GUENIN–CARLUT Avel - TD2 Complex Networks

November 19, 2018

1 Complex Networks - TD 2

1.1 Avel GUÉNIN--CARLUT

1.2 20/11/2018

```
In [1]: import networkx as nx
        import random as rd
        import numpy as np
        import math
        import matplotlib.pyplot as plt
        import copy
```

1.3 I - The Erdös-Rényi random graph model

Below, we will generate and analyse ER - type random graphs. This category of graphs, initially invented for building probabilistic demonstrations of mathematical properties of networks, happen to be the maximal entropy networks for a given number of nodes and edges, hence a central role in complex networks theory.

1.3.1 I.1 - Generating ER graphs

```
In [2]: def er_np (n,p) :

            ## check variables
            if not (type(n) == int):
                print("Improper variable type : n is not an integer !")
                return()
            if n < 0 :
                print("Improper nodes number : n is negative")
                return()
            if (0>p) or (p>1) :
                print("Improper edge probability : p is not a probability")
                return()


            ## define output graph
            G = nx.Graph()
```

```
            ## add nodes
            for i in range(n):
                G.add_node(i+1)

            ## define all possible edges
            possible_edges = set()
            for i in G.nodes:
                for j in G.nodes:
                    if not (i==j) : # no loops !
                        possible_edges.add(frozenset([i,j]))

            ## add edges
            for edge in possible_edges:
                edge = [v for v in edge]
                if rd.random() < p :
                    G.add_edge(edge[0],edge[1])

            return(G)
```

In [3]:
```
def er_nm (n,m) :

    ## check variables
    if not (type(n) == int):
        print("Improper variable type : n is not an integer !")
        return()
    if n < 0 :
        print("Improper nodes number : n is negative")
        return()
    if not (type(m) == int):
        print("Improper variable type : m is not an integer !")
        return()
    if (0>m) or (m>n*(n-1)/2) :
        print("Improper edge number : m is outside possible edge number for n")
        return()


    ## define output graph
    G = nx.Graph()

    ## add nodes
    for i in range(n):
        G.add_node(i+1)

    ## define all possible edges
    possible_edges = set()
    for i in G.nodes:
        for j in G.nodes:
```

```
                if not (i==j) : # no loops !
                    possible_edges.add(frozenset([i,j]))

        ## add edges
        for i in range(m):
            edge = rd.choice(list(possible_edges))
            possible_edges.remove(edge)
            edge = [v for v in edge]
            G.add_edge(edge[0],edge[1])

        return(G)
```

Having defined the graph generating function, we will check their proper functioning.

```
In [4]: print("~~~ checking er_np ~~~\n")
        node_number = []
        edge_distance = []
        loops_number = []
        doublons_number = []
        N = 30

        for i in range(N):
            n = rd.randrange(10,100)
            p = rd.random()

            g = er_np(n,p)

            node_number.append(1 - int(len(g.nodes)==n))
            edge_distance.append(np.log(len(g.edges)/(p*n*(n-1)/2)))


            edges = list(g.edges)

            loops_number.append(len([v for v in edges if len(v) == 1]))

            doublons = edges
            edges = set(edges)
            for v in edges:
                doublons.remove(v)
            doublons_number.append(len(doublons))


        print("Over " + str(N) + " trials, we found " + str(sum(node_number)) + " errors on the
```

```
~~~ checking er_np ~~~

Over 30 trials, we found 0 errors on the node number, 0 loops, and 0 doublons. Mean log distanc
```

```
In [5]: print("~~~ checking er_nm ~~~\n")
        node_number = []
        edge_number = []
        loops_number = []
        doublons_number = []
        N = 30

        for i in range(N):
            n = rd.randrange(10,100)
            m = rd.randrange(0,n*(n-1)/2)

            g = er_nm(n,m)

            node_number.append(1 - int(len(g.nodes)==n))
            edge_number.append(1 - int(len(g.edges)==m))


            edges = list(g.edges)

            loops_number.append(len([v for v in edges if len(v) == 1]))

            doublons = edges
            edges = set(edges)
            for v in edges:
                doublons.remove(v)
            doublons_number.append(len(doublons))


        print("Over " + str(N) + " trials, we found " + str(sum(node_number)) + " errors on the
```

```
~~~ checking er_nm ~~~

Over 30 trials, we found 0 errors on the node number, 0 errors on the edges number, 0 loops, a
```

As you can see, everything is going well.

1.3.2 I.2 - Degree distributions of random graphs

```
In [6]: def compare_edge_count(n,p):
            G_np = er_np(n,p)
            m = np.floor(p*n*(n-1)/2)
            G_nm = er_nm(n,int(m))
            return((len(G_np.edges)/len(G_nm.edges)))
```

```
In [7]: N = 30
        n = 30
        p = 0.3
```

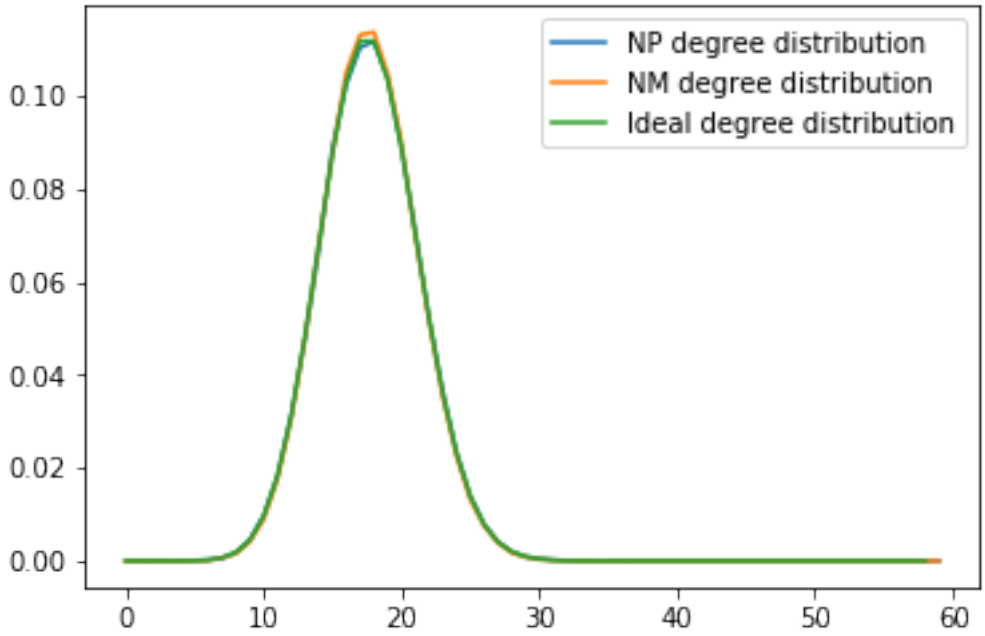
```
        edge_distance = [np.log(compare_edge_count(n,p)) for i in range(N)]

        print("Over " + str(N) + " trials, the mean log distance between NP and NM graph edges
```

```
Over 30 trials, the mean log distance between NP and NM graph edges number was 0.0523626690383
```

As you can see, this ratio never goes very far from 1. This suggest that both NP and NM constructions do have the same expected number of edges.

```
In [8]: def compare_degree_distrib(n,p):

            N = 3000 ## number of graphs over which mean is calculated

            ## Compute np degree distribution
            degrees_np = [0]*n
            for i in range(N):
                g_np = er_np(n,p)
                for node in g_np.nodes :
                    degrees_np[g_np.degree(node)]+=1
            degree_distrib_np= [v/sum(degrees_np) for v in degrees_np]


            ## Compute nm degree distribution
            m = np.floor(p*n*(n-1)/2)
            m = int(m)
            degrees_nm = [0]*n
            for i in range(N):
                g_nm = er_nm(n,m)
                for node in g_nm.nodes :
                    degrees_nm[g_nm.degree(node)]+=1
            degree_distrib_nm = [v/sum(degrees_nm) for v in degrees_nm]

            ## Compute ideal degree distribution
            degree_distrib_ideal = [math.factorial(n-1)/(math.factorial(k)*math.factorial(n-1-k
```

```
            ## Plot
            plt.plot(degree_distrib_np, label = "NP degree distribution")
            plt.plot(degree_distrib_nm, label = "NM degree distribution")
            plt.plot(degree_distrib_ideal, label = "Ideal degree distribution")
            plt.legend()
            plt.show()

            return()

        compare_degree_distrib(60,0.3)
```



Out[8]: ()

As you can see above, both NP and NM follow closely the ideal distribution. To fully establish this result, we would need to show convergence with growing n, but I believe superposing the plots give us a clear enough idea of the situation here.

Even though not displayed here, I tested for an array of n and p : the result is robust.

1.4 II - The Erdös-Rényi model and connected components

1.5 II.1 - Transition in n times p

As different scaling laws and graph property are implicated, we will have to study individual plots traducing scaling properties for varying p times n.

We will plot average result both within and between graph of different size, to control for other parameters than n times p. Tested graph size will be evenly spaced in log space.

While it is clear that higher n will give clearer results, we will stay within the 100 to 10000 range because of hardware limitations.

```
In [9]: ## I define a function extracting sorted connected component sizes
        def comp_size(G):
            components = nx.connected_components(G)
            comp_size = []
            for nodes in components:
                comp_size.append(len(nodes))
            comp_size.sort(reverse=True)
            return(comp_size)
```

```
        ## I define a function producing the quantities relevant to our study : mean biggest a
        ## It takes as parameters the n times p and n range it must review, and the number of
        def np_plot(ntimesp_range,n_range,I):
            plots = {}
            for ntimesp in ntimesp_range:
                print("\n n times p = " + str(ntimesp))
                ## Plot variables initialization
                n_plot = [] ## Graph size
                BCCS = [] ## Biggest connected component size
                sBCCS = [] ## Second biggest connected component size


                for n in n_range:
                    print(n)
                    n_plot.append(n)
                    BCCS.append(0)
                    sBCCS.append(0)

                    for i in range(I): ## Each value is averaged over I trials in order to get
                        p = ntimesp/n
                        G = er_np(n,p)

                        BCCS[-1]+=(comp_size(G)[0])/I
                        sBCCS[-1]+=(comp_size(G)[1])/I

                plots[ntimesp] = [n_plot,BCCS,sBCCS]
            return(plots)
```

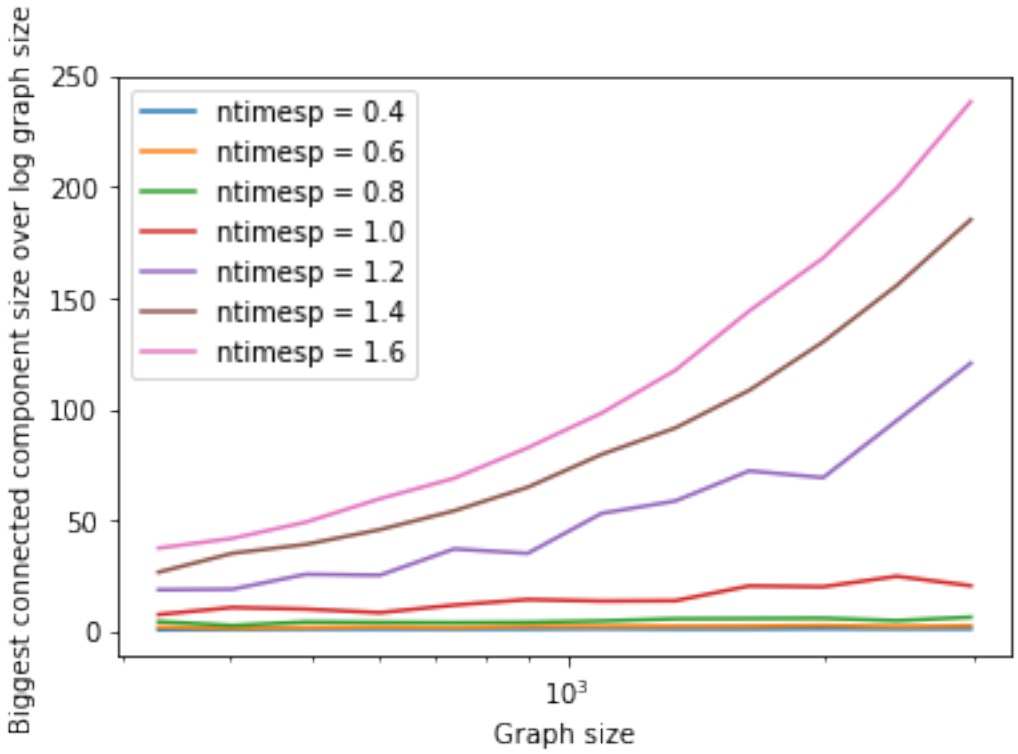
In [10]:
```
        ## Plot parameters initialisation

        n_range = [int(np.floor(np.exp(i/5))) for i in range(29,41)] ### Final run
        ## n_range = [int(np.floor(np.exp(i/5))) for i in range(25,35)] ### Test run

        I = 10 ### Final run
        ## I = 3ã### Test run

        ntimesp_range = [v/5 for v in range(2,9)]


        ## Generating data
        plots = np_plot(ntimesp_range,n_range,I)
```

```
 n times p = 0.4
330
403
```

```
492
601
735
897
1096
1339
1635
1998
2440
2980

 n times p = 0.6
330
403
492
601
735
897
1096
1339
1635
1998
2440
2980

 n times p = 0.8
330
403
492
601
735
897
1096
1339
1635
1998
2440
2980

 n times p = 1.0
330
403
492
601
735
897
1096
1339
```

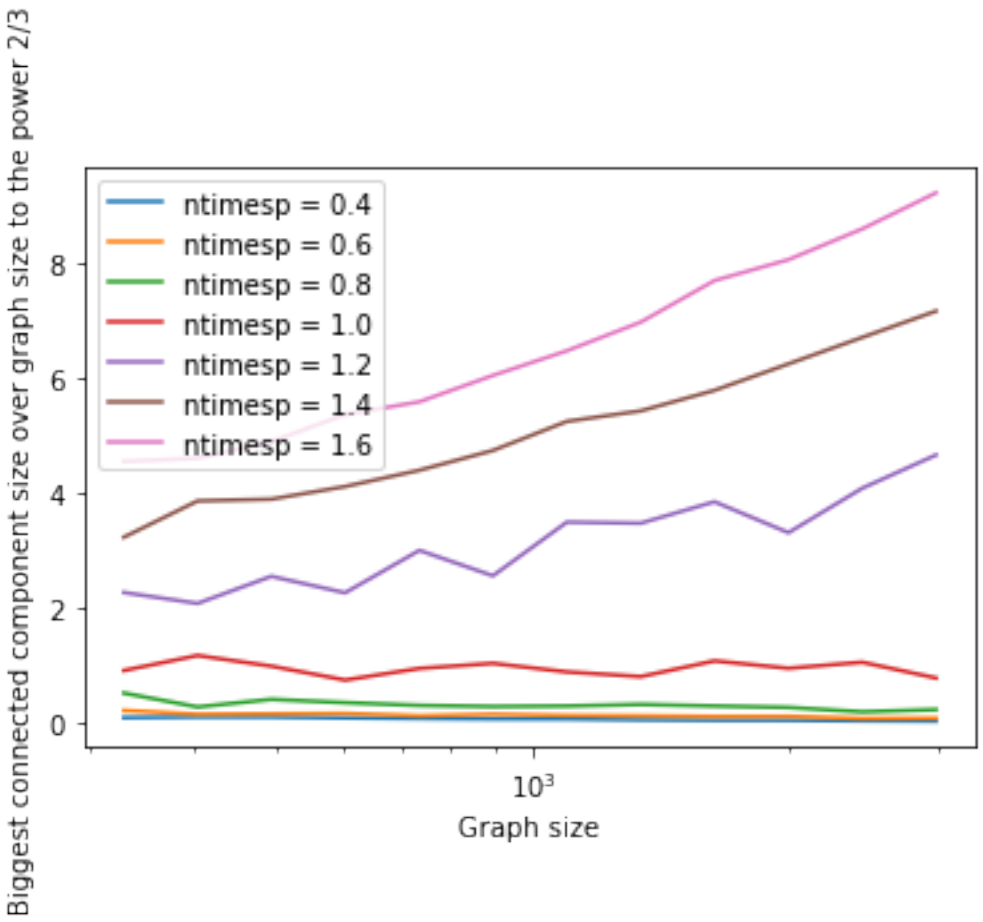
```
1635
1998
2440
2980

 n times p = 1.2
330
403
492
601
735
897
1096
1339
1635
1998
2440
2980

 n times p = 1.4
330
403
492
601
735
897
1096
1339
1635
1998
2440
2980

 n times p = 1.6
330
403
492
601
735
897
1096
1339
1635
1998
2440
2980
```

```
In [11]: ## Plotting biggest component size over log graph size
         for ntimesp in ntimesp_range:
             plt.semilogx(plots[ntimesp][0],plots[ntimesp][1]/np.log(plots[ntimesp][0]),label='

         plt.xlabel("Graph size")
         plt.ylabel("Biggest connected component size over log graph size")
         plt.legend(loc='upper left')
         plt.show()
```



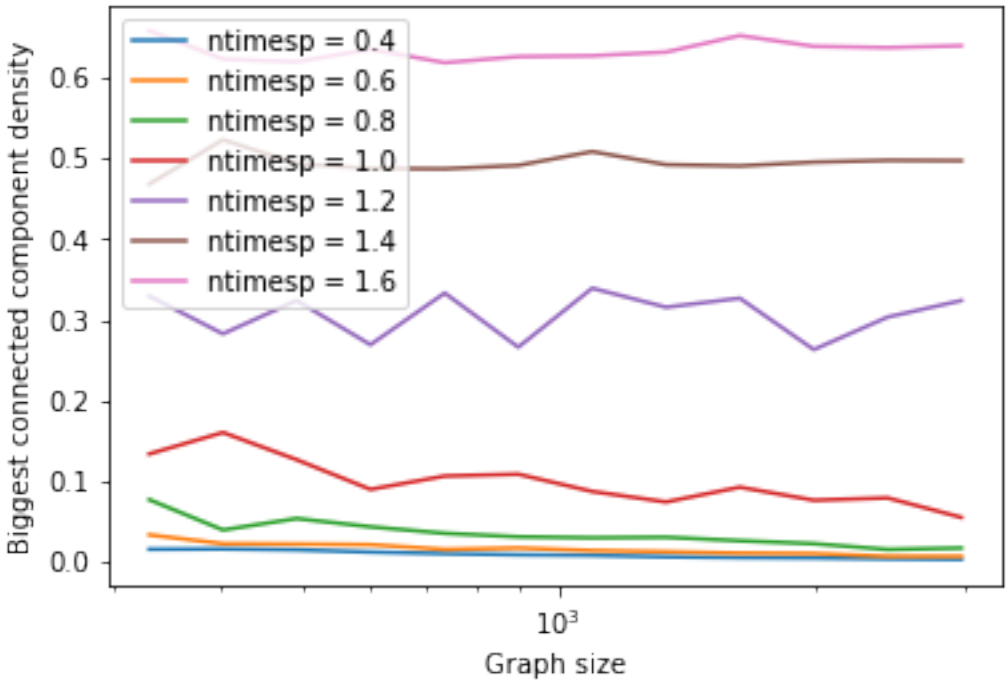
We can see that biggest component size clearly outgrows log graph size for n times $p > 1$, while it is clearly outgrown for $p < 1$. The case $p = 1$ is a littel ambiguous, but it does outgrow log graph size. This confirms the first Erdös-Rényi result.

```
In [12]: ## Plotting biggest component size over graph size to the power 2/3
         for ntimesp in ntimesp_range:
             plt.semilogx(plots[ntimesp][0],plots[ntimesp][1]/np.power(plots[ntimesp][0],2/3),l

         plt.xlabel("Graph size")
         plt.ylabel("Biggest connected component size over graph size to the power 2/3")
         plt.legend(loc='upper left')
         plt.show()
```



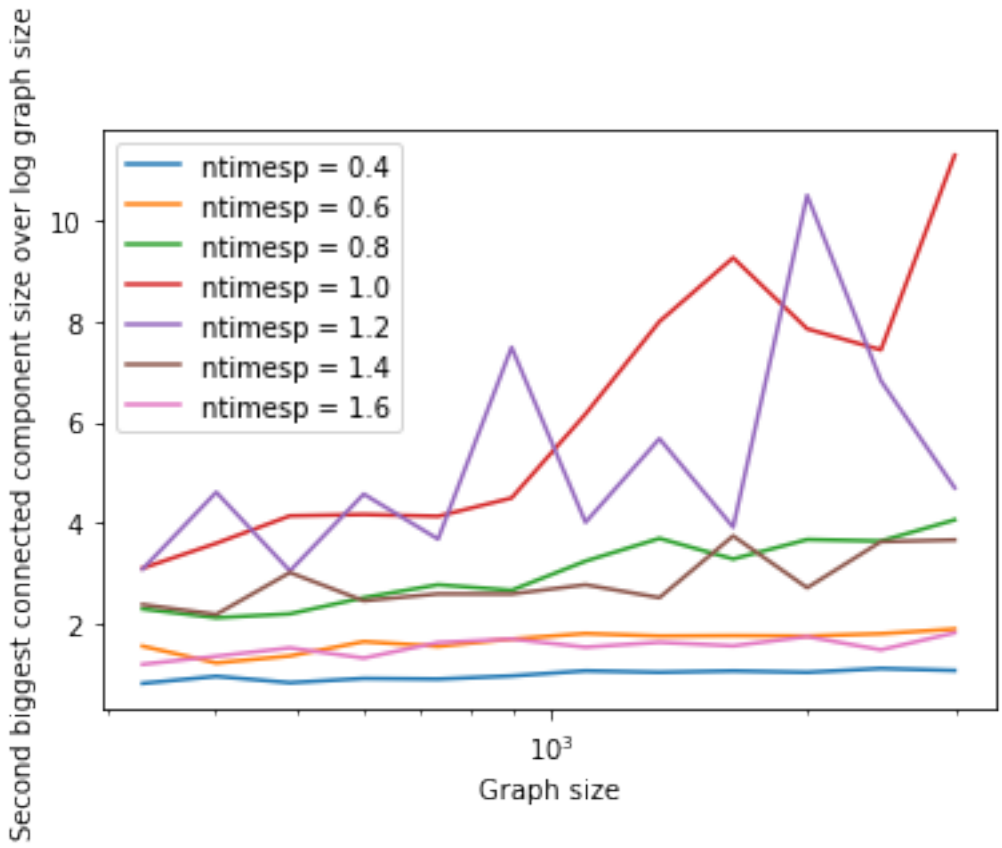
Biggest component size clearly outgrows graph size to the power 2/3 for n times p > 1, while it is clearly outgrown for n times p < 1. For n times p ~ 1, biggest component size seems to grow with graph size to the power 2/3. This confirm the second Erdös-Rényi result.

```
In [13]: ## Plotting biggest component density
         for ntimesp in ntimesp_range:
             plt.semilogx(plots[ntimesp][0],plots[ntimesp][1]/np.abs(plots[ntimesp][0]),label='

         plt.xlabel("Graph size")
         plt.ylabel("Biggest connected component density")
         plt.legend(loc='upper left')
         plt.show()
```



While it is not extremely clear for p = 1, biggest component density does seem to be converging to 0 for all p =< 1, while it stays at finite value for all p > 1. This confirms partially the third Erdös-Rényi result.

```
In [14]: ## Plotting second biggest component size over log graph size
         for ntimesp in ntimesp_range:
             plt.semilogx(plots[ntimesp][0],plots[ntimesp][2]/np.log(plots[ntimesp][0]),label='

         plt.xlabel("Graph size")
         plt.ylabel("Second biggest connected component size over log graph size")
         plt.legend(loc='upper left')
         plt.show()
```



Second biggest component size clearly outgrows log graph size for n times p ~ 1 only. Far from 1, both above and below, it seems to grow with log graph size. This completes confirmation of the third Erdös-Rényi result.

1.5.1 2.1 - Transition in n times p over log n

We will now go on to study the transition in n times p over log n. This will be more straightforward, as we will be able to just plot connectedness probability for different values of n times p over log n.

Except for that, our strategy is essentially similar to above.

```
In [15]: def nlognp_plot(v_range,n_range,I):
             plots = {}
             for v in v_range:
                 print("\n n times p over log n = " + str(v))
                 ## Plot variables initialization
                 connected_plot = [] ## Is the graph connected ?


                 for n in n_range:
                     print(n)
```

```
                    for i in range(I): ## Each value is averaged over I trials in order to ge
                        p = v * np.log(n) / n
                        G = er_np(n,p)

                        connected_plot.append(nx.is_connected(G))

                plots[v] = connected_plot
            return(plots)
```

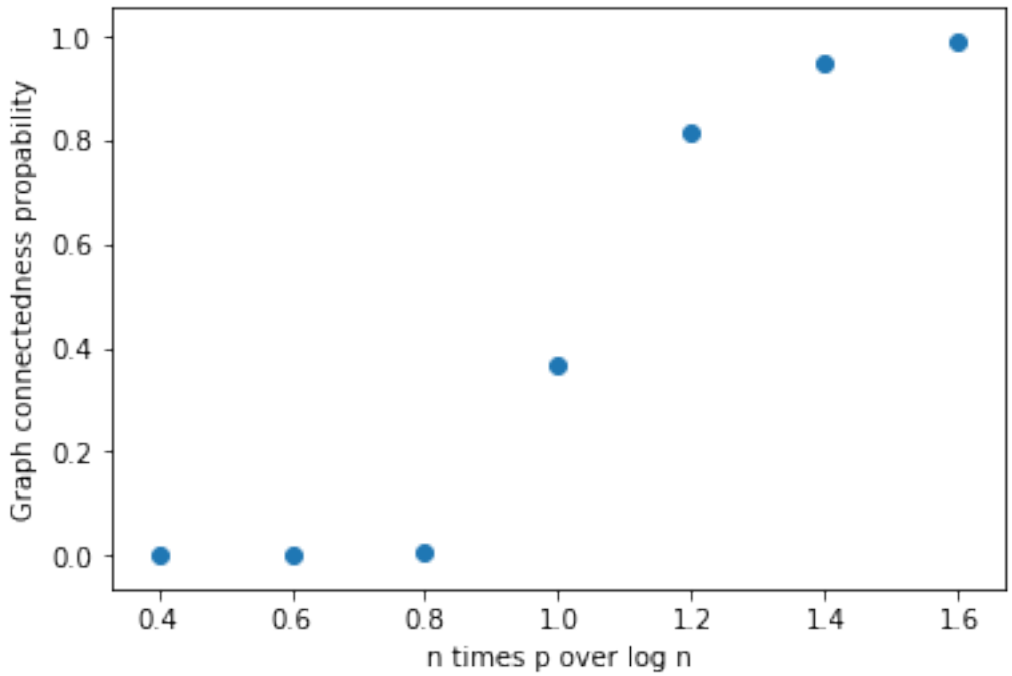
In [16]:
```
## Plot parameters initialisation (n_range and I are reused from previous paragraph)
v_range = [v/5 for v in range(2,9)]

## Generating data
plots = nlognp_plot(v_range,n_range,I)
```

```
 n times p over log n = 0.4
330
403
492
601
735
897
1096
1339
1635
1998
2440
2980

 n times p over log n = 0.6
330
403
492
601
735
897
1096
1339
1635
1998
2440
2980

 n times p over log n = 0.8
330
403
492
```

```
601
735
897
1096
1339
1635
1998
2440
2980

 n times p over log n = 1.0
330
403
492
601
735
897
1096
1339
1635
1998
2440
2980

 n times p over log n = 1.2
330
403
492
601
735
897
1096
1339
1635
1998
2440
2980

 n times p over log n = 1.4
330
403
492
601
735
897
1096
1339
1635
```

```
1998
2440
2980

 n times p over log n = 1.6
330
403
492
601
735
897
1096
1339
1635
1998
2440
2980
```

In [17]:
```
## Plotting biggest component size over log graph size
plt.scatter(v_range,[sum(plots[v])/len(plots[v]) for v in plots.keys()])
plt.xlabel("n times p over log n")
plt.ylabel("Graph connectedness propability")
plt.show()
```



We do observe the transition from near certain non-connectedness to near certain connectedness around 1, with epsilon = 0.3 in our case (relatively low n). This confirms the fourth and fifth Erdös-Rényi results.

While our results are imperfect due to experimental noise, we could confirm every Erdös-Renyi result with convincing accuracy. Note however that transition in p times n and p times n over log n occur slowly, with a characteristic transition of around 0.2 for n varying between 300 and 3000. I expect the transition to be sharper for higher network size. I cannot however show this due to hardware limitations.

1.6 3 - The configuration model of random networks

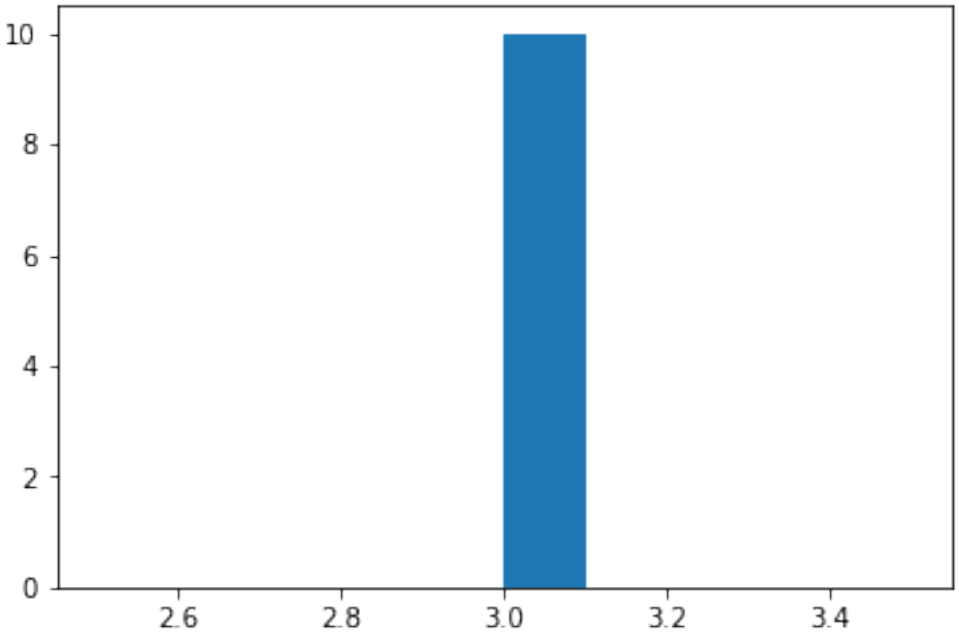
1.6.1 3.1 - Degree sequences obeying a given distribution

3.1.1 - degree_sequence_regular

```
In [9]: def degree_sequence_regular(n,k):
            if n*k%2 == 1:
                print("Error : an " + str(k) + "-regular graph cannot be of size " + str(n) + "
                return()
            else :
                deg = []
                for i in range(n):
                    deg.append(k)
            return(deg)
```

We'll test the function by plotting an histogram for a specific value. We also check parity.

```
In [10]: deg = degree_sequence_regular(1000,3)

         plt.hist(deg,density=True)
         plt.show()
         print("Degree sum parity ? " + str(sum(deg)%2 == 0))
```




```
Degree sum parity ? True
```

Everything seems to be working.

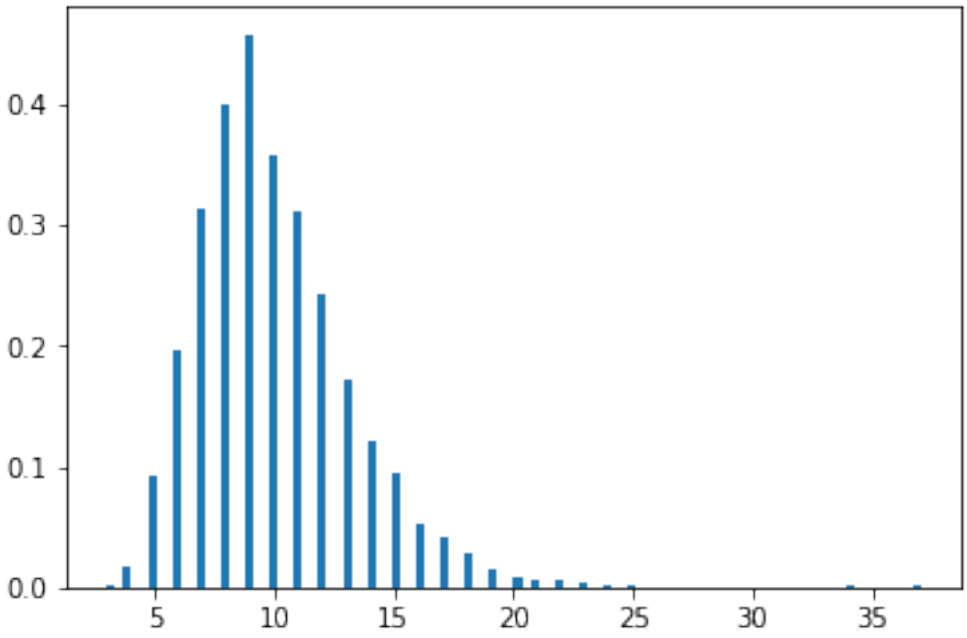
3.1.1 - degree_sequence_lognormal

```
In [11]: def degree_sequence_lognormal(n,m,v):
             degree_sequence = [1]
             mu = np.log(m/np.sqrt(1+v/(m**2)))
             sigma = np.sqrt(np.log(1+v/(m**2)))
             while sum(degree_sequence)%2 == 1:
                 degree_sequence=[]
                 for i in range(n):
                     k = np.round(rd.lognormvariate(mu,sigma))
                     degree_sequence.append(k)
             return(degree_sequence)
```

We'll test the function by plotting an histogram and first moments for a specific value. We also check parity.

```
In [12]: mu = 10
         sigma = 10
         plot_range = 100

         deg = degree_sequence_lognormal(3000,mu,sigma)
```

```
plt.hist(deg,plot_range,density=True)
plt.show()
print("Mean : " + str(np.mean(deg)) + " (Expected = " + str(mu) + ")")
print("Variance : " + str(np.var(deg)) + " (Expected = " + str(sigma) + ")")
print("Somme des degrés paire ? " + str(sum(deg)%2 == 0))
```




```
Mean : 10.0086666667 (Expected = 10)
Variance : 10.1345915556 (Expected = 10)
Somme des degrés paire ? True
```

We can visually check lognormal distribution. Everything seems to be working.

1.6.2 3.2 - Attaching stubs to form a network

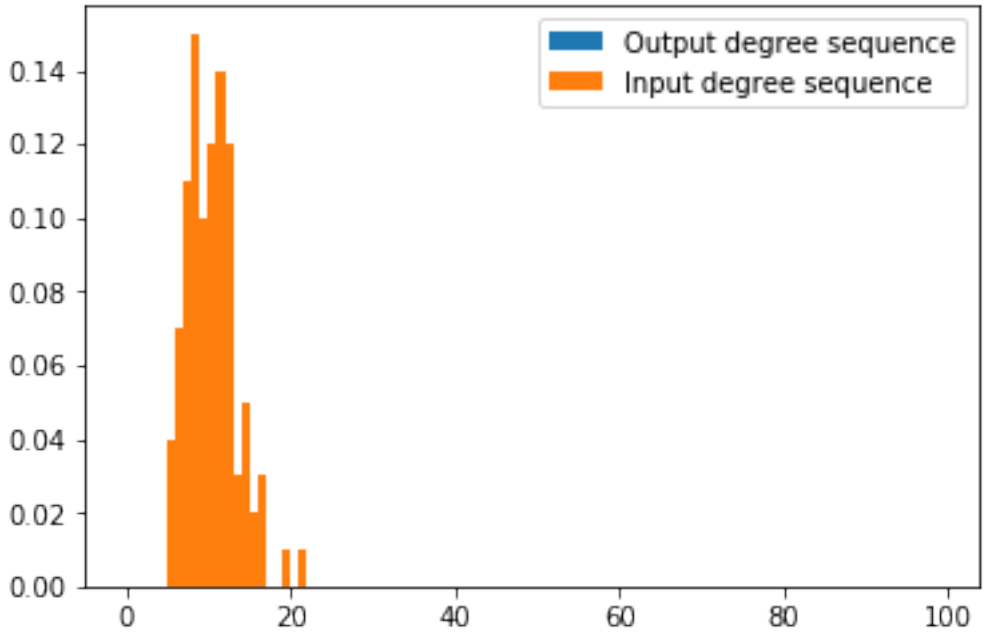
```
In [13]: def configure_sequence(degree_sequence):
             G = nx.MultiGraph()
             node_sequence = range(len(degree_sequence))

             for i in node_sequence:
                 G.add_node(i)

             while sum(degree_sequence) > 0:
                 node_1 = rd.choices(node_sequence,degree_sequence,k=1)[0]
                 degree_sequence[node_1]-=1
```

```
            node_2 = rd.choices(node_sequence,degree_sequence,k=1)[0]
            degree_sequence[node_2]-=1

            G.add_edge(node_1,node_2)

        return(G)
```

We'll test the function by comparing input degree sequence to the independently calculated degree sequence of the output graph for some specific distribution.

```
In [14]: N = 100
         mu = 10
         sigma = 10
         degree_sequence = degree_sequence_lognormal(N,mu,sigma)

         G = configure_sequence(copy.deepcopy(degree_sequence))
         print("Number of nodes = " + str(G.number_of_nodes()) + " (expected = " + str(N) + ")
         print("Number of edges = " + str(G.number_of_edges()) + " (expected = " + str(N*mu/2)

         plot_range = range(100)
         output_degree_sequence = sorted([d for n, d in G.degree()], reverse=True)
         plt.hist([d for n, d in G.degree()],plot_range,density=True,label="Output degree seque
         plt.hist(degree_sequence,plot_range,density=True,label ="Input degree sequence")
         plt.legend()
         plt.show()
```

```
Number of nodes = 100 (expected = 100)
Number of edges = 497 (expected = 500.0)
```



As you can see, both distribution are perfectly aligned : configure_sequence works.

1.6.3 3.3 - Counting double edges and self loops

```
In [15]: def irregular_edge_count(G):
             count = 0
             visited = []

             for edge in G.edges():
                 if edge[0] == edge[1]:
                     count += 1
                     ##print("loop : " + str(edge))

                 elif edge in visited:
                     count+=1
                     ##print("multiedge : " + str(edge) + " (number of edges = " + str(G.numbe

                 visited.append(edge)

             return(count)
```

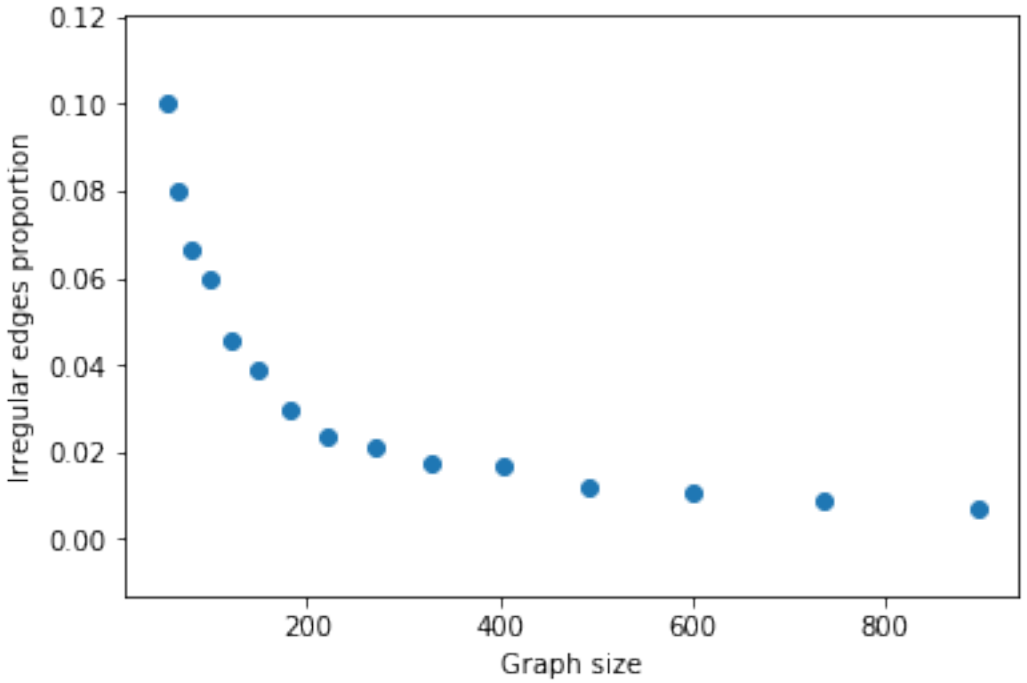
I tested the function by searching for irregularities in edges categorised as irregular, and for irregular edges categorised as regular. I do not show the process as it is cumbersome, but irregular_edges_count works.

I will now plot irregular edge density against network size for a specific degree distribution. While it does not work as proof in the general case, the dynamics we will show hold for any distribution.

```
In [16]: n_range = [int(np.floor(np.exp(i/5))) for i in range(20,35)]
         I = 5

         mu = 10
         sigma = 10

         plots = dict()

         for n in n_range:
             print(n)
             plots[n] = []
             for i in range(I):
                 degree_sequence = degree_sequence_lognormal(n,mu,sigma)
                 G = configure_sequence(copy.deepcopy(degree_sequence))
                 plots[n].append(irregular_edge_count(G)/G.number_of_edges())


         ## Plotting irregular edges proportion against graph size
```

```
        plt.scatter(n_range,[sum(plots[v])/len(plots[v])for v in plots.keys()])
        plt.xlabel("Graph size")
        plt.ylabel("Irregular edges proportion")
        plt.show()
```

```
54
66
81
99
121
148
181
221
270
330
403
492
601
735
897
```



As you can see, irregular edges density decrease to 0 as network size increase. In the large network size approximation, the configuration model can reliably be expected to generate simple

graphs. Alternatively, all irregular edges can be removed without significantly altering degree distribution.