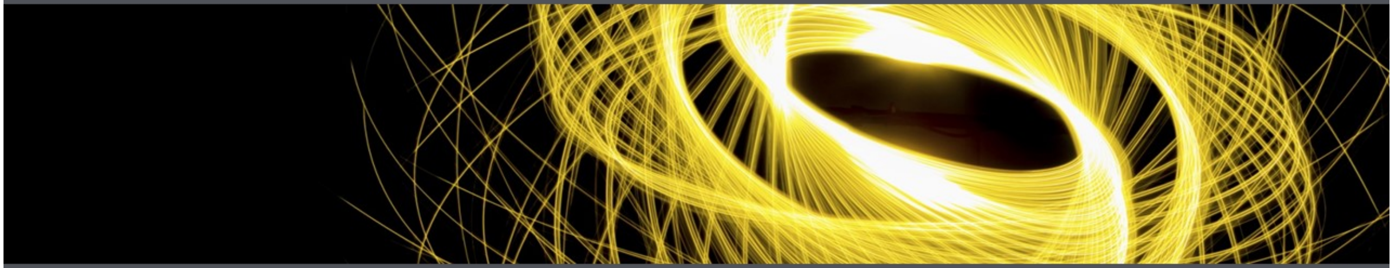
Department of Computer Science

University of Reading

CSMAD21 – Applied Data Science with Python



Networks

University of Reading

Lecture Objectives

- Understand and identify the definition, elements and representation networks.
- Calculate, understand and analyse the statistics of a network.
- Implement NetworkX to visualise and calculate statistics of networks.

Outline

- Networks Definition
- Representation
- NetworkX
- Creating Networks in NerworkX
- Methods
- Properties of nodes and networks
- Statistics
  - Degree Distributions
  - Clustering Coefficient
  - Betweenness Centrality
  - Assortativity
- Random Graphs
- Visualising Networks
- Force direct layout
- Other tools
- Further Reading
- Summary
- Q&A

Networks Definition

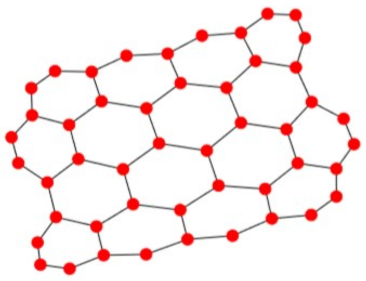
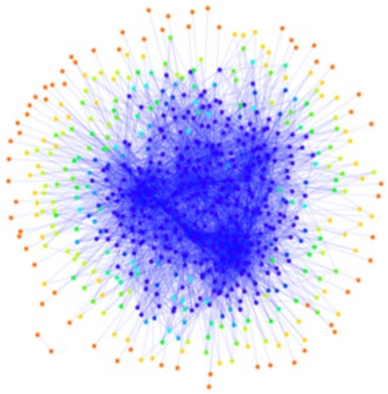
A network or graph $\mathcal{G}$ consists of a set of nodes (or vertices) V and edges E. An edge is a pair of nodes (a, b) denoting the nodes connected by the edge.

$$\mathcal{G} = (V, E)$$

For today we are assuming that the network is *undirected*, meaning that edges have no direction. As a result we consider edges (a, b) and (b, a) to be equivalent.

Networks Definition

- Complex networks are networks that do not have simple structures, but are more complex and harder to model. A lattice (grid) is an example of a simple network, whereas a social network is a complex network.

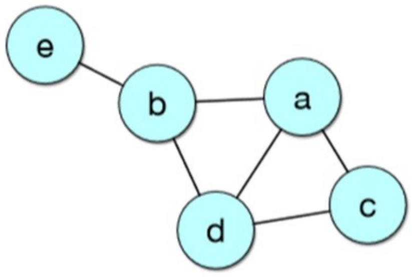



Network Representation – Edge List

Networks can be represented in various different data structures. We can represent networks using a simple list of all the nodes and all the edges in the network:

$$V = \{a, b, c, d, e\}$$

Edges
(a,b)
(a,c)
(a,d)
(b,d)
(b,e)
(c,d)



Network Representation – Edge List

Edge lists

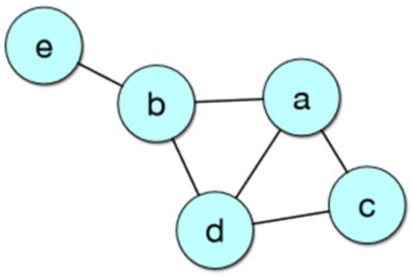
- Good for storing networks in files
- Easy to parse
- Easy to add or delete edges
- Networks are generally sparse - saves space
  - Of $N(N-1)/2$ possible edges only a small fraction generally occur

- Testing if two nodes share an edge means scanning entire list
- Similarly for finding neighbours of a node

Network Representation – Neighbour List

Another data structure that we could use is to build a neighbour list for every node. For every node x we maintain a list of all the y such that $(x, y) \in E$.

Node	Neighbours
a	b,c,d
b	a,d,e
c	a,d
d	a,b,c
e	b



Network Representation – Neighbour List

Neighbour lists

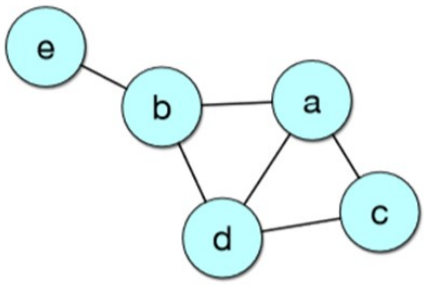
- Very quick to test if nodes share an edge
- Easily find all neighbours of a node
- Good for traversing the network, e.g. in breadth first or depth first search.

- Adding or deleting edges is (slightly) more complex

Network Representation – Adjacency Matrices

An adjacency matrix is a matrix where each row and column corresponds to a node in the network. For each pair of nodes, the corresponding entry in the matrix is 1 if there is an edge between the nodes and 0 otherwise.

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$



Network Representation – Adjacency Matrices

Adjacency matrices

- Testing if two nodes share an edge is simply an array lookup
- Adding or deleting edges is just modifying an array

- If nodes are named we need to maintain mapping between nodes and matrix rows/columns.
- Finding all neighbours of a node means checking every value in a column (or row).
- Can take a lot of space. How much memory would a network with 1 million nodes take?

The Social Network

- A social network data set where dolphins have links between them if they frequently associated with one another.
- Taken from the Koblenz Network Collection by the University of Koblenz–Landau.

Datasouce: Dolphins network dataset – KONECT, April 2017. http://konect.uni-koblenz.de/networks/dolphins.

D. Lusseau, K. Schneider, O. J. Boisseau, P. Haase, E. Slooten, and S. M. Dawson. The bottlenose dolphin community of Doubtful Sound features a large proportion of long-lasting associations. Behavioral Ecology and Sociobiology, 54:396–405, 2003.

The Social Network - Dolphins

- To read in the tab separated network file, we need to read each line in the file. To do this we just treat the file object as an iterator.
- Here l will be a string for each line in the file. The split(x) method splits a string into a list of substrings for each occurence of the character x.

In [1]:

```
# Create an empty edge set
edges = set()

with open("Datasets/dolphins.tsv",'r') as f:
    for l in f:
        a,b = l.split("\t")
        e = (int(a),int(b))
        edges.add(e)
```

In [2]:

```
edges
```

Out[2]:

```
{(9, 4),
 (10, 6),
 (10, 7),
 (11, 1),
 (11, 3),
 (14, 6),
 (14, 7),
 (14, 10),
 (15, 1),
 (15, 4),
 (16, 1),
 (17, 15),
 (18, 2),
 (18, 7),
 (18, 10),
 (18, 14),
 (19, 16),
 (20, 2),
```

Creating a neighbour list network representation

In [3]:

```
# Create an empty dictionary
network = {}

for (a,b) in edges:
    #Check if key is in dictionary
    if a in network:
        network[a].add(b)
    else:
        network[a]={b}
    #Check if key is in dictionary
    if b in network:
        network[b].add(a)
    else:
        network[b]={a}

network
```

Out[3]:

```
{55: {2, 7, 8, 14, 20, 42, 58},
 2: {18, 20, 27, 28, 29, 37, 42, 55},
 20: {2, 8, 31, 55},
 42: {2, 10, 14, 55, 58},
 31: {8, 20, 29, 43, 48},
 29: {2, 9, 21, 31, 48},
 10: {6, 7, 14, 18, 33, 42, 58},
 6: {10, 14, 57, 58},
 37: {2, 21, 24, 38, 40, 41, 60},
 24: {37, 46, 52},
 43: {1, 3, 11, 31, 48, 51},
 3: {11, 43, 45, 62},
 41: {1, 8, 15, 16, 34, 37, 38, 53},
 15: {1, 4, 17, 25, 34, 35, 38, 39, 41, 44, 51, 53},
 52: {5, 12, 19, 22, 24, 25, 30, 46, 51, 56},
 58: {6, 7, 10, 14, 18, 40, 42, 49, 55},
 40: {37, 58},
 60: {4, 9, 16, 37, 46},
 51: {15, 17, 21, 34, 43, 46, 52},
 34: {13, 15, 17, 22, 35, 38, 39, 41, 44, 51},
 59: {39},
 39: {15, 17, 21, 34, 44, 45, 53, 59},
 49: {58},
 46: {9, 16, 19, 22, 24, 25, 30, 38, 51, 52, 60},
 18: {2, 7, 10, 14, 23, 26, 28, 32, 58},
 21: {9, 17, 19, 29, 37, 39, 45, 48, 51},
 9: {4, 21, 29, 38, 46, 60},
 44: {15, 30, 34, 38, 39, 47, 54},
 38: {9, 15, 17, 22, 34, 35, 37, 41, 44, 46, 62},
 45: {3, 21, 35, 39},
 17: {15, 21, 34, 38, 39, 51},
 48: {1, 11, 21, 29, 31, 43},
 22: {19, 30, 34, 38, 46, 52},
 19: {16, 21, 22, 25, 30, 46, 52},
 30: {11, 19, 22, 25, 36, 44, 46, 52, 53},
 11: {1, 3, 30, 43, 48},
 8: {20, 28, 31, 41, 55},
 1: {11, 15, 16, 41, 43, 48},
 62: {3, 38, 54},
 50: {35, 47},
 47: {44, 50},
 54: {44, 62},
 57: {6, 7},
 26: {18, 27, 28},
 14: {6, 7, 10, 18, 33, 42, 55, 58},
 56: {16, 52},
 16: {1, 19, 25, 41, 46, 56, 60},
 28: {2, 8, 18, 26, 27},
 12: {52},
 53: {15, 30, 39, 41},
 32: {18},
 61: {33},
 33: {10, 14, 61},
 7: {10, 14, 18, 55, 57, 58},
 5: {52},
 4: {9, 15, 60},
 35: {15, 34, 38, 45, 50},
 25: {15, 16, 19, 30, 46, 52},
 27: {2, 26, 28},
 13: {34},
 23: {18},
 36: {30}}
```

In [4]:

```
for a1, a2 in edges:
    if a1 == 52:
        print(a1,a2)
```

```
52 24
52 51
52 19
52 46
52 12
52 30
52 5
52 25
52 22
```

In [5]:

```
network[52], network[5]
```

Out[5]:

```
({5, 12, 19, 22, 24, 25, 30, 46, 51, 56}, {52})
```

Calculating some statistics

In [6]:

```
##Degree. Number of edges per node
degrees = []

for neighbours in network.values():
    degrees.append(len(neighbours))
```

In [7]:

```python
degrees[:10]
```

Out[7]:

```
[7, 8, 4, 5, 5, 5, 7, 4, 7, 3]
```

- We can see what the largest and smallest numbers of friends a dolphin has are:

In [8]:

```python
print(max(degrees))
print(min(degrees))
```

```
12
1
```

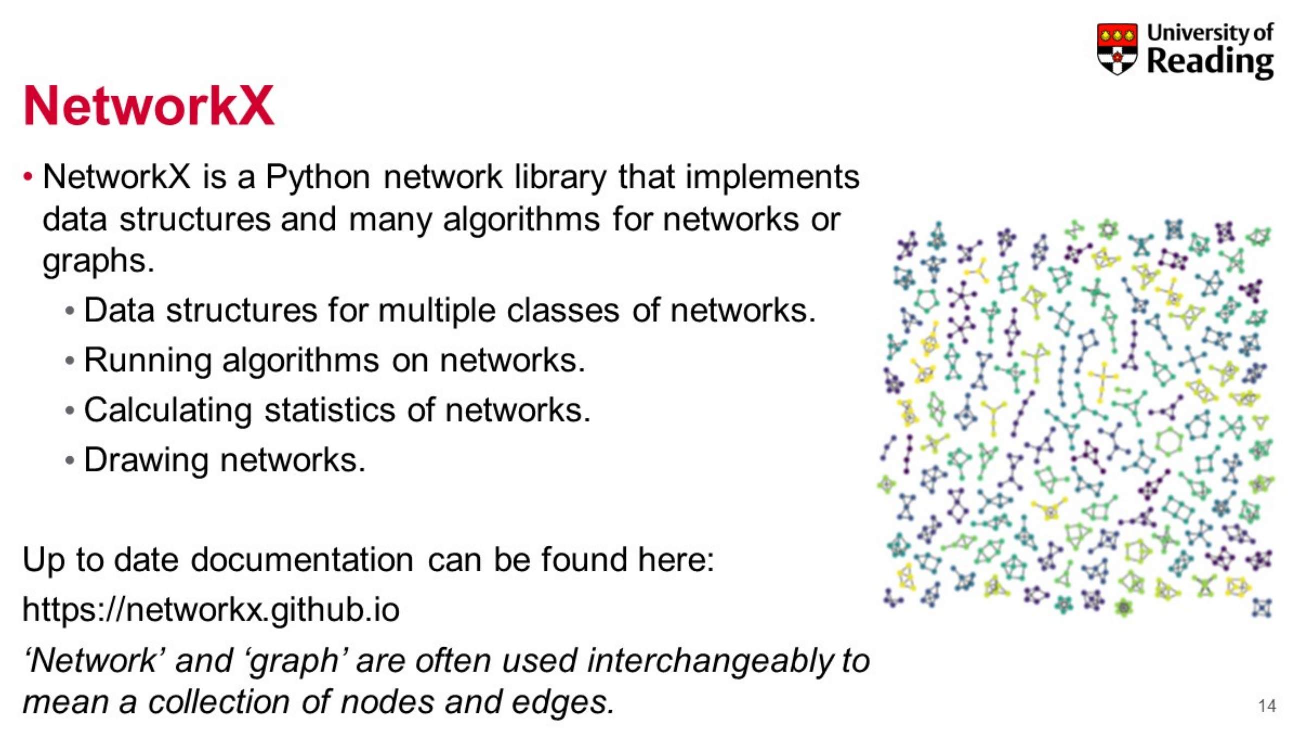
- Taking the average or mean of the degree sequence is easy:

In [9]:

```python
sum(degrees)/len(degrees) # 5.13
```

Out[9]:

```
5.129032258064516
```



University of Reading

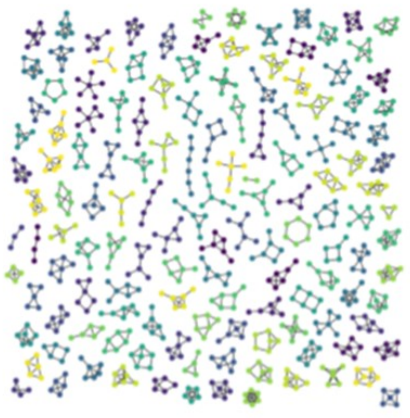
NetworkX

- NetworkX is a Python network library that implements data structures and many algorithms for networks or graphs.
  - Data structures for multiple classes of networks.
  - Running algorithms on networks.
  - Calculating statistics of networks.
  - Drawing networks.

Up to date documentation can be found here:

https://networkx.github.io

'Network' and 'graph' are often used interchangeably to mean a collection of nodes and edges.



14

Creating networks in NetworkX

• We can create a unidirectional network with NetworkX.

Creating networks in NetworkX

In [10]:

```
##We can create an unidirectional network
import networkx as nx
g = nx.Graph()
g.add_node(1)
g.add_node("b")
g.add_edge(1,"b",weight=0.5) # Edges can have attributes
g.add_edge("b",2)
g.add_edge("c",2)
g["b"]
```

Out[10]:

```
AtlasView({1: {'weight': 0.5}, 2: {}})
```

Methods

In [11]:

```
g.edges(), g.number_of_nodes(), g.number_of_edges()
```

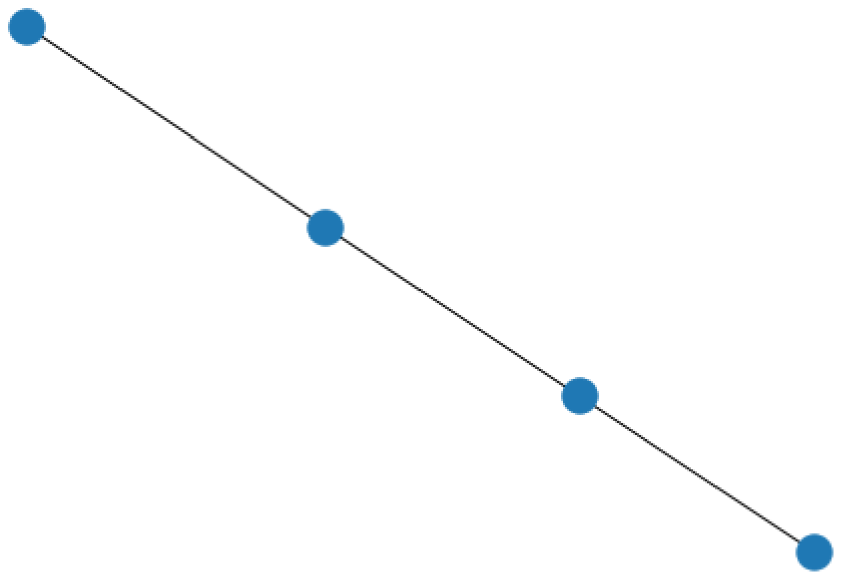
Out[11]:

```
(EdgeView([(1, 'b'), ('b', 2), (2, 'c')]), 4, 3)
```

Initial Visualisation

In [12]:

```
G = g
layout = nx.spring_layout(G)
nx.draw(G)
```



Example with the Dolphins data

In [13]:

```
with open("Datasets/dolphins.tsv") as f:
    el = (list(map(int,l.split())) for l in f)
    g = nx.from_edgelist(el)
```

In [14]:

```
g
```

Out[14]:

```
<networkx.classes.graph.Graph at 0x13e4937cb80>
```

Methods

- For a NetworkX network object g the following methods (and many others) are available:
  - **nodes()** Return a list of the nodes in the network.
  - **edges()** Return a list of the edges in the network.
  - **neighbours(n)** Return the neighbours of node n.
  - **has_node(n)** Test if a node is in the network (you can also use n in g).
  - **has_edge(i,j)** Test if edge (i,j) is in the network
  - **number_of_nodes()** Returns the number of nodes in the network
  - **number_of_edges()** Returns the number of edges in the network

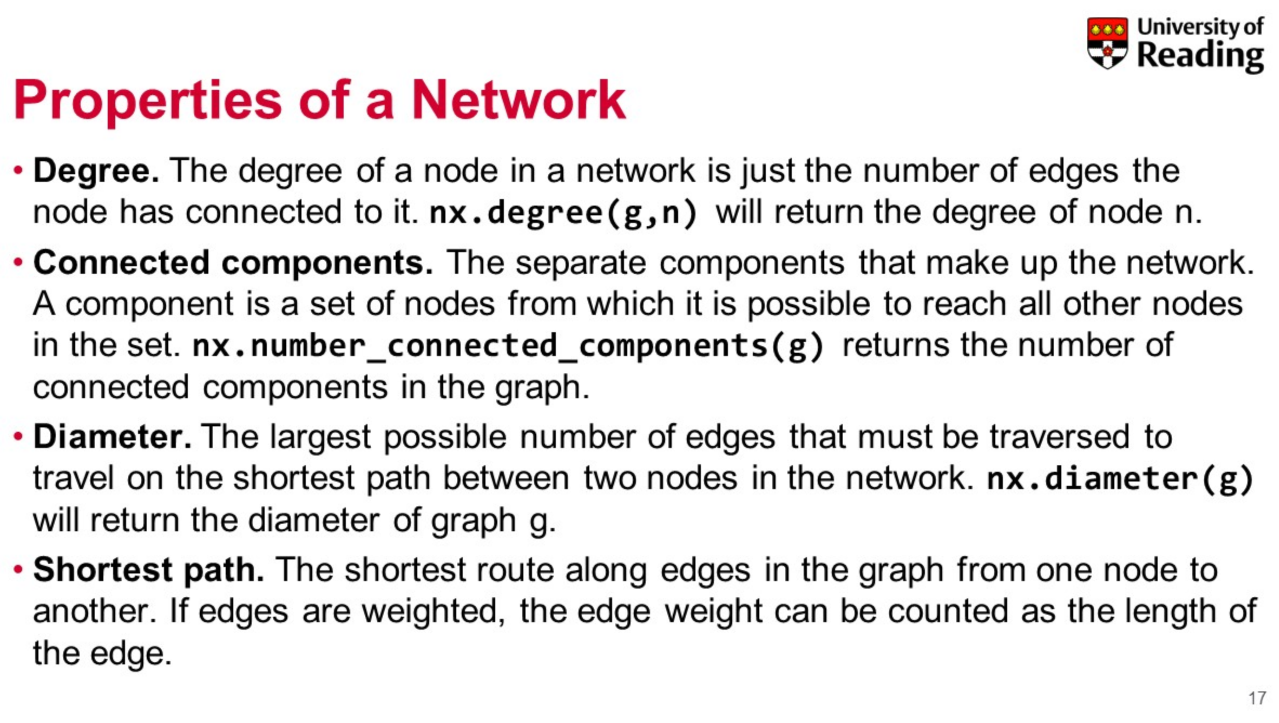
Methods

In [15]:

```
g.number_of_nodes(), g.number_of_edges()
#g.nodes()
#g.edges()
```

Out[15]:

```
(62, 159)
```



Properties of a Network

- **Degree.** The degree of a node in a network is just the number of edges the node has connected to it. **`nx.degree(g,n)`** will return the degree of node n.
- **Connected components.** The separate components that make up the network. A component is a set of nodes from which it is possible to reach all other nodes in the set. **`nx.number_connected_components(g)`** returns the number of connected components in the graph.
- **Diameter.** The largest possible number of edges that must be traversed to travel on the shortest path between two nodes in the network. **`nx.diameter(g)`** will return the diameter of graph g.
- **Shortest path.** The shortest route along edges in the graph from one node to another. If edges are weighted, the edge weight can be counted as the length of the edge.

17

Properties of a Network

In [16]:

```
##Degree
#nx.degree(g,52)
##Connected Components
nx.number_connected_components(g)
##Diameter
nx.diameter(g)
nx.shortest_path(g, 52,5)
```

Out[16]:

```
[52, 5]
```

Degree Distribution

- Counting the number of edges each node has, known as the **degree sequence** of the network, and looking at the frequency with which each occurs, we can generate a **degree distribution**.

Degree distribution

In [17]:

```
nx.degree_histogram(g)
```

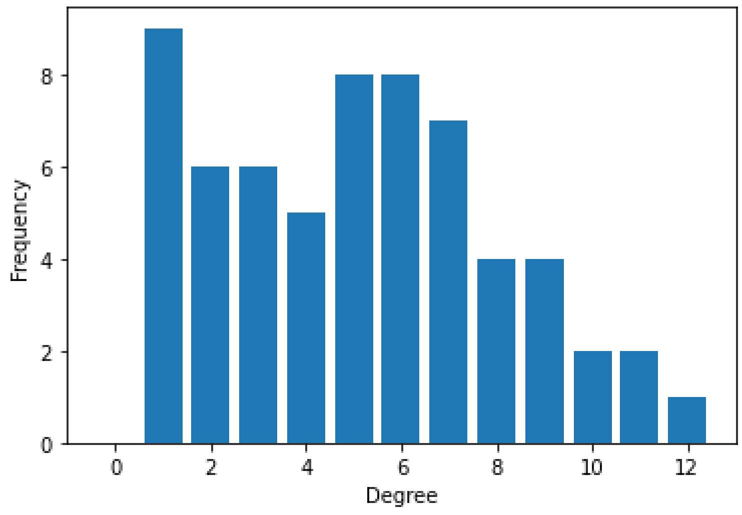
Out[17]:

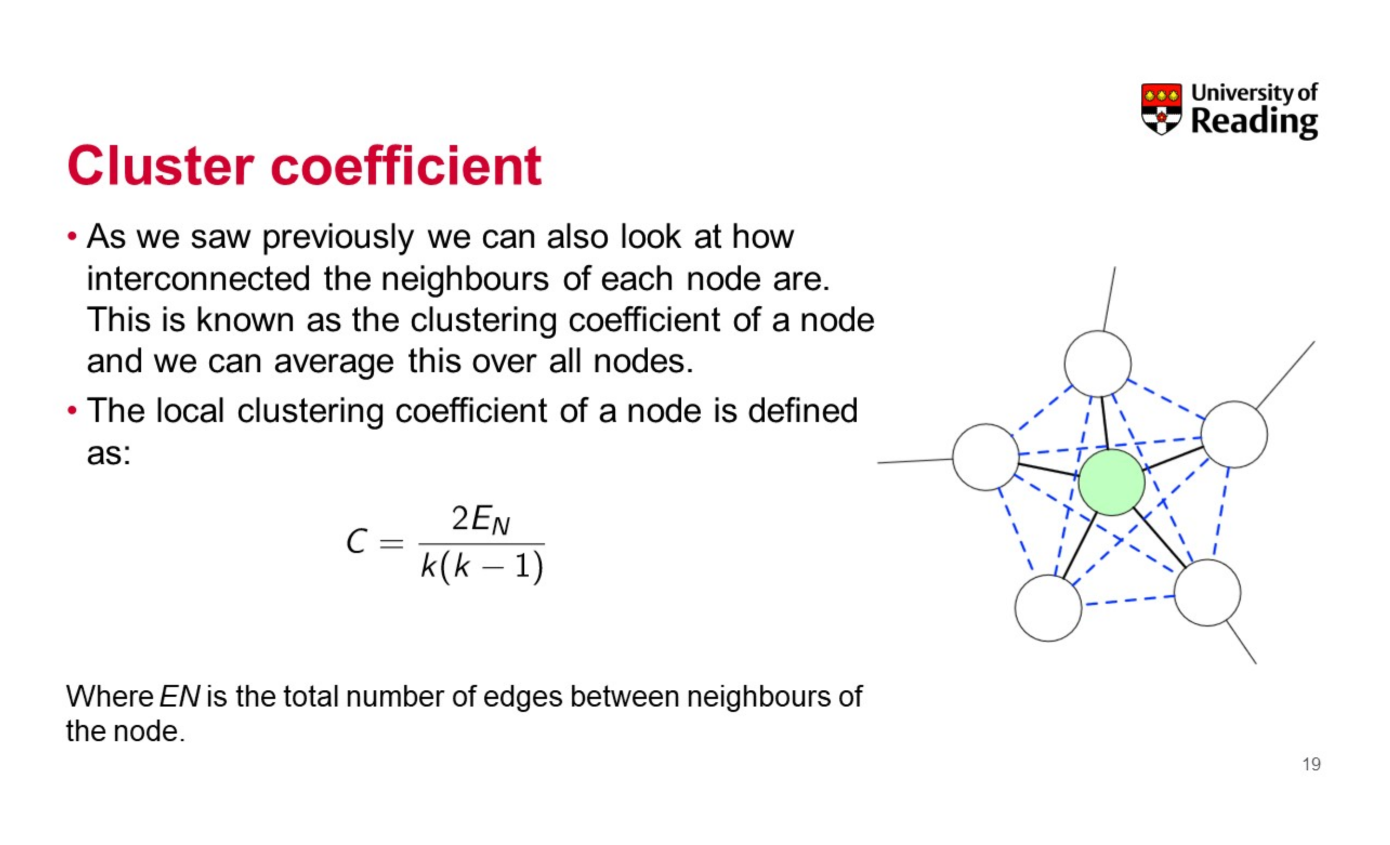
```
[0, 9, 6, 6, 5, 8, 8, 7, 4, 4, 2, 2, 1]
```

In [18]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np

plt.bar(range(0,len(nx.degree_histogram(g))),nx.degree_histogram(g))
plt.xlabel("Degree")
plt.ylabel("Frequency")
plt.show()
```





University of Reading

Cluster coefficient

- As we saw previously we can also look at how interconnected the neighbours of each node are. This is known as the clustering coefficient of a node and we can average this over all nodes.
- The local clustering coefficient of a node is defined as:

$$C = \frac{2E_N}{k(k-1)}$$

Where *EN* is the total number of edges between neighbours of the node.

19

Cluster coefficient

In [19]:

```
cc = nx.clustering(g)
cc
```

Out[19]:

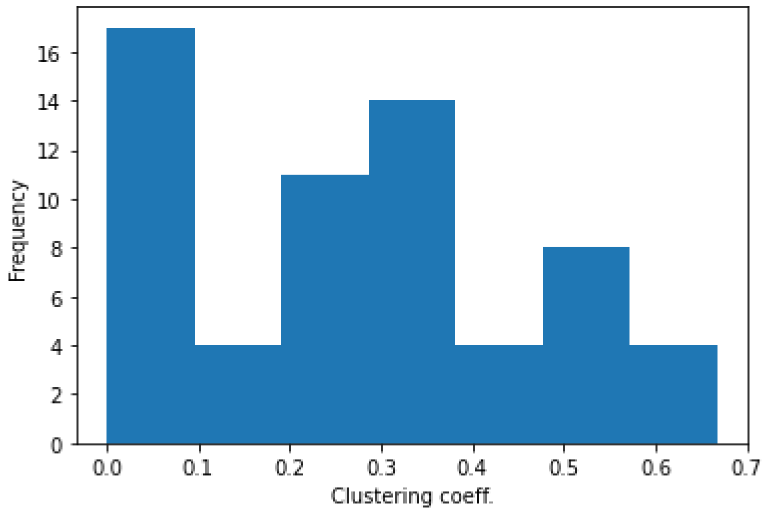
```
{9: 0.26666666666666666,
 4: 0.3333333333333333,
 10: 0.5238095238095238,
 6: 0.5,
 7: 0.5333333333333333,
 11: 0.4,
 1: 0.3333333333333333,
 3: 0.16666666666666666,
 14: 0.5,
 15: 0.25757575757575757,
 16: 0.23809523809523808,
 17: 0.6,
 18: 0.2222222222222222,
 2: 0.14285714285714285,
 19: 0.5238095238095238,
 20: 0.5,
 8: 0.2,
 21: 0.1388888888888889,
 22: 0.5333333333333333,
 23: 0,
 25: 0.5333333333333333,
 26: 0.6666666666666666,
 27: 0.6666666666666666,
 28: 0.4,
 29: 0.3,
 30: 0.25,
 31: 0.3,
 32: 0,
 33: 0.3333333333333333,
 34: 0.3333333333333333,
 13: 0,
 35: 0.3,
 36: 0,
 37: 0.047619047619047616,
 24: 0.3333333333333333,
 38: 0.23636363636363636,
 39: 0.2857142857142857,
 40: 0,
 41: 0.25,
 42: 0.6,
 43: 0.3333333333333333,
 44: 0.23809523809523808,
 45: 0.16666666666666666,
 46: 0.3090909090909091,
 47: 0,
 48: 0.4,
 50: 0,
 51: 0.23809523809523808,
 52: 0.24444444444444444,
 5: 0,
 12: 0,
 53: 0.3333333333333333,
 54: 0,
 55: 0.38095238095238093,
 56: 0,
 57: 0,
 58: 0.3611111111111111,
 49: 0,
 59: 0,
 60: 0.3,
 61: 0,
 62: 0}
```

In [20]:

```
plt.hist(list(cc.values()),bins=7)
plt.ylabel("Frequency")
plt.xlabel("Clustering coeff.")
```

Out[20]:

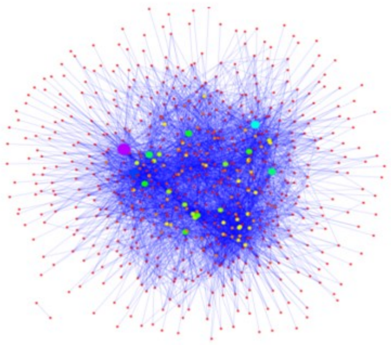
```
Text(0.5, 0, 'Clustering coeff.')
```

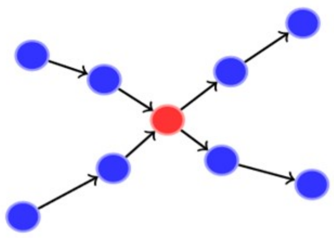


University of Reading

Betweenness Centrality

- Betweenness centrality is a measure of the importance of a node in the network in terms of the possible paths through the network. It is calculated as the fraction of all shortest paths in the network that pass through a node.





- **Nodes with a high betweenness centrality often connect separate regions of the network, and so are generally important in the function of the network.**

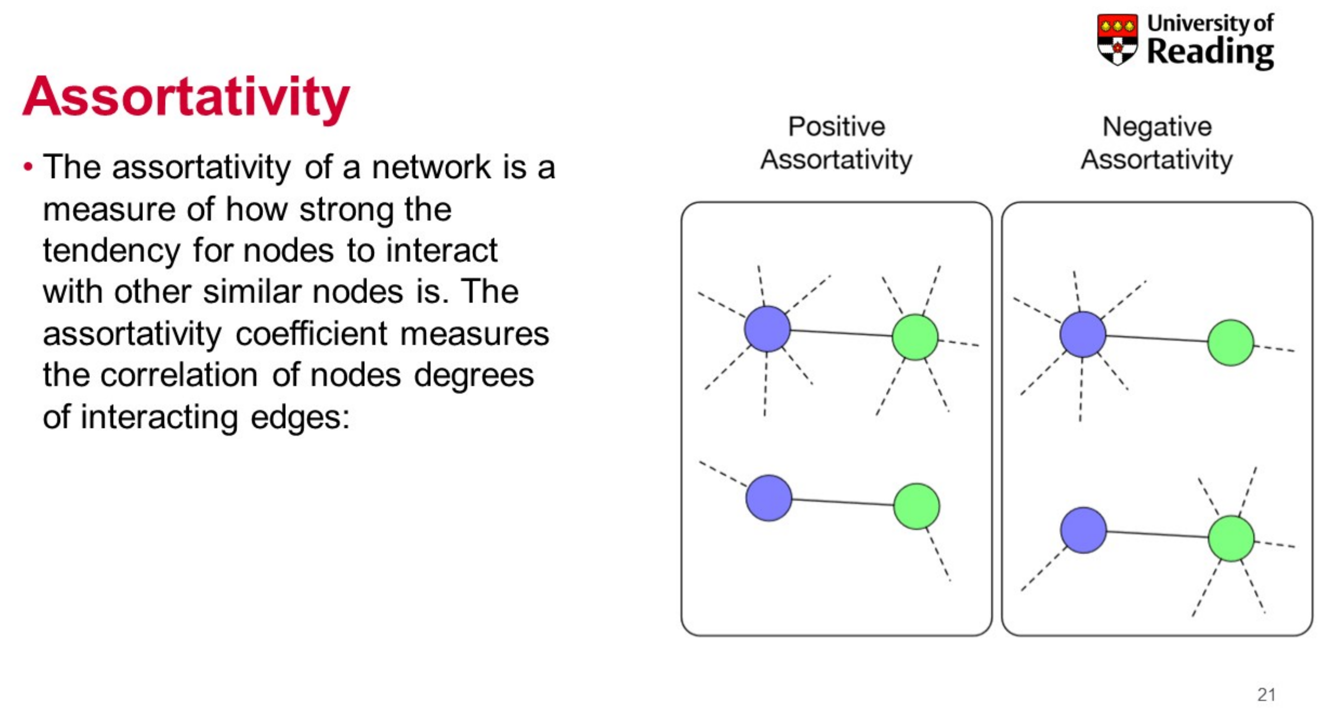
Betweenness Centrality

In [21]:

```
nx.betweenness_centrality(g)
```

Out[21]:

```
{9: 0.022365737598409235,
 4: 0.0023737965131407756,
 10: 0.02089438036159347,
 6: 0.004380300179480508,
 7: 0.029372536747686685,
 11: 0.016092020911693046,
 1: 0.01908259621374376,
 3: 0.00907281243346817,
 14: 0.0528463284386915,
 15: 0.06197200484885411,
 16: 0.03329222098223321,
 17: 0.0033047098620869104,
 18: 0.11430016291546968,
 2: 0.213324435532811,
 19: 0.0148548997169549,
 20: 0.013314394166853183,
 8: 0.11823861926938342,
 21: 0.10264573972090967,
 22: 0.012700653930162124,
 23: 0.0,
 25: 0.0073830434896008665,
 26: 0.0016441148408361526,
 27: 0.004362477231329691,
 28: 0.029236860493157976,
 29: 0.06675695466395656,
 30: 0.06552928249649563,
 31: 0.03305046077177225,
 32: 0.0,
 33: 0.03278688524590164,
 34: 0.057166440117259784,
 13: 0.0,
 35: 0.032694759702956426,
 36: 0.0,
 37: 0.24823719602893804,
 24: 0.04218278563875618,
 38: 0.13856978865859435,
 39: 0.04535223883584539,
 40: 0.070516778539934,
 41: 0.14314951834261747,
 42: 0.023251600670186168,
 43: 0.029157951944837193,
 44: 0.06283060442896508,
 45: 0.012037805849281259,
 46: 0.040670440596470174,
 47: 0.003008466942893172,
 48: 0.0232014761195089,
 50: 0.0009289617486338798,
 51: 0.03341103492742837,
 52: 0.08467725475022554,
 5: 0.0,
 12: 0.0,
 53: 0.0192343448900826,
 54: 0.0011930783242258653,
 55: 0.09912164676351938,
 56: 0.0008774695250105086,
 57: 0.0001366120218579235,
 58: 0.08420468343495603,
 49: 0.0,
 59: 0.0,
 60: 0.020332774690384588,
 61: 0.0,
 62: 0.014194982613015397}
```



University of Reading

Assortativity

- The assortativity of a network is a measure of how strong the tendency for nodes to interact with other similar nodes is. The assortativity coefficient measures the correlation of nodes degrees of interacting edges:

Positive Assortativity
Negative Assortativity

21

Assortavility

In [22]:

```
nx.degree_assortativity_coefficient(g)
```

Out[22]:

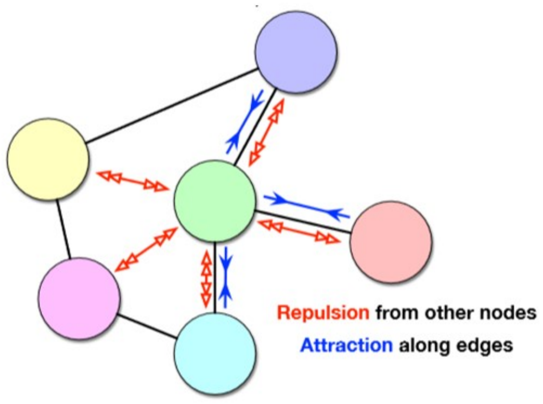
```
-0.04359402821531252
```

Visualising networks

- To visualise networks we need to assign each node a coordinate in two dimensions. There are many different ways of doing this, and the appropriate method to use depends on the structure of the network.
- One way of doing this is to apply force directed layout. For each node in the network, nodes are moved based on forces applied by the other nodes of the network.

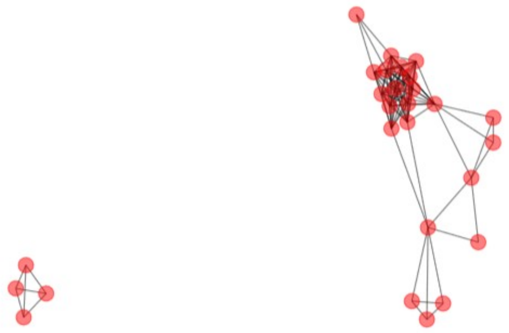
Visualising networks – Force direct layout

- A node is repelled by other nodes, but attracted towards neighbouring nodes in the graph. The overall force to be applied to the node can then be calculated, and it is moved.
- This is done over many iterations, allowing large movements initially and then restricting the magnitude of the distance moved by the node.



Visualising networks

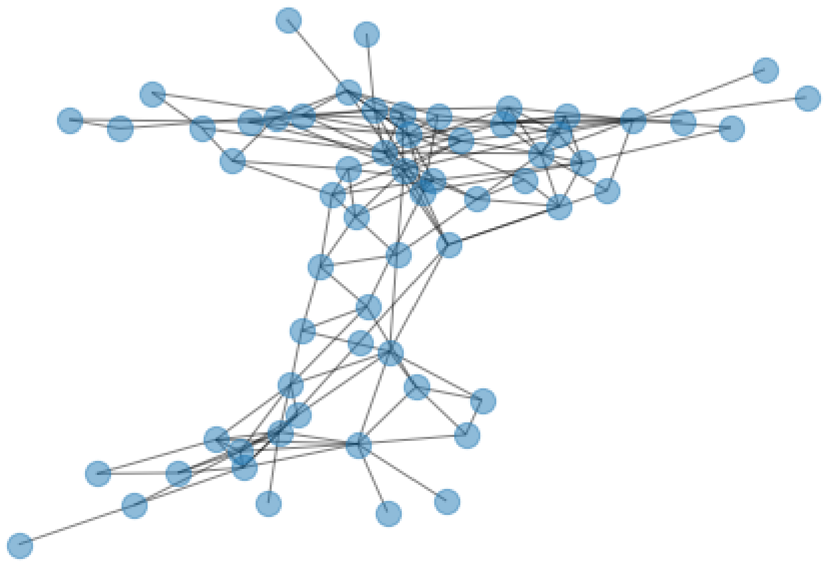
- NetworkX can generate locations for each node applying a force directed layout. This uses the method developed by Thomas Fruchterman and Edward Reingold in the paper Graph drawing by force-directed placement (https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380211102).



Visualising Networks

In [23]:

```
G=g
layout = nx.spring_layout(G)
nx.draw(G,pos=layout,node_size=150,alpha=0.5)
```



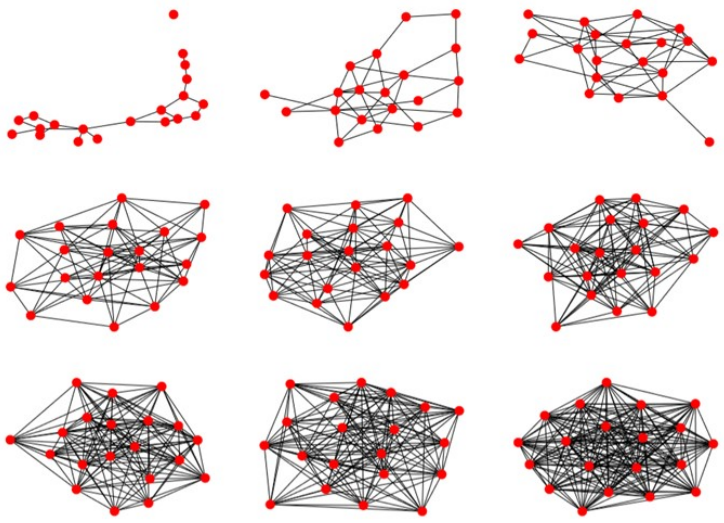
Random Graphs

- Erdős–Rényi random graph models are named after Paul Erdős and Alfréd Rényi who developed the model in the 1950s. They generate random undirected graphs with n nodes.
- The model has a parameter p giving the probability of each edge in the graph being present. That is, for every pair of nodes i and j in the graph, there is a edge between them with probability p.
- Since an undirected graph of n nodes there are $t = \frac{n(n-1)}{2}$
- Total possible edges, forming an edge between them with probability p gives a network or graph with a binomial distribution of the number of edges:

$$|E| \sim \mathrm{Binom}(\tfrac{n(n-1)}{2}, p)$$

Random Graphs

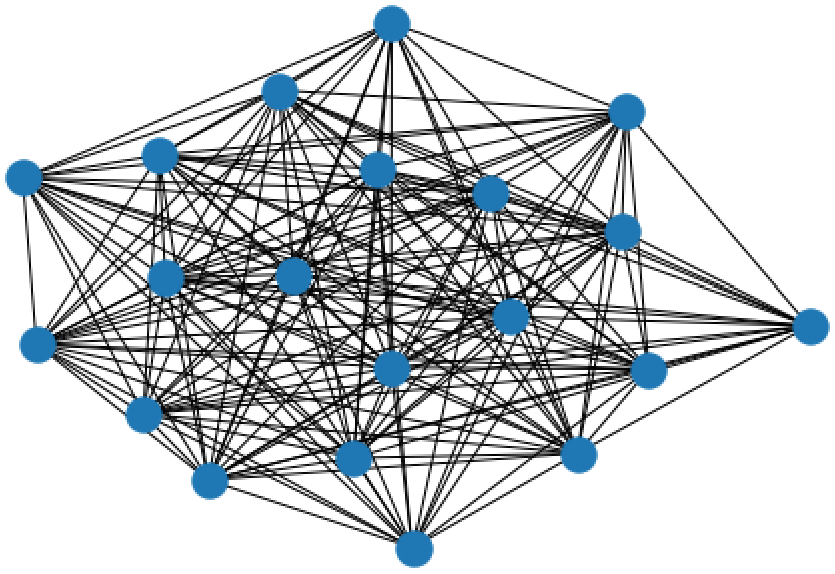
- N = 20, p =0.1 …0.9



Radnom Graphs

In [24]:

```
plt.clf()
ger = nx.fast_gnp_random_graph(20,9/10)
nx.draw(ger)
```



Degree distributions

For some classes of complex network, we often see a degree distribution with a heavy tail:

```
with open("soc-google-plus.tsv") as f:
    el = (list(map(int,l.split())) for l in f)
    G = nx.from_edgelist(el)
ddist = nx.degree_histogram(G)[1:] # Skip 0 degree nodes
plt.loglog(range(1,len(ddist)+1),ddist,'o')
```

Log plots

Sometimes when values vary over different orders of magnitude (e.g. some are in the range 1-10, some are 10000-100000) it can be useful to use the logarithm of values on an axis. *Note that for a box plot this will change the range of the whiskers.* In `matplotlib` this can be done using `plt.semilogx`, `plt.semilogy` or `plt.loglog`.

In [25]:

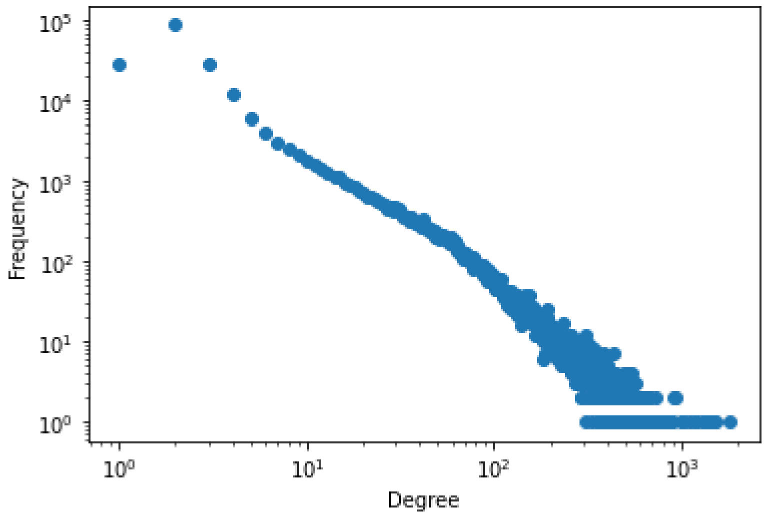
```
with open("Datasets/soc-google-plus.tsv") as f:
    el = (list(map(int,l.split())) for l in f)
    G = nx.from_edgelist(el)
```

In [26]:

```
ddist = nx.degree_histogram(G)[1:] # Skip 0 degree nodes
plt.loglog(range(1,len(ddist)+1),ddist,'o')
plt.xlabel("Degree")
plt.ylabel("Frequency")
```

Out[26]:

```
Text(0, 0.5, 'Frequency')
```

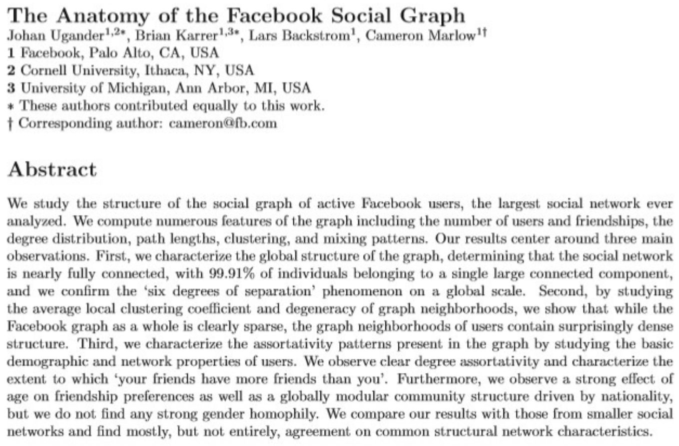


Other Tools

- There are many other tools for visualising networks, a few being:
  - Cytoscape – open source software for network visualisation and analysis, with functionality for analysing biological networks.
  - Gephi – open source network visualisation software.
  - Graphviz – a set of tools for generating layouts and visualisations of graphs. Available as command line tools.

Further Reading

- A nice summary of some interesting properties of the Facebook social network:
- The Anatomy of the Facebook Social Graph (https://arxiv.org/abs/1111.4503)



The Anatomy of the Facebook Social Graph

Johan Ugander[1,2*], Brian Karrer[1,3*], Lars Backstrom[1], Cameron Marlow[1†]

1 Facebook, Palo Alto, CA, USA

2 Cornell University, Ithaca, NY, USA

3 University of Michigan, Ann Arbor, MI, USA

∗ These authors contributed equally to this work.

† Corresponding author: cameron@fb.com

Abstract

We study the structure of the social graph of active Facebook users, the largest social network ever analyzed. We compute numerous features of the graph including the number of users and friendships, the degree distribution, path lengths, clustering, and mixing patterns. Our results center around three main observations. First, we characterize the global structure of the graph, determining that the social network is nearly fully connected, with 99.91% of individuals belonging to a single large connected component, and we confirm the 'six degrees of separation' phenomenon on a global scale. Second, by studying the average local clustering coefficient and degeneracy of graph neighborhoods, we show that while the Facebook graph as a whole is clearly sparse, the graph neighborhoods of users contain surprisingly dense structure. Third, we characterize the assortativity patterns present in the graph by studying the basic demographic and network properties of users. We observe clear degree assortativity and characterize the extent to which 'your friends have more friends than you'. Furthermore, we observe a strong effect of age on friendship preferences as well as a globally modular community structure driven by nationality, but we do not find any strong gender homophily. We compare our results with those from smaller social networks and find mostly, but not entirely, agreement on common structural network characteristics.

Summary

- We can calculate statistics that tell us about the structure of networks.
- Properties of nodes in the network may give us some idea about their function. matplotlib is important to know because it forms the basis of most plotting in Python.
- Different network representations are good for different things

Questions