CS4423 - Networks

Prof. Götz Pfeiffer
School of Mathematics, Statistics and Applied Mathematics
NUI Galway

6. Power Laws and Scale-Free Graphs

Lecture 22: Configuration Model

In [1]:
```
import numpy as np
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
```

The Configuration Model

In principle, random graph can be generated in such a way that they have a prescribed degree sequence.

Idea: Choose numbers d_i, $i \in X$, so that $\sum d_i = 2m$ is an even number.

Then regard each degree d_i as d_i **stubs** (half-edges) attached to node i.

Compute a random **matching** of pairs of stubs and build a graph on X with those (full) edges.

Example. Suppose that $X = \{1, 2, 3, 4, 5\}$ and that we want those nodes to have degrees $d_1 = 3$, $d_2 = 2$ and $d_3 = d_4 = d_5 = 1$.

This gives a list of stubs $(1, 1, 1, 2, 2, 3, 4, 5)$ where each node i appears as often as its degree d_i requires.

A random shuffle of that list is $(1, 3, 4, 1, 2, 1, 5, 2)$.

One way to construct a matching is to simply cut this list in half and match entries of the first half with corresponsing entries in the second half.

$$\begin{matrix} 1 & 3 & 4 & 1 \\ 2 & 1 & 5 & 2 \end{matrix}$$

Note that $\sum d_i = 8 = 2m$ yields $m = 4$ edges ...

A Quick Implementation

In [2]:
```
degrees = [3, 2, 1, 1, 1]
```

Recall that, in Python, list addresses start at 0, and `networkx` default node names do likewise. Let's adopt this convention here.

Now entry 3 in position 0 of the list `degrees` stands for 3 entries 0 in the list of stubs, to be constructed. Entry 2 in position 1 stands for 2 entries 1 in the list of stubs and so on. In general, entry d in position i stands for d entries i in the list of stubs.

Python's list arithmetic (using `m * a` for `m` *repetitions* of a list `a` and `a + b` for the *concatenation* of lists `a` and `b`) can be used to quickly convert a degree sequence into a list of stubs as follows.

In [3]:
```
stubs = [degrees[i] * [i] for i in range(len(degrees))]
stubs
```

Out[3]: `[[0, 0, 0], [1, 1], [2], [3], [4]]`

In [4]:
```
stubs = sum(stubs, [])
stubs
```

Out[4]: `[0, 0, 0, 1, 1, 2, 3, 4]`

Let's call this process the **stubs list** of a list of integers and wrap it into a python function

In [5]:
```
def stubs_list(a):
    return sum([a[i] * [i] for i in range(len(a))], [])
```

In [6]:
```
stubs_list(degrees)
```

Out[6]: `[0, 0, 0, 1, 1, 2, 3, 4]`

How to randomly shuffle this list? The wikipedia page on [random permutations (https://en.wikipedia.org/wiki/Random_permutation#Knuth_shuffles)](https://en.wikipedia.org/wiki/Random_permutation#Knuth_shuffles) recommends a simple algorithm for shuffling the elements of a list `a` in place.

In [7]:
```
from random import randint
def knuth_shuffle(a):
    l = len(a)-1
    for k in range(l):
        j = randint(k, l)
        a[j], a[k] = a[k], a[j]
```

In [8]:
```
a = [1,2,3]
knuth_shuffle(a)
a
```

Out[8]: `[3, 2, 1]`

Let's test whether this shuffle produces uniformly random outcomes ...

In [9]:
```
shuffles = {}
for i in range(1000):
    a = [1,2,3]
    knuth_shuffle(a)
    key = tuple(a)
    shuffles[key] = shuffles.get(key, 0) + 1

print(shuffles)
```

```
{(1, 2, 3): 152, (3, 2, 1): 154, (3, 1, 2): 183, (1, 3, 2): 168, (2, 1, 3): 1
56, (2, 3, 1): 187}
```

But python's `random` module already contains a function `shuffle` which does exactly this.

In [10]:
```
from random import shuffle
```

In [11]:
```
a = [1,2,3]
shuffle(a)
a
```

Out[11]:
```
[1, 2, 3]
```

Let's test whether this shuffle produces uniformly random outcomes ...

In [12]:
```
shuffles = {}
for i in range(1000):
    a = [1,2,3]
    shuffle(a)
    key = tuple(a)
    shuffles[key] = shuffles.get(key, 0) + 1

print(shuffles)
```

```
{(1, 3, 2): 168, (3, 2, 1): 149, (2, 1, 3): 160, (3, 1, 2): 178, (1, 2, 3): 1
69, (2, 3, 1): 176}
```

So we shuffle the stubs ...

In [13]:
```
shuffle(stubs)
stubs
```

Out[13]:
```
[0, 0, 0, 2, 4, 3, 1, 1]
```

Then we match pairs, by cutting the list of stubs into halves and transposing the resulting array of 2 rows ...

In [14]:
```
m = len(stubs) // 2
```

In [15]:
```
edges = [stubs[:m], stubs[m:]]
edges
```

Out[15]:
```
[[0, 0, 0, 2], [4, 3, 1, 1]]
```

In [16]:
```
edges = list(zip(*edges))
edges
```

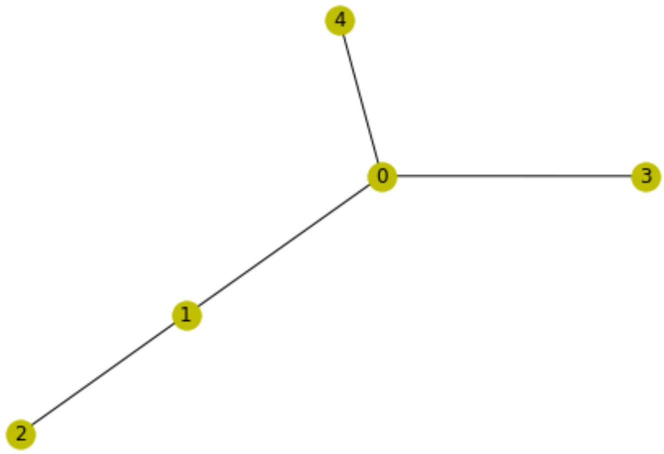
Out[16]:
```
[(0, 4), (0, 3), (0, 1), (2, 1)]
```

In [17]:
```
G = nx.Graph(edges)
```

In [18]:
```
G.number_of_edges()
```

Out[18]: 4

In [19]:
```
nx.draw(G, with_labels=True, node_color='y')
```



In [20]:
```
G.edges()
```

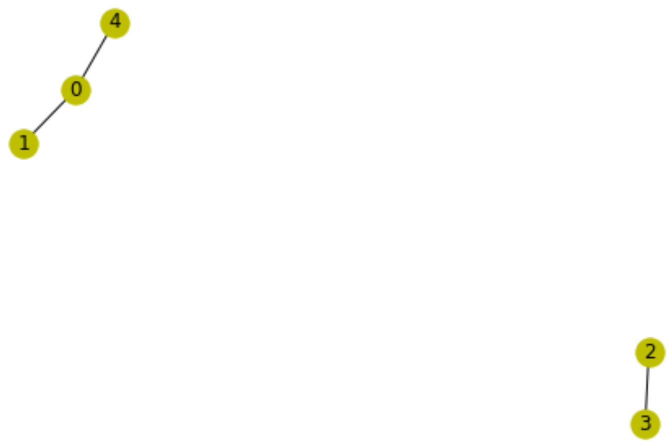
Out[20]: EdgeView([(0, 4), (0, 3), (0, 1), (1, 2)])

All in all, a configuration model can be built as follows.

In [21]:
```
def configuration(degrees):
    m = sum(degrees) // 2  # should check if sum(degs) is even ...
    stubs = stubs_list(degrees)
    shuffle(stubs)
    edges = list(zip(stubs[:m], stubs[m:]))
    return nx.Graph(edges)
```

In [22]:
```
G = configuration([3,2,1,1,1])
nx.draw(G, with_labels=True, node_color='y')
print(G.edges())
```

```
[(0, 1), (0, 4), (3, 2)]
```



Now create a power degree distribution ... and generate a graph. Use $\gamma = 2$, since I [know (https://en.wikipedia.org/wiki/Riemann_zeta_function)](https://en.wikipedia.org/wiki/Riemann_zeta_function) that $\zeta(2) = \pi^2/6$ ...

In [23]:
```
from math import pi
gamma = 2
c = 6/pi/pi
p = [0] + [c * k**(-gamma) for k in range(1,35)]
print(p, sum(p))
```

```
[0, 0.6079271018540267, 0.15198177546350666, 0.06754745576155852, 0.037995443
865876666, 0.024317084074161065, 0.01688686394038963, 0.01240667554804136, 0.
009498860966469166, 0.007505272862395391, 0.006079271018540266, 0.00502419092
4413444, 0.004221715985097407, 0.003597201786118501, 0.00310166888701034, 0.0
027018982304623405, 0.0023747152416172916, 0.0021035539856540716, 0.001876318
2155988477, 0.0016840085923934256, 0.0015198177546350666, 0.00137851950533792
9, 0.001256047731103361, 0.0011492005706125268, 0.0010554289962743518, 0.0009
726833629664427, 0.0008993004465296252, 0.0008339192069328212, 0.000775417221
752585, 0.0007228621900761315, 0.0006754745576155851, 0.0006325984410551787,
0.0005936788104043229, 0.0005582434360459381, 0.0005258884964135179] 0.982380
1579310864
```

Note how $p_0 = 0$ was explicitly prepended to the list `p`. Turn this now into a degree histogram on, say, $n = 50$ nodes.

In [24]:
```
d = [int(50 * x) for x in p]
print(d)
print(sum(d))
```

```
[0, 30, 7, 3, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
42
```

Then recover the degree sequence from the histogram. Incidently, this works exactly as the transformation of the degree sequence above into a list of stubs (why?).

In [25]:
```
print(stubs_list(d))
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 4, 5]
```

In [26]:
```
G = configuration(stubs_list(d))
```

In [27]:
```
G.edges()
```

Out[27]:
```
EdgeView([(41, 34), (41, 31), (41, 18), (41, 6), (41, 9), (34, 39), (31, 40),
(40, 27), (40, 40), (15, 38), (38, 23), (38, 32), (37, 4), (37, 26), (37, 2),
(25, 14), (39, 30), (39, 24), (28, 36), (36, 33), (13, 8), (33, 20), (30, 0),
(7, 3), (11, 10), (19, 16), (32, 35), (35, 21), (12, 29), (22, 5), (1, 17)])
```

In [28]:
```
nx.draw(G, with_labels=True, node_color='y')
```

In [29]:
```
nx.degree_histogram(G)
```

Out[29]:
```
[0, 30, 7, 3, 1, 1]
```

Try again ... This time with $\gamma = 2.5$, an explict value of $c = 2.5$, and $p[1]$ corrected so that $\sum p_k = 1$.

In [30]:
```
gamma = 2.5
c = 2.5
p = [0] + [c * k**(-gamma) for k in range(1,35)]
p[1] = p[1] - (sum(p) - 1)
print(p)
print(sum(p))
```

```
[0, 0.1545054783722426, 0.4419417382415922, 0.16037507477489604, 0.078125, 0.
0447213595499958, 0.028350575726657154, 0.019283901684144247, 0.0138106793200
49757, 0.010288065843621401, 0.007905694150420948, 0.006229573235077761, 0.00
5011721086715501, 0.004102812102257612, 0.00340894441214827, 0.00286887655274
62344, 0.00244140625, 0.0020980590401066864, 0.0018186902808295976, 0.0015887
516196022283, 0.0013975424859373688, 0.0012370628698185509, 0.001101243369605
4257, 0.0009854178358084818, 0.0008859554914580361, 0.0008, 0.000725281564860
148, 0.0006599797315839344, 0.0006026219276295077, 0.0005520075451160876, 0.0
005071505162084871, 0.0004672354371143988, 0.0004315837287515549, 0.000399627
30935651474, 0.0003708879436472942]
0.9999999999999999
```

Lift the degree numbers by $1/2$ so that they are rounded, rather than cut off when converted into integers. Also reduce number of nodes to 25 for better drawings ...

In [31]:
```
d = [int(25 * x + 0.5) for x in p]
print(d)
print(sum(d))
```

```
[0, 4, 11, 4, 2, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
23
```

In [32]:
```
print(stubs_list(d))
```

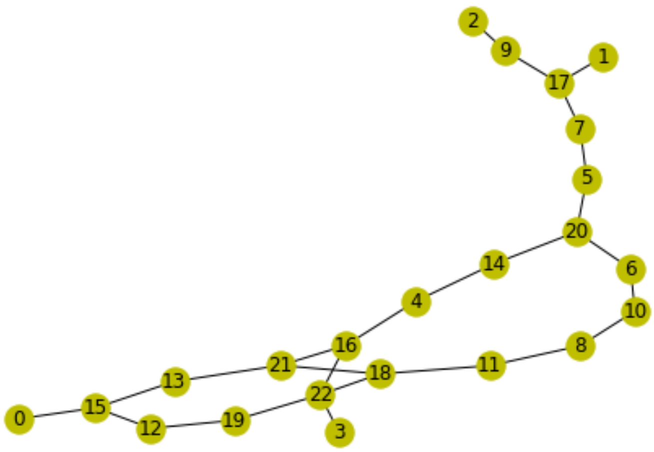
```
[1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 5, 6]
```

In [33]:
```
G = configuration(stubs_list(d))
```

In [34]:
```
G.edges()
```

Out[34]:
```
EdgeView([(19, 22), (19, 12), (22, 18), (22, 3), (22, 16), (18, 11), (18, 21)
, (16, 4), (16, 21), (4, 14), (15, 0), (15, 13), (15, 12), (13, 21), (8, 11),
(8, 10), (1, 17), (17, 7), (17, 9), (21, 21), (10, 6), (6, 20), (7, 5), (5, 2
0), (20, 14), (2, 9)])
```

In [35]:
```
nx.draw(G, with_labels=True, node_color='y')
```

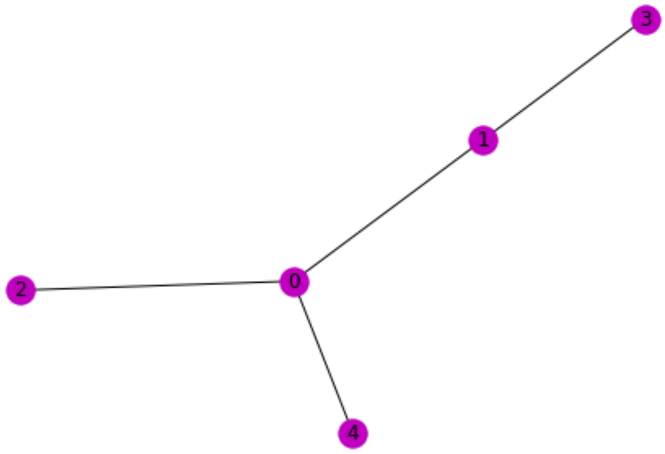


In [36]:
```
nx.degree_histogram(G)
```

Out[36]:
```
[0, 4, 12, 5, 1, 1]
```

In `networkx` , configuration models can be generated with the function `nx.configuration_model` .

In [37]:
```
G = nx.configuration_model(degrees)
nx.draw(G, with_labels=True, node_color='m')
```



In [ ]:
```
```