Table of Contents

CS4423-Networks: Week 10 (19+20 March 2025)

Part 2: Clustering

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/#Week10

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

Modules for this notebook

In [1]:
```
import networkx as nx
import numpy as np
opts = { "with_labels": True,  "node_color": "#004225", "font_color": "white" } # Iri

import random   # some random number generators:random, random_choices
import statistics  # e.g., mean of entries in a list
import math     # for comb (=binomial coef)
import matplotlib.pyplot as plt

np.set_printoptions(precision=2)    # just display arrays to 2 decimal places
np.set_printoptions(suppress=True)
```

Giant Components (again)

In Part 1, we learned about the following result:

Suppose $p(n) = cn^{-1}$ for some positive constant c. (Then the average degree $\langle k \rangle = pn = c$ remains fixed as $n \to \infty$.)

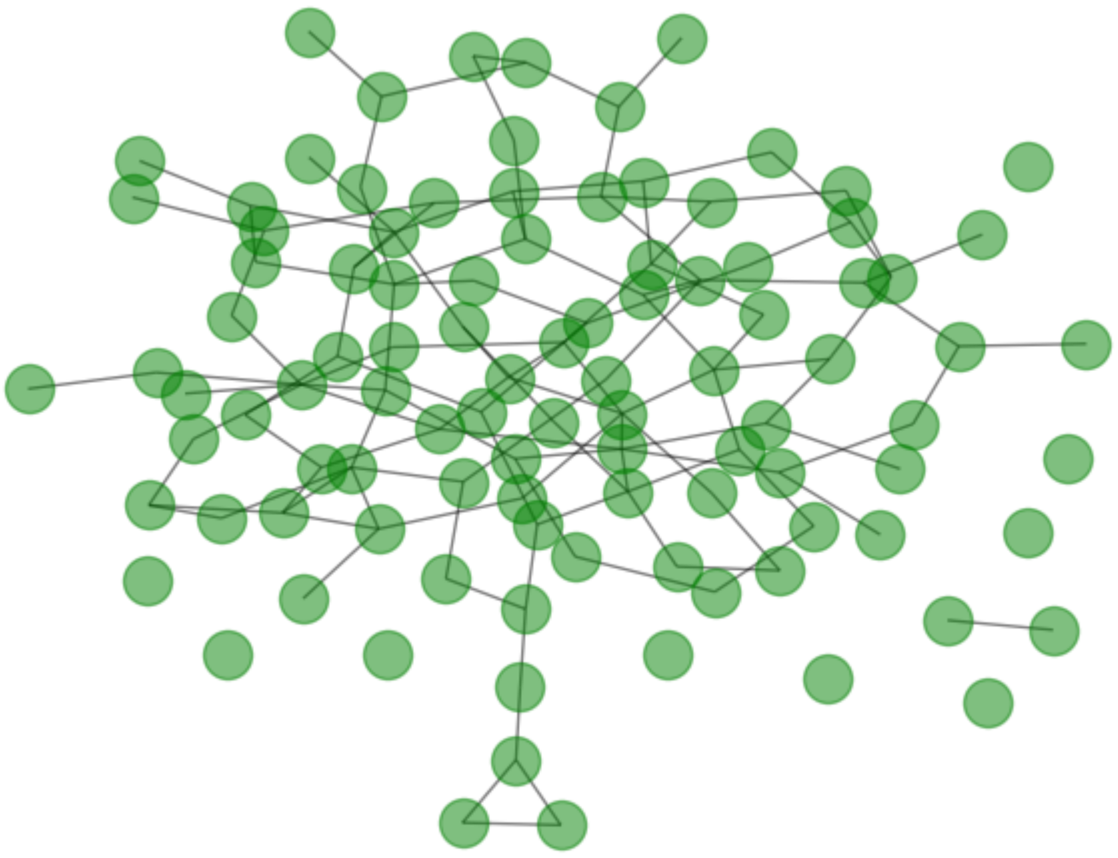
Theorem (Erdős-Rényi).

- If $c < 1$ the graph contains many small components, orders bounded by $O(\ln n)$.
- If $c = 1$ the graph has large components of order $O(n^{2/3})$.
- If $c > 1$ there is a unique **giant component** of order $O(n)$.

There are a few ways we can verify this. One is to fix n, and construct a graph for values of p that correspond to these three ranges:

In [2]:
```
n=100
p=2/n    # take p=0.5/n, 1/n, and 2/m
G = nx.gnp_random_graph(n, p, seed=6)
pos = nx.nx_agraph.graphviz_layout(G, prog="neato", args="")
nx.draw(G, pos, alpha=0.5, node_color="green", with_labels=False)
S = len(max(nx.connected_components(G), key=len))
print(f"p={p}, largest component has {S} nodes")
```

```
p=0.02, largest component has 89 nodes
```



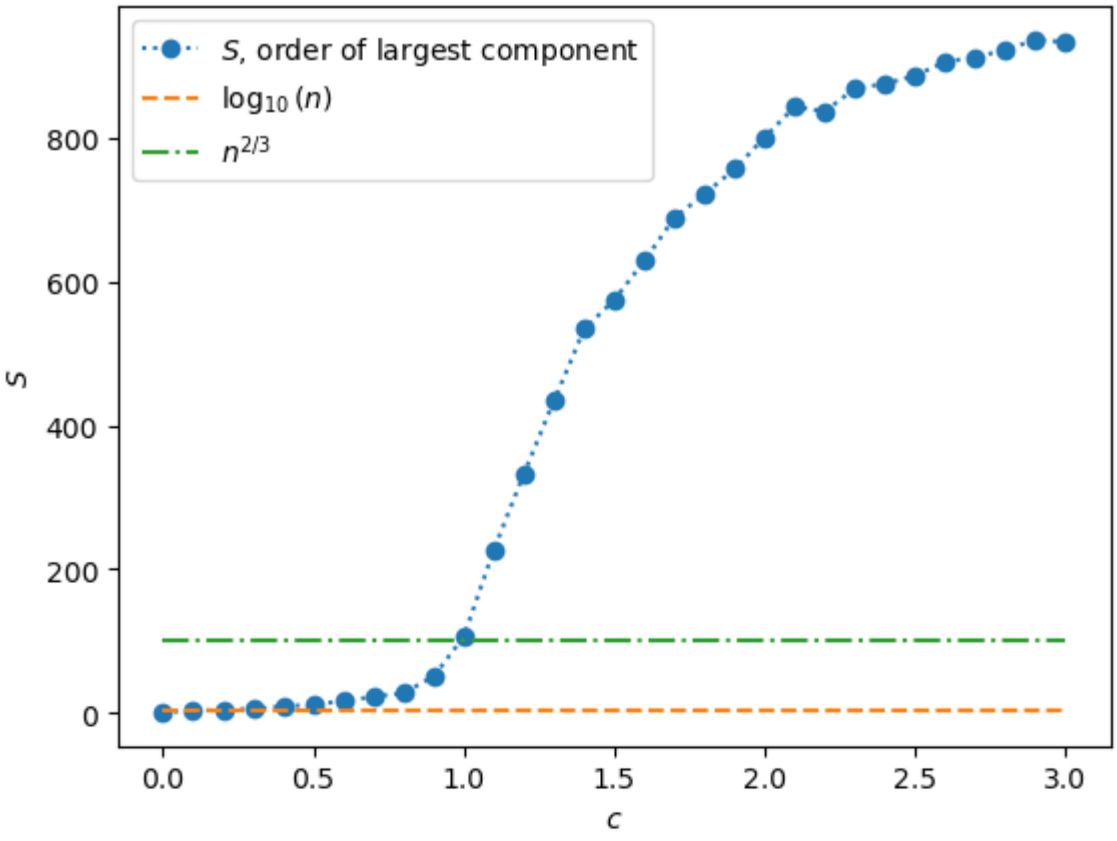
Another way, is to compute the order the largest component for a range of values of c. Note: this test might take about 30 seconds to run.

In [3]:
```
n = 1000    # order of the graphs
c = np.linspace(0, 3,31)  # range of values of c
Ps = (1/n)*c     # corresponding probabilties
```

```
S = []
Runs = 5      # Average over 5 graphs
for p in Ps:
    Ave = 0
    for i in range(Runs):
        G = nx.gnp_random_graph(n,p)
        Ave += len(max(nx.connected_components(G), key=len))
    S += [Ave/Runs]
```

```
In [4]: plt.plot(c, S, ':o', label=r'$S$, order of largest component')
        plt.plot(c, c*0 + math.log10(n),  '--', label=r'$\log_{10}(n)$')
        plt.plot(c, c*0 + n**(2/3),  '-.', label=r"$n^{2/3}$")
        plt.xlabel(r'$c$')
        plt.ylabel(r'$S$')
        plt.legend()
```

```
Out[4]: <matplotlib.legend.Legend at 0x7f92c81d2f90>
```



Characteristic Path Length

We learned in Part 1 that there are several measures of "Small Worldedness", that include

- Small Characteristic Path Length (**CPL**)
- High Clustering (not yet defined)

We asserted in that class that graphs in the G_{ER} models tend to have small Characteristic Path Length, but not high clustering.

Recall: the **characteristic path length** L of G is the average distance between pairs of distinct nodes,

$$L = \frac{1}{n(n-1)} \sum_{i \neq j} d_{ij}.$$

CLP in $G_{ER}(n, m)$

Fact (noted in Part 1): The characteristic path length of a random network in $G_{ER}(n, m)$, or $G_{ER}(n, p)$ is

$$L = \frac{\ln n}{\ln\langle k\rangle}.$$

where $\langle k\rangle$ is the average degree of the network.

Consider $G_{ER}(n, m)$, where we know $\langle k\rangle = 2m/n$.

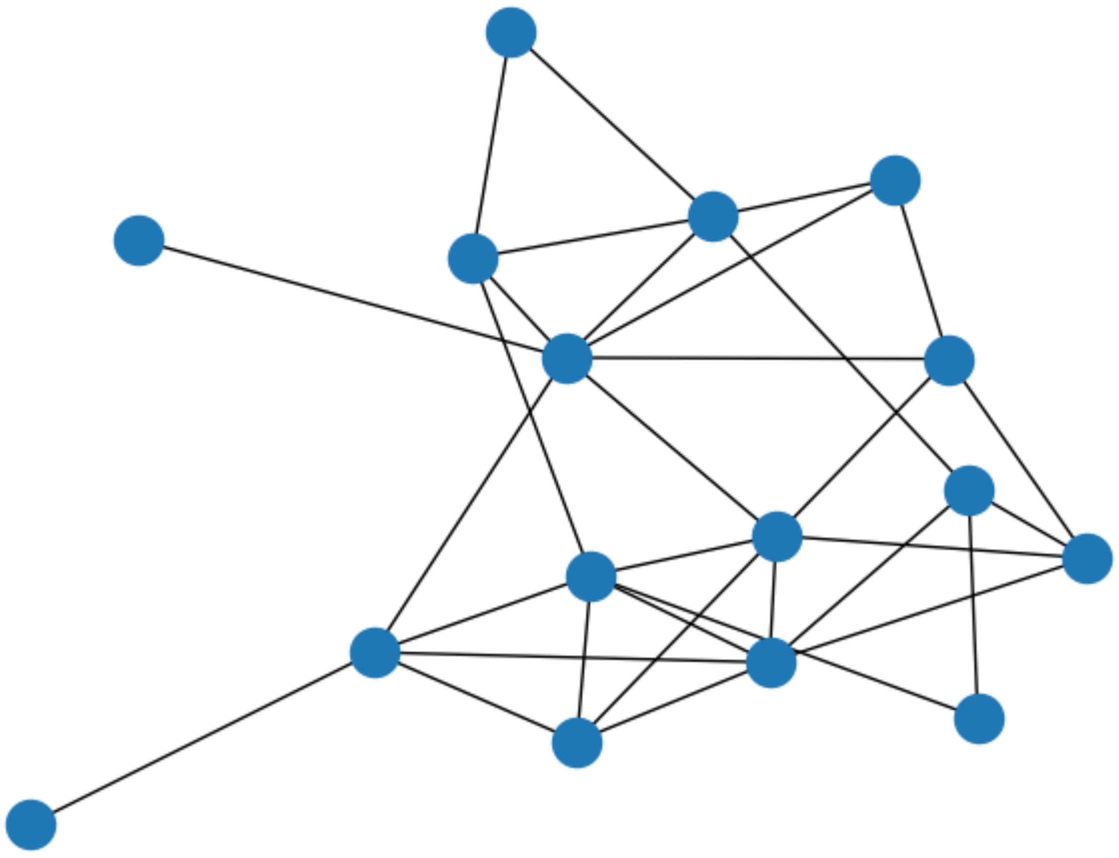
So if $n = 16$ and $m = 32$, then the average node degree in $G(n, m)$ is $\langle k\rangle = \frac{2m}{n} = 4$, and, approximately, $L = \frac{\log_2 16}{\log_2 4} = 2$.

In `networkx`, this is called `average_shortest_path_length`. However, it can only be applied to connected graphs. So, let's choose G from $G_{ER}(16, 32)$ a connected graph:

In [5]:
```python
n = 16
m = 32
G = nx.gnm_random_graph(n,m)
count=0
while (not nx.is_connected(G)):
    count +=1
    G = nx.gnm_random_graph(n,m)
print(f"FYI, took {count} iterations to get a connectted G")
L = nx.average_shortest_path_length(G)
print(f"G_ER({n},{m}) has an average shortest path length of {L:.3f}")
```

```
FYI, took 1 iterations to get a connectted G
G_ER(16,32) has an average shortest path length of 2.017
```

In [6]:
```python
nx.draw(G)
```



Computing CPL by hand

Yesterday, we learned about CPL in the context of eccentricity, in turn computed from by the distance matrix $\mathcal{D}$. We can compute $\mathcal{D}$ for G as:

In [7]:
```
dist = dict(nx.shortest_path_length(G))
Dm = [[dist[i][j] for j in range(n)] for i in range(n)]
print(np.array(Dm))
```

```
[[0 3 3 3 2 3 3 3 2 1 3 3 4 2 3 2]
 [3 0 2 2 2 2 2 2 1 2 2 1 1 1 3 2]
 [3 2 0 2 1 2 2 2 1 2 1 2 3 1 1 1]
 [3 2 2 0 2 1 4 3 1 2 3 2 3 3 2 2]
 [2 2 1 2 0 1 3 3 1 1 2 2 3 2 1 1]
 [3 2 2 1 1 0 3 2 2 2 2 1 2 2 1 2]
 [3 2 2 4 3 3 0 2 3 2 2 2 3 1 3 3]
 [3 2 2 3 3 2 2 0 3 2 1 1 2 1 2 3]
 [2 1 1 1 1 2 3 3 0 1 2 2 2 2 2 1]
 [1 2 2 2 1 2 2 2 1 0 2 2 3 1 2 1]
 [3 2 1 3 2 2 2 1 2 2 0 2 3 1 1 2]
 [3 1 2 2 2 1 2 1 2 2 2 0 1 1 2 3]
 [4 1 3 3 3 2 3 2 2 3 3 1 0 2 3 3]
 [2 1 1 3 2 2 1 1 2 1 1 1 2 0 2 2]
 [3 3 1 2 1 1 3 2 2 2 1 2 3 2 0 2]
 [2 2 1 2 1 2 3 3 1 1 2 3 3 2 2 0]]
```

As we know, the **characteristic path length** L is the sum of all entries in $\mathcal{D}$, divided by the number of pairs of distinct nodes $n(n-1)$.

In [8]:
```
cpl = sum([sum(d) for d in Dm])/(n*(n-1))
```

```
print(f"CPL = {cpl:.3f}")
```

```
CPL = 2.017
```

Small World

Definition (Small-world behaviour). A network $G = (X, E)$ is said to exhibit a **small world behaviour** if its characteristic path length L grows proportionally to the logarithm of the number n of nodes of G:

$$L \sim \ln n.$$

In this sense, the ensembles $G(n, m)$ and $G(n, p)$ of random graphs do exhibit small world behavior (as $n \to \infty$).

Transitivity

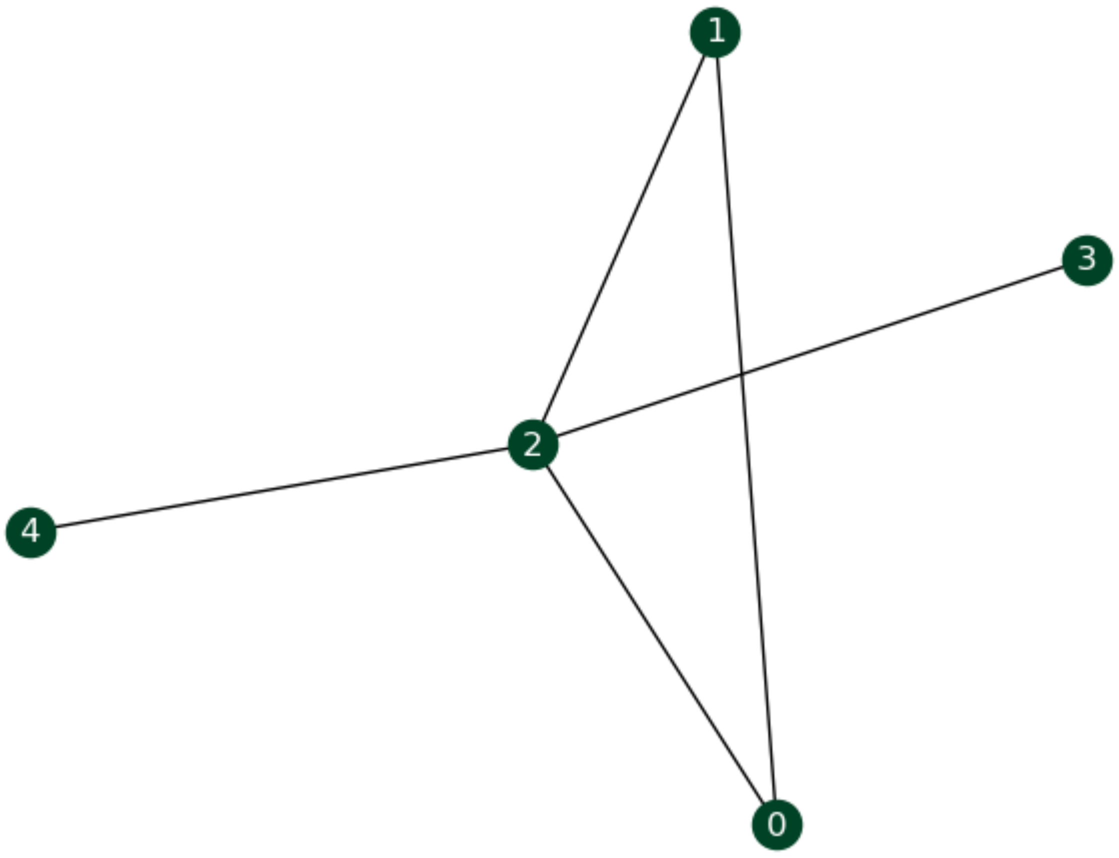
Definition (Graph transitivity). The **transitivity** T of a graph $G = (X, E)$ is the proportion of **transitive** triads, i.e., triads which are subgraphs of **triangles**. This proportion can be computed as follows:

$$T = \frac{3n_{\Delta}}{n_{\wedge}},$$

where n_{Δ} is the number of triangles in G, and $n_{\wedge}$ is the number of triads.

Example:

```
In [9]: G = nx.Graph(((0,1), (1,2), (2,0), (2,3), (2,4)))
        nx.draw(G, **opts)
```



The function `nx.triangles(G)` returns a `python` dictionary reporting for each node of the graph `G` the number of triangles it is contained in.

In [10]:
```
print(nx.triangles(G))
```

```
{0: 1, 1: 1, 2: 1, 3: 0, 4: 0}
```

Overall, each triangle in `G` is thus accounted for 3 times, once for each of its nodes. Hence, the following sum determines this number $3n_\Delta$.

In [11]:
```
NumTriangles = sum(nx.triangles(G).values())/3
print(f"G has {NumTriangles} triangle(s)")
```

```
G has 1.0 triangle(s)
```

As we've seen, the number $n_\wedge$ of triads in `G` can be determined from the graph's degree sequence, as each node of degree k is the central node of exactly $\binom{k}{2}$ triads.

In [12]:
```
NumTriads = 0
for i in G.nodes():
    NumTriads += math.comb(G.degree[i],2)
print(f"G has {NumTriads} triade(s)")
```

```
G has 8 triade(s)
```

In [13]:
```
T = 3*NumTriangles/NumTriads
print(f"Transitivity of G is {T}")
```

```
Transitivity of G is 0.375
```

Of course, there is a built-in function to do this:

```
In [14]: nx.transitivity(G)
```

```
Out[14]: 0.375
```

Transitivity of $G_{ER}(n,p)$

The transitivity of graph in $G_{ER}(n,p)$ is easy to estimate: for every triad, the "third" edge is present with probability p. So

$$T = p,$$

(Or: Compute $3n_\Delta / n_\wedge$ using the explicit formulas from the previous lecture: $n_\Delta = \binom{n}{3}p^3$ and $n_\wedge = 3\binom{n}{3}p^2$.)

Let's check:

```
In [15]: n,p = 100, 0.1
         T=nx.transitivity(nx.gnp_random_graph(n,p))
         print(f"G({n},{p}) has T = {T:.3}")
```

```
G(100,0.1) has T = 0.103
```

Clustering

The concept of **clustering** measures the transitivity of a node, or of an entire graph in a different way.

To define it, we need the concept of an **induced subgraph**.

Induced subgraph

Given $G = (X, E)$ and $Y \subset X$, the induced subgraph of G on Y is the graph $H = \left(Y, E \cap \binom{Y}{2}\right)$.

That is:

- H is a subgraph of G, with node set Y
- H has all possible edges in G for which both nodes are in Y.

(See examples on board).

In `networkx`, we can get an induced subgraph of G, on nodes $\{x, y, z\}$, from `G.subgraph([x,y,z])`

Clustering coefficient

Definition (Clustering coefficient). For a node $i \in X$ of a graph $G = (X, E)$, denote by G_i the subgraph induced on the neighbours of i in G, and by $m(G_i)$ its number of edges.

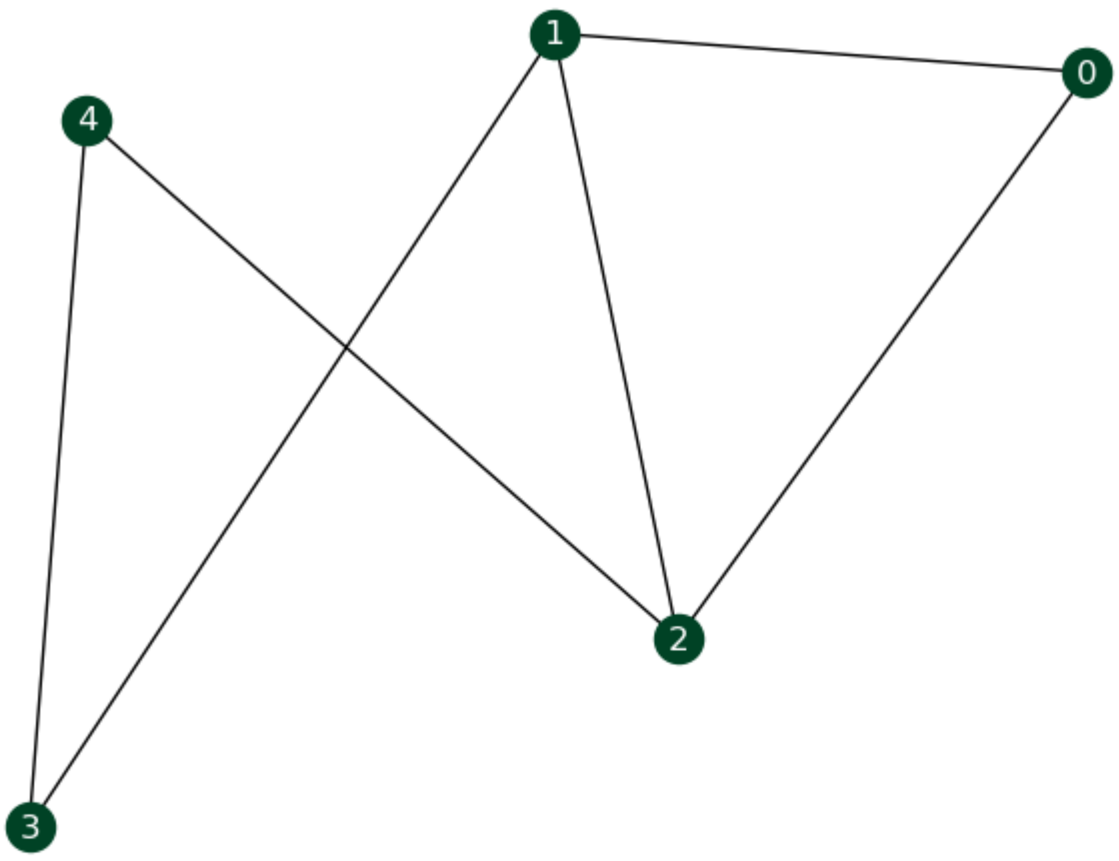
The **node clustering coefficient** c_i of node i is defined as

$$c_i = \begin{cases} \binom{k_i}{2}^{-1} m(G_i), & k_i \geq 2, \\ 0, & \text{else.} \end{cases}$$

That is, the node clustering coefficient measures the proportion of existing edges its **social graph** among the possible edges.

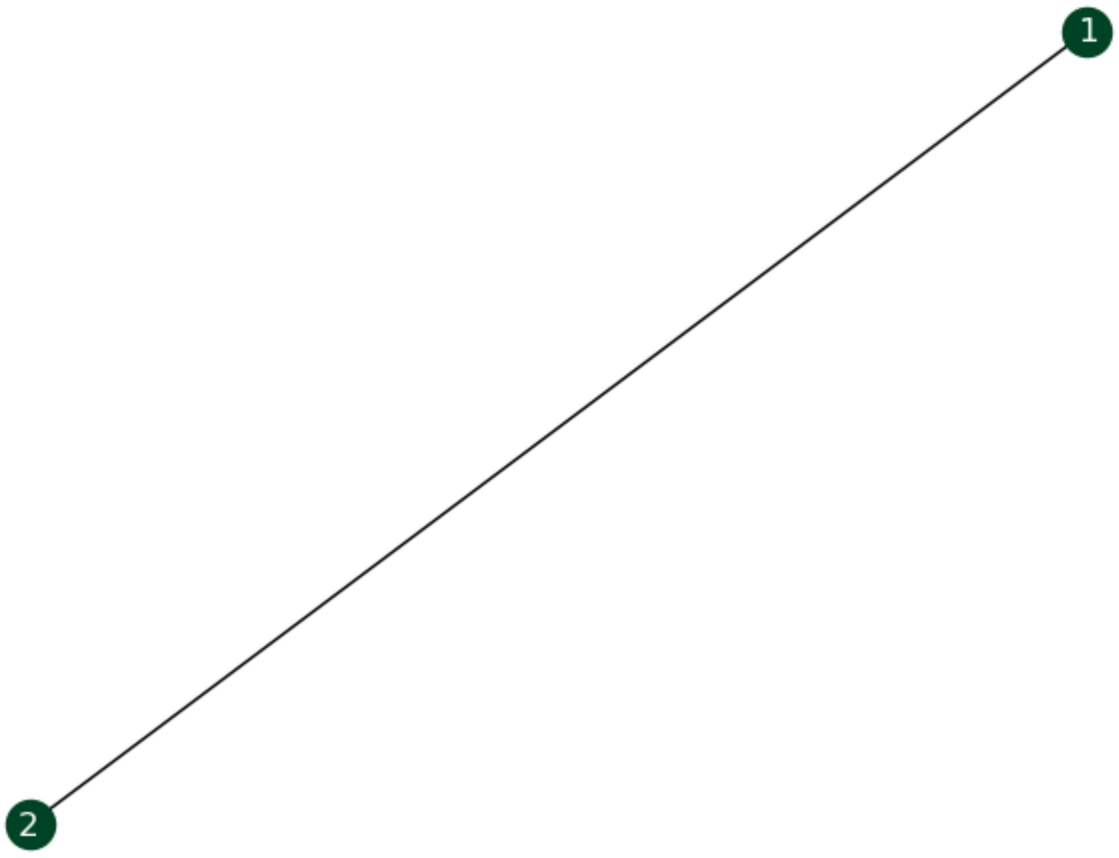
Example:

```
In [16]: G = nx.Graph([(0,1), (0,2), (1,2), (1,3), (2,4), (3,4)])
         nx.draw(G, **opts)
```



The graph induced by the neighbours of 0 is :

```
In [17]: N = nx.neighbors(G, 0)
         S = G.subgraph(list(N))
         nx.draw(S, **opts)
```



Calculate the clustering coefficient for the nodes of G:

```
In [18]: for i in G.nodes():
             k_i = G.degree(i)
             N = nx.neighbors(G, i)
             S = G.subgraph(list(N))
             c_i = S.size()/math.comb(k_i,2)
             print(f"Node {i} has clustering coef {c_i:0.4}")
```

```
Node 0 has clustering coef 1.0
Node 1 has clustering coef 0.3333
Node 2 has clustering coef 0.3333
Node 3 has clustering coef 0.0
Node 4 has clustering coef 0.0
```

Of course, there is a `networkx` function for this:

```
In [19]: nx.clustering(G)
```

```
Out[19]: {0: 1.0, 1: 0.3333333333333333, 2: 0.3333333333333333, 3: 0, 4: 0}
```

Graph Clustering Coefficient

The **graph clustering coefficient** C of G is the average node clustering coefficient,

$$C = \langle c \rangle = \frac{1}{n} \sum_{i=1}^{n} c_i.$$

By definition, $0 \leq c_i \leq 1$ for all nodes $i \in X$, and $0 \leq C \leq 1$.

```
In [20]: nx.average_clustering(G)

Out[20]: 0.3333333333333333
```

Clustering for $G_{ER}(n, p)$

The **node clustering coefficient** of any node i in a $G_{ER}(n, p)$ **random graph** is $c_i = p$. (In any selection of potential edges, by construction a proportion p of them is present in the random graph; this is true in particular for the $\binom{k}{2}$ potential edges between the k neighbors of a node of degree k.)

Thus the **graph clustering coefficient** of a $G_{ER}(n, p)$ **random graph** is

$$C = p.$$

IMPORTANT: Large G_{ER} graphs have few triangles

Note that when $p(n) = \langle k \rangle n^{-1}$ for a fixed expected average degree $\langle k \rangle$ then $C = \langle k \rangle / n \to 0$ for $n \to \infty$. That is **in large G_{ER} random graphs, the number of triangles is negligible**.

In real world networks, one often observes that $C / \langle k \rangle$ does not depend on n (as $n \to \infty$).

Clustering vs Transitivity

For a node $i \in X$, denote by $n_i^{\wedge} = \binom{k_i}{2}$ the number of triads containing i as their central node, and by n_i^{Δ} the actual number of triangles containing i.

Then the node clustering coefficient is $c_i = n_i^{\Delta} / n_i^{\wedge}$, or $n_i^{\Delta} = n_i^{\wedge} c_i$.

Moreover $3n_{\Delta} = \sum_i n_i^{\Delta}$ and $n_{\wedge} = \sum_i n_i^{\wedge}$.

It follows that

$$T = \frac{3n_{\Delta}}{n_{\wedge}} = \frac{1}{n_{\wedge}} \sum_i n_i^{\wedge} c_i$$

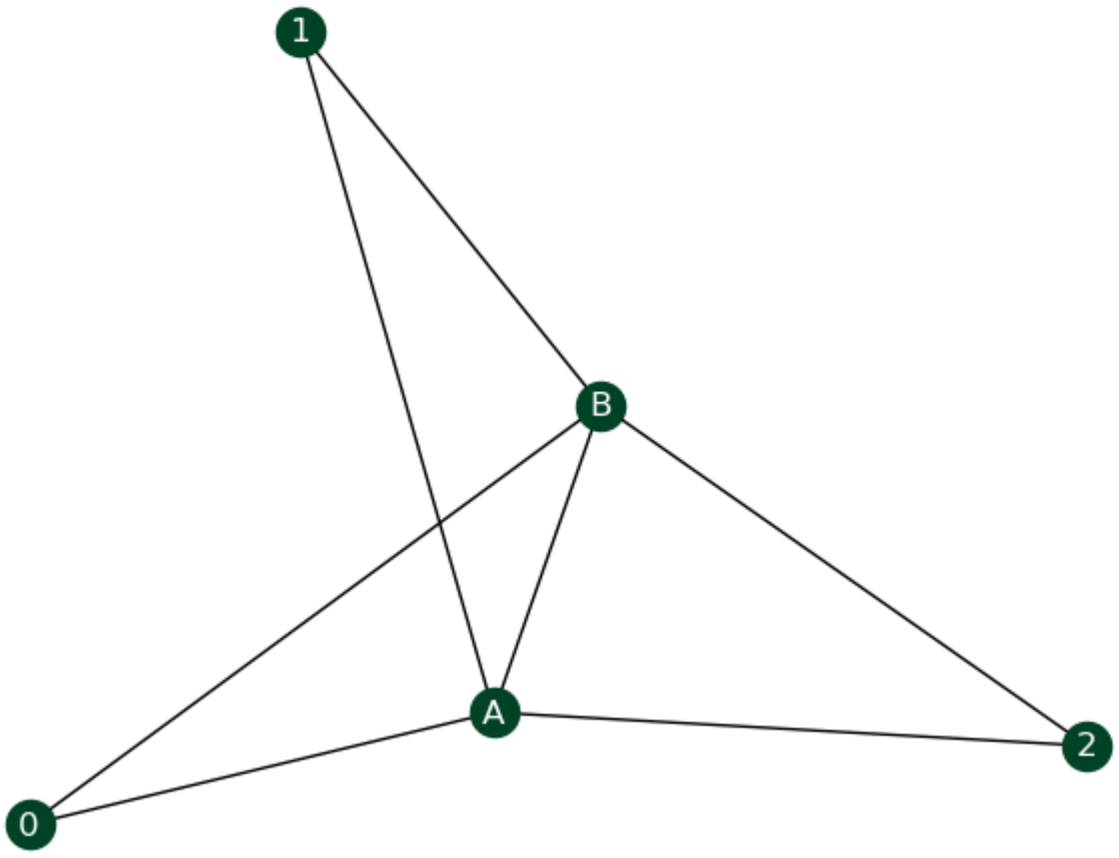
in contrast to

$$C = \frac{1}{n} \sum_i c_i.$$

That is, C is the (plain) **average** of the node clustering coefficients, whereas T is a **weighted average** of node clustering coefficients, giving higher weight to high degree nodes.

The following example illustrates how C and T are different measures. This (very non-random) ensemble of networks has the property that, as $n \to \infty$ here, $T \to 0$ while $C \to 1$.

```
In [21]: n = 3 # try larger and larger n
         G = nx.Graph(["AB"])
         G.add_edges_from([(x, k) for x in "AB" for k in range(n)])
         nx.draw(G, **opts)
```



```
In [22]: nx.average_clustering(G), nx.transitivity(G)
```

```
Out[22]: (0.8, 0.6)
```

- The fact that ER random networks tend to have low transitivity and clustering shows the need of a new kind of (random) network construction that is better at modelling real world networks.
- One idea, developed by Watts and Strogatz in 1998, is to start with some **regular network** that naturally has a **high clustering**, and then to randomly distort its edges, to introduce some **short paths**.

FINISHED HERE THURSDAY