Table of Contents

CS4423-Networks: Week 9 (11+12 March 2025)

Part 2: Computing Random Graphs

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/#Week09

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": "aqua"} # aqua nodes this week

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
```

Random Samples

- Our goal is to randomly select edges on a given vertex set X. That is, pick at random elements from the set $\binom{X}{2}$ of pairs of nodes.
- So we need a procedure

for selecting m from N objects randomly, in such a way that each of the $\binom{N}{m}$ subsets of the N objects is an equally likely outcome.

- We first discuss sampling m values in the range $\{0, 1, \ldots, N-1\}$.

An intuitive approach

Maybe the most obvious approach is to select each number in the desired range with probability $p = m/N$.

- `Python` 's basic random number generator `random.random` returns a random number in the (half-open) interval $[0, 1)$ every time it is called.
- Looping with `a` over `range(N)` : if the randomly generated number is less than p, then we include the current value of `a` , if not we don't.

In [2]:
```
def random_sample_B(N, p):
    """sample elements in range(n) with probability p"""
    sample = []
    for a in range(N):
        if random.random() < p:
            sample.append(a)
    return sample
```

We'll make a few samples with $pN = (0.2)10 = 2$, so we expect to usually get 2 terms in the sample. But it will not always happen.

In [3]:
```
random_sample_B(10,0.2)
```

Out[3]:
```
[]
```

In [4]:
```
random_sample_B(10,0.2)
```

Out[4]:
```
[4]
```

In [5]:
```
random_sample_B(10,0.2)
```

Out[5]:
```
[1, 9]
```

We'd expect this to return a list of pN numbers, which it does (on average)

In [6]:
```
sum_l = 0
N = 100
p = 0.2
for i in range(N):
    S = random_sample_B(N,p)
    sum_l += len(S)
    # print(f"Sample {i:2d} has {len(S)} terms")
print(f"Average is {sum_l/N}")
```

```
Average is 20.31
```

Let's do that for 10,000 runs:

In [7]:
```
c = 100000
sum(len(random_sample_B(N, p)) for i in range(c))/c
```

Out[7]:
```
19.98759
```

Choosing exactly m terms

To randomly select exactly m numbers from from $0, 1, \ldots, N-1$, we use a modification of this procedure [see Knuth: The Art of Computer Programming, Vol. 2, Section 3.4.2, Algorithm S] :

- The number a should be selected with probability $\frac{m-c}{N-a}$,

if c items have already been selected.

- *Can you explain why this works?*

In [8]:
```
def random_sample_A(N, m):
    sample = []
    for a in range(N):
        if (N - a) * random.random() < m - len(sample):
            sample.append(a)
    return sample
```

Let's see a small example. Note that they all have 4 terms in the samples.

In [9]:
```
N = 10
m = 4
print( random_sample_A(N, m) )
print( random_sample_A(N, m) )
print( random_sample_A(N, m) )
```

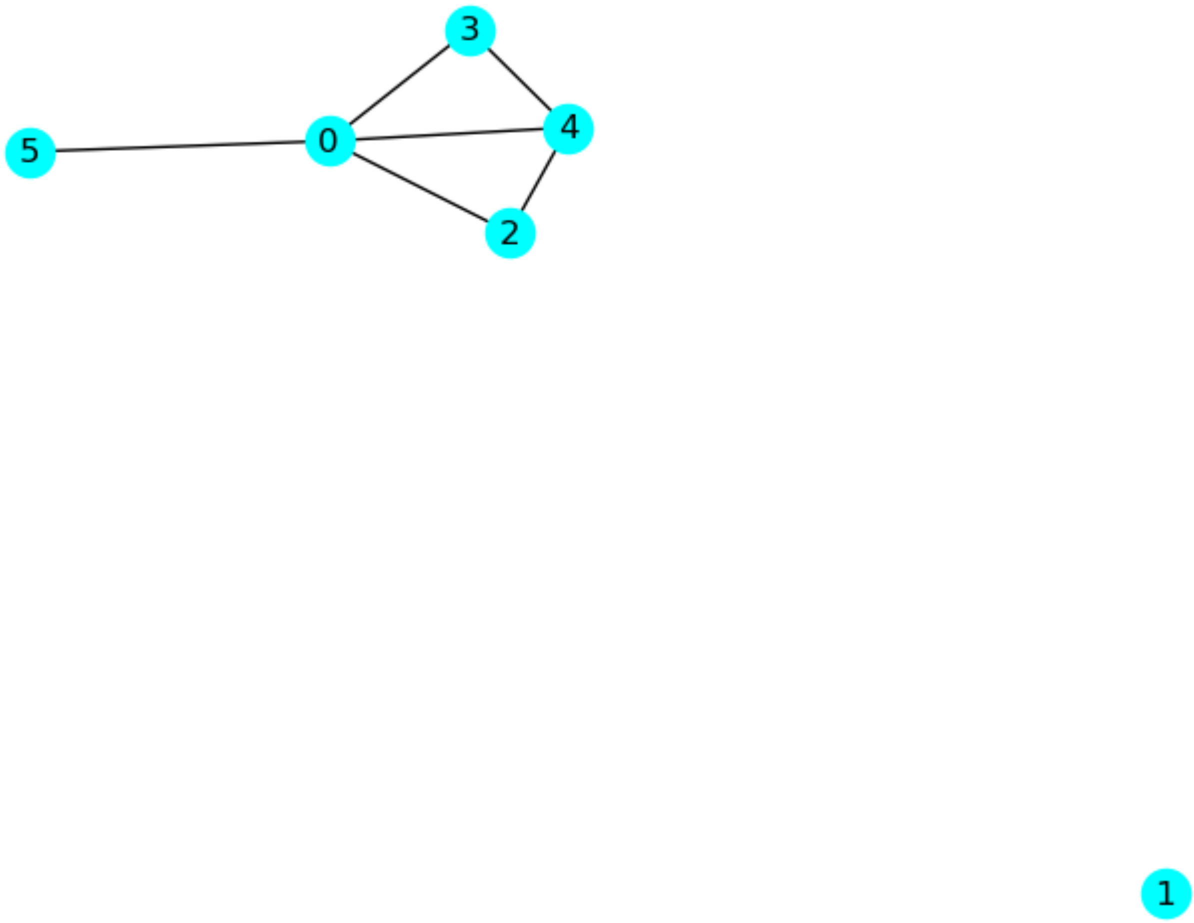
```
[2, 4, 7, 8]
[1, 4, 7, 8]
[5, 6, 7, 9]
```

Computing $G_{ER}(n, m)$

We can easily adapt the above procedure to compute examples of graphs in $G_{ER}(n, m)$.

But here we'll use the `networkx` random graph constructor, `gnm_random_graph` , to do this.

In [10]:
```python
n,m = 6,6
G1 = nx.gnm_random_graph(n, m)
nx.draw(G1, **opts)
```



Computing $G_{ER}(n, p)$

Our own function

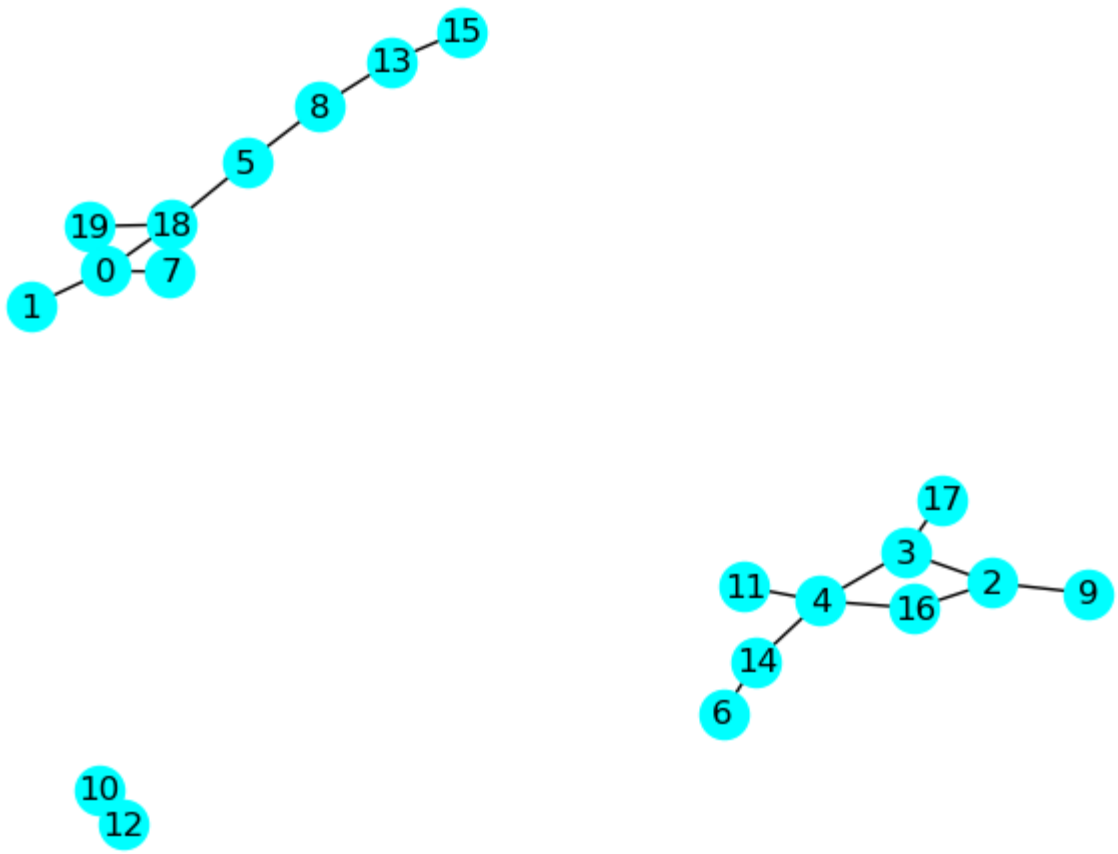
Here is a simple approach to computing a sample from $G_{ER}(n, p)$:

In [11]:
```python
def random_graph_B(n, p):
    """construct a random type B graph  with n nodes and edge probability p"""
    G = nx.empty_graph(n)
    for x in range(n):
        for y in range(x):
            if random.random() < p:
                G.add_edge(x, y)
    return G
```

In [12]:
```python
n = 20
p = 0.2
N = n*(n-1)/2
```

In [13]:
```python
%time
G2 = random_graph_B(n, p)
nx.draw(G2, **opts)
print(f"G2 has {G2.size()} edges. Expeced number is {p*N}")
```

```
CPU times: user 4 µs, sys: 1 µs, total: 5 µs
Wall time: 8.11 µs
G2 has 20 edges. Expeced number is 38.0
```

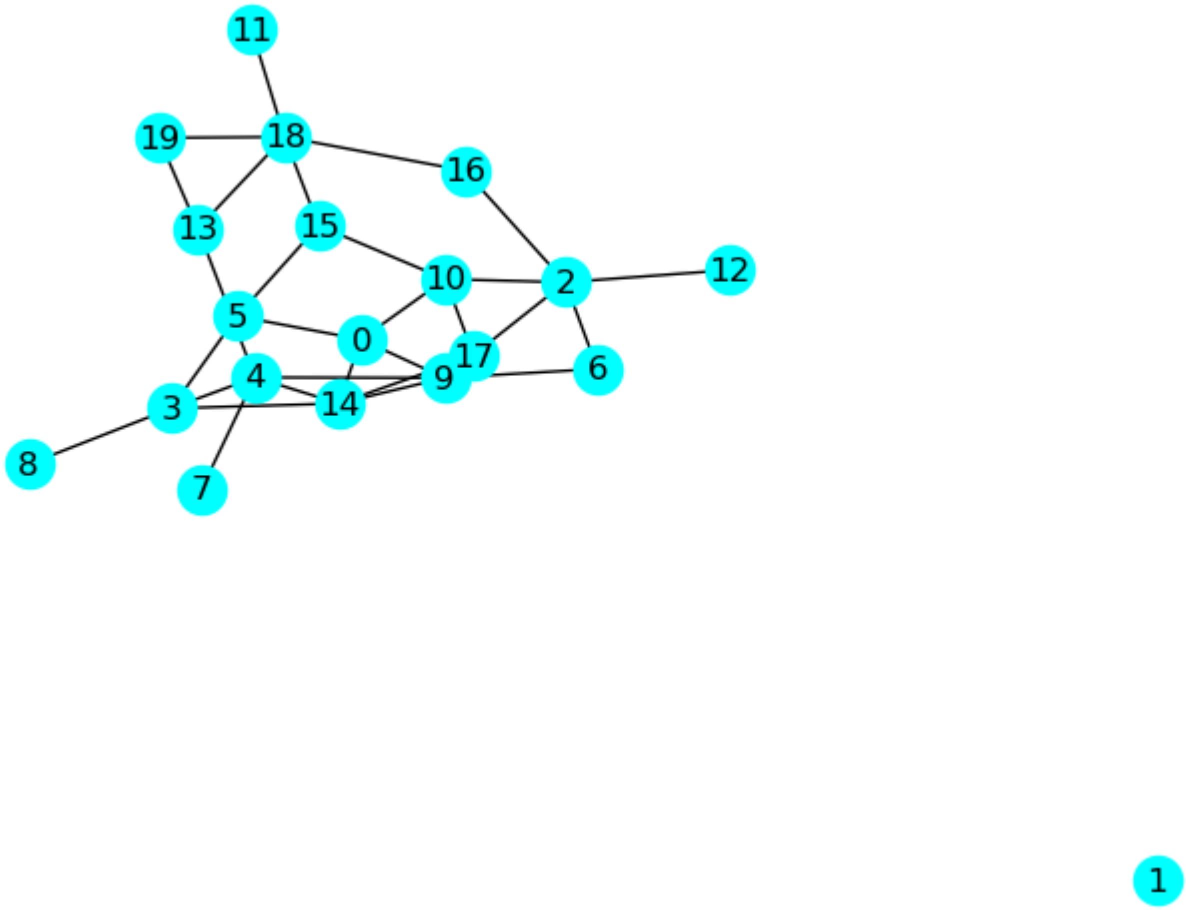


The `gnp_random_graph()` function

The `networkx` version of this random graph constructor is called `gnp_random_graph` and should produce the same random graphs with the same probability (but should be more efficient for large networks).

In [14]:
```
G3 = nx.gnp_random_graph(n, p)
nx.draw(G3, **opts)
print(f"G3 has {G3.size()} edges. Expeced number is {p*N}")
```

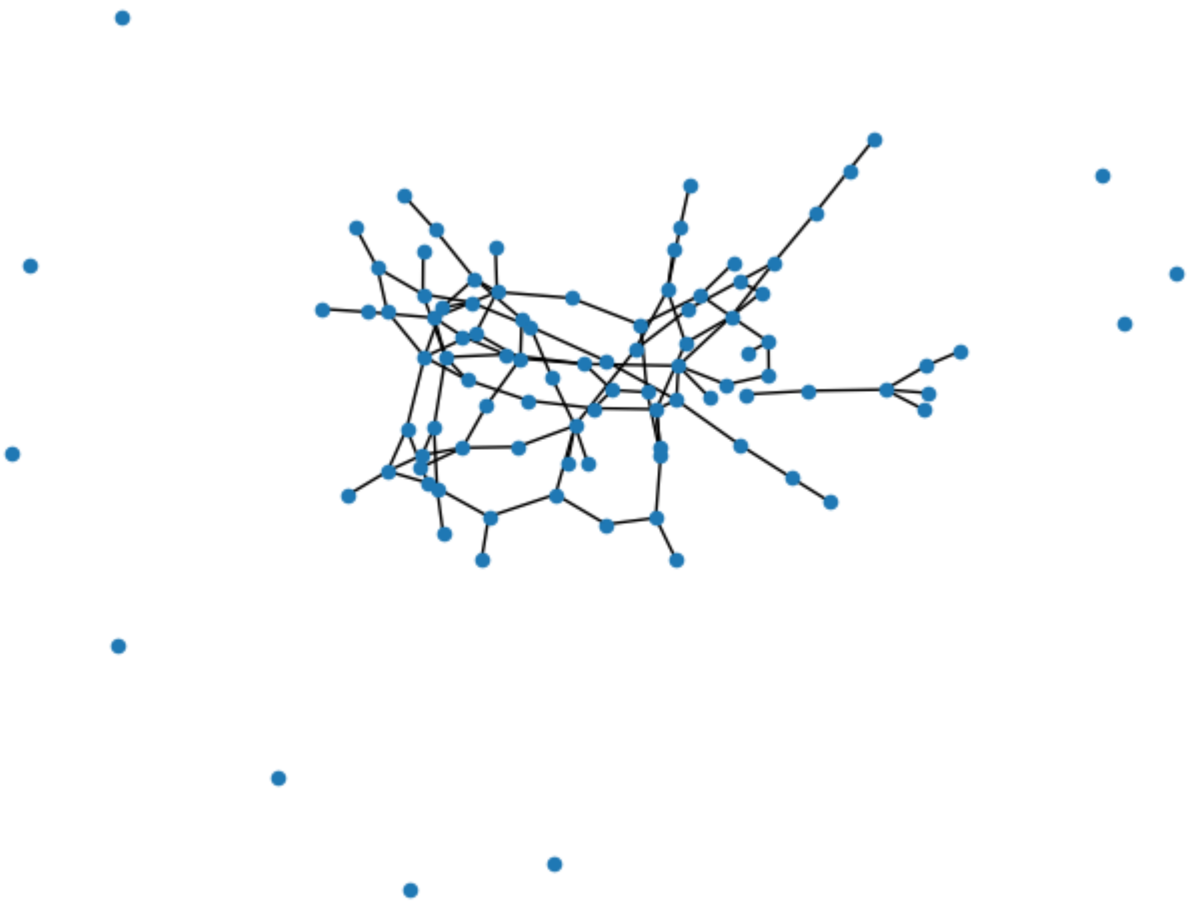
```
G3 has 29 edges. Expeced number is 38.0
```



In [15]:

```
n = 100
p = 0.02
N = n*(n-1)/2
G4 = nx.gnp_random_graph(n, p)
nx.draw(G4, node_size=20)
print(f"G4 has {G4.size()} edges. Expeced number is {p*N}")
plt.savefig("W09-cover.png")
```

```
G4 has 113 edges. Expeced number is 99.0
```



Expected size

We know that any graph drawn from $G_{ER}(n,m)$ has size m (with probability 1).

For $G_{ER}(n,p)$ the *expected size* is pN. Let's check that:

In [16]:
```
n = 100
N = math.comb(n,2) # "combination" = "binomial coef"
p = 0.01
num_trials = 1000
sum_of_sizes = 0
for i in range(num_trials):
    G = nx.gnp_random_graph(n,p)
    sum_of_sizes += G.size()
ave_size = sum_of_sizes/num_trials
print(f"For this selection, average size is {ave_size}; expected is pN={p*N}")
```

```
For this selection, average size is 49.614; expected is pN=49.5
```

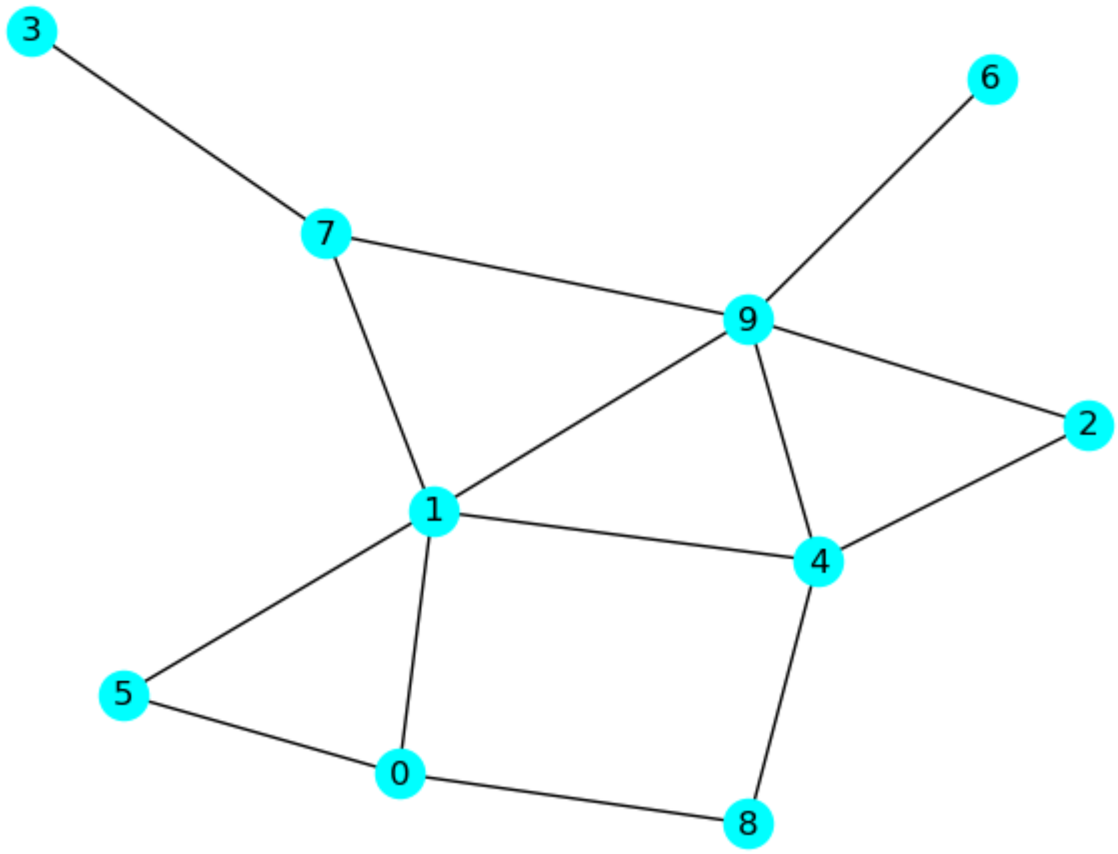
Expected Average Degree

In Part 1, we noted that, for $G_{ER}(n,m)$, the the expected **size** of a graph is $\bar{m} = m$ as every graph G in $G_{ER}(n,m)$ has exactly m edges.

It follows that the expected **average degree** is $\langle k \rangle = \frac{2m}{n}$, as every graph has average degree $2m/n$.

Let's verify that:

In [17]:
```
n = 10
m = 14
G = nx.gnm_random_graph(n,m)
nx.draw(G, **opts)
```



Get the degree sequence:

In [18]:
```
degree_sequence = [d for n, d in G.degree()]
print(degree_sequence)
```

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

Compute the mean value, and compare with $< k >= 2m/n$.

In [19]:
```
mean_deg = statistics.mean(degree_sequence)
print(f"Averge degree is {mean_deg}, and 2m/n = {2*m/n}")
```

```
Averge degree is 2.8, and 2m/n = 2.8
```

$G_{ER}(n, p)$

We learned in Part 1 that the degree distribution in a random graph in $G_{ER}(n, p)$ is a *binomial distribution*

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

That is, in the $G_{ER}(n, p)$ model, the *probability that a node has degree k is p_k*.

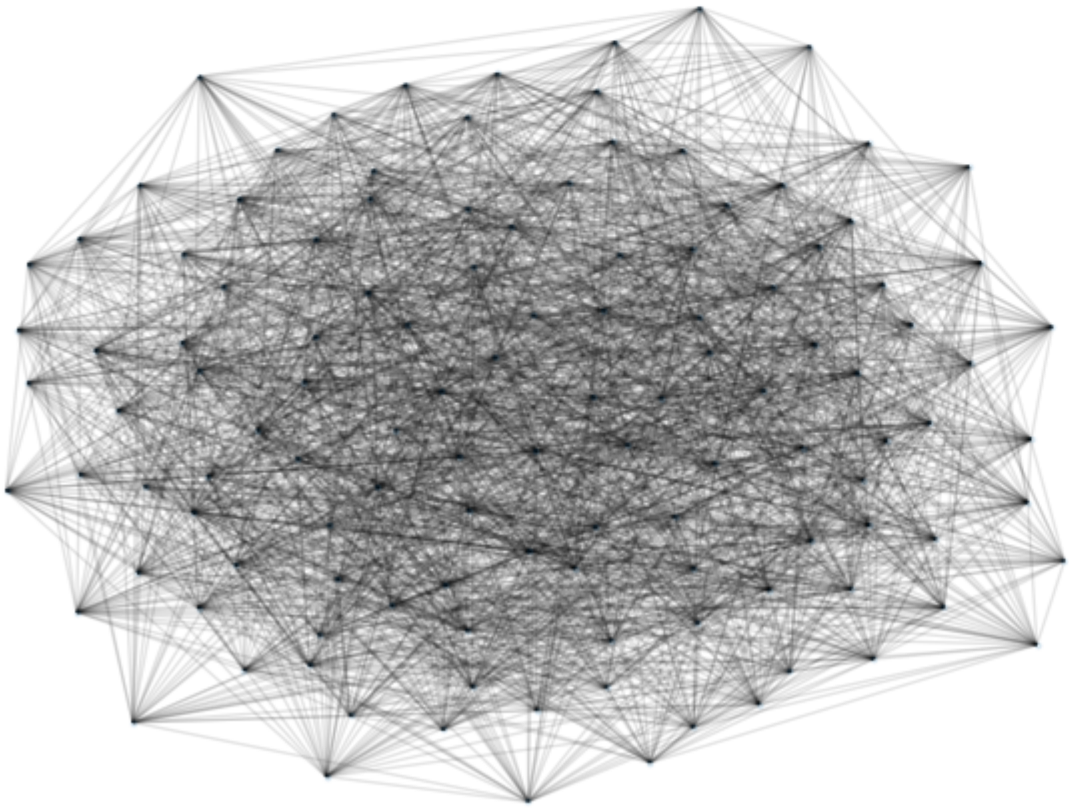
Let's check some examples.

In Part 1, we considered an example for Q3(c) of the 2023/24 exam paper: suppose one constructed a graph G on 120 nodes by tossing a (fair, 6-sided) die once for each possible edge, adding the edge only if the die shows 3 or 6. Then pick a node at random in this graph. What is the probability that this node has degree 50?

Set n and p and make a graph

In [20]:
```
n = 120
p = 1.0/3.0
G = nx.gnp_random_graph(n,p)
```

In [21]:
```
nx.draw(G, node_size=3, alpha=0.1 )
```



From the theory:

In [22]:
```
k=50
p50 = math.comb(n-1,k)*(p**k)*(1-p)**(n-1-k)
print(p50)
```

```
0.010555313148364344
```

In practice:

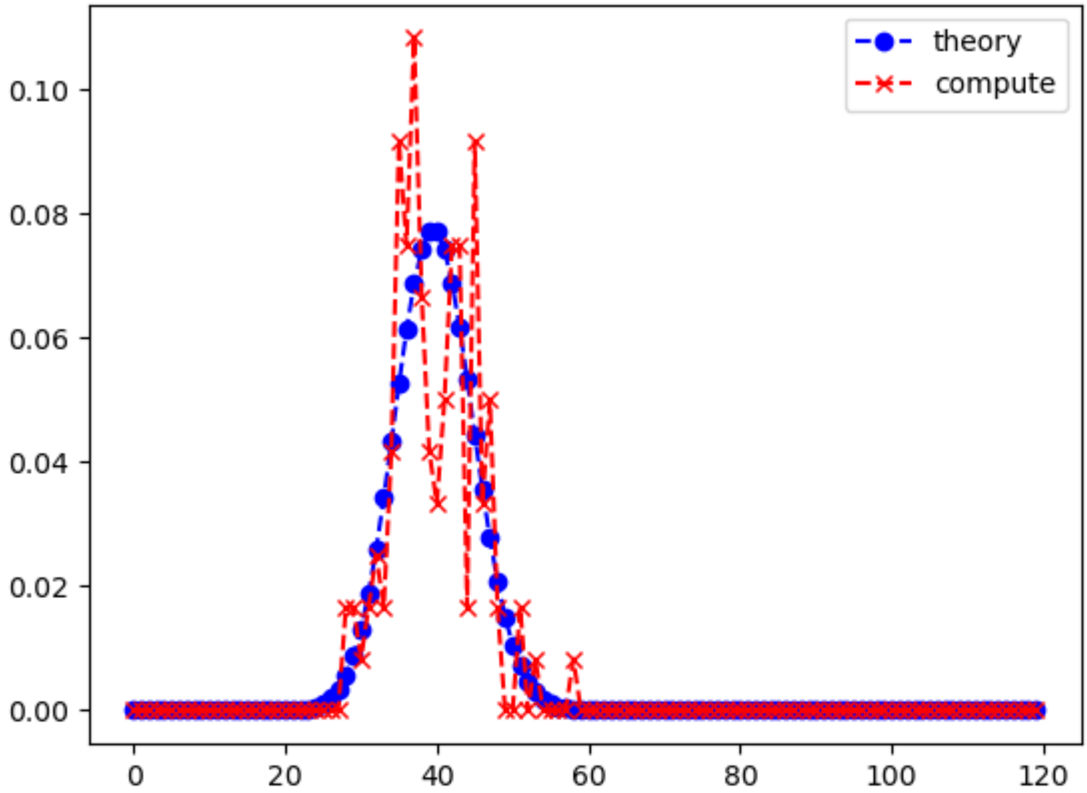
In [23]:
```
def count_k_in_G(G,k):
    count = 0
    for i in range(n):
        if (G.degree(i) == k):
            count +=1
    return(count)
print(count_k_in_G(G,50)/n)
```

```
0.0
```

These numbers may not agree terribly well... let's check for all k, and plot

In [24]:
```
P1 = [math.comb(n-1,k)*(p**k)*(1-p)**(n-1-k) for k in range(n)]
p2 = [count_k_in_G(G,k)/n for k in range(n)]
plt.plot(P1, marker='o', linestyle='--', color='b', label='theory')
plt.plot(p2, marker='x', linestyle='--', color='r', label='compute')
plt.legend()
```

Out[24]:
```
<matplotlib.legend.Legend at 0x7f33c3e55040>
```

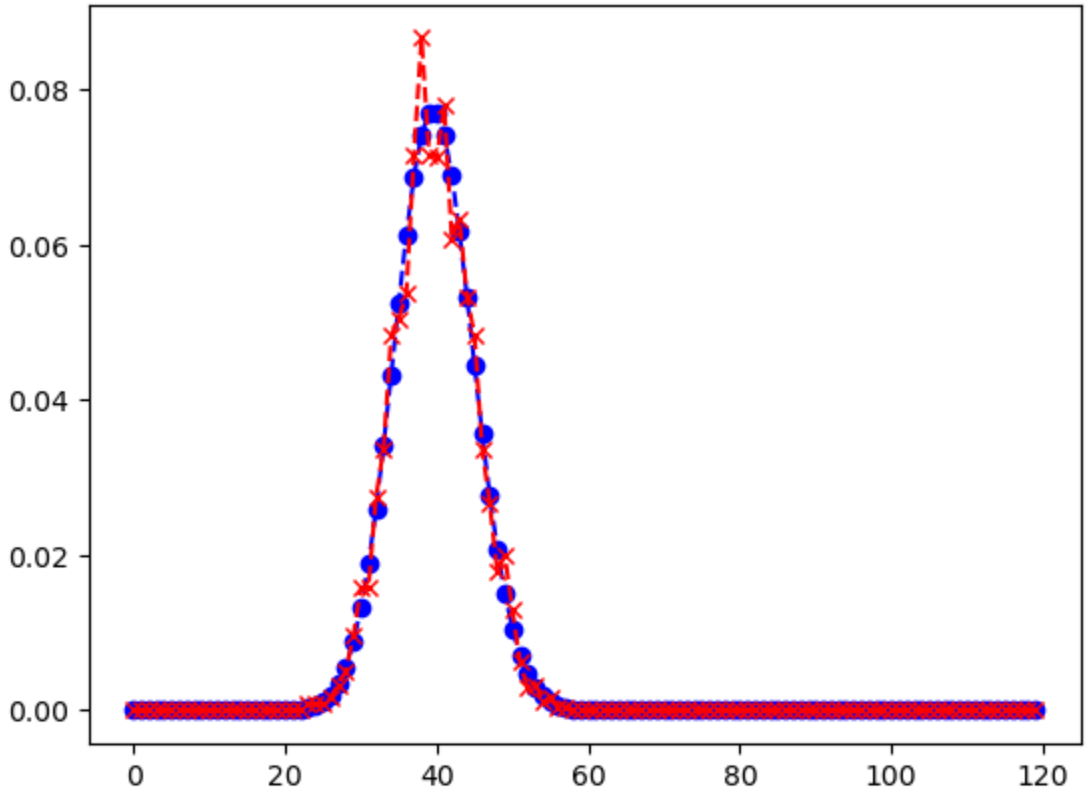


That looks reasonable, but would be more convincing if we averaged over a number of randomly drawn graphs:

In [25]:
```python
P1 = [math.comb(n-1,k)*(p**k)*(1-p)**(n-1-k) for k in range(n)]
P2 = np.zeros(n)
num_draws = 20
for run in range(num_draws):
    G = nx.gnp_random_graph(n,p)
    P2 = P2 + [count_k_in_G(G,k)/n/num_draws for k in range(n)]

plt.plot(P1, marker='o', linestyle='--', color='b', label='theory')
plt.plot(P2, marker='x', linestyle='--', color='r', label='compute')
```

Out[25]: [<matplotlib.lines.Line2D at 0x7f33c2122d20>]

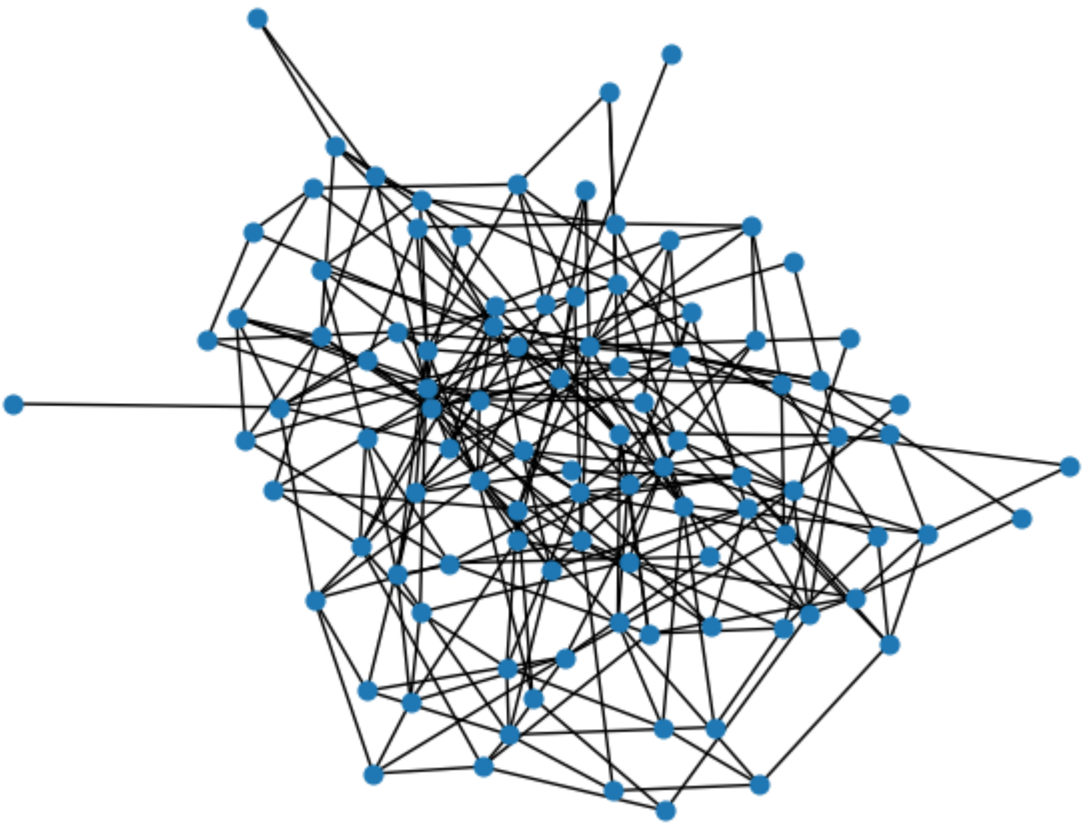


$p = p(n)$

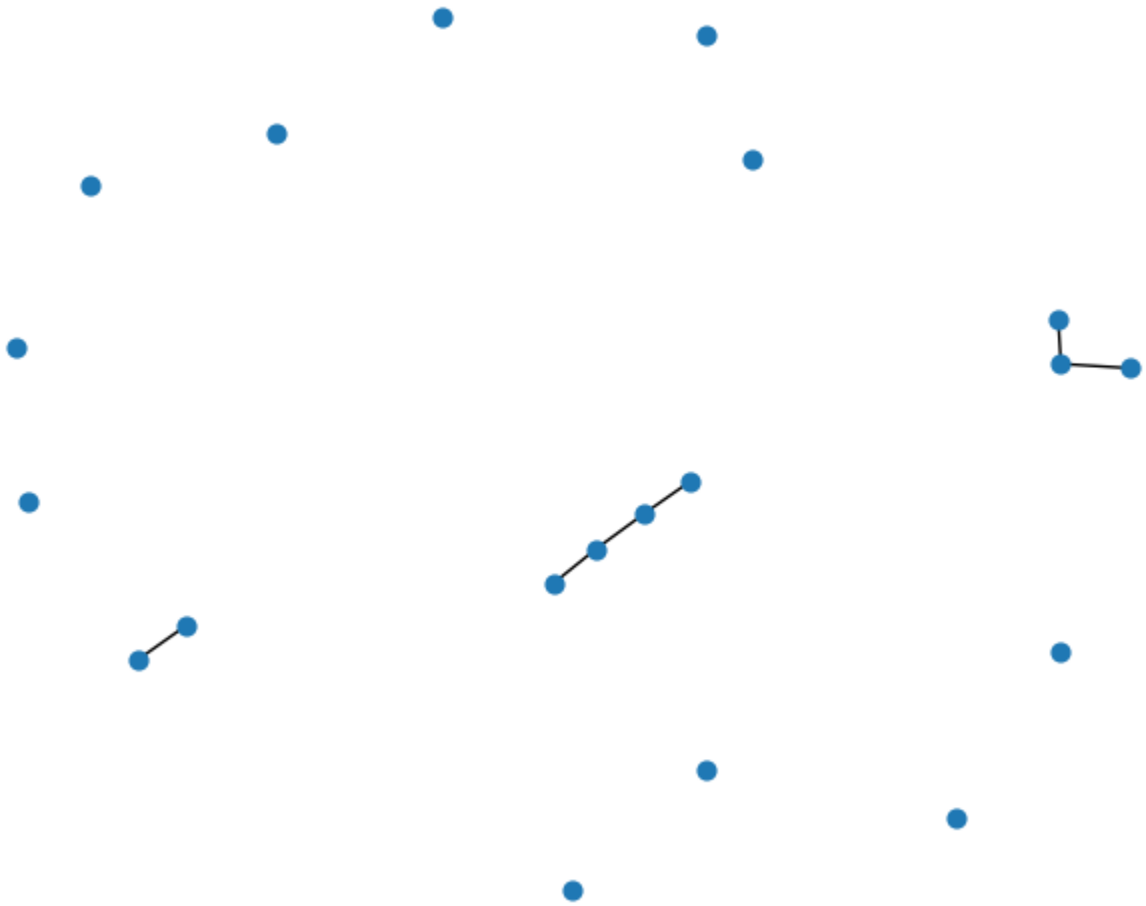
In a way, it does not make sense to compare $G_{ER}(n_1, p)$ with $G_{ER}(n_2, p)$. If n_1 and n_2 are very different, the resulting graphs can have different structures.

Lets look at 2 examples. In both we have $p = 0.05$, but we'll have $n_1 = 100$ and $n_2 = 20$.

In [26]:
```python
n1 = 100
p = 0.05
G1 = nx.gnp_random_graph(n1,p)
nx.draw(G1, node_size=40)
```



In [27]:
```
n2 = 20
G2 = nx.gnp_random_graph(n2,p)
nx.draw(G2, node_size=40)
```



FINISHED HERE THURSDAY

This will lead us to a discussion on "The Giant Connected Component".

Definition (Giant Component). A connected component of a graph G is called a **giant component** if its number of nodes increases with the order n of G as some positive power of n.