COMPLEX & SOCIAL NETWORK ANALYSIS

Enrico Franchi (efranchi@ce.unipr.it)

https://www.dropbox.com/s/43f7c84iolxfvg2/csnap.pdf

A **social network** is a structure composed by actors and their relationships

Actor: person, organization, role ...

Relationship: friendship, knowledge...

A **social networking system** is system allowing users to:

- construct a profile which represents them in the system;
- create a list of users with whom they share a connection
- navigate their list of connections and that of their friends

(Boyd, 2008)

So, what is a **complex network**?

A **complex network** is a network with non-trivial topological features—features that do not occur in simple networks such as lattices or random graphs but often occur in real graphs. (Wikipedia). Foggy.

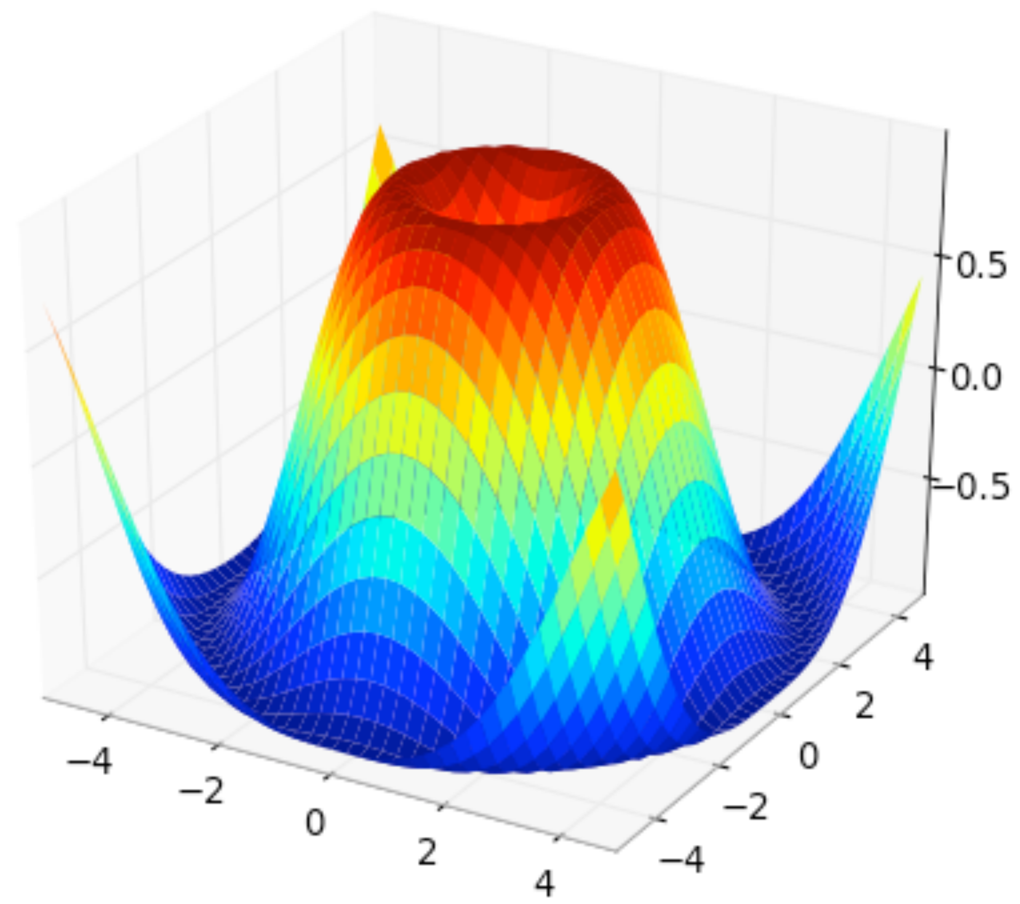
COMPLEX NETWORKS

A **complex network** is a network with non-trivial topological features—features that do not occur in simple networks such as lattices or random graphs but often occur in real graphs.

- Non-trivial topological features (what are topological features?)
- Simple networks: lattices, regular or random graphs
- Real graphs
- Are social networks complex networks?

TOOLS

- NumPy
- SciPy.org
- Matplotlib
- IPython
- NetworkX



BASIC NOTATION

Network

$$G = (V, E) \quad E \subset V^2$$

$$\{(x,x) | x \in V\} \cap E = \varnothing$$

Adjacency Matrix

$$\mathbf{A}_{ij} = \begin{cases} 1 \text{ if (i,j) } \in E \\ 0 \text{ otherwise} \end{cases}$$

Directed Network

$$k_i^{\text{in}} = \sum_j \mathbf{A}_{ji}$$

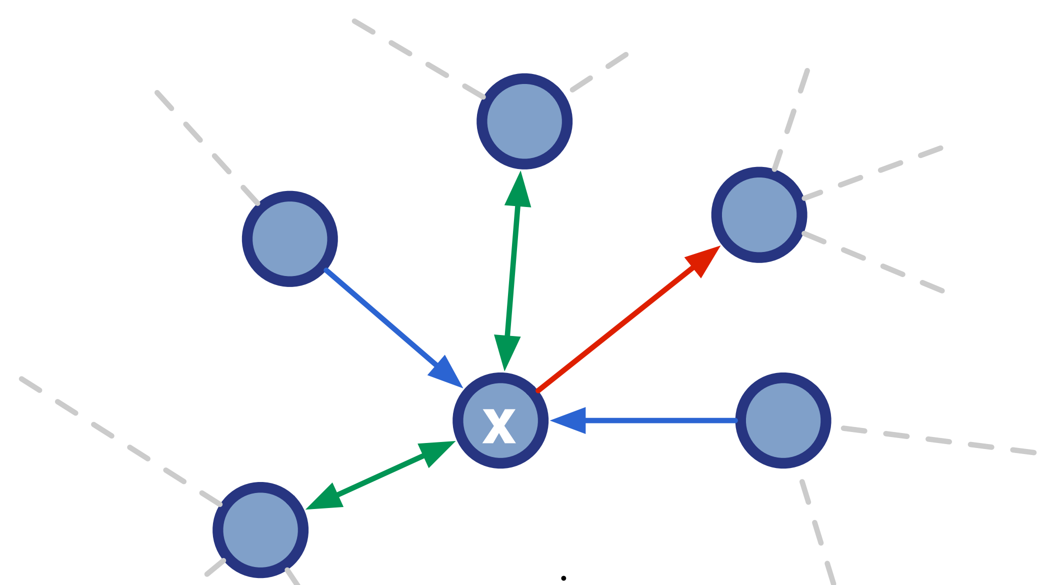
$$k_i^{\text{out}} = \sum_j \mathbf{A}_{ij}$$

$$k_i = k_i^{\text{in}} + k_i^{\text{out}}$$

Undirected Network

A symmetric

$$k_i = \sum_j \mathbf{A}_{ji} = \sum_j \mathbf{A}_{ij}$$



$$k_x^{\text{in}} = 4$$

$$k_x^{\text{out}} = 3$$

BASIC NOTATION

Network

$$G=(V,E) \quad E\subset V^2$$

$$\{(x,x) \mid x\in V\}\cap E=\varnothing$$

Adjacency Matrix

$$\mathbf{A}_{ij}=\begin{cases}1 \text{ if (i,j) } \in E\\ 0 \text{ otherwise}\end{cases}$$

Assume the network connected!

Directed Network

$$k_i^{\text{in}}=\sum_j \mathbf{A}_{ji}$$

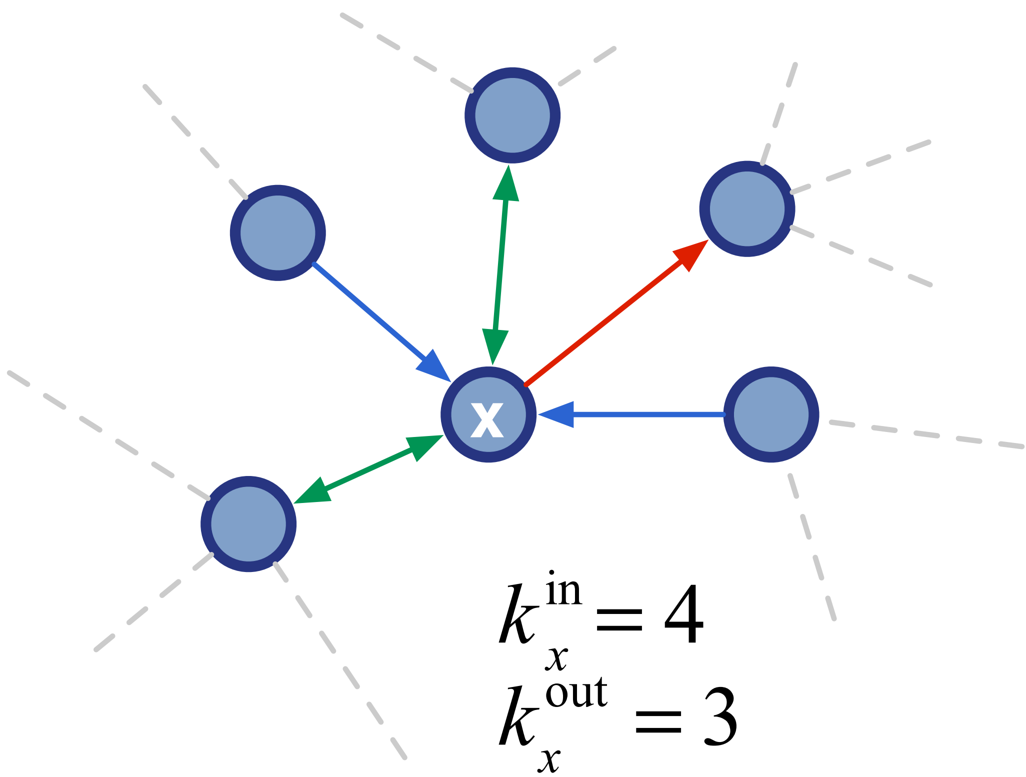
$$k_i^{\text{out}}=\sum_j \mathbf{A}_{ij}$$

$$k_i=k_i^{\text{in}}+k_i^{\text{out}}$$

Undirected Network

$$A \quad \text{symmetric}$$

$$k_i=\sum_j \mathbf{A}_{ji}=\sum_j \mathbf{A}_{ij}$$



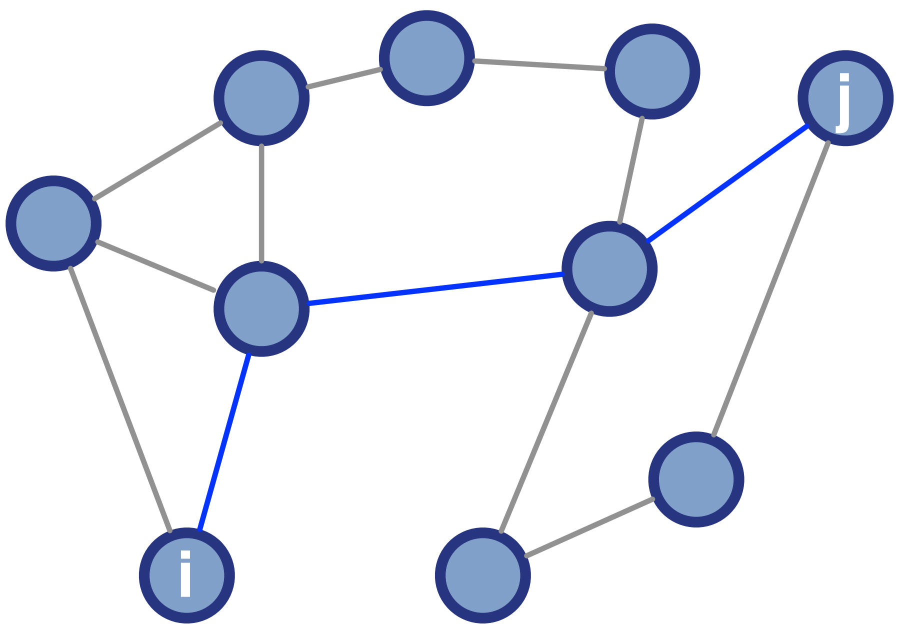
$$k_x^{\text{in}}=4$$

$$k_x^{\text{out}}=3$$

Path $p = \langle v_0, \ldots, v_k \rangle$ $(v_{i-1}, v_i) \in E$

$$v_0 \overset{p}{\rightsquigarrow} v_k$$

Path Length: $\mathrm{length}(p)$ Set of paths from *i* to *j*: $\mathrm{Paths}(i, j)$



Shortest path length: $L(i,j) = \min\left(\left\{\mathrm{length}(p) \middle| p \in \mathrm{Paths}(i,j)\right\}\right)$

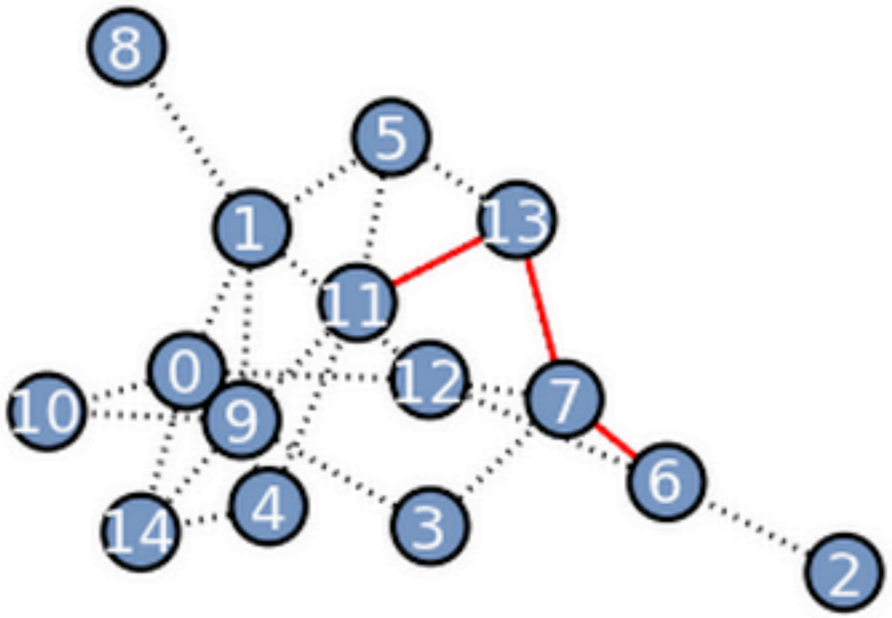
Shortest/Geodesic path: $i \overset{p}{\rightarrow} j = \arg\min\left(\{\mathrm{length}(r)\,|r \in \mathrm{Paths}(i,j)\,\}\right)$

```python
import networkx as nx
```

```python
G = nx.erdos_renyi_graph(15, 0.2)
```

```python
p = nx.shortest_path(G, 6, 11)
```

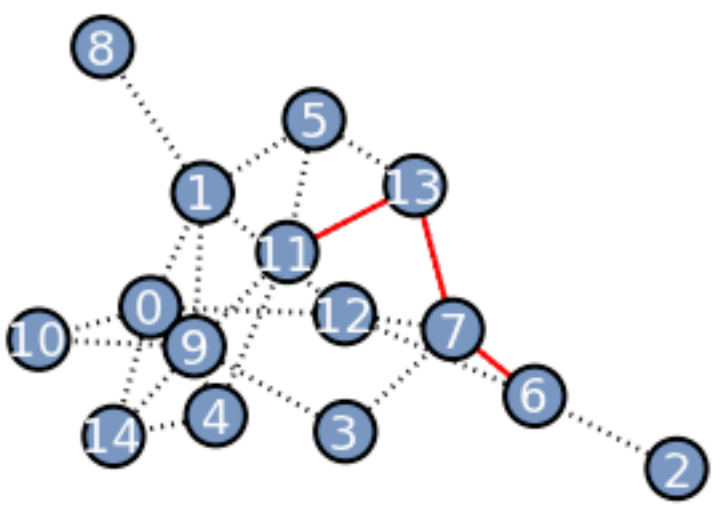
```python
pos = nx.spring_layout(G); # positions for all nodes
nx.draw_networkx_nodes(G, pos, node_color='#6E8EBD', node_size=500, linewidths=2.0);
nx.draw_networkx_labels(G, pos, font_size=18, font_color='w');
nx.draw_networkx_edges(G, pos, width=2.0, style='dotted');
nx.draw_networkx_edges(G, pos, edgelist=zip(p, p[1:]), width=2.0, edge_color='r');
plt.axis('off'); None
```



Average geodesic distance $$\ell(i) = (n-1)^{-1} \sum_{k \in V \setminus \{i\}} L(i,j)$$

Average shortest path length $$\ell = n^{-1} \sum_{i \in V} \ell(i)$$

Characteristic path length $$\mathrm{CPL} = \mathrm{median}\left(\left\{ \ell(i) \middle| i \in V \right\}\right)$$



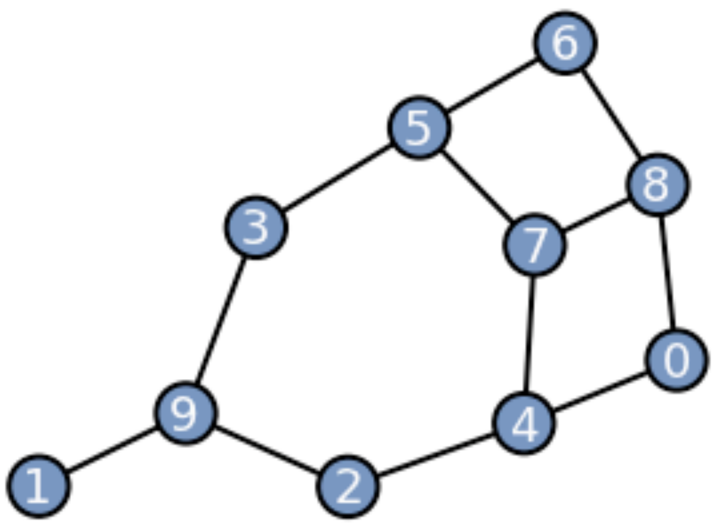

```
nx.average_shortest_path_length(G)
```

```
2.1904761904761907
```

```python
import networkx as nx
```

```python
G = nx.read_dot('small.dot');
```

```python
pos = nx.spring_layout(G)
labels = dict(zip(G.nodes(), range(len(G))))
nx.draw_networkx_nodes(G, pos, node_color='#6E8EBD', node_size=500, linewidths=2.0)
nx.draw_networkx_labels(G, pos, labels=labels, font_size=18, font_color='w')
nx.draw_networkx_edges(G, pos, width=2.0)
plt.axis('off'); None
```



$\left[\mathbf{A}^k\right]_{ij}$ number of paths of length k from i to j

```python
A = nx.to_numpy_matrix(G)
```

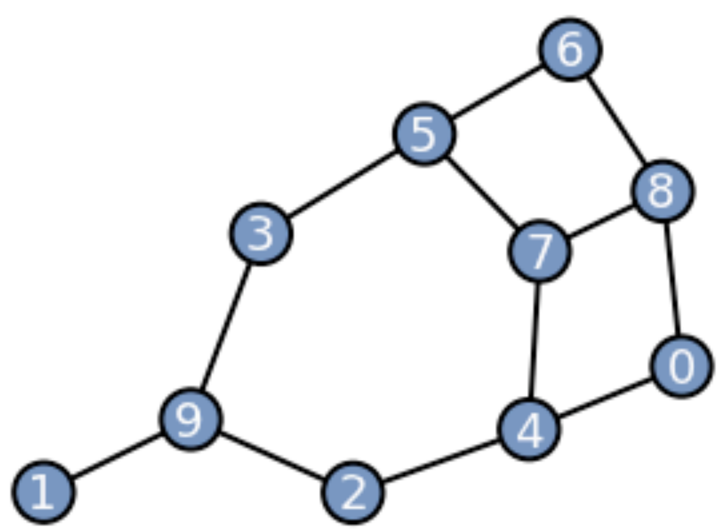
```python
A * A
```

```
matrix([[ 2.,  0.,  1.,  0.,  0.,  0.,  1.,  2.,  0.,  0.],
        [ 0.,  1.,  1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 1.,  1.,  2.,  1.,  0.,  0.,  0.,  1.,  0.,  0.],
        [ 0.,  1.,  1.,  2.,  0.,  0.,  1.,  1.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  3.,  1.,  0.,  0.,  2.,  1.],
        [ 0.,  0.,  0.,  0.,  1.,  3.,  0.,  0.,  2.,  1.],
        [ 1.,  0.,  0.,  1.,  0.,  0.,  2.,  2.,  0.,  0.],
        [ 2.,  0.,  1.,  1.,  0.,  0.,  2.,  3.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  2.,  2.,  0.,  0.,  3.,  0.],
        [ 0.,  0.,  0.,  0.,  1.,  1.,  0.,  0.,  0.,  3.]])
```

```python
import networkx as nx
```

```python
G = nx.read_dot('small.dot');
```

```python
pos = nx.spring_layout(G)
labels = dict(zip(G.nodes(), range(len(G))))
nx.draw_networkx_nodes(G, pos, node_color='#6E8EBD', node_size=500, linewidths=2.0)
nx.draw_networkx_labels(G, pos, labels=labels, font_size=18, font_color='w')
nx.draw_networkx_edges(G, pos, width=2.0)
plt.axis('off'); None
```



$\left[\mathbf{A}^k\right]_{ij}$ number of paths of length *k* from *i* to *j*

```python
A = nx.to_numpy_matrix(G)
```

```python
A * A
```

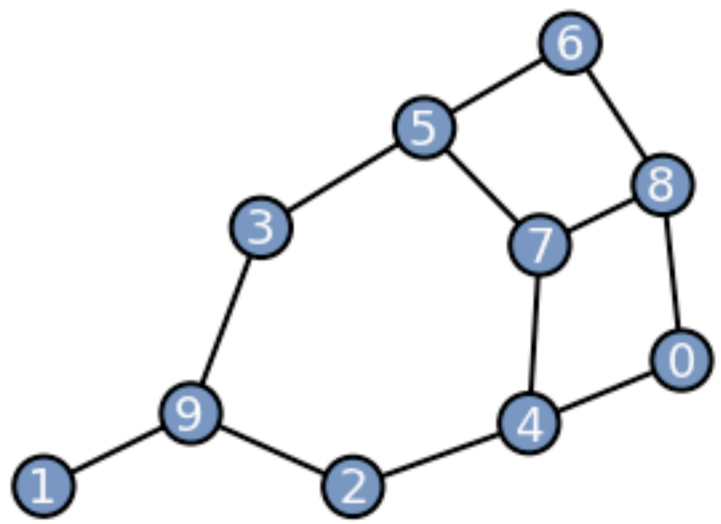
```
matrix([[ 2.,  0.,  1.,  0.,  0.,  0.,  1.,  2.,  0.,  0.],
        [ 0.,  1.,  1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 1.,  1.,  2.,  1.,  0.,  0.,  0.,  1.,  0.,  0.],
        [ 0.,  1.,  1.,  2.,  0.,  0.,  1.,  1.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  3.,  1.,  0.,  0.,  2.,  1.],
        [ 0.,  0.,  0.,  0.,  1.,  3.,  0.,  0.,  2.,  1.],
        [ 1.,  0.,  0.,  1.,  0.,  0.,  2.,  2.,  0.,  0.],
        [ 2.,  0.,  1.,  1.,  0.,  0.,  2.,  3.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  2.,  2.,  0.,  0.,  3.,  0.],
        [ 0.,  0.,  0.,  0.,  1.,  1.,  0.,  0.,  0.,  3.]])
```

number of paths!

```python
import networkx as nx
```

```python
G = nx.read_dot('small.dot');
```

```python
pos = nx.spring_layout(G)
labels = dict(zip(G.nodes(), range(len(G))))
nx.draw_networkx_nodes(G, pos, node_color='#6E8EBD', node_size=500, linewidths=2.0)
nx.draw_networkx_labels(G, pos, labels=labels, font_size=18, font_color='w')
nx.draw_networkx_edges(G, pos, width=2.0)
plt.axis('off'); None
```



$\left[\mathbf{A}^k\right]_{ij}$ number of paths of length k from i to j

```python
A = nx.to_numpy_matrix(G)
```

```python
A * A
```

```
matrix([[ 2.,  0.,  1.,  0.,  0.,  0.,  1.,  2.,  0.,  0.],
        [ 0.,  1.,  1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 1.,  1.,  2.,  1.,  0.,  0.,  0.,  1.,  0.,  0.],
        [ 0.,  1.,  1.,  2.,  0.,  0.,  1.,  1.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  3.,  1.,  0.,  0.,  2.,  1.],
        [ 0.,  0.,  0.,  0.,  1.,  3.,  0.,  0.,  2.,  1.],
        [ 1.,  0.,  0.,  1.,  0.,  0.,  2.,  2.,  0.,  0.],
        [ 2.,  0.,  1.,  1.,  0.,  0.,  2.,  3.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  2.,  2.,  0.,  0.,  3.,  0.],
        [ 0.,  0.,  0.,  0.,  1.,  1.,  0.,  0.,  0.,  3.]])
```

number of paths!

degree

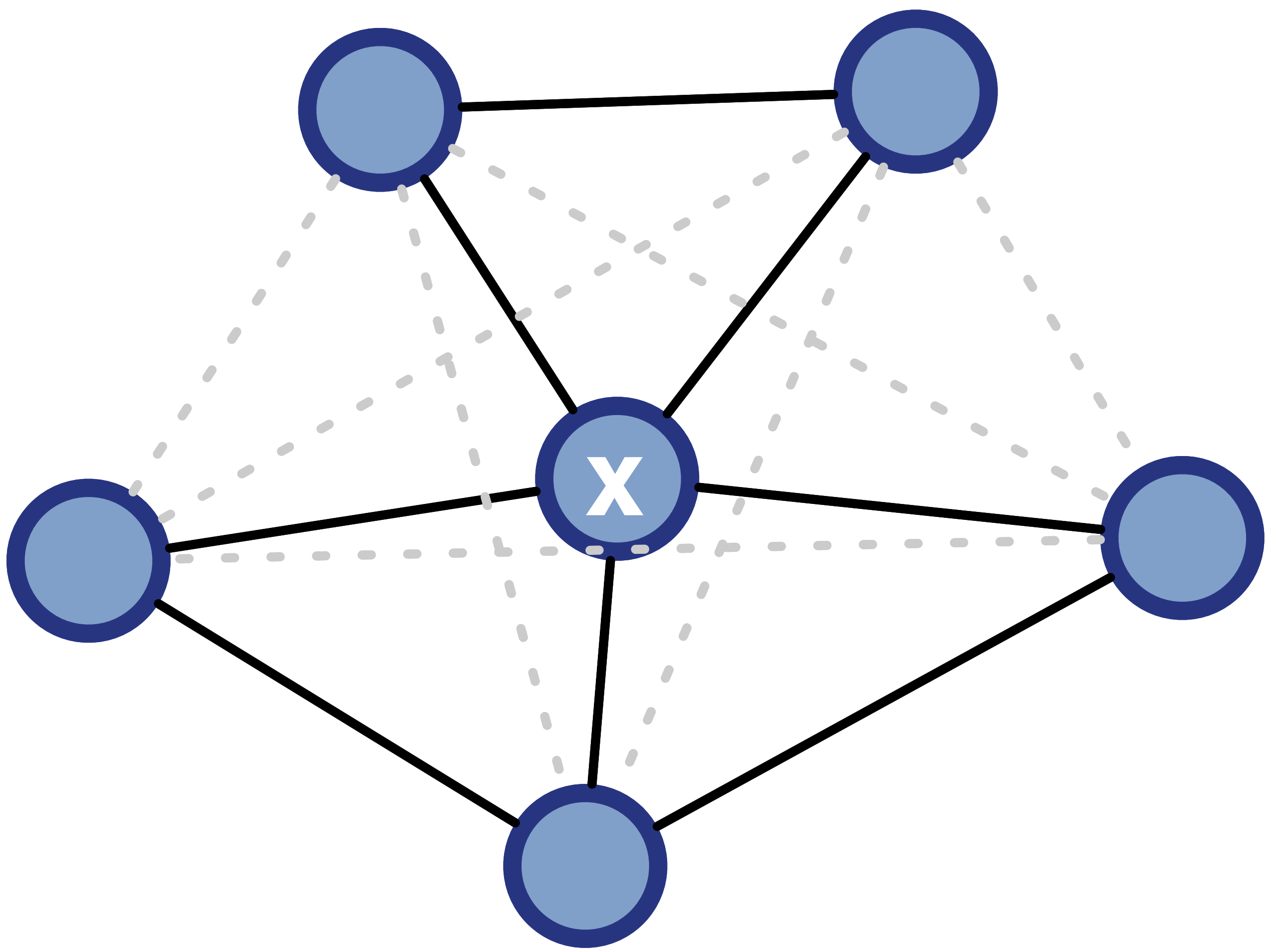
CLUSTERING COEFFICIENT

Local Clustering Coefficient $$C_i = \binom{k_i}{2}^{-1} T(i)$$

T(*i*): # distinct triangles with *i* as vertex

Clustering Coefficient

$$C = \frac{1}{n}\sum_{i \in V} C_i$$



- Measure of **transitivity**
- High CC → "resilient" network
- Counting triangles

$$\Delta(G) = \sum_{i \in V} N(i) = \frac{1}{6}\mathrm{trace}\left(\mathbf{A}^3\right)$$

CLUSTERING COEFFICIENT

Local Clustering Coefficient $$C_i = \binom{k_i}{2}^{-1} T(i)$$

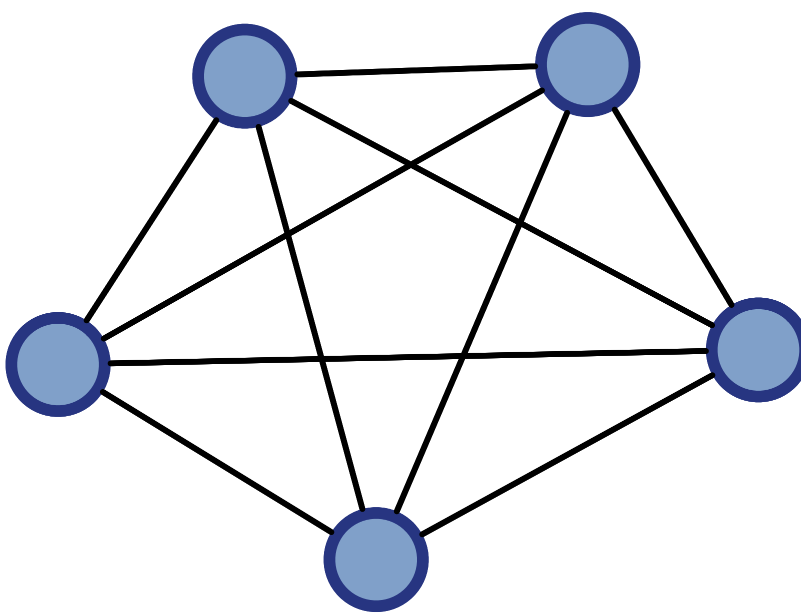
$T(i)$: # distinct triangles with i as vertex

Clustering Coefficient

$$C = \frac{1}{n}\sum_{i \in V} C_i$$

- Measure of **transitivity**
- High CC → "resilient" network
- Counting triangles

$$\Delta(G) = \sum_{i \in V} N(i) = \frac{1}{6}\operatorname{trace}\left(\mathbf{A}^3\right)$$



$$k_x = 5$$

$$\binom{k_x}{2}^{-1} = 10$$

CLUSTERING COEFFICIENT

Local Clustering Coefficient $$C_i = \binom{k_i}{2}^{-1} T(i)$$

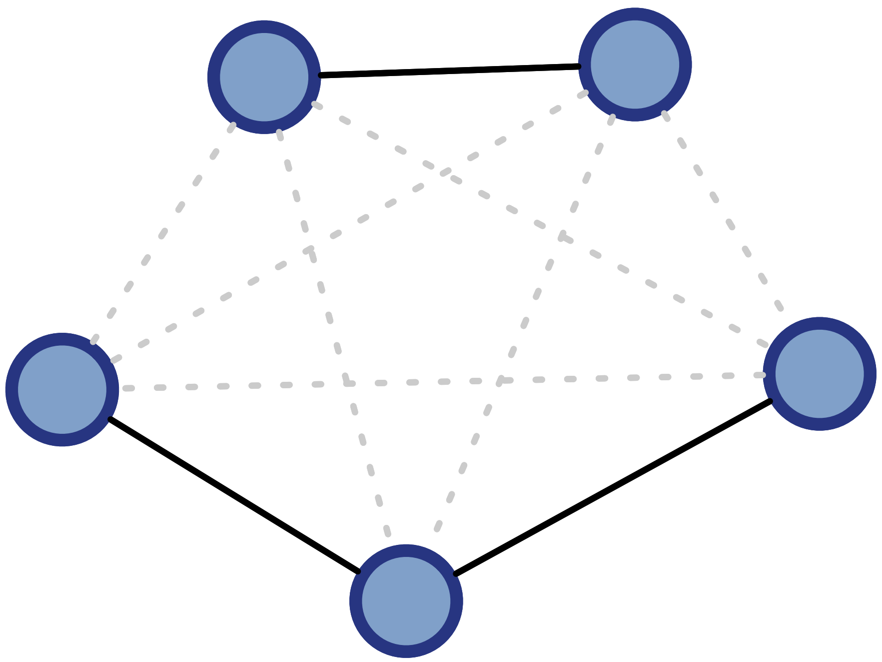
T(*i*): # distinct triangles with *i* as vertex

Clustering Coefficient

$$C = \frac{1}{n} \sum_{i \in V} C_i$$

- Measure of **transitivity**
- High CC → “resilient” network
- Counting triangles

$$\Delta(G) = \sum_{i \in V} N(i) = \frac{1}{6} \operatorname{trace}\left(\mathbf{A}^3\right)$$



$$T(x) = 3$$

CLUSTERING COEFFICIENT

Local Clustering Coefficient $$C_i = \binom{k_i}{2}^{-1} T(i)$$

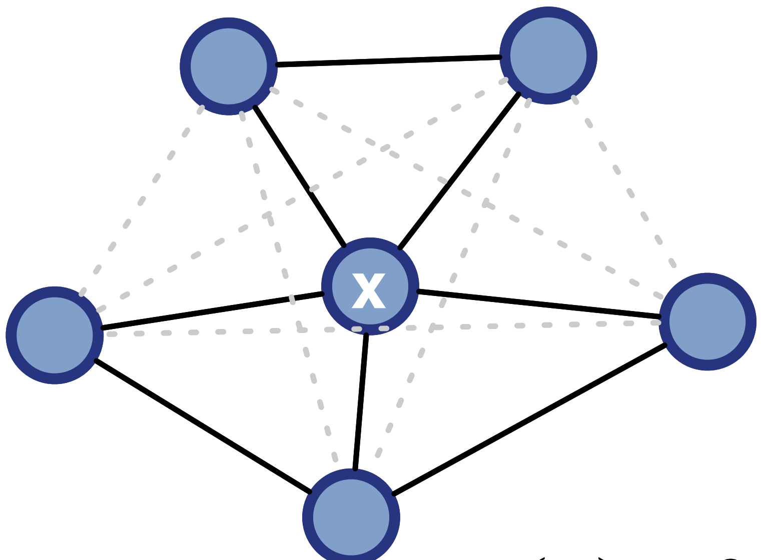
T(*i*): # distinct triangles with *i* as vertex

Clustering Coefficient

$$C = \frac{1}{n} \sum_{i \in V} C_i$$

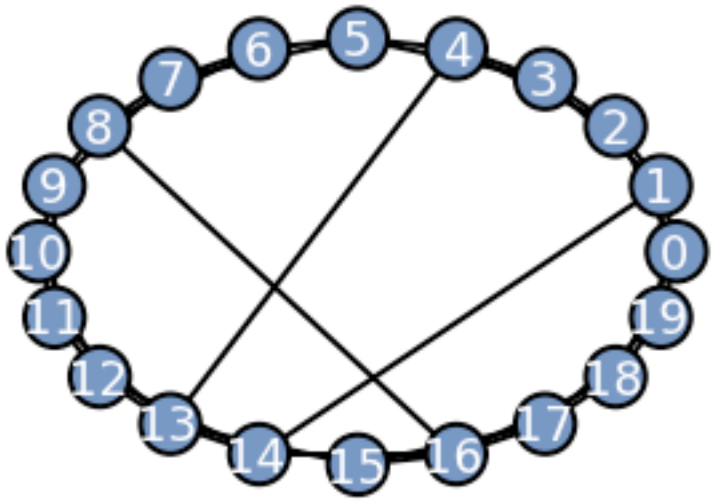
- Measure of **transitivity**
- High CC → “resilient” network
- Counting triangles

$$\Delta(G) = \sum_{i \in V} N(i) = \frac{1}{6} \operatorname{trace}\left(\mathbf{A}^3\right)$$



$$T(x) = 3 \quad \binom{k_x}{2}^{-1} = 10$$

$$k_x = 5 \qquad C_x = 0.3$$



$[\mathbf{A}^k]_{ij}$ number of paths of length *k* from *i* to *j*

$[\mathbf{A}^3]_{ii}$ All paths of length 3 starting and ending in *i* → **triangles**

```
A = nx.to_numpy_matrix(G)
```

```
nx.average_clustering(G)
```

```
0.41833333333333333
```

```
def clustering_coefficient(A):
    trs = diag(A**3) / 2. # better use eigenvalues
    degrees = np.asarray(A.sum(axis=1)).squeeze()
    significant_indices = degrees > 1
    max_neighbor_edges = ((degrees * (degrees - 1)) / 2)[significant_indices]
    local_ccs = trs[significant_indices] / max_neighbor_edges
    return np.average(local_ccs)
```

```
clustering_coefficient(A)
```

```
0.41833333333333333
```

DEGREE DISTRIBUTION

- Every “node-wise” property can be studies as an average, but it is most interesting to study the whole distribution.

- One of the most interesting is the “degree distribution”

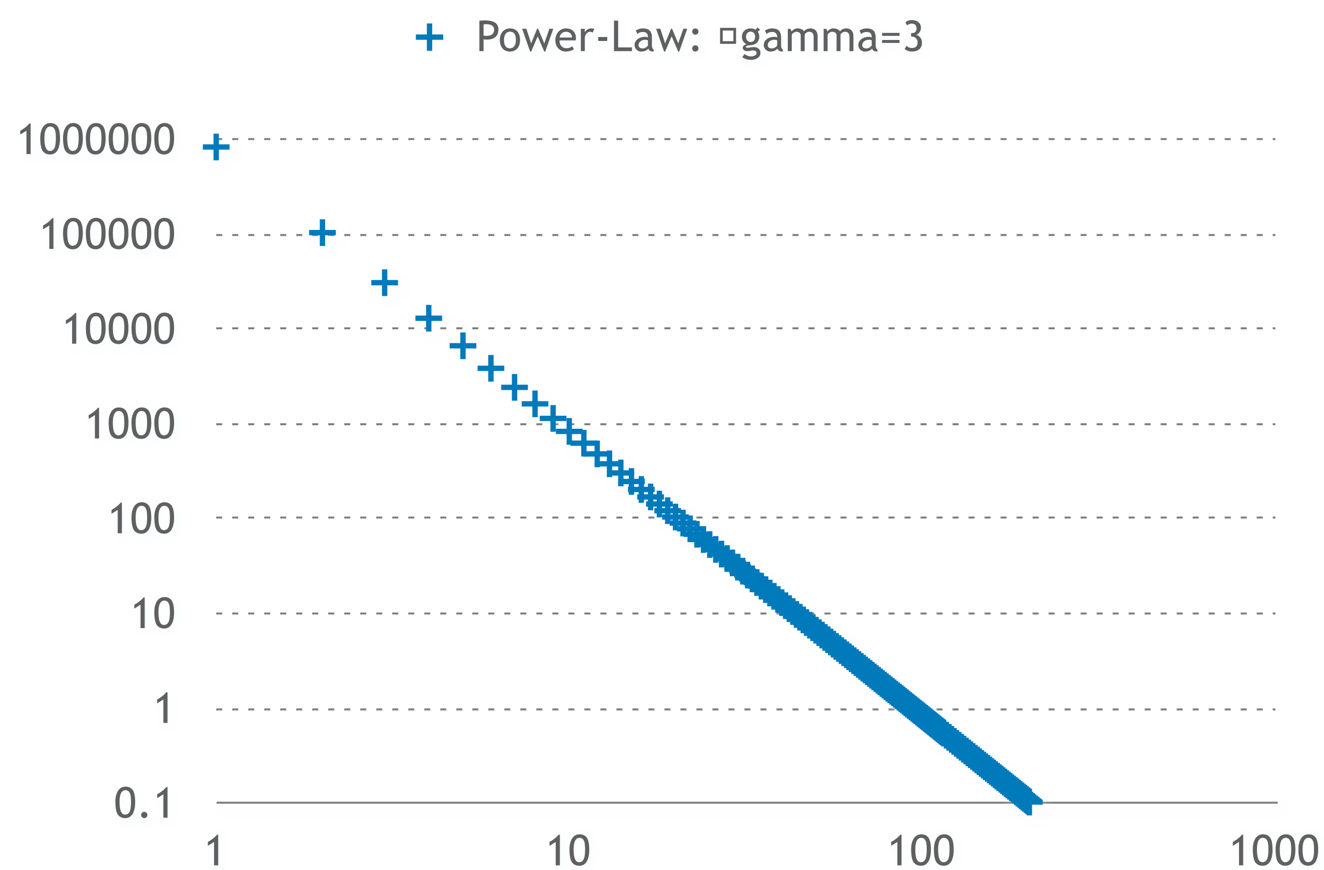
$$p_x = \frac{1}{n} \# \left\{ i \middle| k_i = x \right\}$$

Many networks have power-law degree distribution.

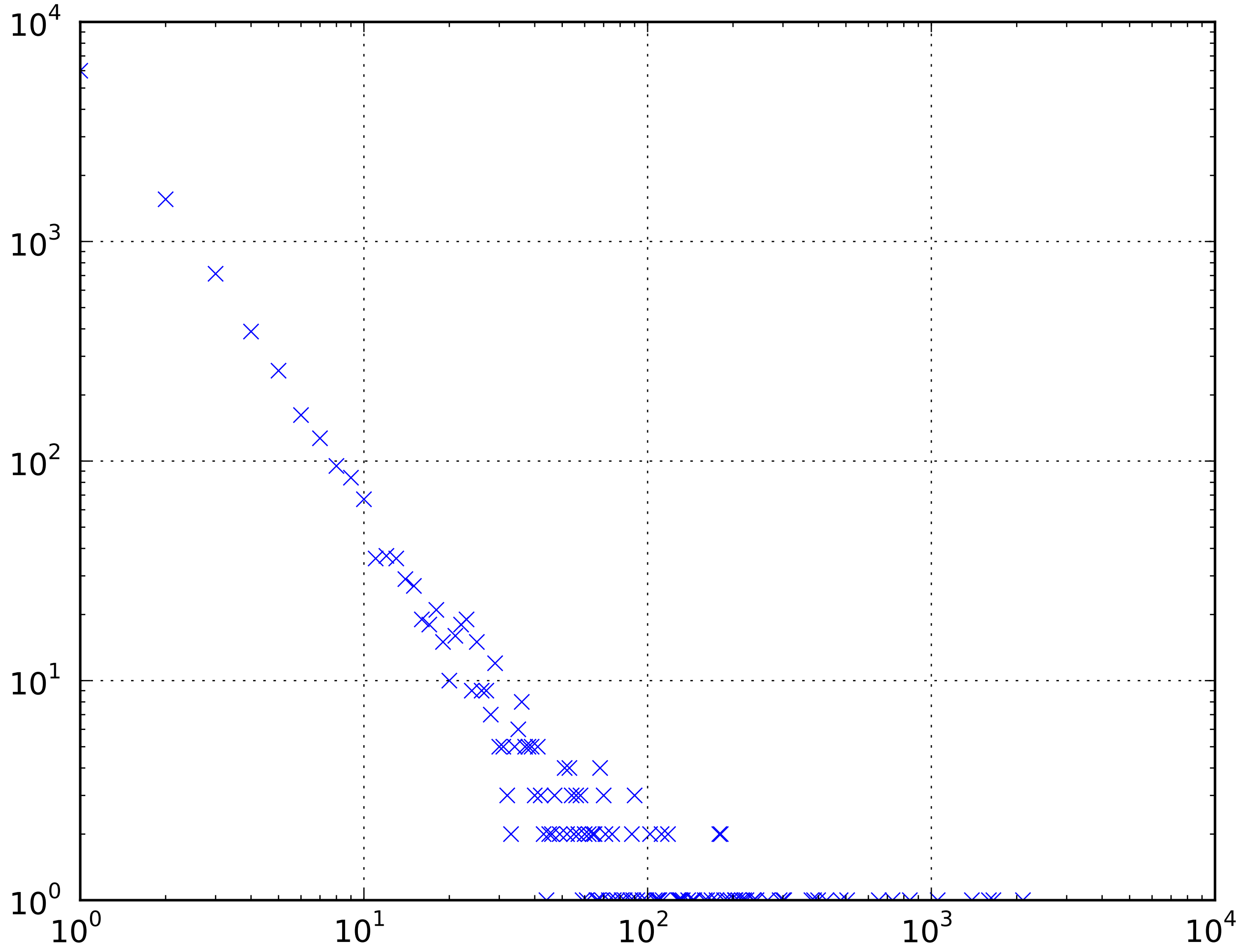
$$p_k \propto k^{-\gamma} \qquad \gamma > 1$$

$$\langle k^r \rangle = ?$$

- Citation networks
- Biological networks
- WWW graph
- Internet graph
- ***Social Networks***



log-log plot



Generated with random generator

80-20 Law

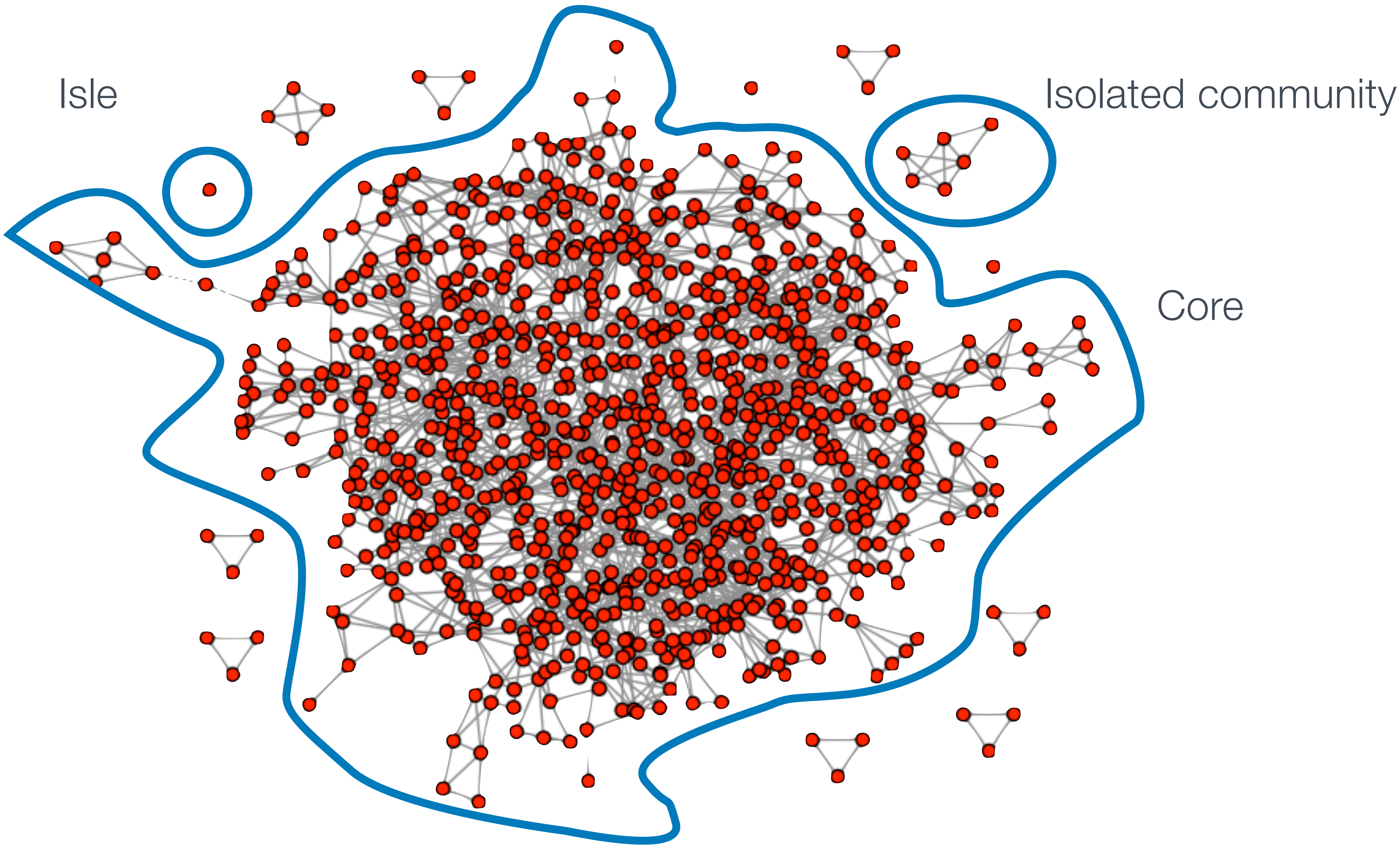
Few nodes account for the vast majority of links

Most nodes have very few links

This points towards the idea that we have a core with a fringe of nodes with few connections.

... and it it proved that implies super-short diameter

HIGH LEVEL STRUCTURE



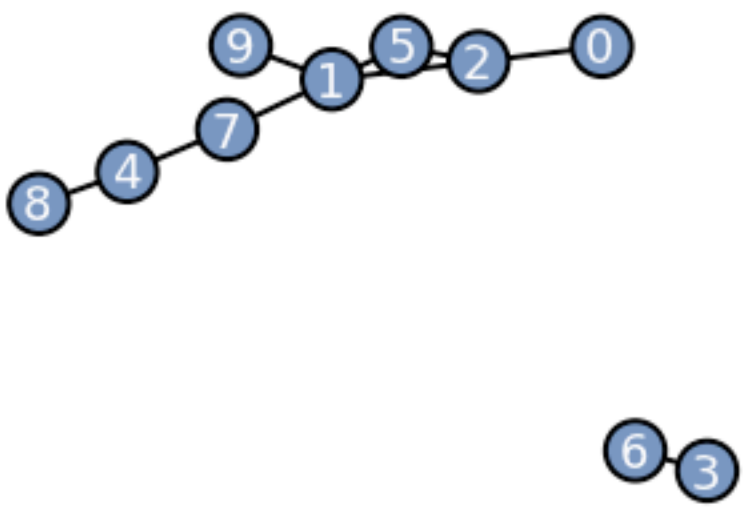
CONNECTED COMPONENTS

Most features are computed on the core

Directed/Undirected

Undirected: strongly connected = weakly connected

Directed: strongly connected != weakly connected



The adjacency matrix is primitive

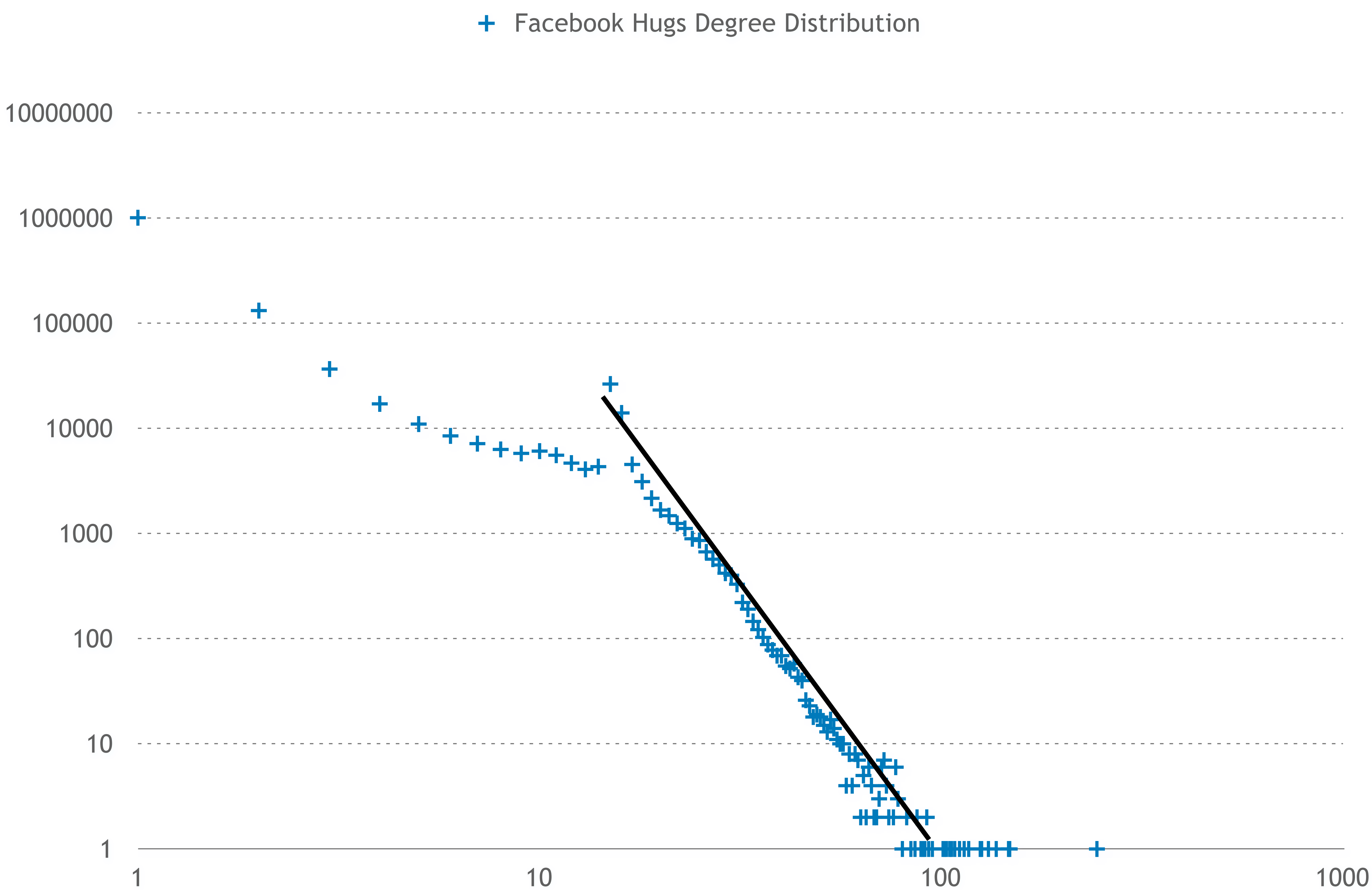
iff the network is connected

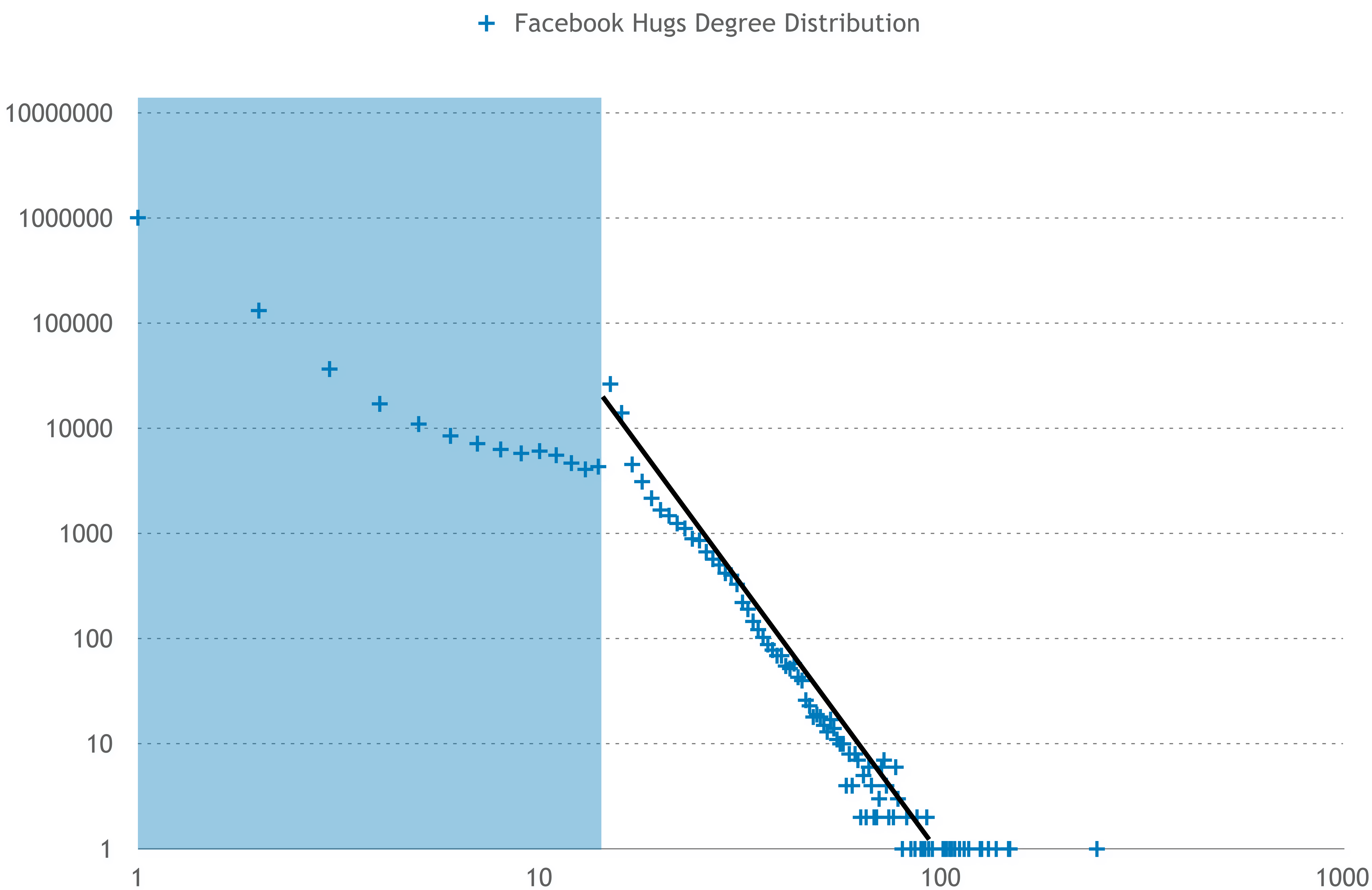
```
nx.connected_components(G)
```

```
[[0, 1, 2, 4, 5, 7, 8, 9], [3, 6]]
```

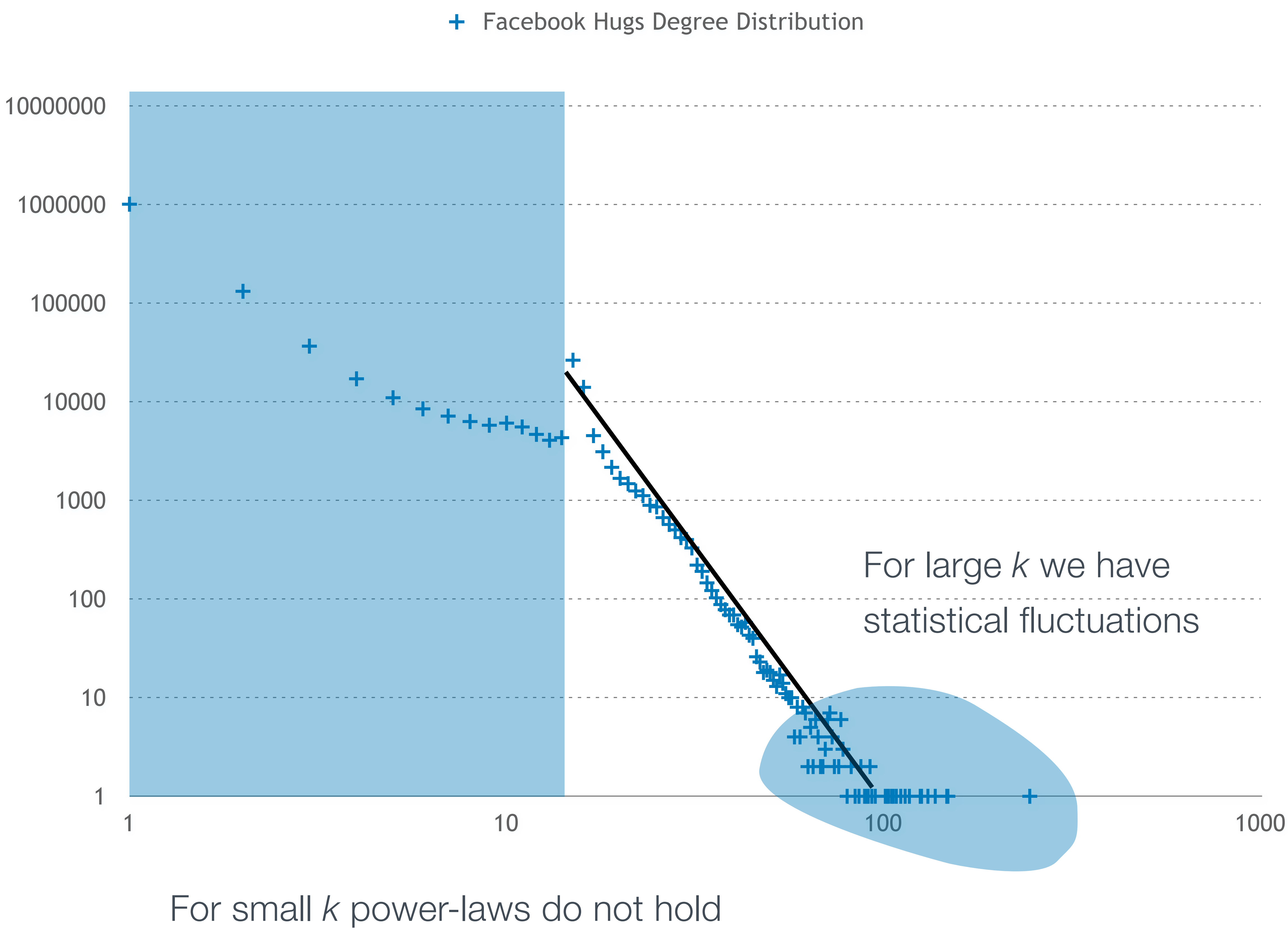
```
sparse.cs_graph_components(nx.to_scipy_sparse_matrix(G))
```

```
(2, array([0, 0, 0, 1, 0, 0, 1, 0, 0, 0], dtype=int32))
```

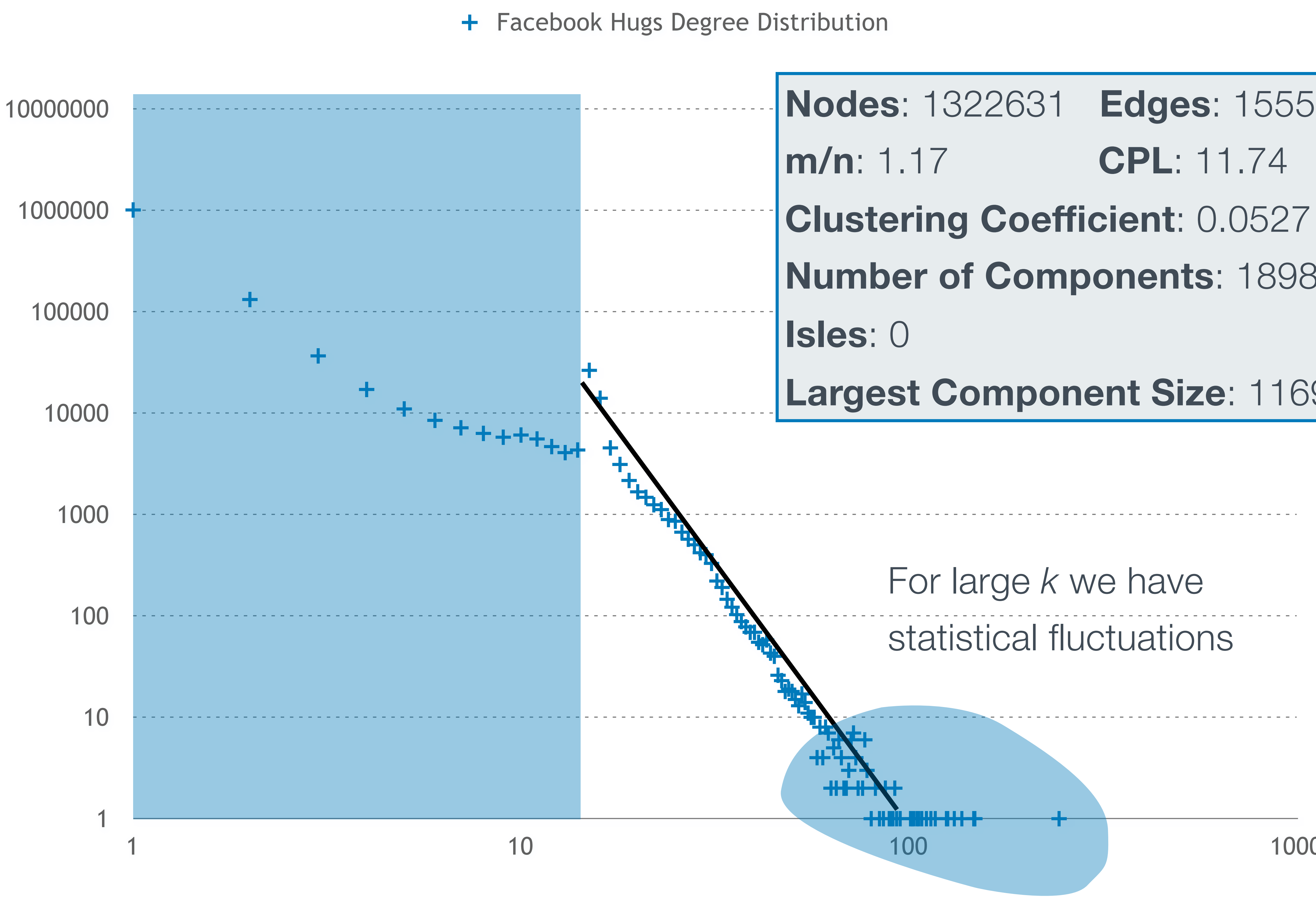

Facebook Hugs Degree Distribution
10000000
1000000
100000
10000
1000
100
10
1
1
10
100
1000



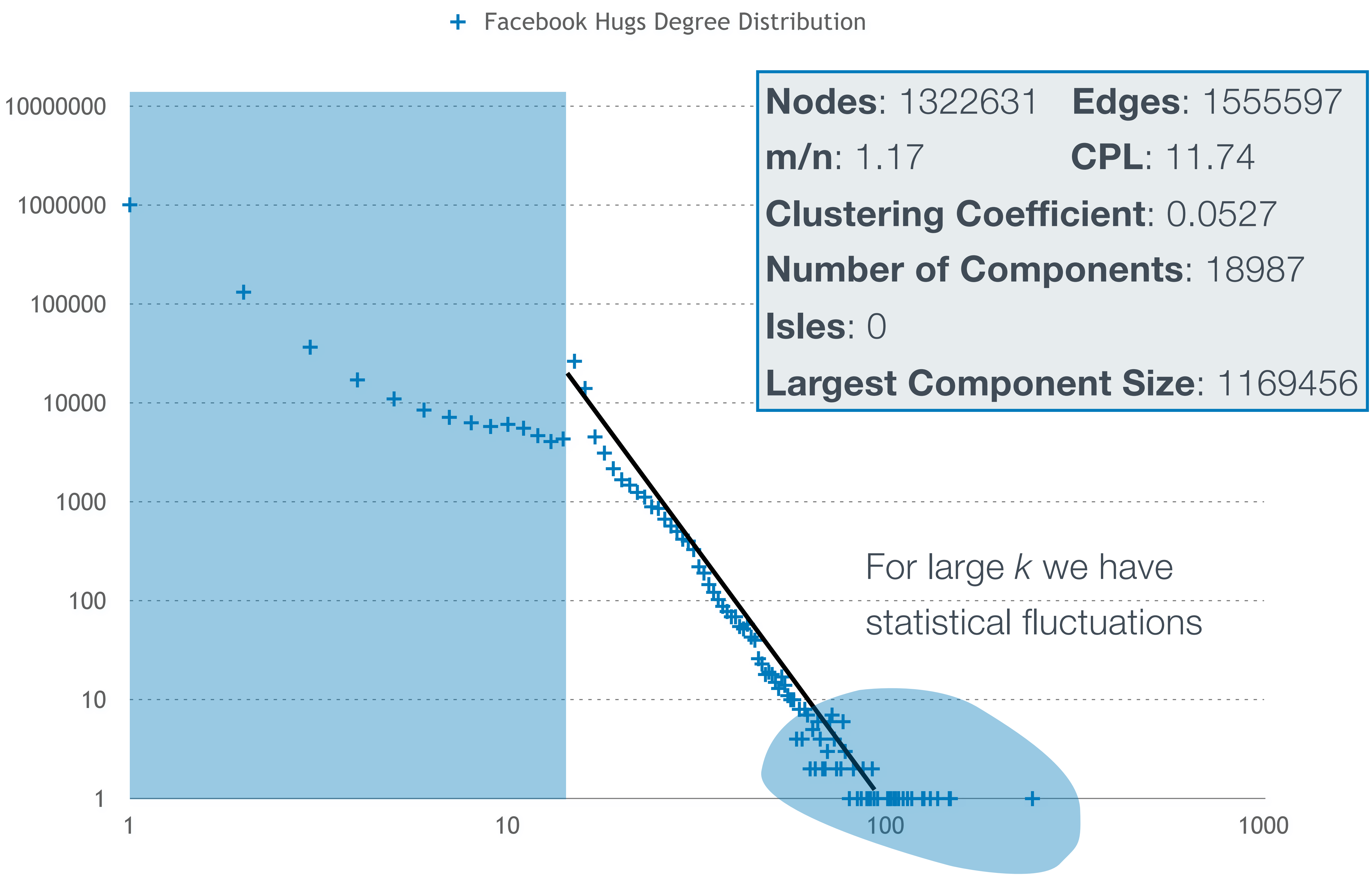
For small *k* power-laws do not hold



Facebook Hugs Degree Distribution
10000000
1000000
100000
10000
1000
100
10
1
1
10
100
1000
For large *k* we have
statistical fluctuations
For small *k* power-laws do not hold



For small *k* power-laws do not hold



For small *k* power-laws do not hold

Moreover, many distributions are wrongly identified as PLs

Online Social Networks

OSN	**Refs.**	**Users**	**Links**	**<k>**	***C***	**CPL**	***d***	**γ**	***r***
Club Nexus	Adamic et al	2.5 K	10 K	8.2	0.2	4	13	*n.a.*	*n.a.*
Cyworld	Ahn et al	12 M	191 M	31.6	0.2	3.2	16		-0.13
Cyworld T	Ahn et al	92 K	0.7 M	15.3	0.3	7.2	*n.a.*	*n.a.*	0.43
LiveJournal	Mislove et al	5 M	77 M	17	0.3	5.9	20		0.18
Flickr	Mislove et al	1.8 M	22 M	12.2	0.3	5.7	27		0.20
Twitter	Kwak et al	41 M	1700 M	*n.a.*	*n.a.*	4	4.1		*n.a.*
Orkut	Mislove et al	3 M	223 M	106	0.2	4.3	9	1.5	0.07
Orkut	Ahn et al	100 K	1.5 M	30.2	0.3	3.8	*n.a.*	3.7	0.31
Youtube	Mislove et al	1.1 M	5 M	4.29	0.1	5.1	21		-0.03
Facebook	Gjoka et al	1 M	*n.a.*	*n.a.*	0.2	*n.a.*	*n.a.*		0.23
FB H	Nazir et al	51 K	116 K	*n.a.*	0.4	*n.a.*	29		*n.a.*
FB GL	Nazir et al	277 K	600 K	*n.a.*	0.3	*n.a.*	45		*n.a.*
BrightKite	Scellato et al	54 K	213 K	7.88	0.2	4.7	*n.a.*		*n.a.*
FourSquare	Scellato et al	58 K	351 K	12	0.3	4.6	*n.a.*		*n.a.*
LiveJournal	Scellato et al	993 K	29.6 M	29.9	0.2	4.9	*n.a.*		*n.a.*
Twitter	Java et al	87 K	829 K	18.9	0.1	*n.a.*	6		0.59
Twitter	Scellato et al	409 K	183 M	447	0.2	2.8	*n.a.*		*n.a.*

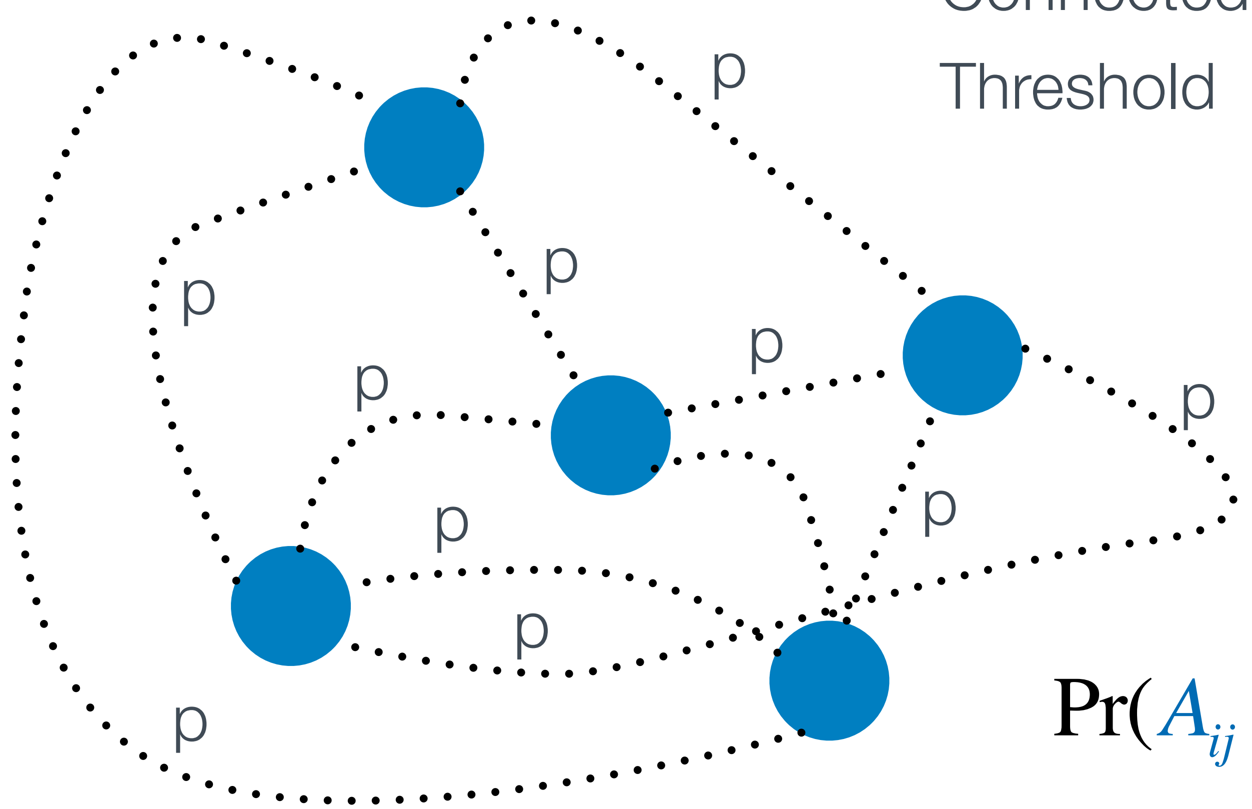
Erdös-Rényi Random Graphs

$$G(n, p)$$

$$G(n,m)$$

Ensembles of Graphs

When describe values of properties, we actually the expected value of the property



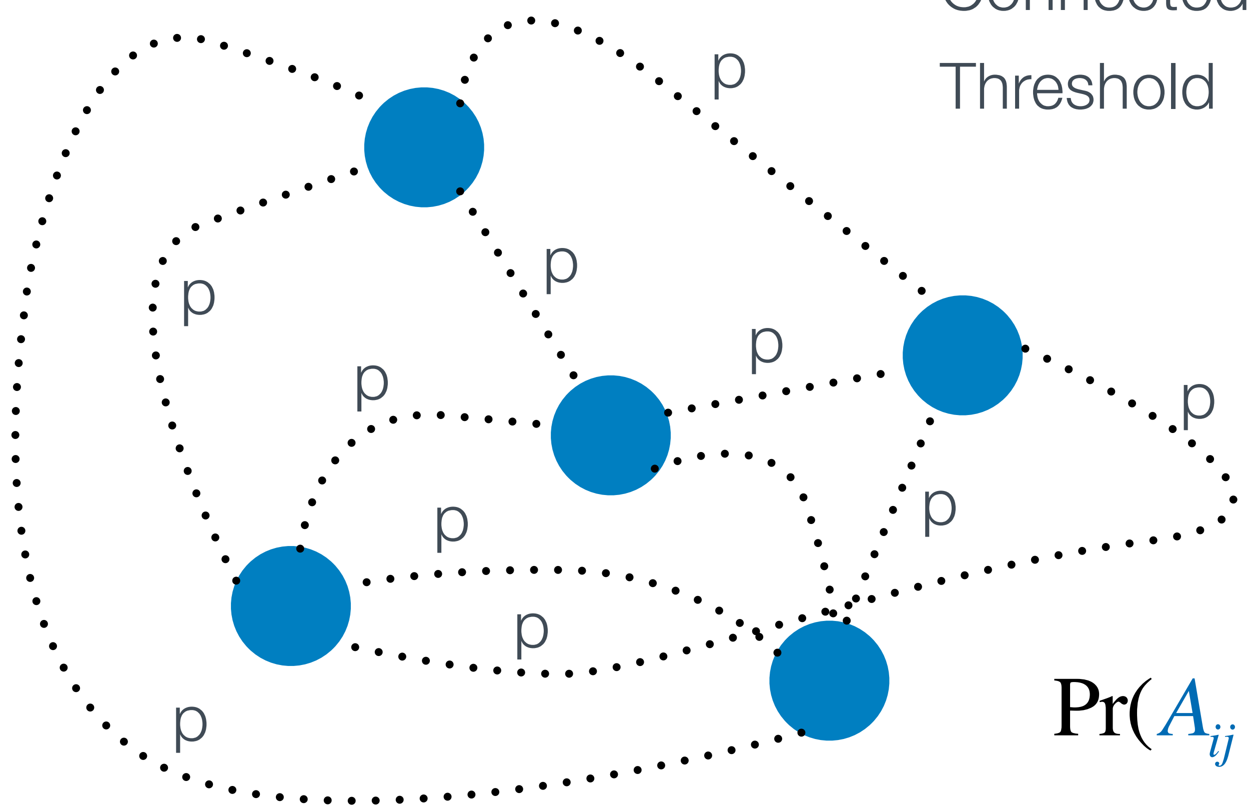
Connectedness Threshold $\log n / n$

$$\Pr(A_{ij} = 1) = p$$

Erdös-Rényi Random Graphs

$$G(n,p)$$

$$G(n,m)$$



Connectedness

Threshold $\log n / n$

$$\Pr(A_{ij} = 1) = p$$

Ensembles of Graphs

When describe values of properties, we actually the expected value of the property

$$d \coloneqq \langle d \rangle = \sum_G \Pr(G) \cdot d(G) \propto \frac{\log n}{\log \langle k \rangle}$$

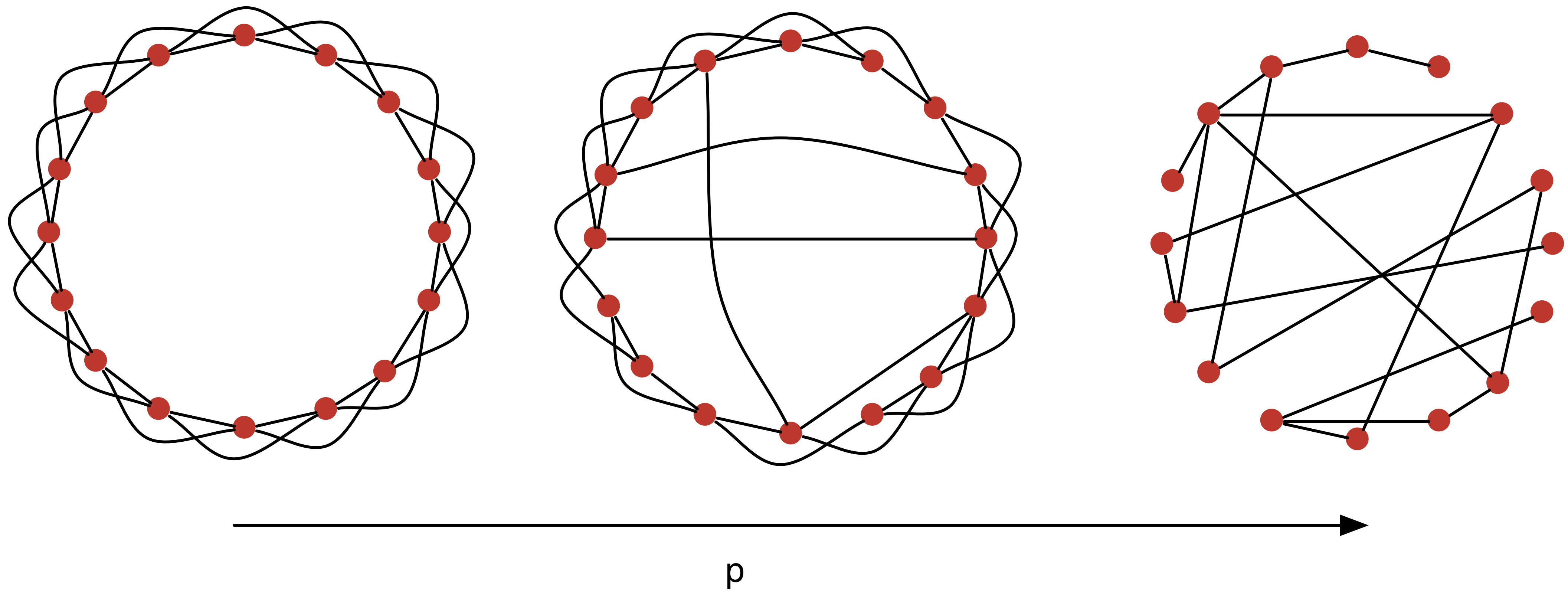
$$\Pr(G) = p^m (1-p)^{\binom{n}{2} - m}$$

$$C = \langle k \rangle (n-1)^{-1} = p$$

$$\langle m \rangle = \binom{n}{2} p$$

$$p_k = \binom{n-1}{k} p^k (1-p)^{n-1-k}$$

$$\boxed{n \to \infty \qquad p_k = e^{-\langle k \rangle} \frac{\langle k \rangle^k}{k!}}$$



Watts-Strogatz Model

In the modified model, we only add the edges.

$$k_i = \kappa + s_i$$

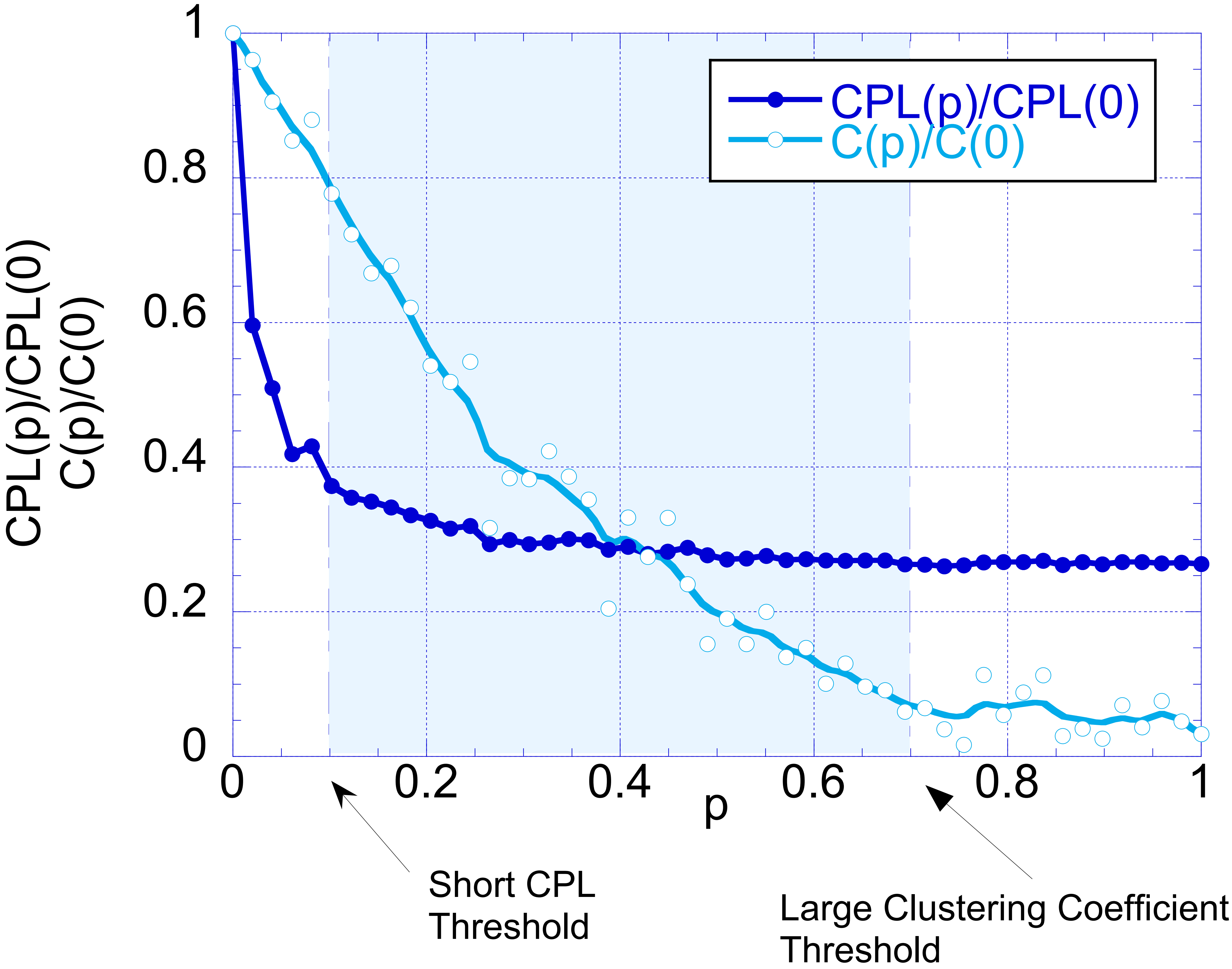
Edges in the lattice

added shortcuts

$$C \to \frac{3(\kappa - 2)}{4(\kappa - 1) + 8\kappa p + 4\kappa p^2}$$

$$\ell \approx \frac{\log(np\kappa)}{\kappa^2 p}$$

Strogatz-Watts Model - 10000 nodes k = 4



Barabási-Albert Model

Connectedness Threshold $\dfrac{\log n}{\log \log n}$

```
BARABASI-ALBERT-MODEL(G,M0,STEPS)
  FOR K FROM 1 TO STEPS
    N0 ← NEW-NODE(G)
    ADD-NODE(G,N0)
    A ← MAKE-ARRAY()
    FOR N IN NODES(G)
      PUSH(A, N)
      FOR J IN DEGREE(N)
        PUSH(A, N)
    FOR J FROM 1 TO M
      N ← RANDOM-CHOICE(A)
      ADD-LINK (N0, N)
```

$$p_k \propto x^{-3}$$

$$\ell \approx \frac{\log n}{\log \log n}$$

$$C \approx n^{-3/4}$$

Scale-free entails short CPL

Transitivity disappears with network size

No analytical proof available

ANALYSIS

- There are many network features we can study
- Let’s discuss some algorithms for the ones we studied so-far
- Also consider the *size* of the networks (> 1M nodes), so algorithmic costs can become an issue

Dijkstra Algorithm (single source shortest path)

```
from heapq import heappush, heappop
# based on recipe 119466
def dijkstra_shortest_path(graph, source):
    distances = {}
    predecessors = {}
    seen = {source: 0}
    priority_queue = [(0, source)]

    while priority_queue:
        v_dist, v = heappop(priority_queue)
        distances[v] = v_dist

        for w in graph[v]:
            vw_dist = distances[v] + 1
            if w not in seen or vw_dist < seen[w]:
                seen[w] = vw_dist
                heappush(priority_queue,(vw_dist,w))
                predecessors[w] = v

    return distances, predecessors
```

$$O(m \cdot \text{push}_Q + n \cdot \text{ex-min}_Q) = O(m \log n + n \log n)$$

Dijkstra Algorithm (single source shortest path)

```
from heapq import heappush, heappop
# based on recipe 119466
def dijkstra_shortest_path(graph, source):
    distances = {}
    predecessors = {}
    seen = {source: 0}
    priority_queue = [(0, source)]

    while priority_queue:
        v_dist, v = heappop(priority_queue)
        distances[v] = v_dist

        for w in graph[v]:
            vw_dist = distances[v] + 1
            if w not in seen or vw_dist < seen[w]:
                seen[w] = vw_dist
                heappush(priority_queue,(vw_dist,w))
                predecessors[w] = v

    return distances, predecessors
```

$O(m\cdot \text{push}_Q + n\cdot \text{ex-min}_Q) = O(m \log n + n \log n)$

Dijkstra Algorithm (single source shortest path)

```
from heapq import heappush, heappop
# based on recipe 119466
def dijkstra_shortest_path(graph, source):
    distances = {}
    predecessors = {}
    seen = {source: 0}
    priority_queue = [(0, source)]

    while priority_queue:
        v_dist, v = heappop(priority_queue)
        distances[v] = v_dist

        for w in graph[v]:
            vw_dist = distances[v] + 1
            if w not in seen or vw_dist < seen[w]:
                seen[w] = vw_dist
                heappush(priority_queue,(vw_dist,w))
                predecessors[w] = v

    return distances, predecessors
```

$O(m \cdot \text{push}_Q + n \cdot \text{ex-min}_Q) = O(m \log n + n \log n)$

Computational Complexity of ASPL:

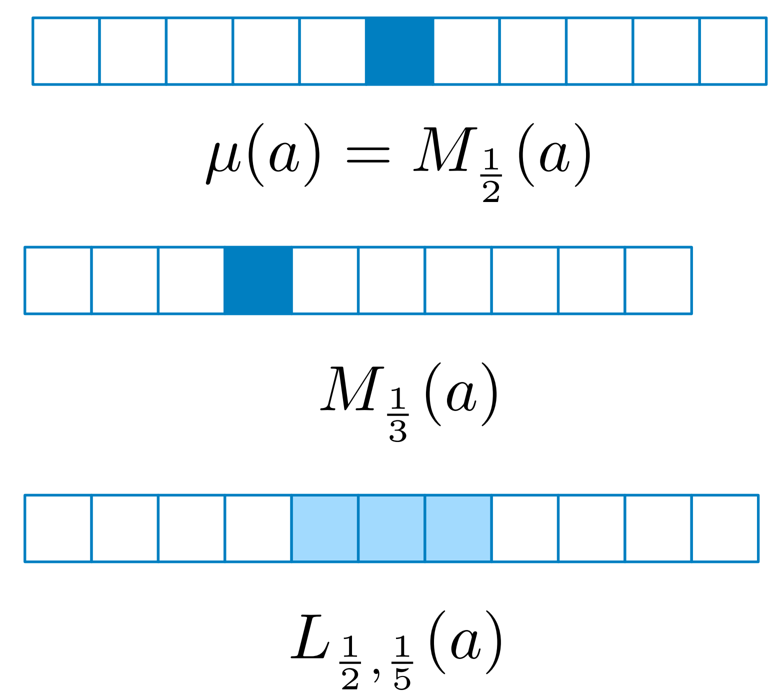
All pairs shortest path matrix based (parallelizable): $\Theta\left(n^3\right)$

All pairs shortest path Dijkstra w. Fibonacci Heaps: $O\left(n^2 \log n + nm\right)$

Computing the CPL

$x = M_q(S)$ q#S elements are ≤ than x and (1-q)#S are > than x

$x \in L_{q\delta}(S)$ q#S(1-δ) elements are ≤ than x or (1-q)#S(1-δ) are > than x



Huber Method

$$s = \frac{2}{q^2} \ln \frac{2}{\epsilon} \frac{\left(1-\delta\right)^2}{\delta^2}$$

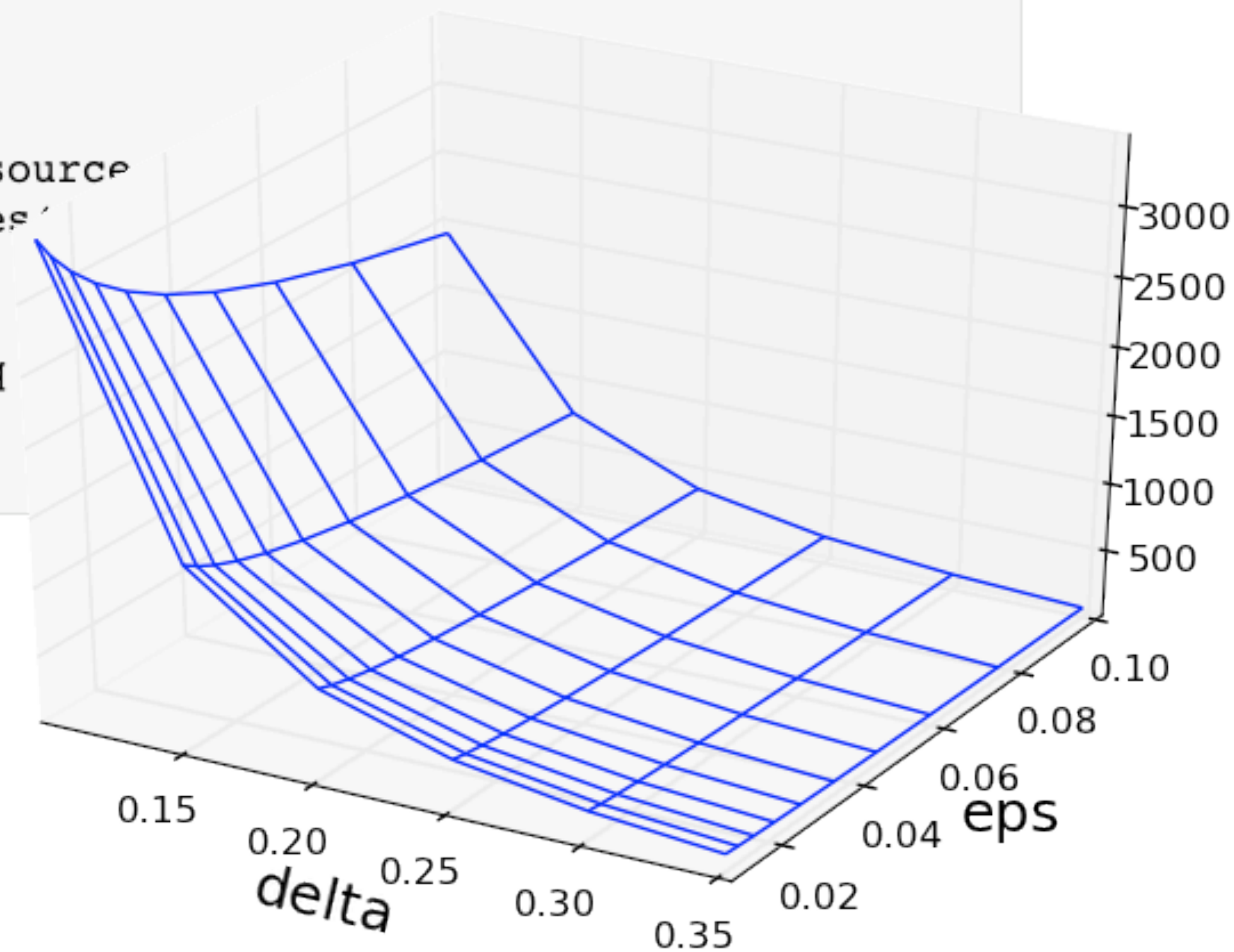
Let R a random sample of S such that #R=s, then $M_q(R) \in L_{q\delta}(S)$ with probability p = 1-ε.

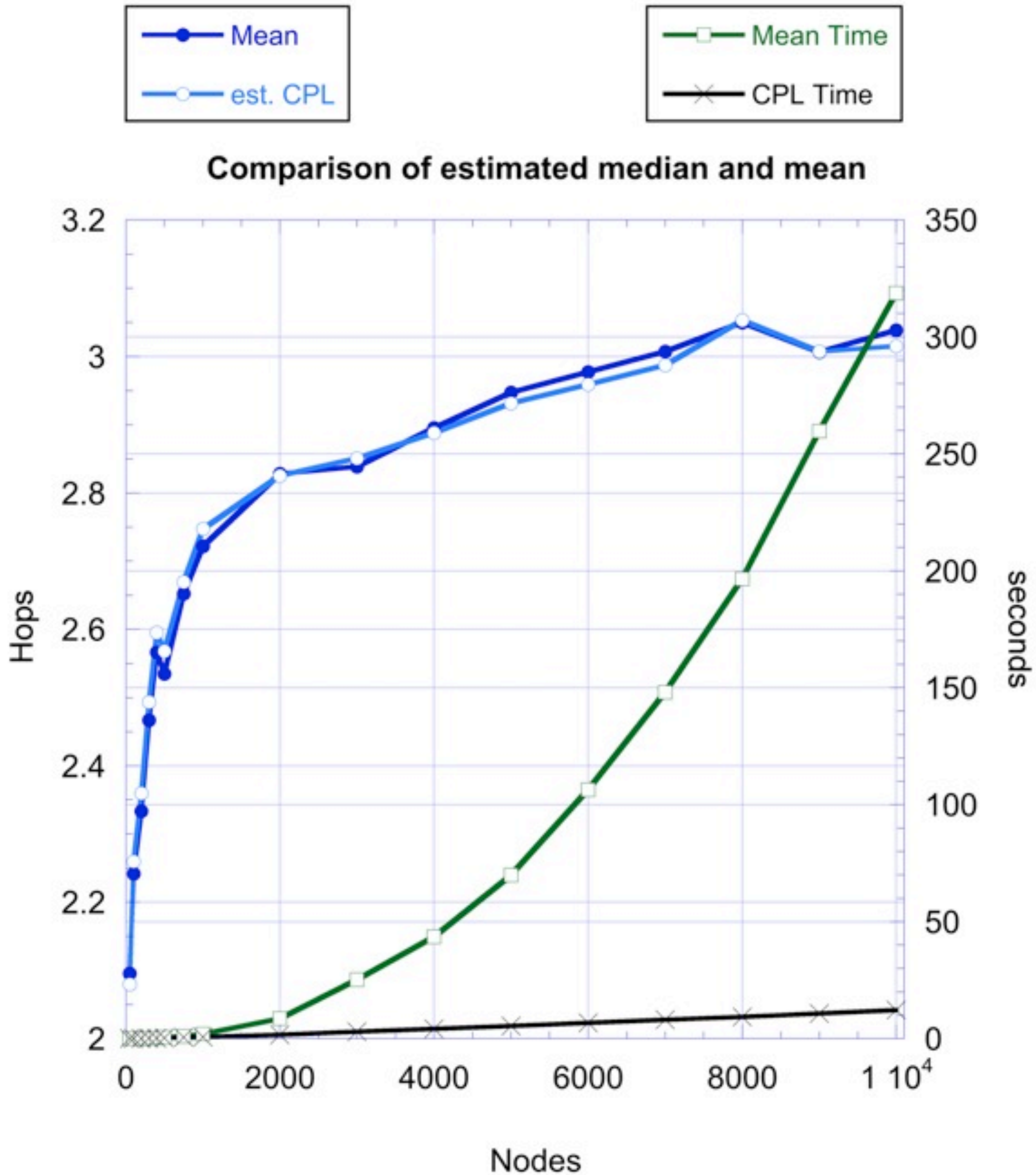
```python
def estimate_s(q, delta, eps):
    delta2 = delta * delta
    delta3 = (1 - delta) * (1 - delta)
    return (2. / (q * q)) * math.log(2. / eps) * delta3 / delta2
```

```python
def approximate_cpl(graph, q=0.5, delta=0.15, eps=0.05):
    assert isinstance(graph, networkx.Graph)
    s = estimate_s(q, delta, eps)
    s = int(math.ceil(s))
    if graph.number_of_nodes() <= s:
        sample = graph.nodes_iter()
    else:
        sample = random.sample(graph.adj.keys(), s)

    averages = []
    for node in sample:
        path_lengths = networkx.single_source_shortest_path_length(graph, node)
        average = sum(path_lengths.values()) / float(len(path_lengths))
        averages.append(average)
    averages.sort()
    median_index = int(len(averages) * q + 1)
    return averages[median_index]
```

```python
def estimate_s(q, delta, eps):
    delta2 = delta * delta
    delta3 = (1 - delta) * (1 - delta)
    return (2. / (q * q)) * math.log(2. / eps) * delta3 / delta2
```

```python
def approximate_cpl(graph, q=0.5, delta=0.15, eps=0.05):
    assert isinstance(graph, networkx.Graph)
    s = estimate_s(q, delta, eps)
    s = int(math.ceil(s))
    if graph.number_of_nodes() <= s:
        sample = graph.nodes_iter()
    else:
        sample = random.sample(graph.adj.keys(), s)

    averages = []
    for node in sample:
        path_lengths = networkx.single_source
        average = sum(path_lengths.values
        averages.append(average)
    averages.sort()
    median_index = int(len(averages) * q
    return averages[median_index]
```





$$s = \frac{2}{q^2} \ln \frac{2}{\epsilon} \frac{(1-\delta)^2}{\delta^2}$$

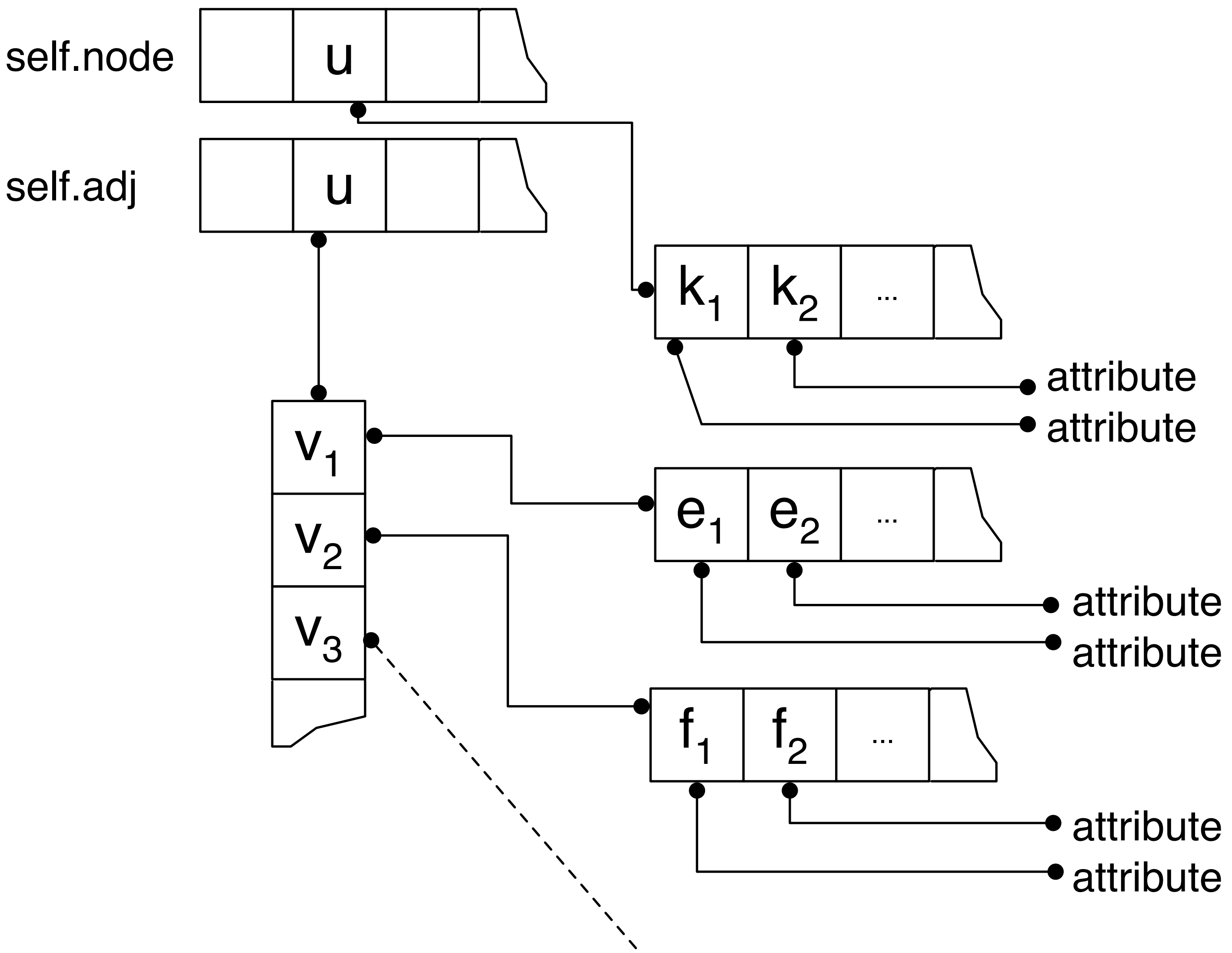
HOW ABOUT THE MEMORY?

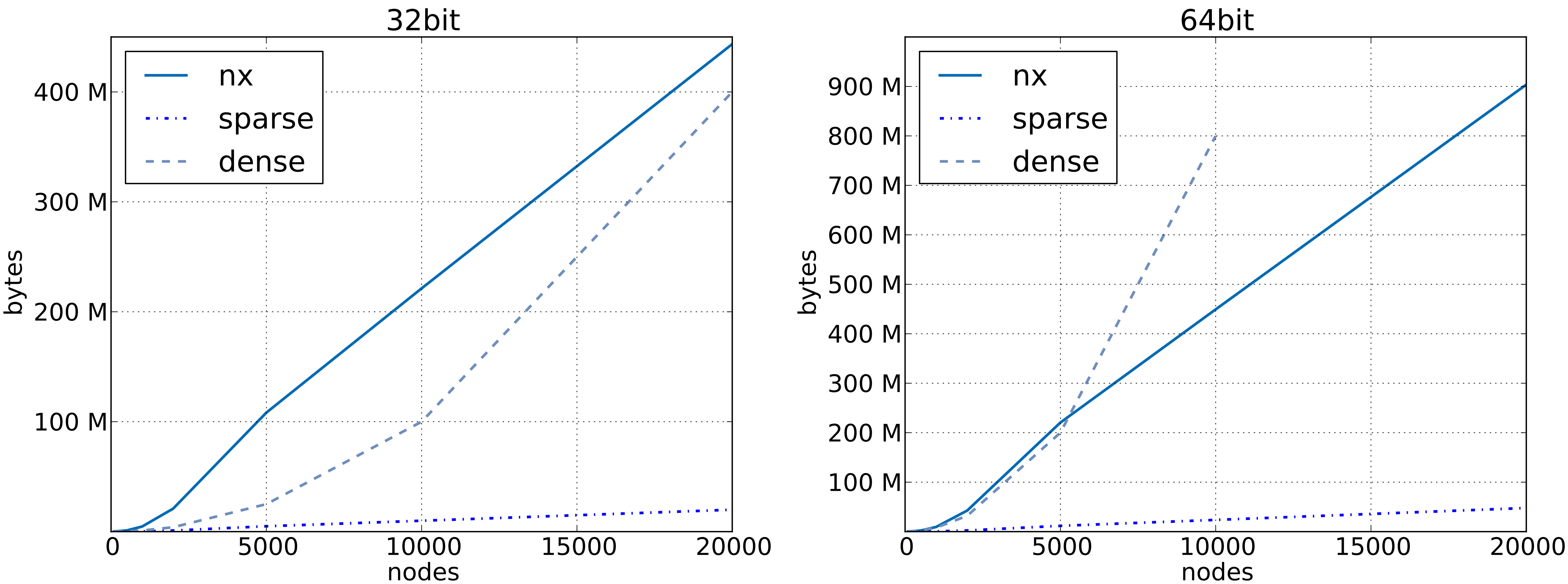
- Different representations
- Different trade-offs (space/time)
- How easy is metadata to manipulate
- Disk/RAM

POPULAR REPRESENTATIONS

- Adjacency List
- Incidence List
- Adjacency Matrix (using sparse matrices)
- Incidence Matrix (using sparse matrices)

```python
class AdjacencyListGraph(object):
    def __init__(self):
        self.node = {}
        self.adj = {}

    def add_node(self, node, **attrs):
        if node not in self.adj:
            self.adj[node] = {}
            self.node[node] = attrs
        else: # update attr even if node already exists
            self.node[node].update(attrs)

    def add_edge(self, u, v, **attrs):
        if u not in self.adj:
            self.adj[u] = {}
            self.node[u] = {}
        if v not in self.adj:
            self.adj[v] = {}
            self.node[v] = {}

        datadict=self.adj[u].get(v,{})
        datadict.update(attrs)

        self.adj[u][v] = datadict
        self.adj[v][u] = datadict
```





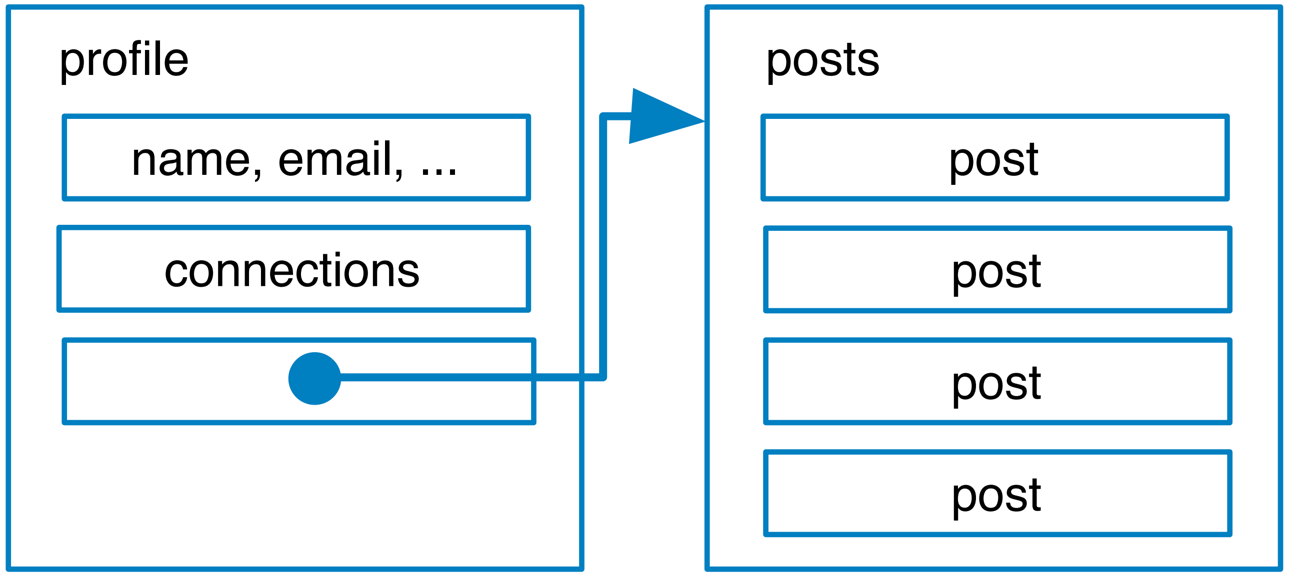
32 bit variant

Nodes	Edges	NX bytes	Sparse bytes	Dense bytes
100	733	198000	7734	10000
500	3922	1037976	41224	250000
1000	19518	4621688	199184	1000000
2000	96941	20927728	977414	4000000
5000	487686	108248888	4896864	25000000
10000	987274	221310552	9912744	100000000
20000	1986718	443525880	19947184	400000000

DISK BASED SOLUTIONS

- HDF5 (Pytables, h5py)
- Map-Reduce (Hadoop)
- Graph DBs (Riak, Neo4J, Allegro)
- “NoSQL graphs” (Mongo, ...)
- SQL DBs (PostgreSQL)

How about social networking applications?



When a user opens his page, his contacts profiles are read to get their posts.

A user that has many friends (high degree) has his profile read more often.

The degree distribution becomes the profile accesses distribution.

Very good for caching!

And what about the clustering coefficient? Friends of friends tend to be friends...

Minimize: $$\chi(G) = \sum_{v,u \in V^2} (\mathrm{loc}(v) - \mathrm{loc}(u))\, A_{uv}$$

"location of v profile"

unfortunately NP complete

We can however use community detection to improve locality.

We define a cluster to be a subgraph with some cohesion.

Different cluster definition exist, with different trade-offs.

But profiles in a cluster tend to be accessed together, so that actually we can store the information "close" (disk-layout, sharding)

We can also study the correlations between geography and clustering and hopefully use that info.

In general, for SNS knowing the network structure gives insight on how to optimize stuff.

NETWORK PROCESSES

- Study of processes that occur on real networks
- “Network destruction”: models malfunctions in the network
- “Idea/Disease” diffusion over networks
- Link prediction

Network Destruction Process

```
def attack(graph, centrality_metric):
    graph = graph.copy()
    steps = 0
    ranks = centrality_metric(graph)
    nodes = sorted(graph.nodes(), key=lambda n: ranks[n])

    while nx.is_connected(graph):
        graph.remove_node(nodes.pop())
        steps += 1
    else:
        return steps
```

	Power-Law Cluster	Random
Random Attack	220	**10**
PageRank driven	**19**	149
Betweenness driven	**22**	157
Degree	**19**	265

```python
import networkx as nx
```

```python
CLOSING_FIXTURE = 'closing fixture'
FRUIT = 'fruit'
ANTARTIC_BIRD = 'antartic bird'

INFECTED = 'infected'
IMMUNE = 'immune'
CLEAN = 'clean'

INFECTION_RATE = {CLOSING_FIXTURE: 0.05,
                  FRUIT: 0.05,
                  ANTARTIC_BIRD: 0.05}
```

```python
def infection_step(G, has_updated_antivirus):
    for node, attributes in G.nodes(data=True):
        is_infect = attributes.get('status', INFECTED)
        if has_updated_antivirus(node, attributes):
            G.add_node(node, status=IMMUNE)
        elif is_infect:
            propagate_infection(G, node, attributes)
```

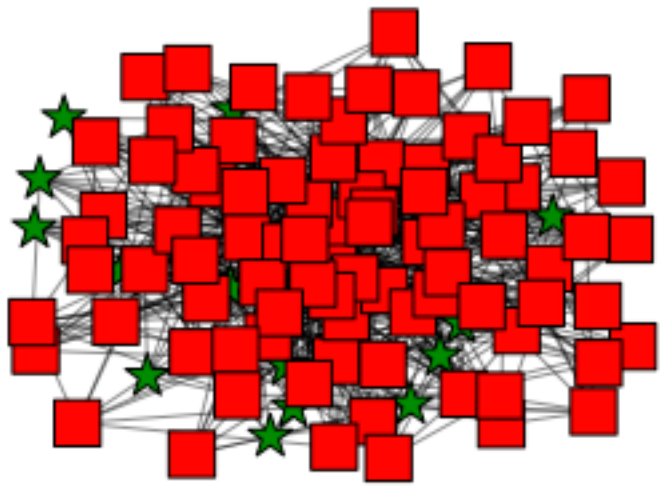
```python
def propagate_infection(G, node, attributes):
    node_os = attributes['os']
    for neighbor, neighbor_attributes in G.nodes(G.neighbors(1)):
        if (node_os == neighbor_attributes['os']
            and neighbor_attributes.get('status') != IMMUNE
            and rand() < INFECTION_RATE[node_os]):
            G.add_node(neighbor, status=INFECTED)
```

```python
def partition_graph(G, attribute_name):
    partitions = {}
    for node, attributes in G.nodes(data=True):
        partitions.setdefault(attributes[attribute_name], []).append(node)
    return partitions
```

```python
def draw_graph(G):
    color = {INFECTED: 'r',
             CLEAN: 'b',
             IMMUNE: 'g'}
    shape = {INFECTED: 's',
             CLEAN: 'o',
             IMMUNE: '*'}
    pos = nx.spring_layout(G)
    status_partitions = partition_graph(G, 'status')
    for partition_name, partition in status_partitions.iteritems():
        nx.draw_networkx_nodes(G, pos, nodelist=partition,
                               node_color = color[partition_name],
                               node_shape = shape[partition_name])
    nx.draw_networkx_edges(G, pos, alpha=0.5)
    axis('off')
```

```
G = process(100)
```

```
draw_graph(G)
```



```
by_os = partition(G, 'os')
```

```
by_status = partition(G, 'status')
```

```
set(by_os[CLOSING_FIXTURE]) & set(by_status[IMMUNE])
```

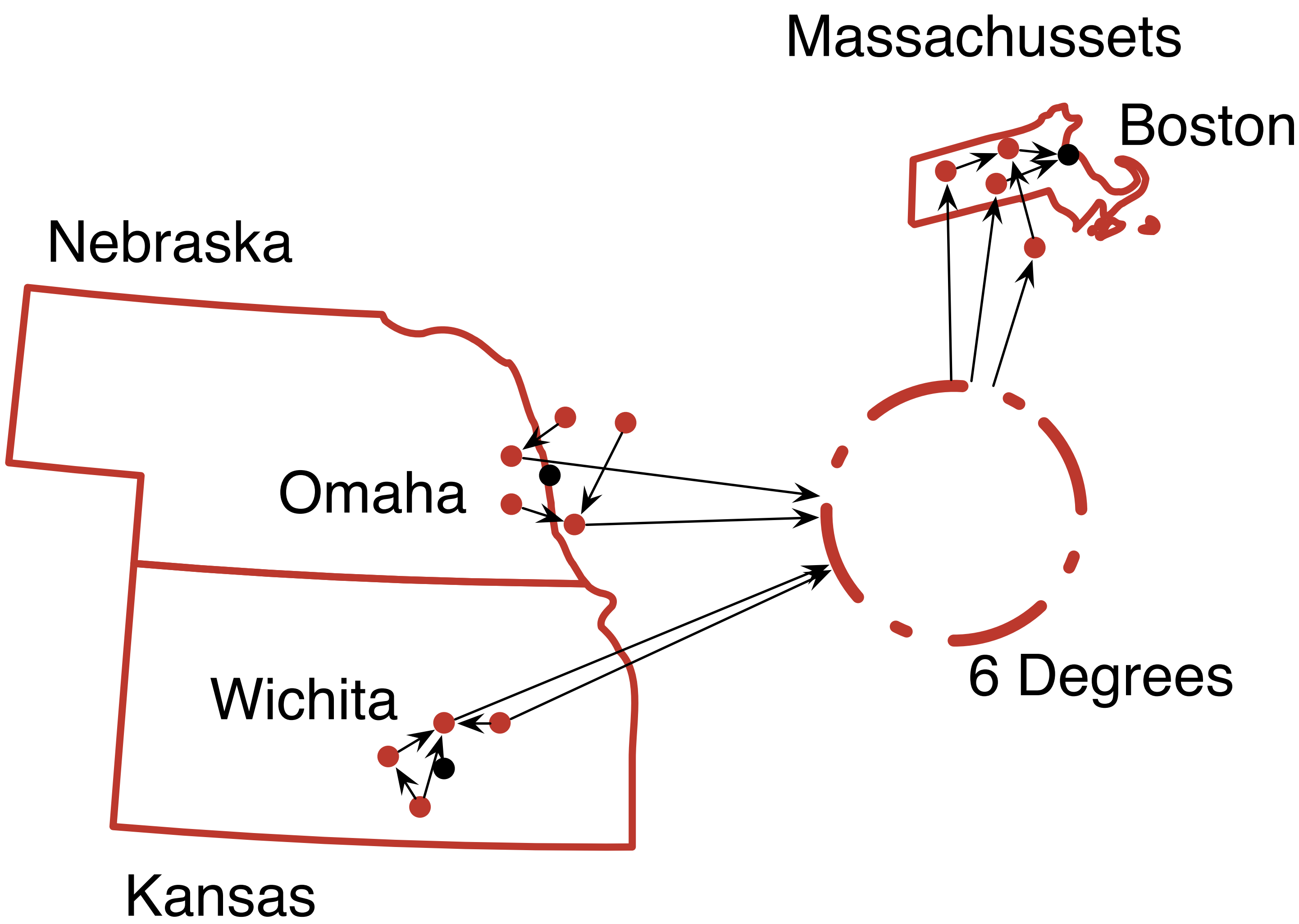
```
set([10, 51])
```

```python
def install_os(G, closing_fixture=0.89, fruit=0.1, antartic_bird=0.01):
    for node in G:
        random_value = rand()
        if random_value < closing_fixture:
            chosen_os = CLOSING_FIXTURE
        elif random_value < closing_fixture + fruit:
            chosen_os = FRUIT
        else:
            chosen_os = ANTARTIC_BIRD
        G.add_node(node, os=chosen_os)
```

```python
def initial_status(G, infection_p, immune_p=None):
    if immune_p is None:
        immune_p = infection_p / 100.
    for node in G:
        random_value = rand()
        if random_value < immune_p:
            G.add_node(node, status=IMMUNE)
        elif random_value < immune_p + infection_p:
            G.add_node(node, status=INFECTED)
        else:
            G.add_node(node, status=CLEAN)
```

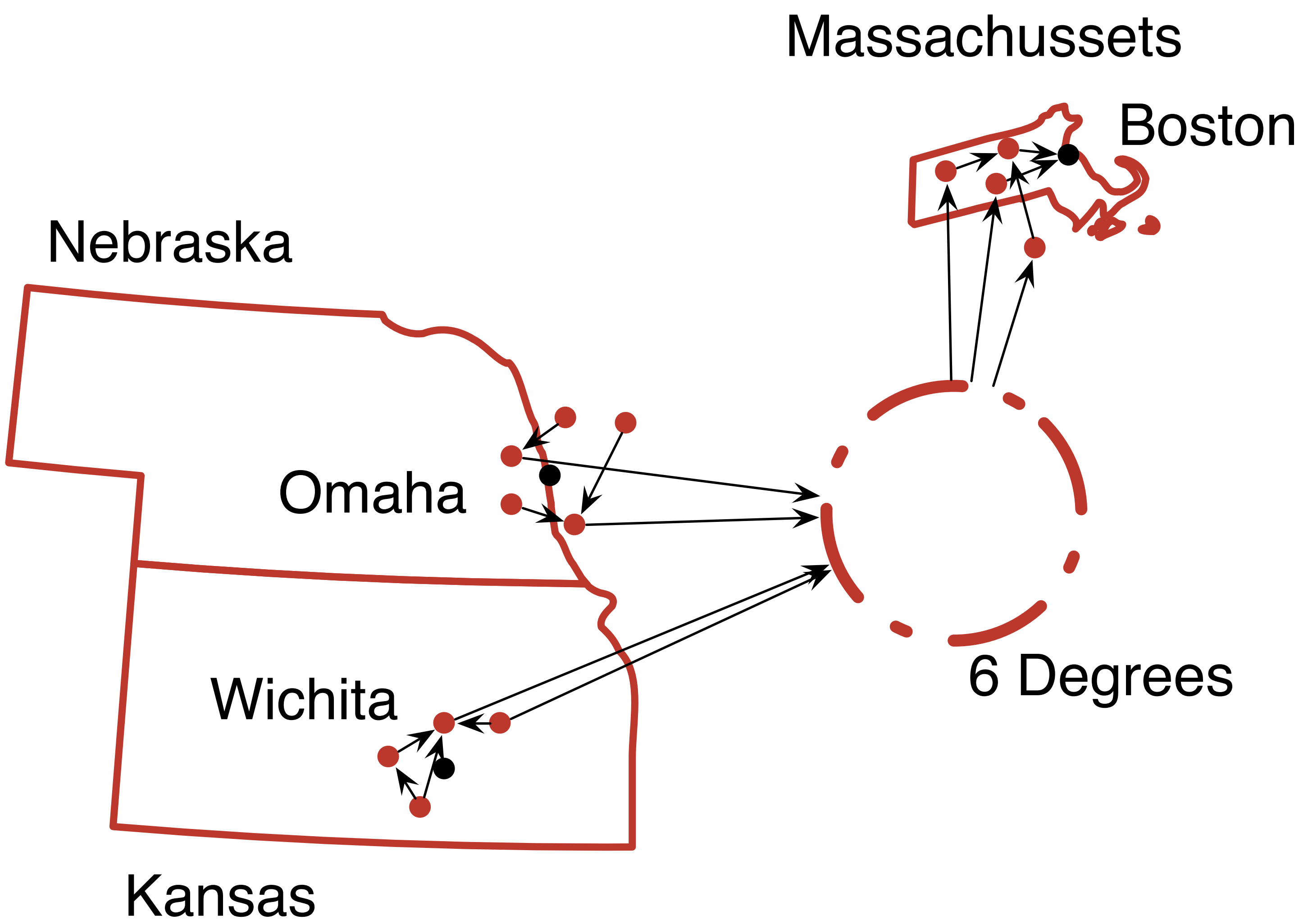
```python
def simple_antivirus(node, attributes, p=0.00005):
    if attributes['os'] == ANTARTIC_BIRD:
        return True
    else:
        return rand() < p
```

Thanks for your kind attention.



Milgram’s Experiment

- Random people from Omaha & Wichita were asked to send a postcard to a person in Boston:
- Write the name on the postcard
- Forward the message only to people personally known that was more likely to know the target



1st run: 64/296 arrived, most delivered to him by 2 men

2nd run: 24/160 arrived, 2/3 delivered by “Mr. Jacobs”

$2 \leq$ hops ≤ 10; μ=5.x

CPL, hubs, ...
... and Kleinberg’s Intuition

Milgram’s Experiment

- Random people from Omaha & Wichita were asked to send a postcard to a person in Boston:
- Write the name on the postcard
- Forward the message only to people personally known that was more likely to know the target

References

[1] Ahn, Y. Y., Han, S., Kwak, H., Moon, S., and Jeong, H. 2007. Analysis of topological characteristics of huge online social networking services Proceedings of the 16th International Conference on World Wide Web. 835–844.
[2] Adamic, L., Buyukkokten, O., and Adar, E. 2003. A social network caught in the web. First Monday. 8, 6, 29.
[3] Barabási, A. L. and Albert, R. 1999. Emergence of Scaling in Random Networks. Science. 286, 509–512.
[4] Bergenti, F., Franchi, E., and Poggi, A. 2011. Selected Models for Agent-based Simulation of Social Networks Proceedings of the 3rd Symposium on Social Networks and Multiagent Systems (SNAMAS~'11). 27–32.
[5] Dodds, P. S., Muhamad, R., and Watts, D. J. 2003. An experimental study of search in global social networks. Science. 301, 5634, 827–829.
[6] Dunbar, R. I. M. 1992. Neocortex size as a constraint on group size in primates. Journal of Human Evolution. 22, 6, 469–493.
[7] Erd\H{o}s, P. and Rényi, A. 1959. On random graphs. Publicationes Mathematicae. 6, 26, 290–297.
[8] Gjoka, M., Kurant, M., Butts, C. T., and Markopoulou, A. 2010. Walking in Facebook: A Case Study of Unbiased Sampling of OSNs Proceedings of IEEE INFOCOM '10.
[9] Java, A., Song, X., and Finin, T. 2007. Why we twitter: understanding microblogging usage and communities Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis. 1–10.
[10] Killworth, P. D. and Bernard, H. R. 1979. The Reversal Small-World Experiment. Social Networks. 1, 2, 159–192.
[11] Kleinberg, J. 2006. Complex networks and decentralized search algorithms Proceedings of the International Congress of Mathematicians (ICM). 3, 1–26.
[12] Kwak, H., Lee, C., Park, H., and Moon, S. 2010. What is Twitter, a Social Network or a News Media? Proceedings of the 19th international conference on World wide web. 591–600.
[13] Milgram, S. 1967. The small world problem. Psychology Today. 1, 1, 61–67.
[14] Mislove, A., Marcon, M., Gummadi, K. P., Druschel, P., and Bhattacharjee, B. 2007. Measurement and analysis of online social networks Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement. 29–42.
[15] Nazir, A., Raza, S., and Chuah, C. N. 2008. Unveiling facebook: a measurement study of social network based applications Proceedings of the 8th ACM SIGCOMM conference on Internet measurement. 43–56.
[16] Newman, M. E. J. and Watts, D. J. 1999. Renormalization group analysis of the small-world network model. Physics Letters A. 263, 341–346.
[17] Nguyen, V. and Martel, C. 2005. Analyzing and characterizing small-world graphs Proceedings of the 16th annual ACM-SIAM Symposium on Discrete Algorithms. 311–320.
[18] Scellato, S., Mascolo, C., Musolesi, M., and Latora, V. 2010. Distance matters: Geo-social metrics for online social networks Proceedings of the 3rd conference on Online social networks. 1–8.
[19] Watts, D. J. and Strogatz, S. 1998. Collective dynamics of small-world networks. Nature. 393, 6684, 440–442.