Lecture 8
Network Centrality: Hubs and Robustness

Chapter 3 Book: A First Course in Network Science

The idea of the project is that you adopt methods of <u>at least 3 lectures</u> to build your data Science story. In groups you need to list the contribution of each member.

03/07	Lecture 8	Centralities and Network Resilience		
03/14	Lecture 9	Clustering Networks	Assignment 3_II	
03/21	Lecture 10			
03/28	Spring Break	-	-	
04/04	Lecture 11	Scaling in Transportation Networks	Assignment 4 (Project Preparation)	
04/11	Lecture 12	OSMNx		
04/18	Lecture 13	Dynamics on Networls	Assignment 5 (Paper Draft)	
04/25	Lecture 14	Class Summary		
05/02	Project Presentations			
05/03	Project Presentations			
05/04	Project Presentations			

Assignments 1-4: 60%
Assignment 5, Paper draft: 15%
10% Final project paper
5% Project presentation

Outline

- Centrality measures
- The friendship paradox
- Ultra-small worlds
- Robustness
- Core decomposition

CIVENG C88-CYPLAN C88 > Files > Lectures > Lecture 8

→ Instructors: Making Course Materials Accessible (click to expand)

Search for files

0 items selected

+ Folder | Upload

- Data Science for Smart C
  - course_image
  - HW Solutions
  - HW Templates
  - Lectures
    - Lecture 1
    - Lecture 2
    - Lecture 3
    - Lecture 4
    - Lecture 5

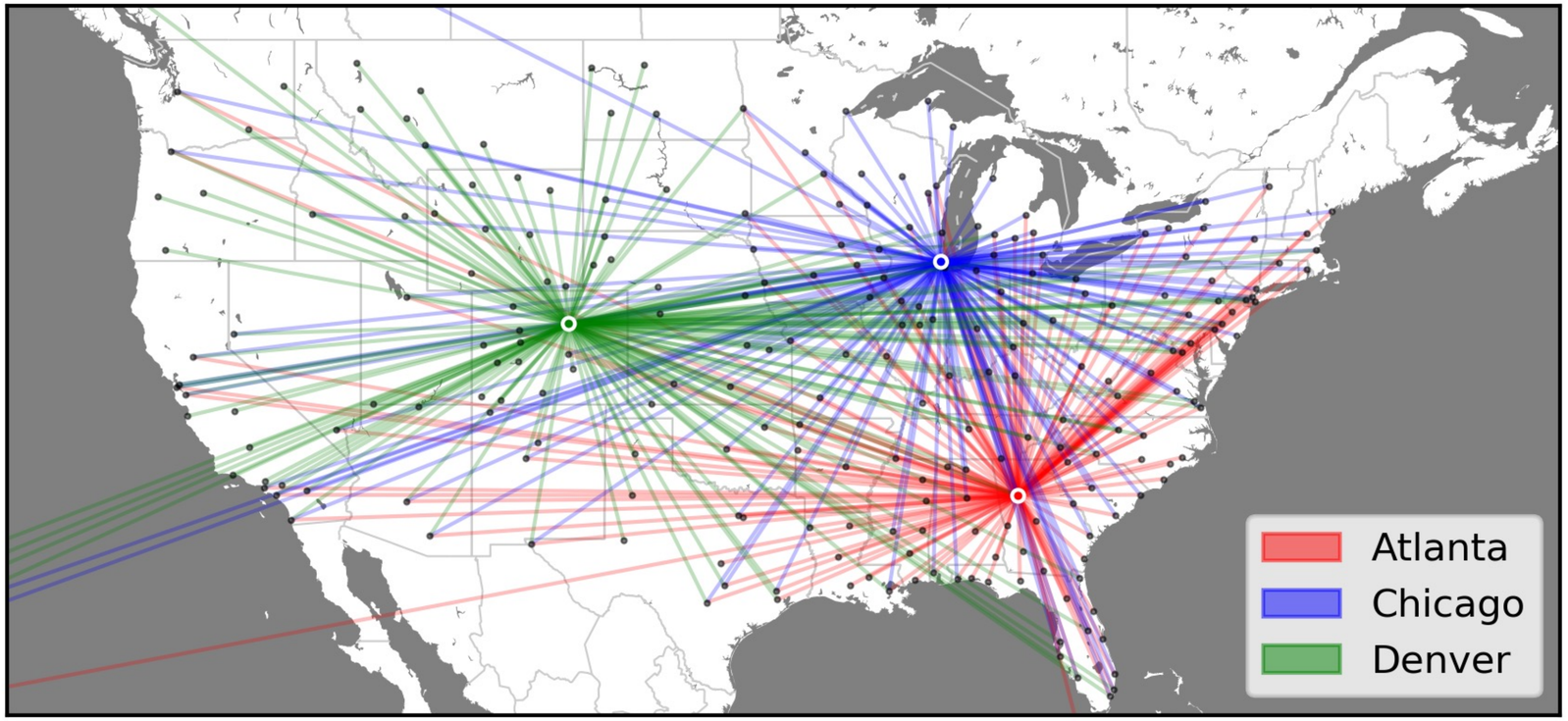
Name ▲	Date Created	Date Modified	Modified By	Size
centralities.ipynb	1:34pm	1:34pm		331 KB
ia-enron-only.edges	1:34pm	1:34pm		4 KB
Lecture8_CentralityTu	1:34pm	1:34pm		244 KB

0% of 68.7 GB used

All My Files

Centrality Measures

Real networks are heterogeneous, apart from the degree there are other nodes properties to measure the importance of a node.



Some nodes (and links) are much more important (**central**) than others!

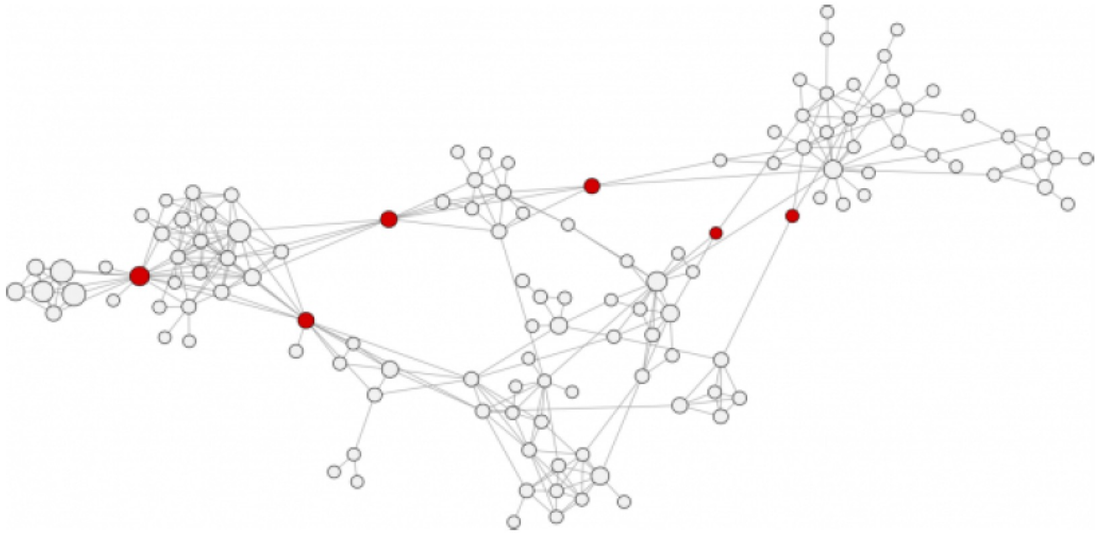
Centrality measures

- **Centrality**: measure of importance of a node
- **Sample Centrality Measures**:

  1. Degree
  2. Closeness
  3. Betweenness

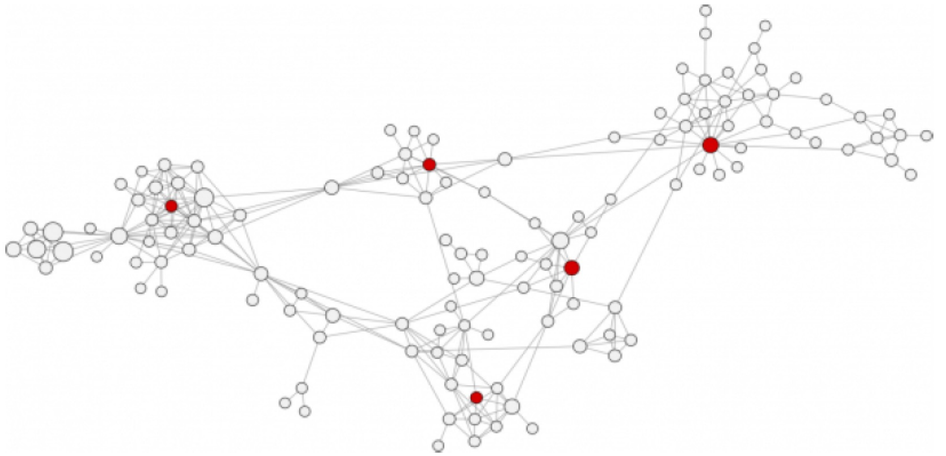
Betweenness centrality

If High Betweenness Centrality is high
A node bridges many shortest paths



If Closeness Centrality is high the sum of shortest paths to all other nodes is low

Closeness centrality



Note: node size is selected proportional to the degree

Degree

- **Degree of a node: number of neighbors of the node**

$$k_i = \text{number of neighbors of node } i$$

- High-degree nodes are called **hubs**

- **Average degree of the network:**

$$\langle k \rangle = \frac{\sum_i k_i}{N} = \frac{2L}{N}$$

```
G.degree(2) # returns the degree of node 2
G.degree()  # dict with the degree of all nodes of G
```

Closeness

Idea: a node is the more central the *closer* it is to the other nodes, on average

$$g_i = \frac{1}{\sum_{j \neq i} \ell_{ij}}$$

where ℓ_{ij} is the distance between nodes *i* and *j*

Closeness *is the inverse of the sum of the distance between node i, to all other nodes in the network*

```
nx.closeness_centrality(G, node) # closeness centrality
                                 # of node
```

Betweenness

Idea: a node is the more central the *more often it is crossed by paths*

$$b_i = \sum_{h \neq j \neq i} \frac{\sigma_{hj}(i)}{\sigma_{hj}}$$

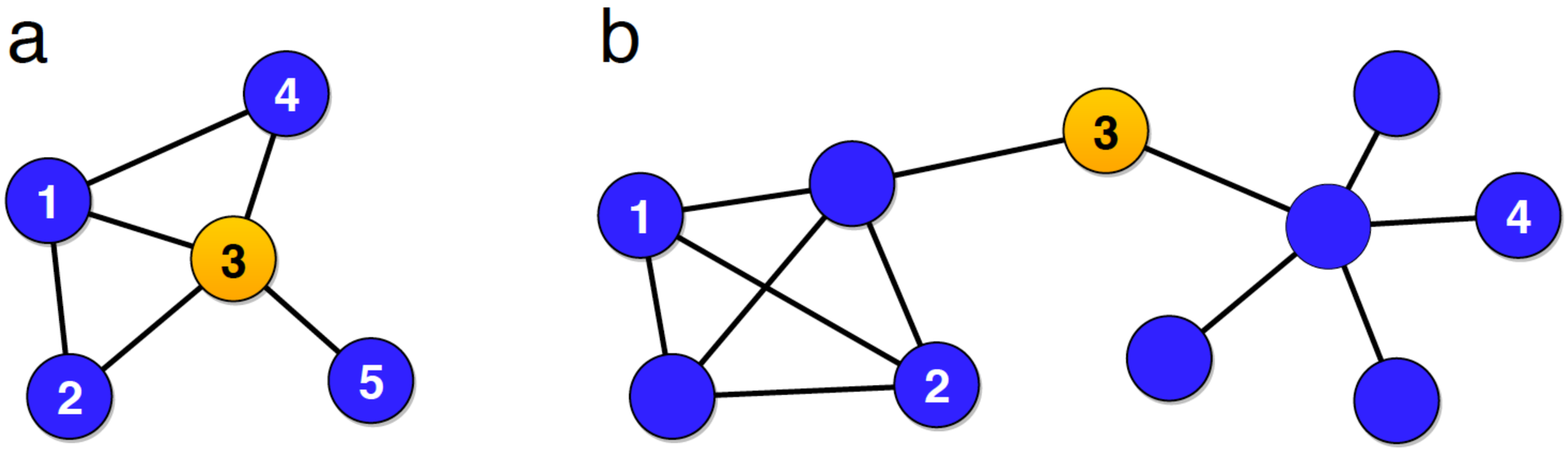
σ_{hj} = number of shortest paths from h to j

$\sigma_{hj}(i)$ = number of shortest paths from h to j running through i

σ_{hj} is of the order of (N-1)(N-1) $\sigma_{hj}(i)$ depends on the network location of i

Betweenness

Hubs usually have high betweenness, but there can be nodes with high betweenness **that are not hubs**



Betweenness

- Betweenness can be easily extended to links
- **Link betweenness**: fraction of shortest paths among all possible node pairs that pass through the link

```
nx.betweenness_centrality(G)       # dict nodes ->
                                   # betweenness centrality
nx.edge_betweenness_centrality(G) # dict links ->
                                   # betweenness centrality
```

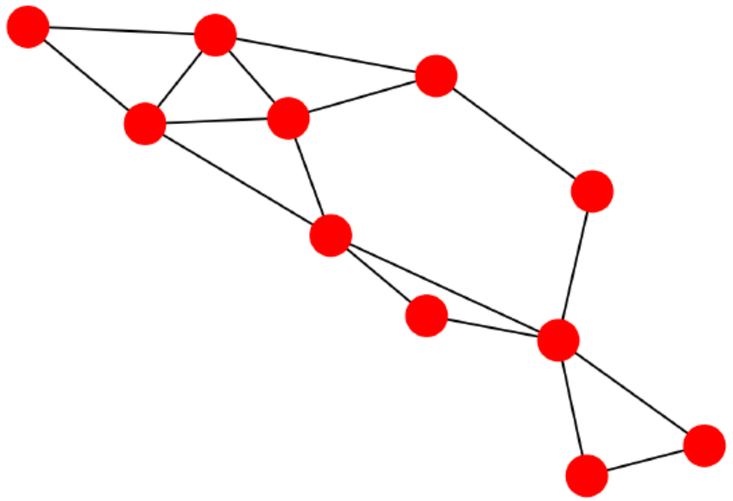
Note: This is important for Road Network Analysis

Centrality distributions

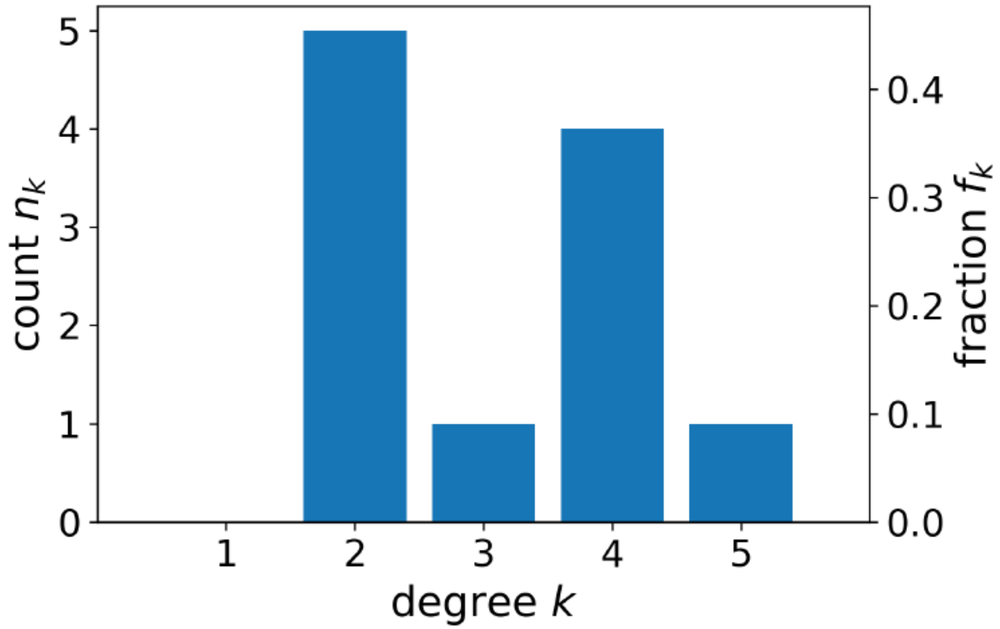
Statistical approach

Instead of focusing on individual nodes and links, we consider **classes** of nodes and links with similar properties

Centrality distributions



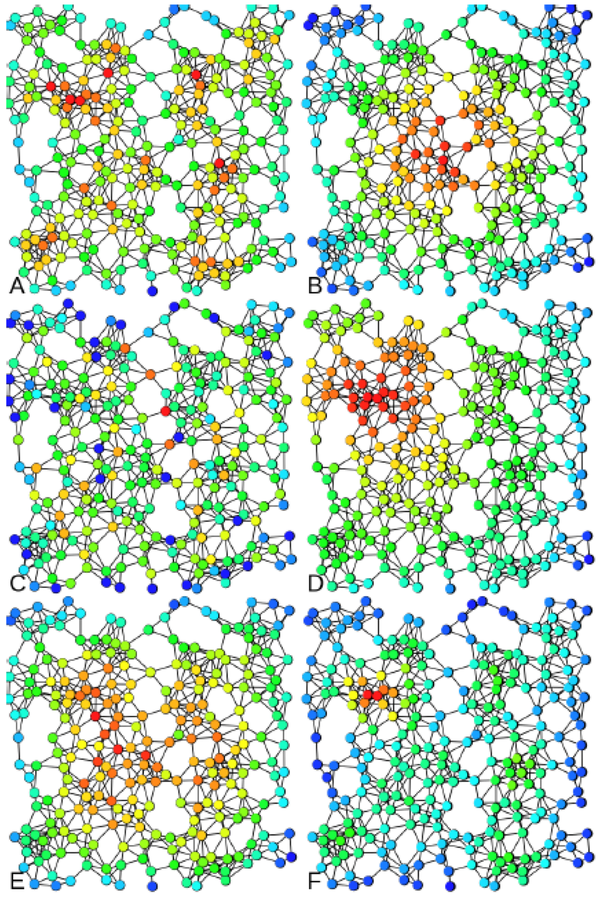
Histogram



- When $N \rightarrow Large, f_k$ becomes the **probability** p_k of having degree k

- **p_k versus k is the probability distribution of node degree**
  - **We can build PDFs with**

There are many examples of Centralities



from Claudio Rocchini

A) Degree centrality
B) Closeness centrality
C) Betweenness centrality
D) Eigenvector centrality
E) Katz centrality
F) Alpha centrality

Eigenvector Centrality: Nodes with high eigenvector centrality are those that have many connections to other important nodes.

Katz Centrality: It is based on the idea that a node is important if it is connected to other important nodes. However, unlike eigenvector centrality, Katz centrality takes into account not only direct connections but also indirect connections through longer paths in the network.

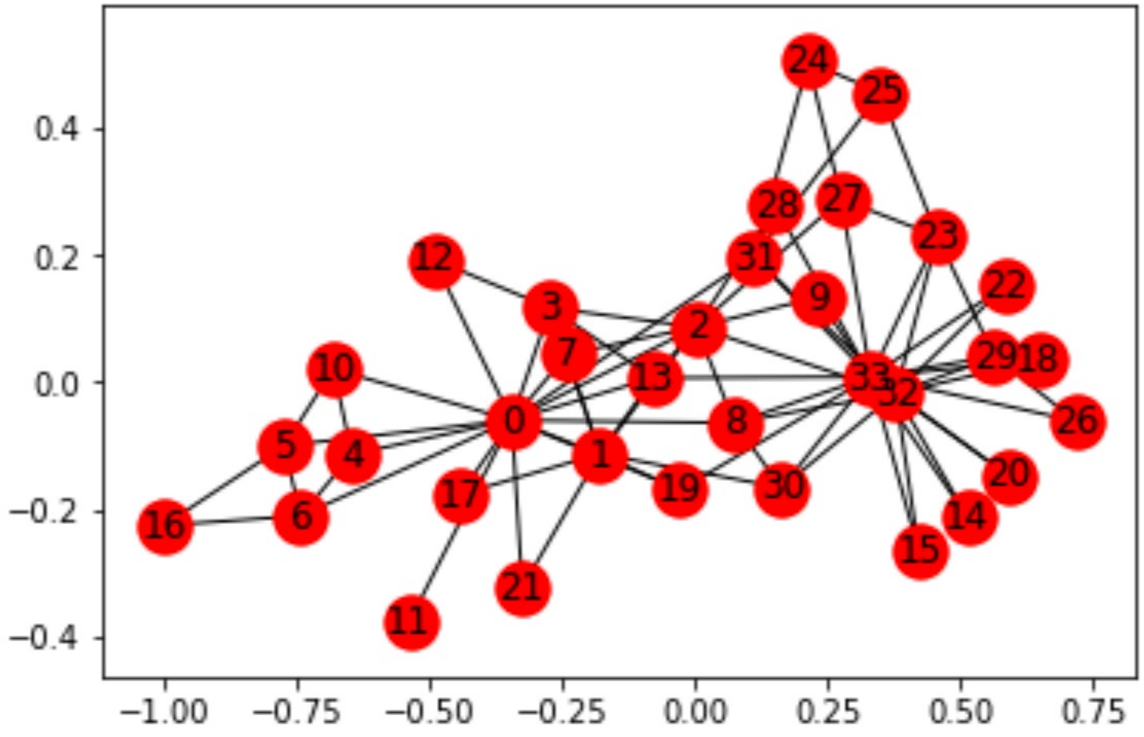
Alpha, is similar to Katz centrality but allows flexible weights in the path calculation.

centralities.ipynb

In [2]:
```
G = nx.karate_club_graph()
pos = nx.spring_layout(G) # Fix node positions on all pictures
```

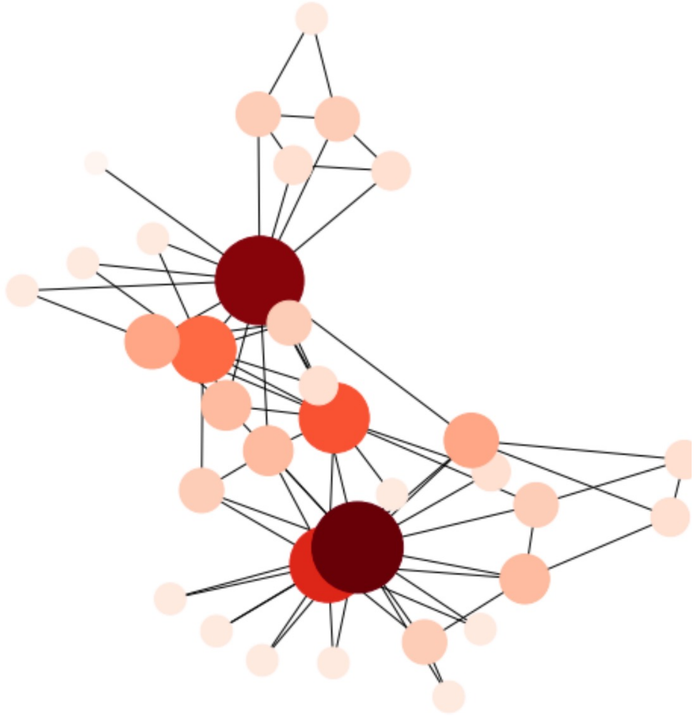
Empirical network within networkx

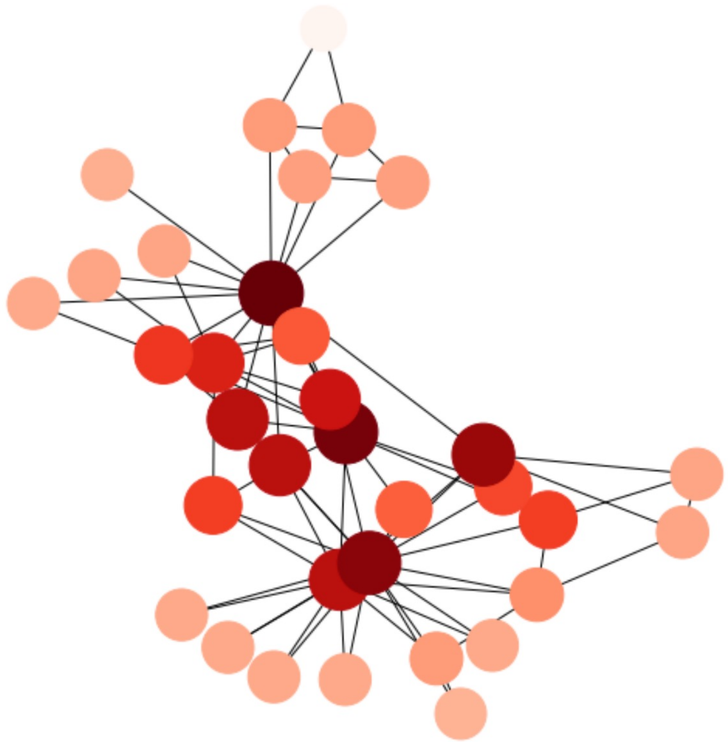
In [3]:
```
# Original network
nx.draw_networkx(G, pos)
```

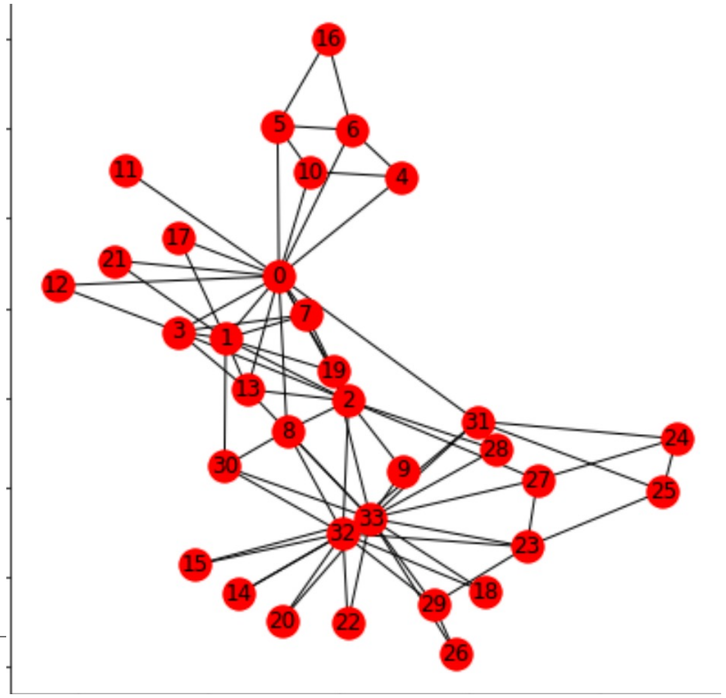


```python
# Degree centrality
dc = nx.degree_centrality(G)
plt.figure(2, figsize=(7,7))
coord = nx.spring_layout(G)
nx.draw(G,
          pos,
          nodelist=dc.keys(),
          node_size = [d*7000 for d in dc.values()],
          node_color=dc.values(),
          font_size=8,
          cmap=plt.cm.Reds,
          )
```

```python
# Closeness centrality
cl = nx.closeness_centrality(G)
plt.figure(1, figsize=(7,7))
coord = nx.spring_layout(G)
nx.draw(G,
          pos,
          nodelist=cl.keys(),
          node_size = [d*3000 for d in cl.values()],
          node_color=cl.values(),
          font_size=8,
          cmap=plt.cm.Reds,
          )
```



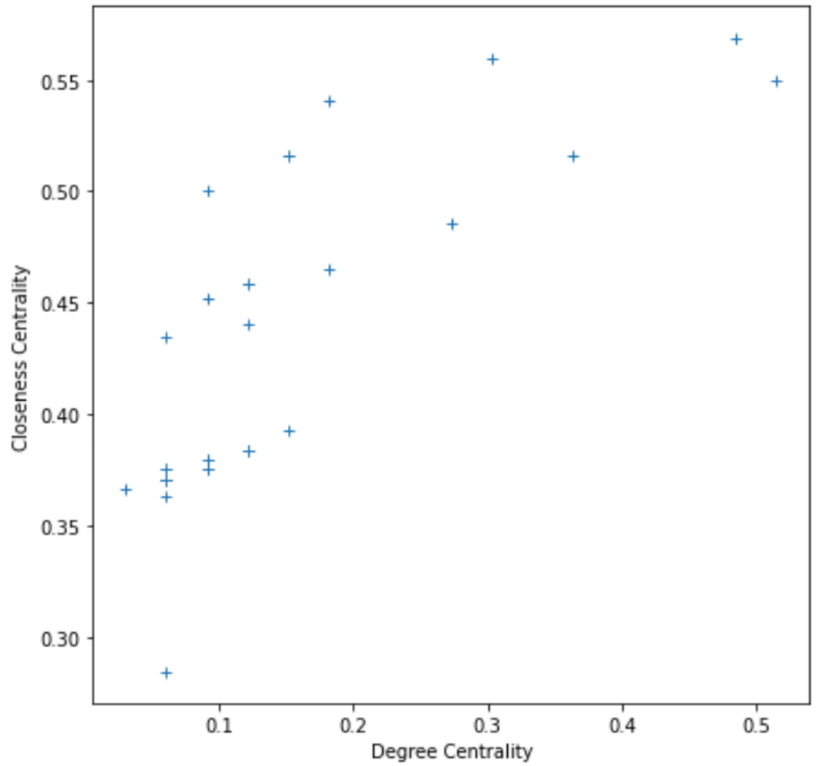




Scatter plot of the two properties

```
1  # Plot degree-closeness
2  xdata = list(dc.values())
3  ydata = list(cl.values())
4  plt.figure(1, figsize=(7,7))
5  plt.plot(xdata,ydata, '+')
6  plt.xlabel('Degree Centrality')
7  plt.ylabel('Closeness Centrality')
```

```
Text(0, 0.5, 'Closeness Centrality')
```

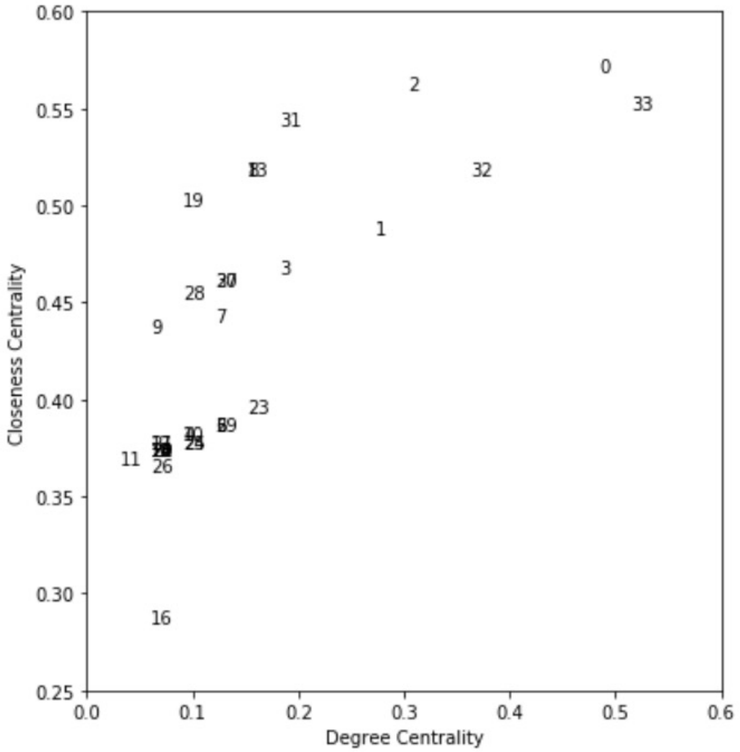


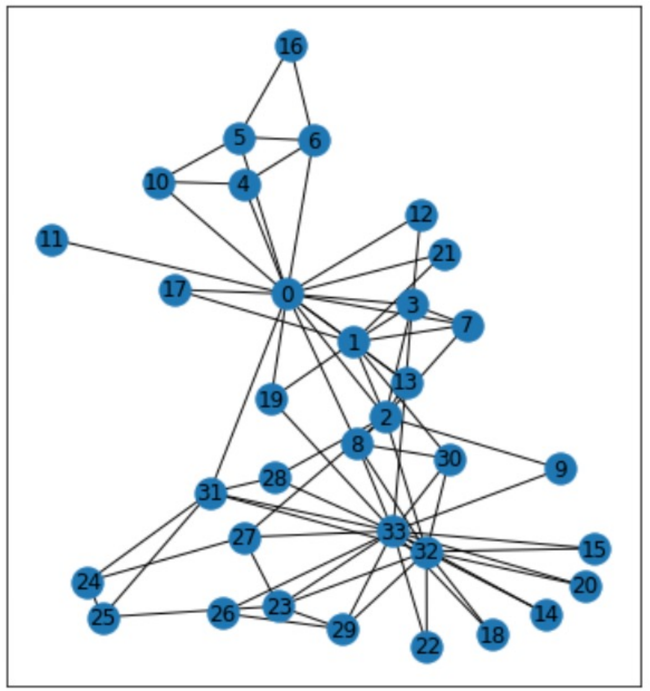
Annotated scatter plot

```
# Adding node ids in the scatter plot:
fig = plt.figure(1, figsize=(14,7))
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)

for v in range(len(dc)):
    ax1.text(x = xdata[v], y = ydata[v], s=str(v))
ax1.set_xlim(0, 0.6)
ax1.set_ylim(0.25, 0.6)
ax1.set_xlabel('Degree Centrality')
ax1.set_ylabel('Closeness Centrality')

ax2 = nx.draw_networkx(G, pos)
```



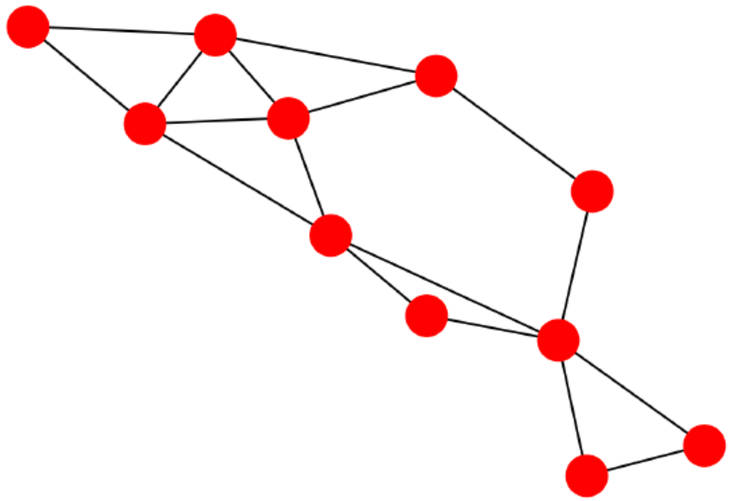


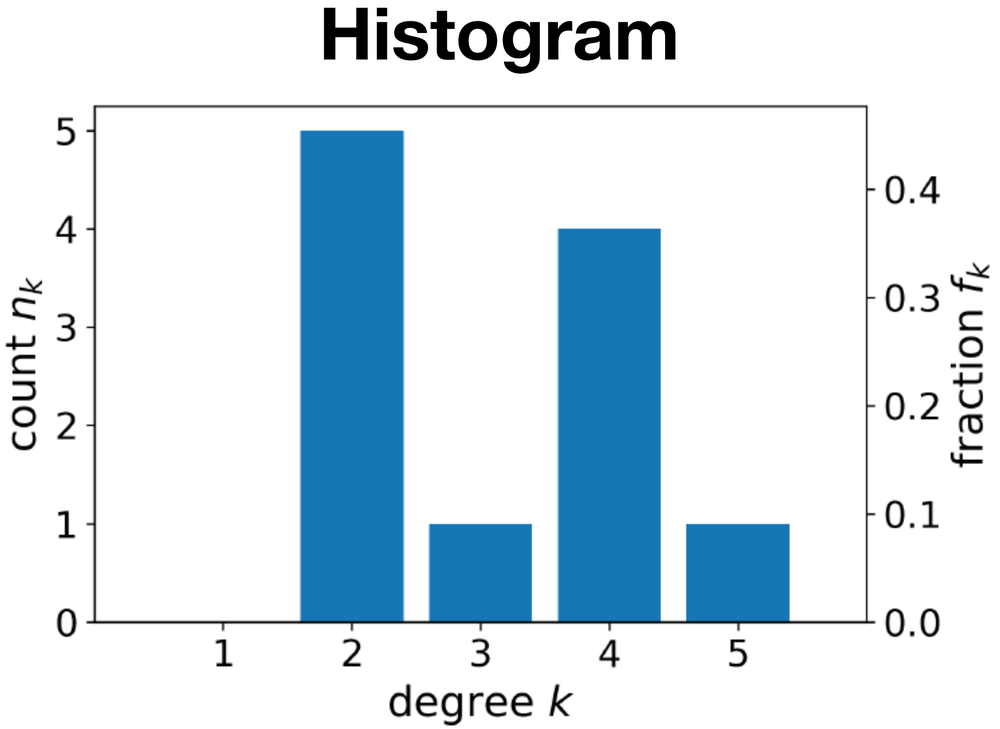
Centrality distributions

Statistical approach

Instead of focusing on individual nodes and links, we consider **classes** of nodes and links with similar properties

Centrality distributions





- When $N \rightarrow Large, f_k$ becomes the **probability** p_k of having degree k
- p_k versus k is the **probability distribution** of node degree

Logarithmic scale

- **Question**: how to plot a probability distribution if the variable spans a large range of values, from small to (very) large?

- **Answer**: use the **logarithmic** scale

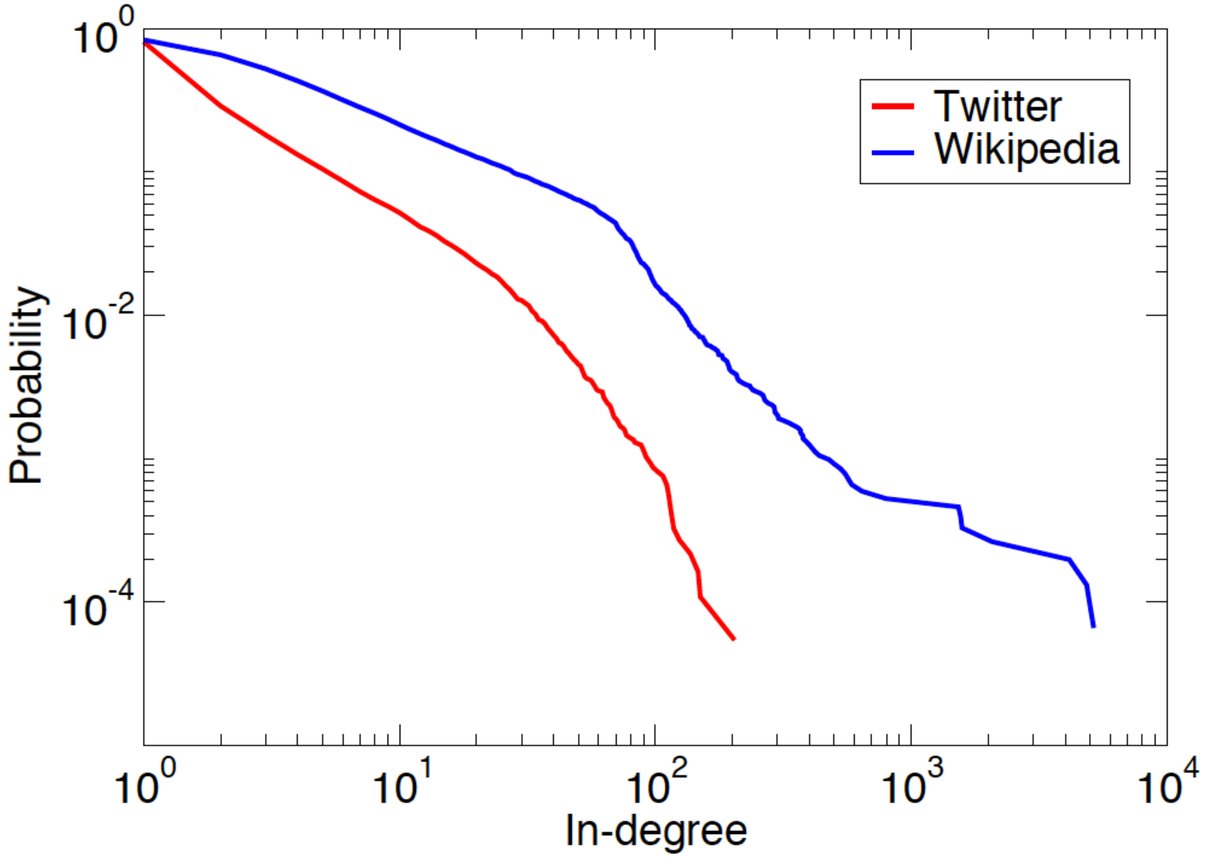
- **How to do it**: report the logarithms of the values on the x- and y-axes

$$\log_{10} 10 = 1$$

$$\log_{10} 1{,}000 = \log_{10} 10^3 = 3$$

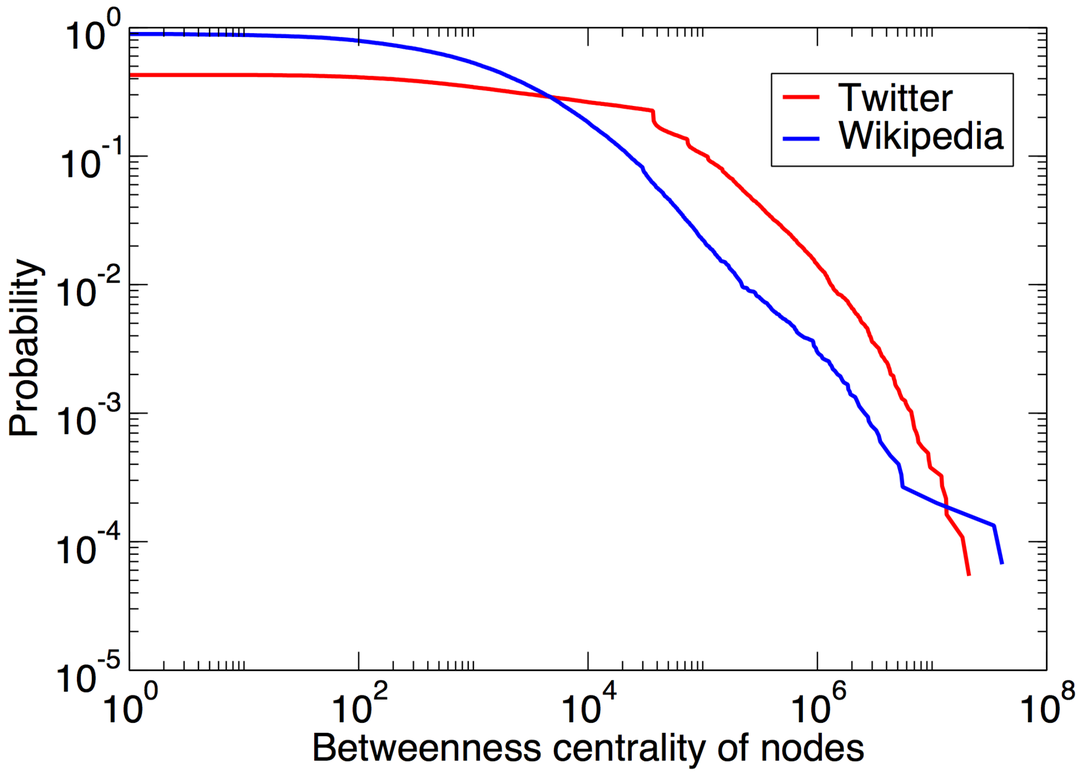
$$\log_{10} 1{,}000{,}000 = \log_{10} 10^6 = 6$$

Degree distributions



Heavy-tail distributions: the variable goes from small to large values

Betweenness distributions



Heavy-tail distribution: the variable goes from small to large values

Degree distributions

- The **heterogeneity parameter κ** says how broad the distribution is:

$$\kappa = \frac{\langle k^2 \rangle}{\langle k \rangle^2}$$

$$\langle k \rangle = \frac{\sum_i k_i}{N} = \frac{2L}{N}; \langle k^2 \rangle = \frac{\sum_i k_i^2}{N}$$

- If most degrees have the same value, say k_0 :

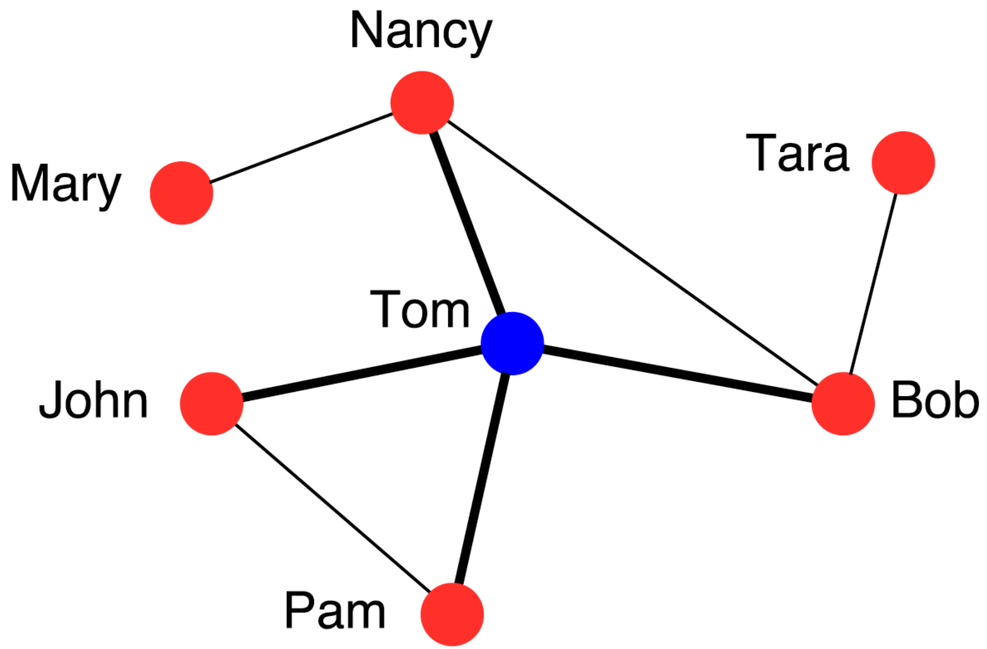
$$\langle k \rangle \approx k_0, \langle k^2 \rangle \approx k_0^2 \Longrightarrow \kappa \approx 1$$

- If the distribution is very heterogeneous: $\kappa \gg 1$

Most real world networks are heterogeneous

Network	Nodes (N)	Links (L)	Average degree ($\langle k \rangle$)	Maximum degree (k_{max})	Heterogeneity parameter (κ)
Facebook Northwestern Univ.	10,567	488,337	92.4	2,105	1.8
IMDB movies and stars	563,443	921,160	3.3	800	5.4
IMDB co-stars	252,999	1,015,187	8.0	456	4.6
Twitter US politics	18,470	48,365	2.6	204	8.3
Enron Email	36,692	367,662	10.0	1,383	14.0
Wikipedia math	15,220	194,103	12.8	5,171	38.2
Internet routers	190,914	607,610	6.4	1,071	6.0
US air transportation	546	2,781	10.2	153	5.3
World air transportation	3,179	18,617	11.7	246	5.5
Yeast protein interactions	1,870	2,277	2.4	56	2.7
C. elegans brain	297	2,345	7.9	134	2.7
Everglades ecological food web	69	916	13.3	63	2.2

Friendship paradox

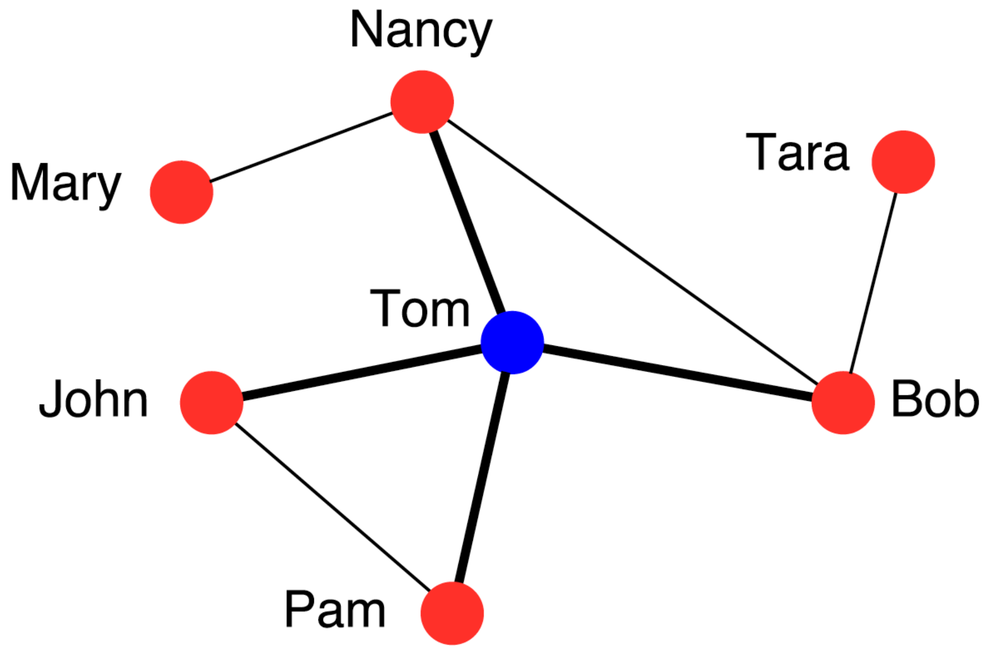


- Average degree of nodes = 16/7 = **2.29**
- Average degree of the neighbors of a node = **2.83 > 2.29**

M,N,B,Ta,To,P,J: 3+(1+4+3)/3+(1+3+4)/3+3+(3+3+2+2)/4+(4+2)/2+(4+2)/2 = 19.83/7 = 2.83

- *Our friends have more friends than we do, on average* (**friendship paradox**)

Friendship paradox



- By choosing *nodes at random*, Tom has **the same chance** to be picked as everybody else
- By choosing *links at random*, Tom has a **higher chance** to be picked than everybody else

Friendship paradox

- **Question:** Where does the friendship paradox come from?

- **Answer:**
  1. By averaging the degree of the nodes, we pick them at random
  2. By averaging the degree of the neighbors, we choose them by following links: nodes with degree *k* will be counted *k* times, which inflates the average

- The *more hubs, the stronger* the effect

Zhao *et al. Applied Network Science* (2021) 6:93
https://doi.org/10.1007/s41109-021-00431-1

Applied Network Science

RESEARCH Open Access

A network-based group testing strategy for colleges

Alex Zhao[1*], Kavin Kumaravel[1], Emanuele Massaro[2] and Marta Gonzalez[1]

*Correspondence:
axyzhao@berkeley.edu
[1] University of California, Berkeley, Berkeley, USA
Full list of author information is available at the end of the article

Abstract

Group testing has recently become a matter of vital importance for efficiently and rapidly identifying the spread of Covid-19. In particular, we focus on college towns due to their density, observability, and significance for school reopenings. We propose a novel group testing strategy which requires only local information about the underlying transmission network. By using cellphone data from over 190,000 agents, we construct a mobility network and run extensive data-driven simulations to evaluate the efficacy of four different testing strategies. Our results demonstrate that our group testing method is more effective than three other baseline strategies for reducing disease spread with fewer tests.

Keywords: Covid-19, Colleges, Testing

https://appliednetsci.springeropen.com/articles/10.1007/s41109-021-00431-1

Published class project from CE88 Spring 2020

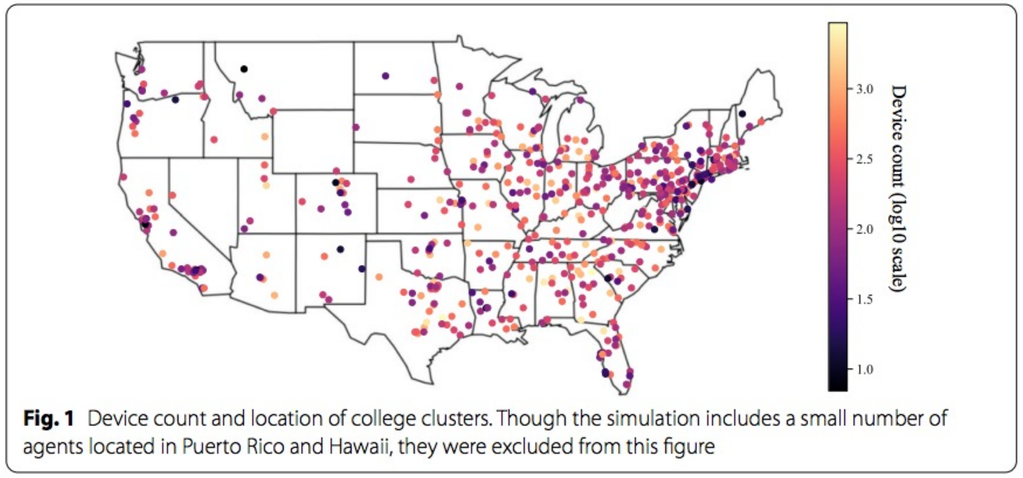


Fig. 1 Device count and location of college clusters. Though the simulation includes a small number of agents located in Puerto Rico and Hawaii, they were excluded from this figure

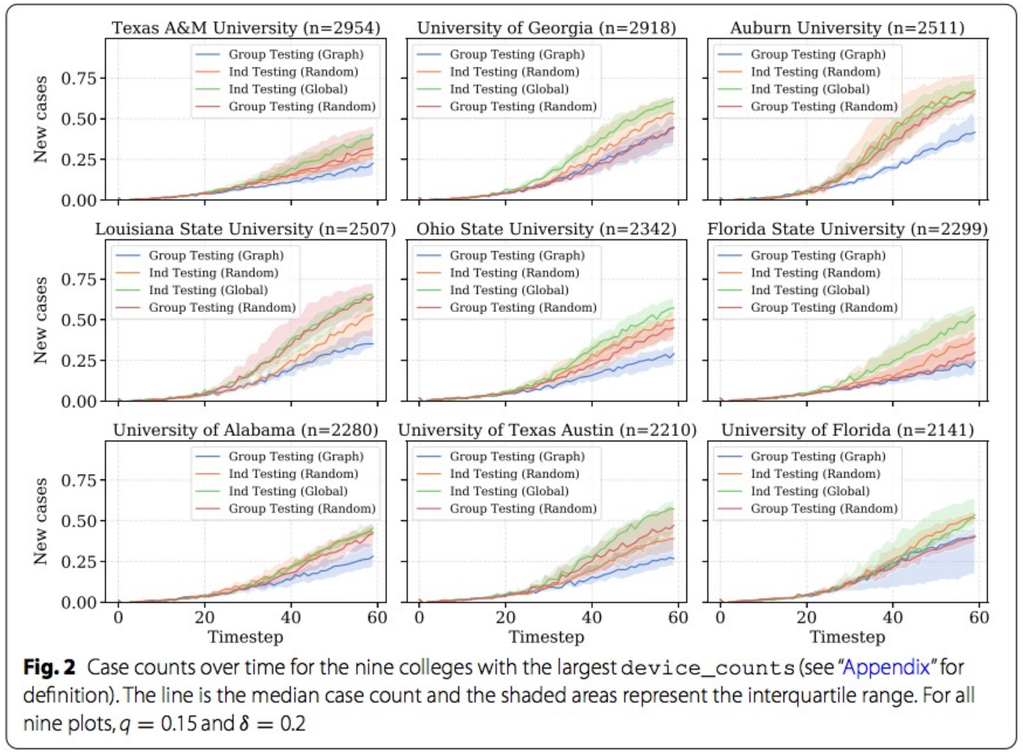
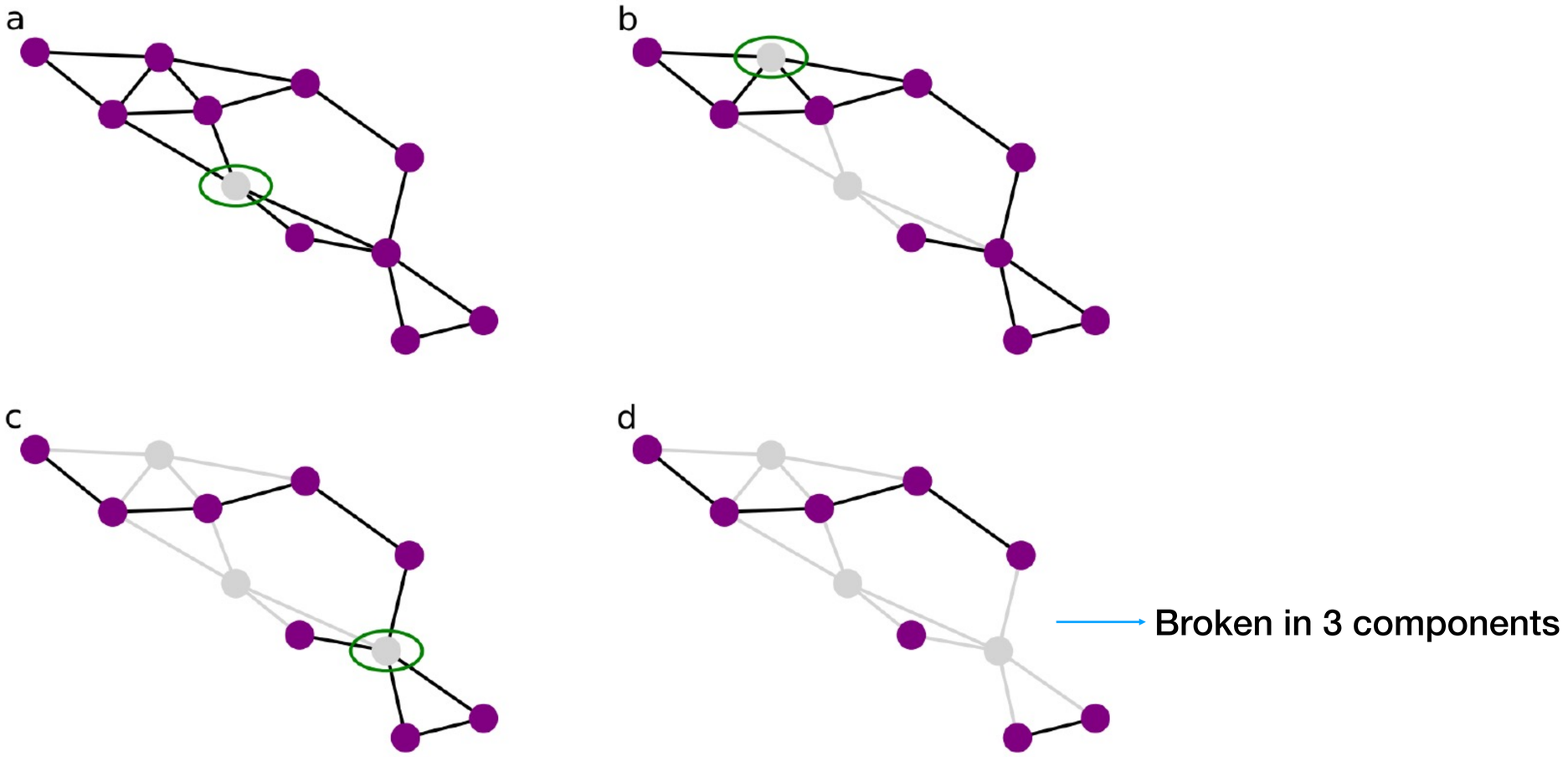


Fig. 2 Case counts over time for the nine colleges with the largest `device_counts` (see "Appendix" for definition). The line is the median case count and the shaded areas represent the interquartile range. For all nine plots, $q = 0.15$ and $\delta = 0.2$

Method:

- Built a model of contact network based on the trips.
- SafeGraph Social Distancing Metrics record the movement of anonymized mobile devices on the census block group (CBG) level.
- Simulated 190,000 agents.
- We grouped the CBGs with an undergraduate population greater than 40% by agglomerative clustering with a distance threshold of ten miles. The resulting 533 clusters can be seen in Fig. 1

Robustness



Robustness

- A system is **robust** if the failure of some of its components does not affect its function
- **Question:** how can we define the robustness of a network?
- **Answer:** we remove nodes and/or links and see what happens to its structure
- **Key point:** *connectedness*
- If the Internet were not connected, it would be impossible to transmit signals (e.g., emails) between routers in different components

Robustness

- **Robustness test:** checking how the connectedness of the network is affected as more and more nodes are removed
- **How to do it:** plot the relative size S of the largest connected component as a function of the fraction of removed nodes
- We suppose that the network is initially connected: there is only one component and $S = 1$
- As more and more nodes (and their links) are removed, the network is progressively broken up into components and S goes down

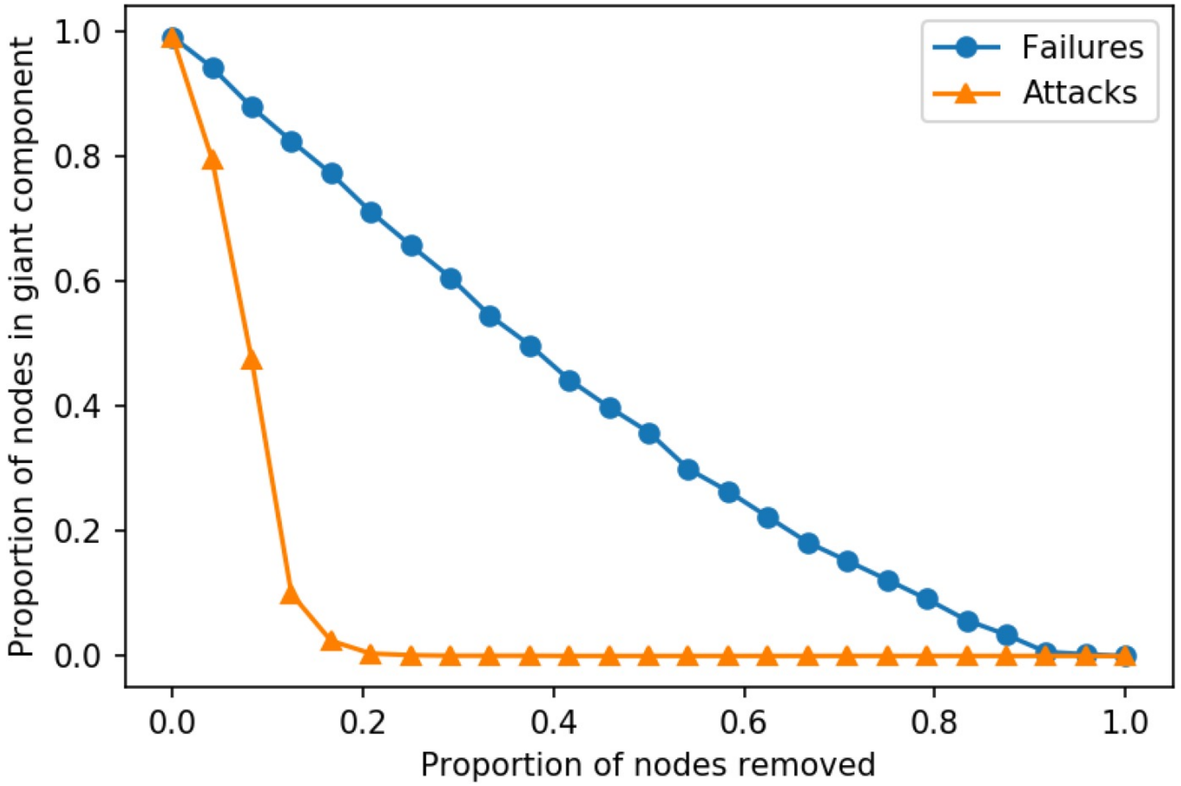
Robustness

- **Two strategies:**
  1. **Random failures:** nodes break down randomly, so they are all chosen with the **same probability**
  2. **Attacks:** hubs are deliberately targeted — the larger the **degree**, the higher the probability of removing the node

- In the first approach, we remove a fraction *f* of nodes, chosen at random

- In the second approach, we remove the fraction *f* of nodes with largest degree, from the one with largest degree downwards

Robustness



Conclusion: real networks are robust against random failures but fragile against targeted attacks!

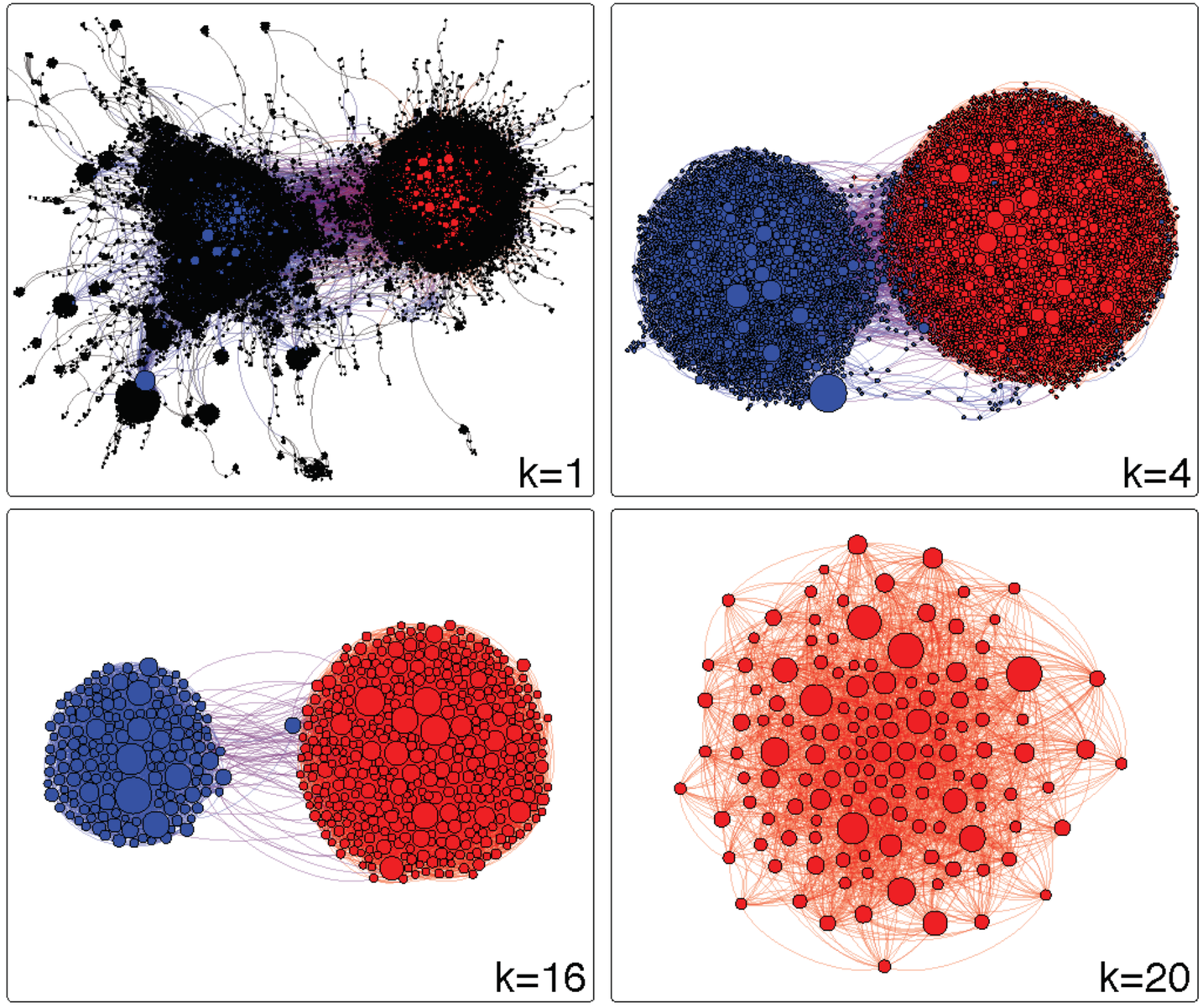
Core decomposition

- **Core:** dense part of the network, with high-degree nodes

- **Core decomposition:** procedure to identify denser and denser cores, by removing nodes of progressively higher degree. If we remove all nodes with degree $k-1$ or lower, the remaining portion of the network is called $\boldsymbol{k}$**-core**

- $\boldsymbol{k}$**-core decomposition procedure:** start with k=0
  1. Recursively remove all nodes with degree k, until none are left
  2. The set of removed nodes is the $\boldsymbol{k}$**-th shell**, while the remaining ones form the $(\boldsymbol{k+1})$**-core**
  3. If there is no node left, terminate. Otherwise, increment k by one and repeat from step 1

Core decomposition



Core decomposition

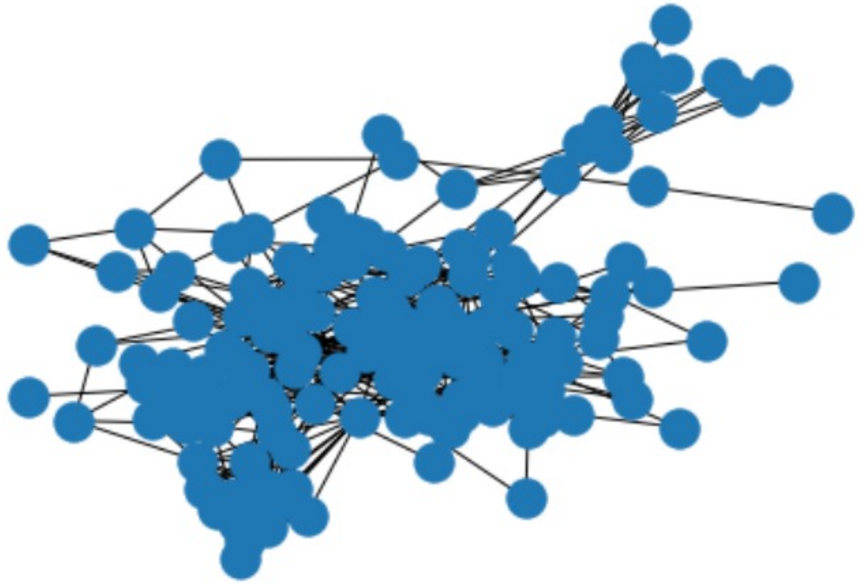
Core decomposition helps to visualize large networks, by pruning low-degree nodes and showing only the densest parts

```
nx.core_number(G)    # return dict with core number of each node
nx.k_shell(G,k)      # subnetwork induced by nodes in k-shell
nx.k_core(G,k)       # subnetwork induced by nodes in k-core
nx.k_core(G)         # innermost (max-degree) core subnetwork
```

Lecture18_CentralityTutorial.ipynb

```
]:  1  G = nx.read_edgelist('ia-enron-only.edges', nodetype=int)
    2  print(nx.info(G))
    3  nx.draw(G)
```

```
Graph with 143 nodes and 623 edges
```



```
# Unweighted, undirected
17 1
72 1
3 2
19 2
20 2
27 2
29 2
31 2
48 2
65 2
67 2
69 2
71 2
132 2
136 2
ia-enron-only.edges
```

Caveat: by default, `nx.read_edgelist` assumes node names are strings

Edge lists are a simple, plain text format for storing graphs. Since this simple file format doesn't contain information about data types, all node names are assumed to be strings by default. When the node names are given by integers, as they are in this example, we should specify the `nodetype=int` keyword argument to avoid confusion with the node names.

Finding the node with max degree

We can apply the `max` function to get the maximum node according to some criterion. In our case, we want to compare the nodes by their degree:

In [6]:
```
highest_degree_node = max(G.nodes, key=G.degree)
highest_degree_node
```

Out[6]: 105

In [7]:
```
G.degree(highest_degree_node)
```

Out[7]: 42

So we know that node `105` has 42 edges connected to it. Let's look at that calculation a little closer:

```
max(G.nodes, key=G.degree)
```

2. Centrality distributions

We've found the most central node in the network, but often we want to summarize information about the centrality of all nodes. For example, in addition to the maxmimum degree, we often want the mean and median degree in a network.

The first step in summarizing this information is to get a sequence of all of the centrality values in the graph. We don't care about the node names anymore, we just want a sequence of numbers. We can start with the example of degree:

:
```
1 degree_sequence = [G.degree(n) for n in G.nodes]
```

:
```
1 degree_sequence
```

:
```
[30,
 2,
 17,
 12,
 13,
 16,
 17,
 10,
 11,
 18,
 20,
 15,
```

```
|:
1 import statistics
2
3 print('Mean degree:', statistics.mean(degree_sequence))
4 print('Median degree:', statistics.median(degree_sequence))
```

```
Mean degree: 8.713286713286713
Median degree: 8
```

For other measures of centrality, which are calculated for the whole network at once and return a dict, we can use the dict's `.values()`.

```
|:
1 betweenness = nx.centrality.betweenness_centrality(G)
2 betweenness_sequence = list(betweenness.values())
3
4 print('Mean betweenness:', statistics.mean(betweenness_sequence))
5 print('Median betweenness:', statistics.median(betweenness_sequence))
```

```
Mean betweenness: 0.013950388838012454
Median betweenness: 0.0037492834763570323
```

```
betweenness = nx.centrality.betweenness_centrality(G)
highest_betweenness_node = max(G.nodes, key=betweenness.get)
highest_betweenness_node
```

```
105
```

```
betweenness[highest_betweenness_node]
```

```
0.19400604344556296
```

Random failure

Whenever we're going to engage in a distructive process, we want to make a copy of the network graph to attack so that we can easily get back to the original state:

```
:  1  C = G.copy()
```

The full simulation is going to work like this, beginning from a fresh copy of the network:

1. Measure size of the network largest component as compared to the original network size
2. Select M nodes at random and remove them
3. Repeat until there are less than M nodes

```
number_of_steps = 25
M = G.number_of_nodes() // number_of_steps
M
```

(// means floor division, 143//25 =5)

```
num_nodes_removed = range(0, G.number_of_nodes(), M)
```

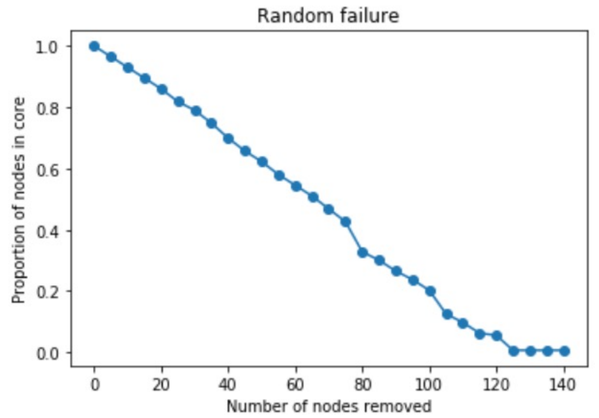
The loop is fairly simple. At each step, we need to record the fraction of remaining nodes in the core.

```
N = G.number_of_nodes()
C = G.copy()
random_attack_core_proportions = []
for nodes_removed in num_nodes_removed:
    # Measure the relative size of the network core
    core = next(nx.connected_components(C))
    core_proportion = len(core) / N
    random_attack_core_proportions.append(core_proportion)

    # If there are more than M nodes, select M nodes at random and remove them
    if C.number_of_nodes() > M:
        nodes_to_remove = random.sample(list(C.nodes), M)
        C.remove_nodes_from(nodes_to_remove)
```

```
plt.title('Random failure')
plt.xlabel('Number of nodes removed')
plt.ylabel('Proportion of nodes in core')
plt.plot(num_nodes_removed, random_attack_core_proportions, marker='o')
```

```
[<matplotlib.lines.Line2D at 0x7f7d7ead2860>]
```



At every step C has 5 less nodes

Targeted attack

Simulating a targeted attack is similar, except that instead of choosing randomly, we will select the M most central nodes at each step. To accomplish this we want something like the `max` function used earlier to get the most central node, but able to get the top M nodes. We can use Python's `sorted` function in a similar way to `max` to first sort the nodes by centrality in descending, or reverse, order. Once sorted by degree, we take the first M nodes in the list:

```
]:
1  nodes_sorted_by_degree = sorted(G.nodes, key=G.degree, reverse=True)
2  top_degree_nodes = nodes_sorted_by_degree[:M]
3  top_degree_nodes
```

```
]: [105, 17, 95, 48, 132]
```

With that in mind, the loop looks similar. Let's put all the pieces together here:

```
]:
1  N = G.number_of_nodes()
2  number_of_steps = 25
3  M = N // number_of_steps
4
5  num_nodes_removed = range(0, N, M)
6  C = G.copy()
7  targeted_attack_core_proportions = []
8  for nodes_removed in num_nodes_removed:
9      # Measure the relative size of the network core
10     core = next(nx.connected_components(C))
11     core_proportion = len(core) / N
12     targeted_attack_core_proportions.append(core_proportion)
13
14     # If there are more than M nodes, select top M nodes and remove them
15     if C.number_of_nodes() > M:
16         nodes_sorted_by_degree = sorted(C.nodes, key=C.degree, reverse=True)
17         nodes_to_remove = nodes_sorted_by_degree[:M]
18         C.remove_nodes_from(nodes_to_remove)
```

(143//25 =5)

[0,5,10,15,20,…,140] range from 0 to 143 with 5 increments

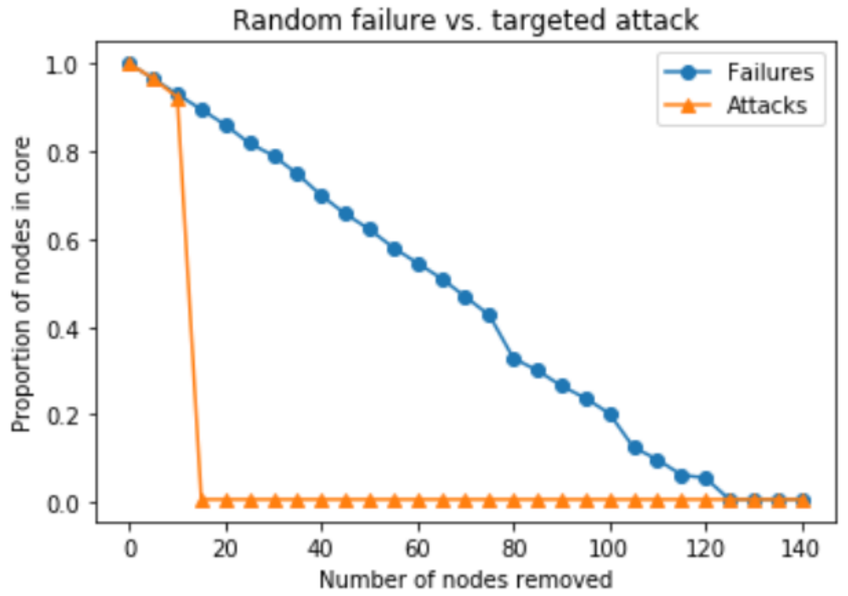
At every step C has 5 less nodes

As we can see, the effect is dramatic. The network is totally disconnected after removing relatively few central nodes in the graph.

Pyplot can plot these curves on the same plot for additional effect:

```
[5]:
1  plt.title('Random failure vs. targeted attack')
2  plt.xlabel('Number of nodes removed')
3  plt.ylabel('Proportion of nodes in core')
4  plt.plot(num_nodes_removed, random_attack_core_proportions, marker='o', label='Failures')
5  plt.plot(num_nodes_removed, targeted_attack_core_proportions, marker='^', label='Attacks')
6  plt.legend()
```

```
[5]: <matplotlib.legend.Legend at 0x7f7d7e81ff60>
```



The network is not robust under targeted attacks