Meaning: * Does the sequence actually converge? * Does it return the correct values?

We won't study the theory of that - but will check an example.

Here is an implementation. We'll just do 10 iterations. By rights, we should use a while loop to iterate until successive estimates are sufficiently close to each other.

[9]:
```
n = G1.order()
u = np.ones((n,1)); u=u/np.linalg.norm(u)
for i in range(10):
    v = A1 @ u  # update u
    l = v[0]/u[0] # appriximate the eigenvalue
    u = v/np.linalg.norm(v) # normalise it
```

The result we get is as follows (compare yourself with the value computed earlier)

[10]:
```
print(u)
```

```
[[0.242]
 [0.447]
 [0.447]
 [0.447]
 [0.378]
 [0.447]]
```

1.2.3 Computing it in `networkx`

To compute eigenvector centraility in `networkx`, we can use the `nx.eigenvector_centrality` function, which returns a dictionary.

[11]:
```
CE = nx.eigenvector_centrality(G1)
print(yaml.dump(CE))  # looks better than "print(CV)"

print(f"\nThe Eigenvector  centrality of Node 3 is {CE[3]:.3f}")
```

```
0: 0.21095390422598534
1: 0.38965701954264753
2: 0.38965701954264753
3: 0.38965701954264753
4: 0.45705572814102585
5: 0.5395375177211617
```

```
The Eigenvector  centrality of Node 3 is 0.390
```

1.3 Closeness Centrality

We learned yesterday that the **normalised closeness centrality** of node i is

$$C_i^C = \frac{n-1}{\sum_{j=1}^{n} d_{ij}}.$$

To compute this, for all nodes, we could construct the *distance matrix* for the graph. For that, we need to compute the distance between every pair of nodes. As we learned last week. that can be done with *BFS*. We learned how to do that in Week 6 (Part 1). Here is a different implementation…

- The following `python` function implements BFS for shortest distance from a previous lecture.

- It takes a graph $G = (X, E)$ and a vertex $x \in X$ as its arguments.
- It returns a **dictionary**, which assigns to each node its distance to x.

[12]:
```
def distances(G, x):
    # 1. init: set up the dictionary and a queue
    dists = { y: None for y in G } # distances
    Q = Queue() # queue of nodes to be visited
    dists[x] = 0
    Q.put(x)

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

    # 3. stop here
    return dists
```

Let's check it works for Node 0

[13]:
```
distances(G1,0)
```

[13]:
```
{0: 0, 1: 2, 2: 2, 3: 2, 4: 3, 5: 1}
```

Next we use these values to build the *distance matrix*, D_1

[14]:
```
D1 = np.zeros_like(A1)
for i in range(n):
    d_i =  distances(G1,i)
    D1[i,:]=list(d_i.values())
```

[15]:
```
print(D1)
```

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

Now compute the *distance sum* vector, $\vec{s}$

[16]:
```
n=G1.order()
s = D1 @ np.ones((n,1))
print(s)
```

```
[[10.]
 [ 8.]
 [ 8.]
 [ 8.]
 [ 8.]
 [ 6.]]
```

Finally, compute the Closeness Centrality vector:

[17]:
```
CC = (n-1)/s   # note: using entrywise division
print(CC)
```

```
[[0.5  ]
 [0.625]
 [0.625]
 [0.625]
 [0.625]
 [0.833]]
```

[ ]:
```

```

Compare with the `networkx` function:

[18]:
```
print(nx.closeness_centrality(G1))
```

```
{0: 0.5, 1: 0.625, 2: 0.625, 3: 0.625, 4: 0.625, 5: 0.8333333333333334}
```

1.4 Betweenness Centrality

From yesterday: the **betweenness centrality**, c_i^B of node i is defined as

$$c_i^B = \sum_j \sum_k \frac{n_i(j,k)}{n(j,k)}, \qquad j \neq k \neq i$$

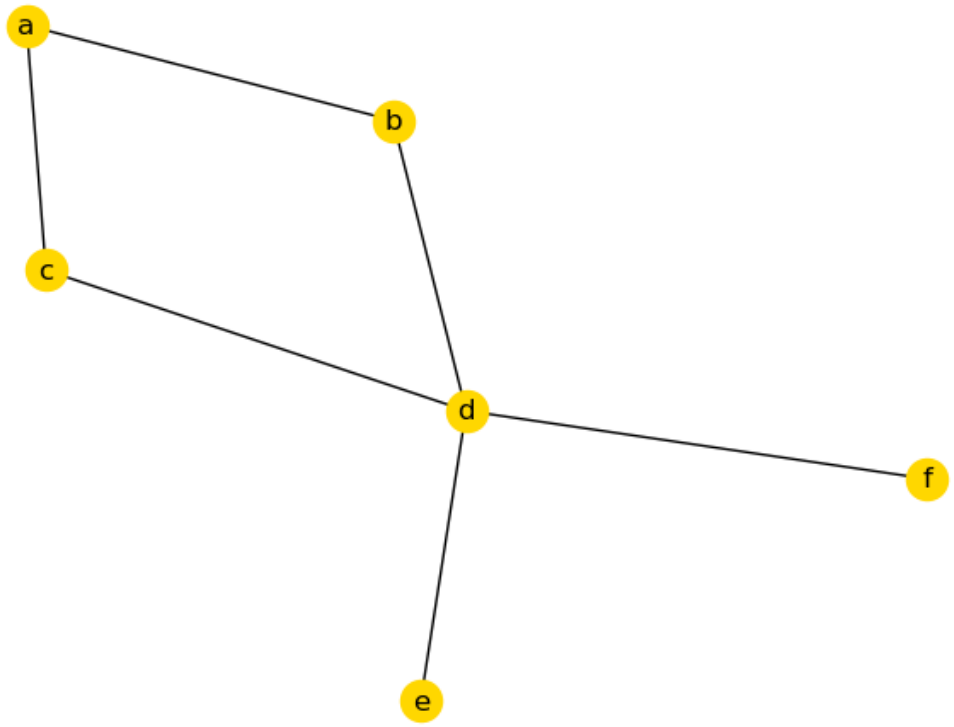
where * $n(j,k)$ denotes the *number* of shortest paths from node j to node k, and * $n_i(j,k)$ denotes the number of those shortest paths *passing through* node i.

Them the **normalised betweenness centrality**, C_i^B of node i is

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

Before we delve into the algorithms, let's take a simple network to study:

[19]:
```
G4 = nx.Graph()
G4.add_edges_from(['ab','ac','bd','cd','de','df']) # Example
nx.draw(G4,**opts)
```



The quantities, particularly, $n_i(j,k)$, can take some work to compute. Yet again, we use a variant on **BFS**.

First for any given any node, we need to compute all its **predecessors** on the shortest paths between it and every other node. That is, if `z` is a predecessor of `x` if it is a neighbour `x`, and on the shortest path between `x` and `y`.

This is then used to count the *number* of shortest paths between a pair of nodes.

Our function works as follows: 1. Takes the graph G and node x as inputs 2. Returns a dictionary, `preds` where `preds[y]` is the list of predecessors of x in the paths from y to x.

[20]:
```
def predecessors(G, x):
    """ Computes the predecssors of Node x in G"""
    # 1. init: set up the two dictionaries and queue
    dists = { y: None for y in G } # distances
    preds = { y: [] for y in G }
    Q = Queue()
    dists[x] = 0 #
    Q.put(x)

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

    # 3. stop here
    return preds
```

Let's check it it works by computing all the predecessors of `a`:

[21]:
```
p = predecessors(G4,'a') ## check our work
print(p)
```

```
{'a': [], 'b': ['a'], 'c': ['a'], 'd': ['b', 'c'], 'e': ['d'], 'f': ['d']}
```

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

[22]:
```
def shortest_paths(G, x, y):
    if x == y:
        return [[x]]
    paths = []
    pred_x_y = predecessors(G, x)[y] # predicessors of x in paths x to y
    # print(f"preds of {y} are {pred_x_y}")  # uncomment for more info
    for z in pred_x_y:
        for path in shortest_paths(G, x, z):
            paths.append(path + [y])
    return paths
```

Check if it works

[23]:
```
shortest_paths(G4, 'a', 'f')
```

[23]:
```
[['a', 'b', 'd', 'f'], ['a', 'c', 'd', 'f']]
```

Finally, we can compute the *betweenness* of a node:

[24]:
```
def betweeness(G):
    CB = { i : 0.0 for i in G }
    n = G.order()
    for i in G:
        for j in G:
            for k in G:
                paths_jk = shortest_paths(G, j, k)
                n_jk = len(paths_jk)
                n_i_jk = 0
                for p in paths_jk:
                    if i in p[1:-1]: # exclude enpoint
                        n_i_jk+=1
                CB[i] += n_i_jk/n_jk
        CB[i] /= ((n-1)*(n-2)) # normalise
    return(CB)
```

[25]:
```
betweeness(G4)
```

[25]:
```
{'a': 0.05, 'b': 0.15, 'c': 0.15, 'd': 0.75, 'e': 0.0, 'f': 0.0}
```

Naturally, this can also be done in `networkx`:

[26]:
```
nx.betweenness_centrality(G4)
```

[26]:
```
{'a': 0.05,
 'b': 0.15000000000000002,
 'c': 0.15000000000000002,
 'd': 0.75,
 'e': 0.0,
 'f': 0.0}
```

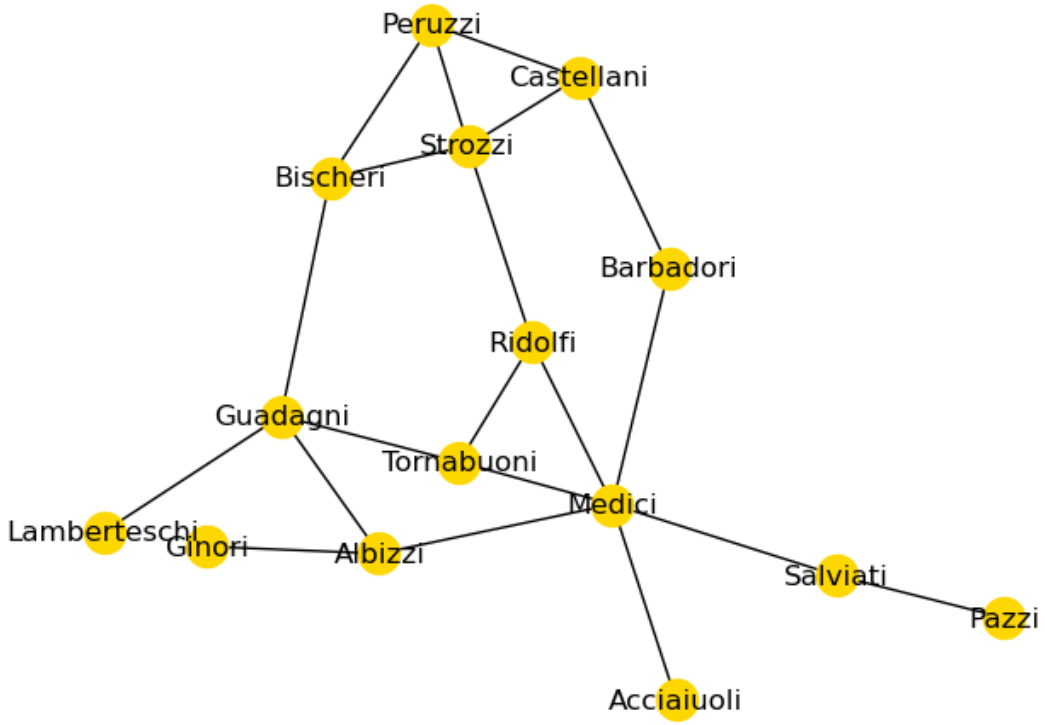
1.5 Example: 15th-century Florentine marriages

There is a famous network used to represent the marriage network of sixteen families in Florence, originally developed to showed how the Medici family gained power and took control of Florence by creating a high number of inter-marriages with the other families; see Wikipedia

1.5.1 The example

[27]:
```
FFG = nx.florentine_families_graph()
print(f"There are {FFG.order()} nodes and {FFG.size()} links in the network.")
pos = nx.spring_layout(FFG, seed=0) # record for layer use.
nx.draw(FFG, **opts, pos=pos)
```

```
There are 15 nodes and 20 links in the network.
```



1.5.2 Compute centralities

Let's compute the centralities of each (using `networkx` methods):

[28]:
```
CD = nx.degree_centrality(FFG)
CE = nx.eigenvector_centrality(FFG)
CC = nx.closeness_centrality(FFG)
CB = nx.betweenness_centrality(FFG)
```

Let's display the results in a `pandas` data frameL

[29]:
```
pd.DataFrame({
    'Key': list(CD.keys()),
    'Degree': list(CD.values()),
    'Eigenv': list(CE.values()),
    'Closen': list(CC.values()),
    'Betwee': list(CB.values())
}).sort_values('Degree', ascending=False)
```

[29]:

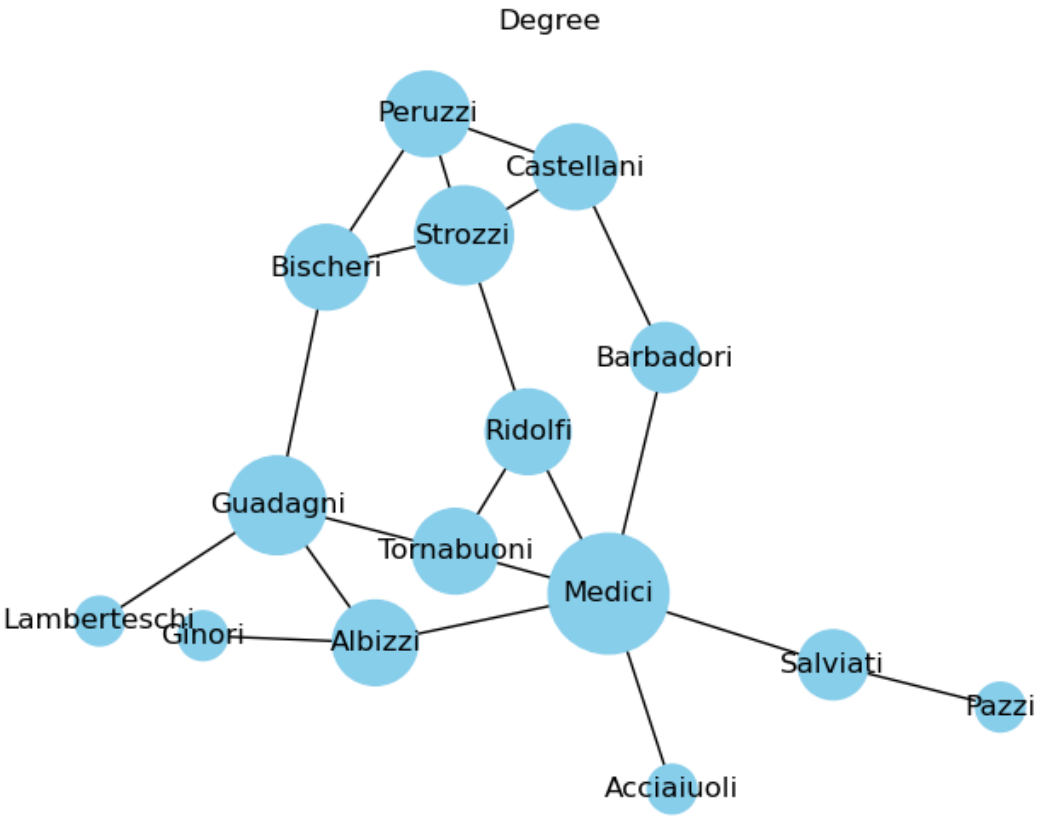
	Key	Degree	Eigenv	Closen	Betwee
1	Medici	0.428571	0.430315	0.560000	0.521978
4	Strozzi	0.285714	0.355973	0.437500	0.102564
12	Guadagni	0.285714	0.289117	0.466667	0.254579
2	Castellani	0.214286	0.259020	0.388889	0.054945
3	Peruzzi	0.214286	0.275722	0.368421	0.021978
6	Ridolfi	0.214286	0.341554	0.500000	0.113553
7	Tornabuoni	0.214286	0.325847	0.482759	0.091575
8	Albizzi	0.214286	0.243961	0.482759	0.212454
11	Bischeri	0.214286	0.282794	0.400000	0.104396
5	Barbadori	0.142857	0.211706	0.437500	0.093407
9	Salviati	0.142857	0.145921	0.388889	0.142857
0	Acciaiuoli	0.071429	0.132157	0.368421	0.000000
10	Pazzi	0.071429	0.044815	0.285714	0.000000
13	Ginori	0.071429	0.074925	0.333333	0.000000
14	Lamberteschi	0.071429	0.088793	0.325581	0.000000

1.5.3 Drawing graphs based on centrality

We'll finish by plotting the graphs again, but this time using the centralities measures to control the node sizes:

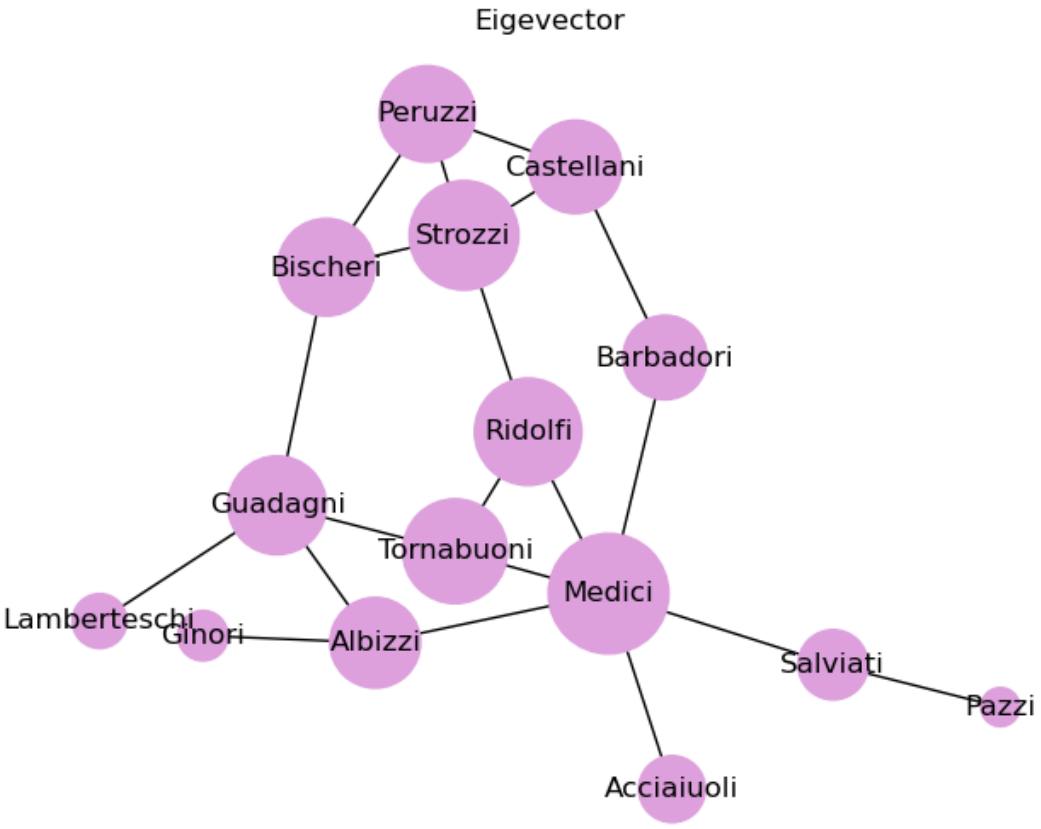
[30]:
```
node_sizes = [CD[node]*6000 for node in FFG.nodes()]
nx.draw(FFG, with_labels=True, node_size=node_sizes, node_color='skyblue',␣
 ↪pos=pos)
plt.title('Degree')
```

[30]:
```
Text(0.5, 1.0, 'Degree')
```



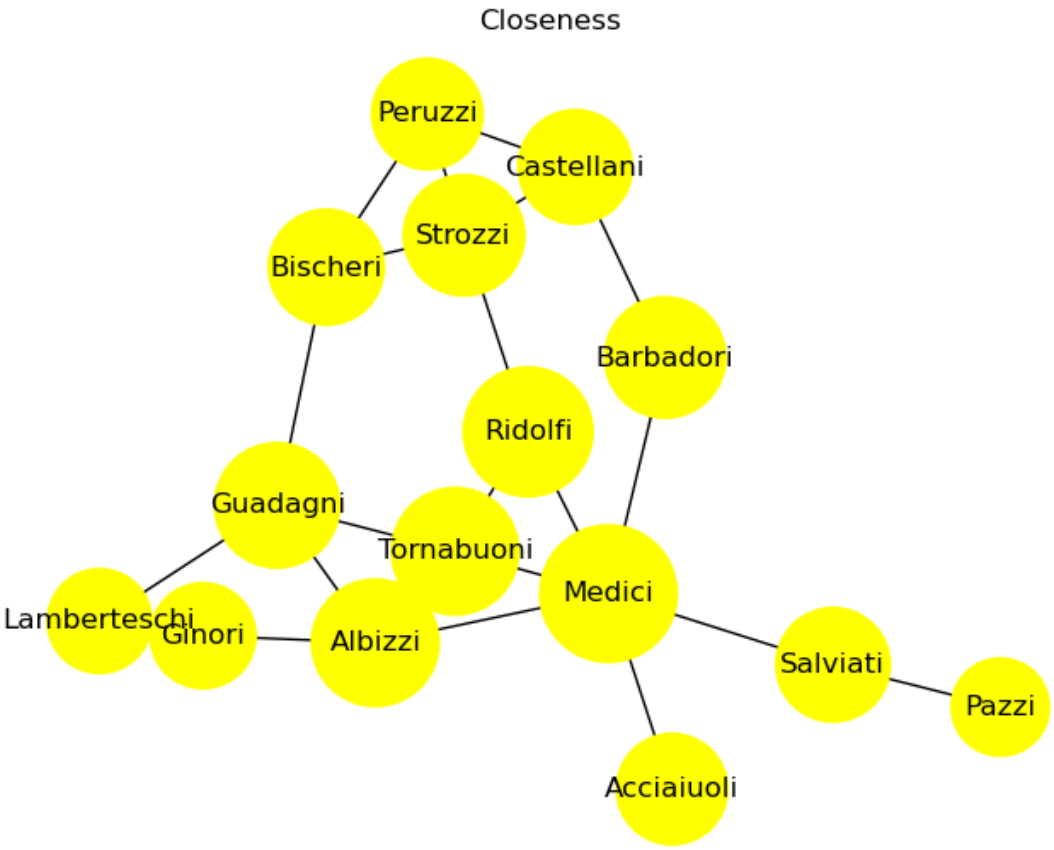
[31]:
```
node_sizes = [CE[node] * 6000 for node in FFG.nodes()]
nx.draw(FFG, with_labels=True, node_size=node_sizes, node_color='plum', pos=pos)
plt.title('Eigevector')
```

[31]: Text(0.5, 1.0, 'Eigevector')



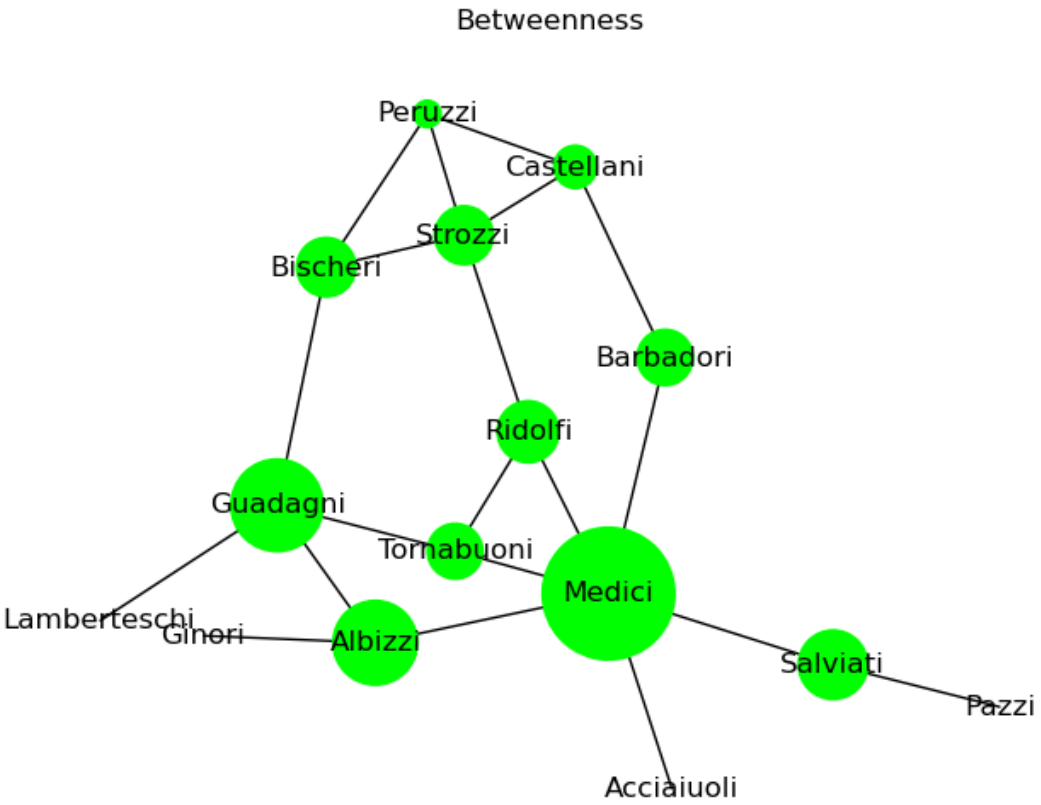
[32]:
```
node_sizes = [CC[node]*6000 for node in FFG.nodes()]
nx.draw(FFG, with_labels=True, node_size=node_sizes, node_color='yellow',␣
 ↪pos=pos)
plt.title('Closeness')
```

[32]:
```
Text(0.5, 1.0, 'Closeness')
```



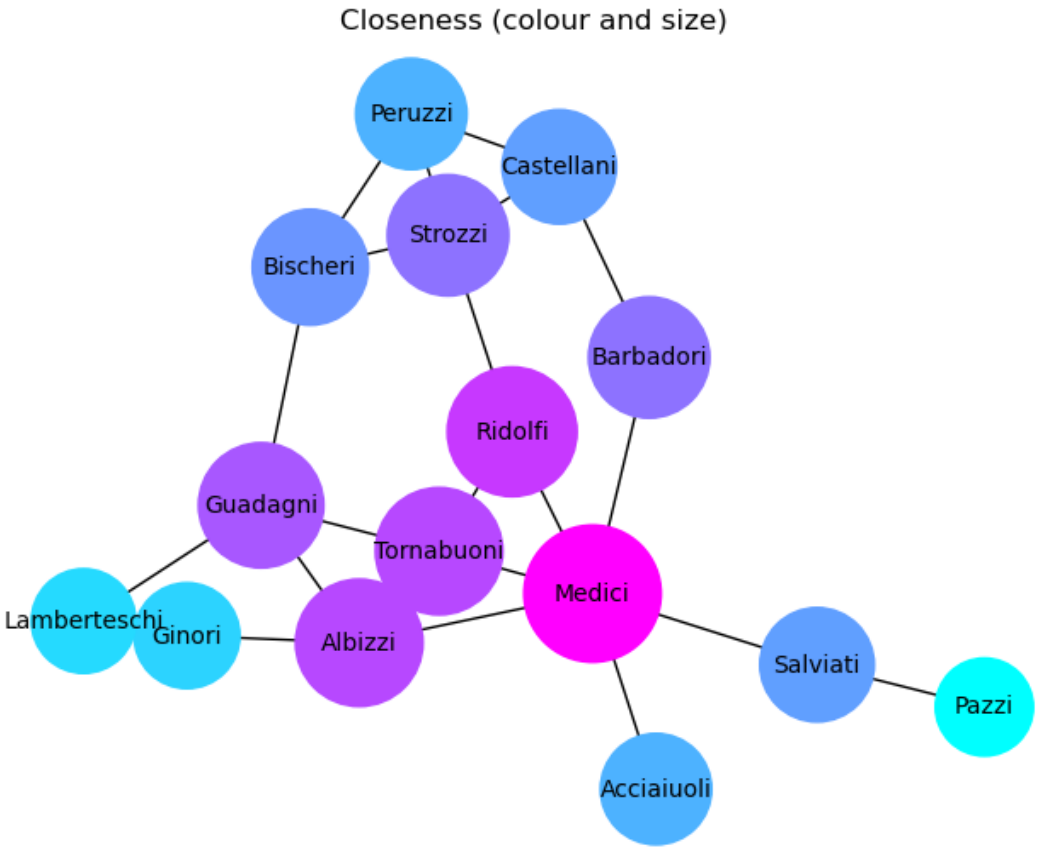
[33]:
```
node_sizes = [CB[node]*6000 for node in FFG.nodes()]
nx.draw(FFG, with_labels=True, node_size=node_sizes, node_color='lime', pos=pos)
plt.title('Betweenness')
```

[33]: Text(0.5, 1.0, 'Betweenness')



[34]:
```
node_sizes = [CC[node]*6000 for node in FFG.nodes()]
node_colors = [CC[node] for node in FFG.nodes()]
nx.draw(FFG, with_labels=True, node_size=node_sizes, node_color=node_colors,␣
 ↪cmap=plt.cm.cool, font_size=10, pos=pos)
plt.title('Closeness (colour and size)')
```

[34]:
```
Text(0.5, 1.0, 'Closeness (colour and size)')
```



1.6 Code corner (not covered in class explicitly)

This is a list a list of functions, and coding ideas, used in this notebook.

How to make a dictionary from two lists: one of keys, one of values, using `zip`. In this case, we'll make one based on the list of nodes, and vector of degree centralities:

[35]:
```
degree_vector # Note this is a (6,1) array, not a (6,) array: need to flatten
```

[35]:
```
array([[0.2],
       [0.4],
       [0.4],
       [0.4],
       [0.6],
       [0.8]])
```

[36]:
```
CD_dict = dict(zip(range(6), list(degree_vector.flatten() )))
print(CD_dict)
```

```
{0: 0.2, 1: 0.4, 2: 0.4, 3: 0.4, 4: 0.6, 5: 0.8}
```

Finished here Thursday