POLITECNICO DI TORINO

Complex networks: theory and applications
Laboratory report

Andrea Senacheribbe
s265392

Master in Communications and Computer Networks Engineering

4th July 2019

Assignment 1: Centrality indices

1.1 Introduction

In this first assignment, we evaluate the centrality of the nodes in a medium size graph (some thousands of nodes), using different centralities measures.
The graph chosen for the following analysis is a protein-protein interaction (PPI) graph in Homo Sapiens (taken from [1]). The nodes represents proteins and two nodes are connected by an edge if they have some kind of interaction.
The centralities in protein-protein interaction networks are a powerful tool to understand which proteins play an important role and are essential for the organism.

1.2 Graph properties

First of all, the graph is read from file and processed to remove self-loops and extract only the largest connected component. The resulting graph has $n = 5973$ vertices and $m = 145778$ edges. It is stored by employing its adjacency matrix represented in sparse matrix format. The structure of the matrix is shown in the sparsity plot of fig. 1.

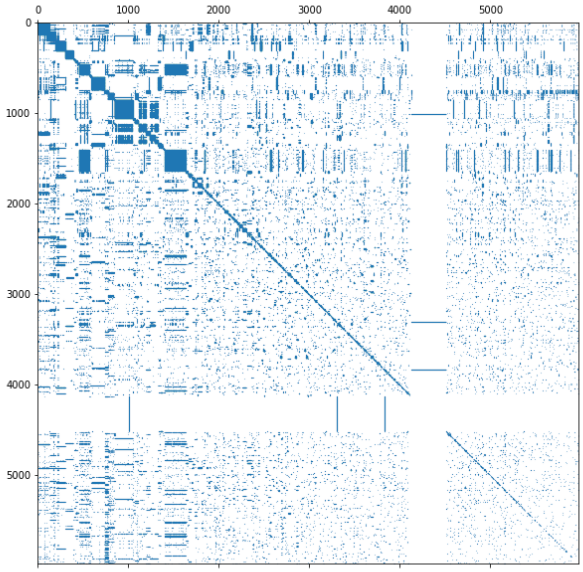


Figure 1: Sparsity of PPI graph with $n = 5973$ and $m = 145778$

For completeness we also report in fig. 2 the degree distribution through the survival function (inverse cumulative) in log log scale. We can notice that the distribution doesn't follow a powerlaw, since its plot has not a linear behaviour in the log log plot.

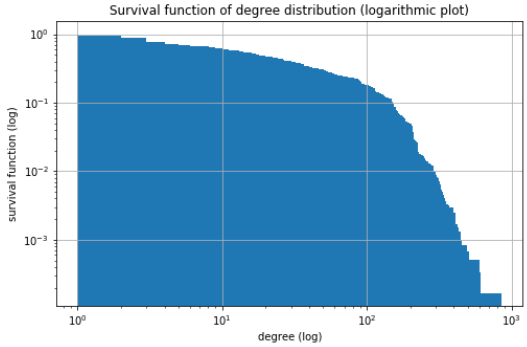


Figure 2: Survival function of degree distribution

1.3 Degree centrality

The first and easier centrality index studied is the degree centrality. It is defined for node i as

$$C_{deg}(i) = \frac{deg(i)}{\max_j deg(j)} \tag{1}$$

where $deg(i)$ is the degree of node i.
The measure is normalized dividing by the maximum degree, so that $0 \leq C_{deg} \leq 1$. The value of C_{deg} for each node is depicted in fig. 3.

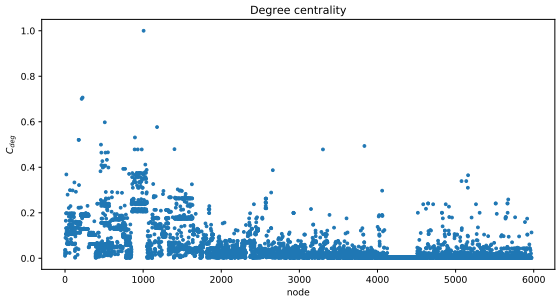


Figure 3: Value of degree centrality for each node in the graph

Even if very simple to compute, the degree centrality can give us useful indications of which node is more important in the network. Its limitation is that it only counts the number of adjacent nodes and doesn't exploit the structure of the graph.

1.4 Katz centrality

Katz centrality tries to assign an importance to each vertex according to the importance of its neighbours. This is an iterative process that at the end should converge to a final solution, which is the resulting centrality for every node. For a vertex i in the graph, the Katz centrality is defined as

$$x_i = \alpha \sum_j A_{ij} x_j + \beta_i \tag{2}$$

where A_{ij} is the adjacency matrix of the graph and $\beta_i = 1$ to give an initial importance to all vertices.

To guarantee convergence the factor α is chosen such that $\alpha < 1/\lambda_1$ with λ_1 largest eigenvalue of A. For our network, $1/\lambda_1 \approx 0.004807$ and so we chose $\alpha = 0.002$. λ_1 was calculated using numerical methods on sparse matrix.

The above eq. (2) can also be expressed in matrix form

$$\mathbf{x} = \alpha \mathbf{A}\mathbf{x} + \mathbf{1} \implies (\mathbf{I} - \alpha \mathbf{A})\mathbf{x} = \mathbf{1} \tag{3}$$

and solved for $\mathbf{x}$ using specific solvers for sparse matrices, without expanding $\mathbf{A}$ to dense format.

The Katz centrality C_{ka} (above was indicated as $\mathbf{x}$ for brevity) is calculated in both ways: with 1000 iterations of eq. (2) and by solving the system in eq. (3). With both methods, we obtained the same result up to approximation errors ($\approx 10^{-15}$). Fig. 4 shows the value of C_{ka} for each node.

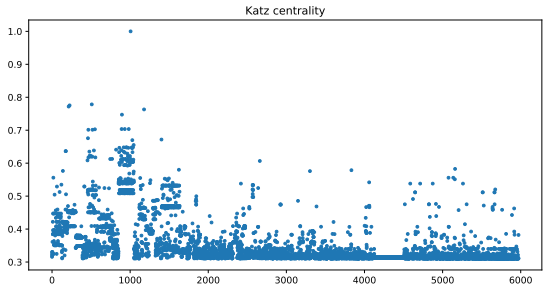


Figure 4: Value of Katz centrality for each node in the graph

1.5 Betweenness centrality

The last centrality considered is the betweenness centrality. It evaluates the importance of a node i according to how many shortest paths pass through it. In particular,

$$C_{betw}(i) = \sum_{s,t \neq i} \frac{\sigma_i(s,t)}{\sigma(s,t)} \tag{4}$$

where $\sigma_i(s,t)$ is the number of shortest path from s to t passing through i and $\sigma(s,t)$ is the total number of shortest path from s to t.
Notice that in case of parallel shortest paths, their effect on the centrality is split equally among them (we divide by their number $\sigma(s,t)$).

A simple algorithm to calculate betweenness centrality can be to compute all the shortest paths between any couple of nodes (s,t) and count how many times these paths pass through node i, weighting by the number of shortest path from s to t. This algorithm runs in $O(n^3)$, which can be too much for large networks.
A better solution for sparse graphs (as our case) is the Brandes' algorithm which runs in $O(nm)$. The basic idea is to perform an augmented BFS that keeps also the count of how many shortest paths are found. The graph is visited then in the reverse order to aggregate the results. A more detailed explanation of the algorithm is given in [2], while the pseudocode, which we

implemented in actual code, is available in the original paper [3]. In fig. 5 the result of the algorithm on our graph is reported. The values are normalized so that $C_{betw} \in [0, 1]$.

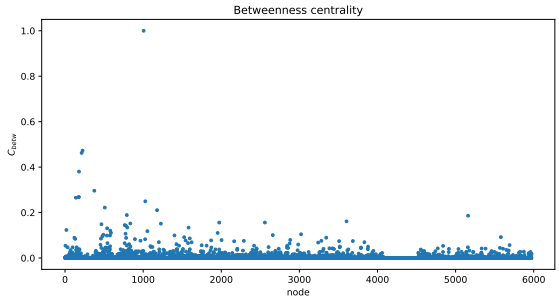
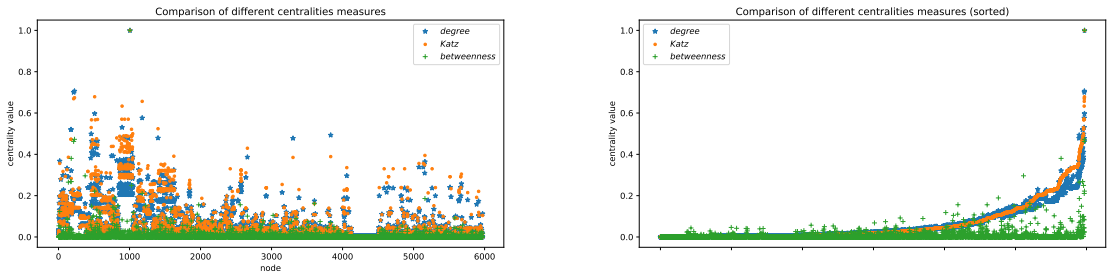


Figure 5: Value of betweenness centrality for each node in the graph

1.6 Final comparison and conclusions

We can now draw a comparison among the different metrics considered. Degree, Katz and betweenness centralities for each node are plotted in fig. 6a. Instead in fig. 6b, they are represented all sorted according to increasing Katz centrality.
We can notice that degree and Katz centrality are correlated and both follow the same trend. Betweenness centrality instead produces low values for most of the nodes (see also fig. 5), and peaks only for some of them, which are the central nodes in the networks where most of the paths pass. The different trend is due to the completely different way to compute the metric: betweenness looks for shortest paths in the whole network, while the others looks only to the adjacent nodes.
Interestingly, all the centralities assume value 1 (maximum) for node #1006. Therefore they all agree this is the most important node in the graph.

In our case of protein-protein interaction graph, this indicates that #1006 is the most important protein, which can be the focus to further studies and analyses by domain experts. But this result was obtained only by looking to the mathematical structure of the graph, without any labelling or knowledge about what the nodes represent. This proves the effectiveness of these methods, which can be applied in many different fields of science.



(a) for every node

(b) sorted (increasing C_{ka})

Figure 6: Comparison of the different centralities indices

Assignment 2: Epidemic processes over the graph

2.1 Introduction

In this assignment, we simulated different epidemic processes on a real graph. The chosen network is a small snapshot of Facebook graph, obtained from users participating in an app ([4]). Each node represents a person, while the edges are friendship between people.
In fig. 7, the sparsity of the adjacency matrix of the graph is reported. The blocky structure indicates the presence of different communities, ie part of the graph which are closely connected together.
It is interesting to study epidemic processes on social networks, because they are a good model for the spread of information across people.
Moreover, since our graph shows this community structure, we can try to see how the information spread inside a community and from one community to another.

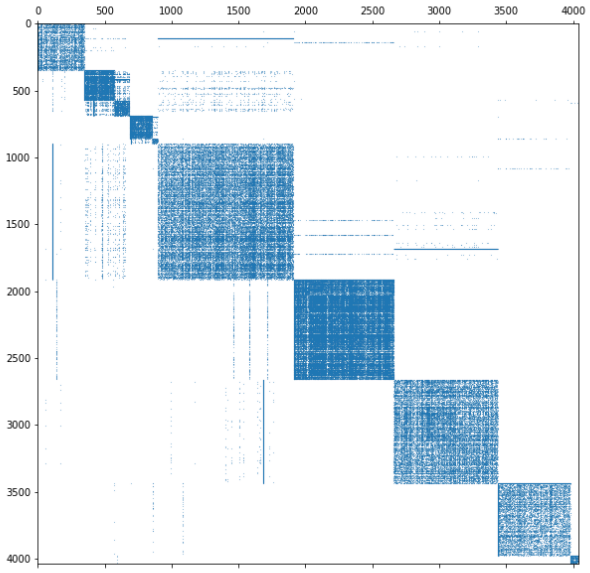


Figure 7: Sparsity of social network graph with $n = 4039$ and $m = 88234$

2.2 Simple SI model

The first model studied is a simple SI model: when a node is infected, it can infect with a probability p each adjacent node. To implement this model, a FIFO queue is used to keep track of which infected node is to be processed and has to spread the infection to the neighbours. More details are reported in algorithm 1. The function *simple_model* outputs:

- $n\_infected$, a vector containing the number of infected nodes for each generation (generation is increased each time the infection spread to a neighbour node and it can represent the distance from the source node)
- $infected$, a vector that for each node associates 1 if was infected in the process, 0 otherwise.

To test the model, we simulate an infection starting from node #2300, which is in the middle of one community block. We choose different values of p (probability of spreading the infection) to test its effect on the process. Following a Monte Carlo approach, 1000 simulations are run for each value of p and the outputs are averaged.
The result of simulation are shown in fig. 8, where we plot the fraction of infected nodes against the generations. $p = 1$ is the degenerate case where all the neighbouring nodes get the infection and this is equivalent to a graph search (BFS).
From the graph we can notice that by increasing the generation, we have more nodes infected, since the algorithm is let run more and propagate for longer the infection. Moreover by increasing p, we have an higher probability to spread the infection, therefore an higher fraction of nodes gets infected.

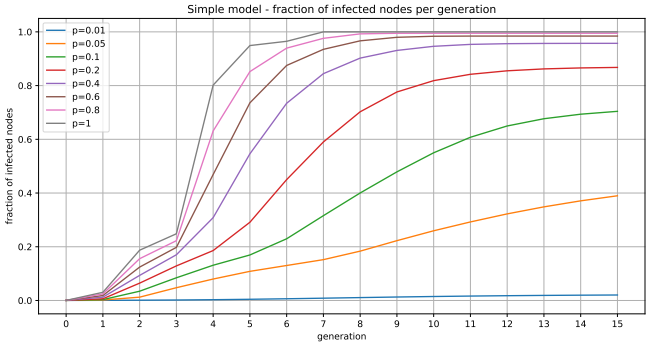


Figure 8: Number of infected nodes per generation for the simple model algorithm

In fig. 9 instead, the model has been run with $p = 0.1$ for 8 generations, starting again from node #2300 and averaging 1000 simulations. The plot shows for each node the probability of it to be infected (1 means that in all 1000 simulations it was infected, 0 if it was never infected). We can clearly notice the the information started in node #2300 spreads easily in the community of the node itself and to the adjacent community of node #1500 (probably because the two communities are well connected). The information struggles instead to be received from the other block communities.

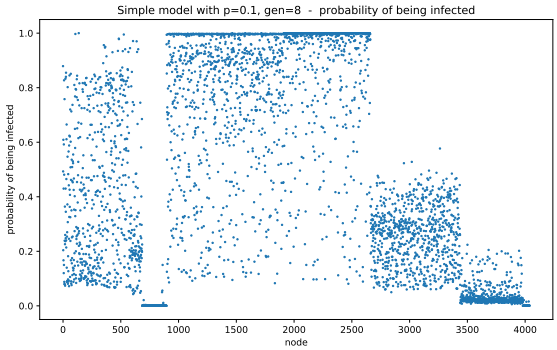


Figure 9: Probability of being infected for the simple model algorithm

Algorithm 1 Simple SI model

```
1: function SIMPLE_MODEL(graph, p, max_gen, starting)
2:     n_infected ← vector, size max_gen + 1, full of 0
3:     infected ← vector, size n, full of 0
4:     queue ← empty queue
5:                                  ▷ queue contains a tuple of 2 elements: node id and generation
6:
7:     enqueue ((starting, 0)) → queue
8:     infected[starting] ← 1
9:     n_infected[0] ← 1
10:
11:    while queue not empty and g ≤ max_gen do
12:        dequeue ((v, g)) ← queue
13:        for i ∈ neighbours of v do
14:            if infected[i] = 0 and random() < p then        ▷ random() ∼ Uniform[0, 1]
15:                infected[i] ← 1
16:                n_infected[g + 1]+ = 1
17:                enqueue ((i, g + 1)) → queue
18:            end if
19:        end for
20:    end while
21:
22:    return cumsum(n_infected)/n and infected
23: end function
```

2.3 Bootstrap percolation

As second epidemic process, we implemented the bootstrap percolation: a node gets infected if at least r of its neighbours are infected. The pseudocode presented in algorithm 2 is very similar to algorithm 1, with the difference that *starting* is now a vector of initial infected nodes and the addition of the vector *pressure* which counts how many infected nodes are adjacent to the considered node. When the latter becomes $\geq r$ the considered node becomes infected itself and it's added in the queue.

The results of the simulations are shown in fig. 10, for different values of r. The starting points are 10 nodes inside the community of nodes from #2000 to #2500 (the same considered with the previous model). No Monte Carlo approach is needed, since once fixed the starting nodes, the spreading is deterministic. As we can notice from the picture, for the case $r = 5$ the infection is not able to propagate, because too many adjacent nodes are required to get infected. Smaller values of r makes easier to pass the infection. The case $r = 1$ is a degenerate case, which is a simple graph search.

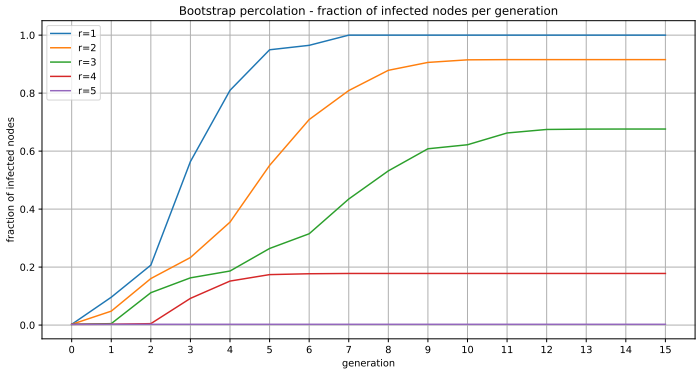


Figure 10: Number of infected nodes per generation for bootstrap percolation

Algorithm 2 Bootstrap percolation

function BOOTSTRAP_PERCOLATION($graph, r, max\_gen, starting$)
    $n\_infected \leftarrow$ vector, size $max\_gen + 1$, full of 0
    $infected \leftarrow$ vector, size n, full of 0
    $pressure \leftarrow$ vector, size n, full of 0
    $queue \leftarrow$ empty queue
                ▷ $queue$ contains a tuple of 2 elements: node id and generation

    enqueue $((starting, 0)) \rightarrow queue$
    $infected[starting] \leftarrow 1$
    $n\_infected[0] \leftarrow length(starting)$

    while $queue$ not empty and $g \leq max\_gen$ **do**
        dequeue $((v, g)) \leftarrow queue$
        for $i \in$ neighbours of v **do**
            $pressure[i] += 1$
            if $infected[i] = 0$ and $pressure[i] \geq r$ **then**
                $infected[i] \leftarrow 1$
                $n\_infected[g + 1] += 1$
                enqueue $((i, g + 1)) \rightarrow queue$
            end if
        end for
    end while

    return $cumsum(n\_infected)/n$ and $infected$
end function

2.4 Bootstrap percolation (stochastic version)

Finally the bootstrap percolation presented in section 2.3 is modified to allow different values of r. Now instead of fixing it to a constant value, it is expressed as a vector that associates for each number of infected neighbours a probability to get the infection.
For example, the following mapping $r = [1 \to 0, \quad 2 \to 0.4, \quad 3 \to 1]$ means:

- if a node has 1 infected neighbour $\to$ probability 0 of becoming infected
- if a node has 2 infected neighbours $\to$ probability 0.4 of becoming infected
- if a node has 3 or more infected neighbours $\to$ probability 1 of becoming infected

The sampling of the probability distribution is performed each time the number of adjacent infected neighbours changes. The algorithm to implement this version is exactly the same as algorithm 2, with a small modification in the **If statement**.

Using the mapping above, we run 1000 simulations and take the average. The starting points are the same 10 nodes between #2000 and #2500 considered before. The results are shown in fig. 11 and compared with a deterministic bootstrap with $r = 2$ and $r = 3$. As expected since our mapping is in the middle between $r = 2$ and $r = 3$ (we allow sometimes $r = 2$ but always $r = 3$), the curve representing the fraction of infected nodes is bounded by the two deterministic cases.

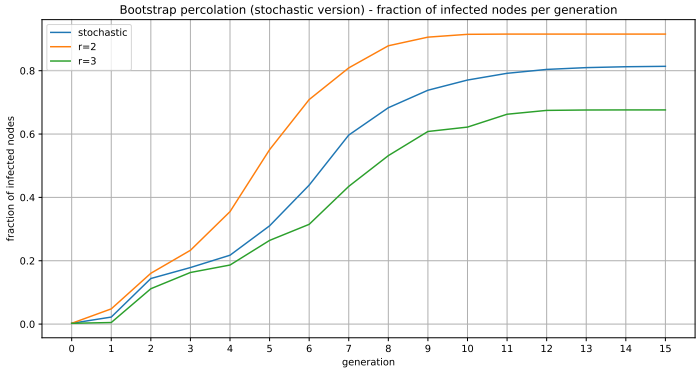


Figure 11: Number of infected nodes per generation for bootstrap percolation (stochastic version)

Assignment 3: Erdős–Rényi model

3.1 Introduction and graph generation

The objective of this third assignment is to generate and test the properties of a $G(n, p)$ graph. To generate it, we propose the algorithm 3. Essentially for each node i, we add an edge to n_edges random previous nodes, with $n\_edges \sim Binomial(i, p)$. To tackle the fact we only add edges to previous nodes, the sparse matrix is finally added to its transposed version, obtaining an undirected graph.
The time complexity of this algorithm is $O(n + m)$, in opposition to the naive approach of sampling a random variable for each possible entry of the matrix (would be $O(n^2)$, too much for large n).

Algorithm 3 Generate $G(n, p)$

```
1: function GENERATE_GNP(n, p)
2:     graph ← sparse matrix, size n × n
3:
4:     for i = 0 → n do
5:         n_edges ← Binomial(i, p)
6:         if n_edges > 0 then
7:             choices ← pick n_edges random elements from 0...i (no repetitions)
8:             graph[i, choices] = 1
9:         end if
10:    end for
11:
12:    return graph + transpose(graph)
13: end function
```

The sparsity of a $G(1000, 0.1)$ is reported in fig. 12. The sparsity of the matrix is uniform (we don't have gaps or recognizable structures) and the number of edges ($m = 49721$) is compatible with the expected number of edges we know from theory

$$\mathbf{E}(m) = \frac{n(n-1)p}{2} = 49950 \tag{5}$$

3.2 Testing $G(n, p)$ properties

To test the properties of the Erdős–Rényi model, we run several simulations varying both n and $p(n)$. For each choice of the two parameters, 100 simulations are performed and the results are averaged using a Monte Carlo approach. The full set of results produced are reported in table 2. Here we focus instead on the results for $n = 100000$ (table 1), which is the largest n we simulated (so more representative for an asymptotic analysis).

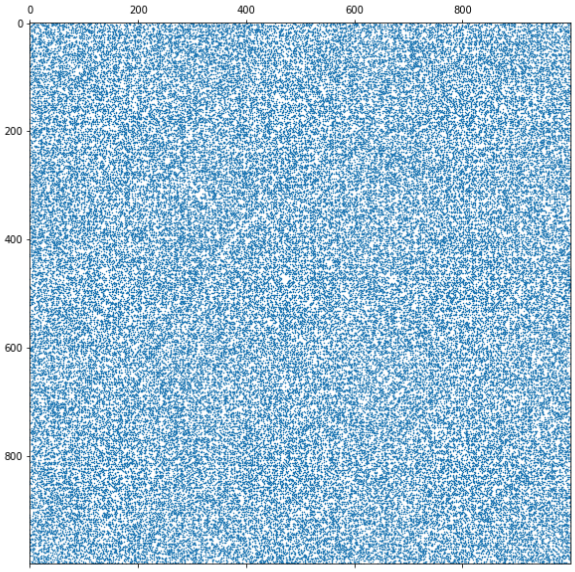


Figure 12: Sparsity of $G(n, p)$ with $n = 1000$ and $m = 49721$

3.2.1 Distribution of node degree

According to theory, the distribution of node degree should follow a binomial distribution (light tail). The minimum and maximum values of the node degree for all different n and $p(n)$ are reported in table 1, while in fig. 13 the case $n = 100000$ and $p(n) = 2log(n)/n$ is shown.
As we can notice from the plot, the simulation fits perfectly a $Binomial(n-1, p)$.

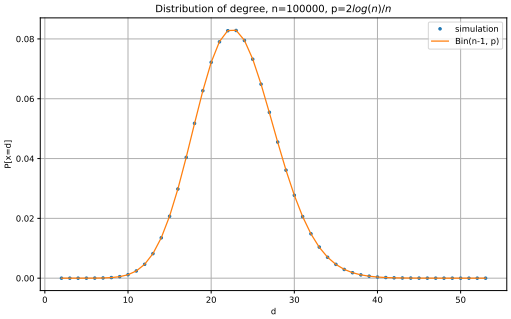


Figure 13: Distribution of degree $G(n, p)$ with $n = 100000$ and $p(n) = 2log(n)/n$

3.2.2 Diameter

Calculating the exact diameter of a graph, ie the distance between the farthest nodes in the graph, is computationally expensive and not viable for large graphs like the one we are considering. We perform then an approximated calculation, which gives us a lower bound on the diameter. We sample randomly 20 nodes in the largest connected component and we compute the distance from them toward all the other nodes in the largest connected component.

The diameter is considered as the maximum distance found in this way. The results are reported in table 1. The first 2 rows should be excluded since the largest connected component is very small (we don't have a giant component). But from the third row, we can notice that $G(n,p)$ shows a small diameter ($\sim log(n)$), which decreases with increasing $p(n)$.

3.2.3 Giant component

The giant component is a connected component in which the number of nodes scales with n. To test its presence, we count the number of nodes in the largest connected component (*1st cc*) and we consider it a giant component if it has at least $\frac{n}{10}$ nodes. In the column *giant* of table 1 we report the fraction of graphs that present this giant component (we run a Monte Carlo simulation with 100 realisation for each n and $p(n)$, 0 means none of the 100 graphs have this property, 1 all have it).
The transition from no giant to giant is abrupt between $0.9/n$ and $1.1/n$. As expected from theory, $1/n$ is a threshold function for the presence of the giant component. Moreover from the column *2nd cc*, we can read the size of the second largest connected component. Again as expected from theory, when there is a giant component the 2nd connected component is a small component that scales at most with $log(n)$. The giant component is therefore unique.

3.2.4 Connectivity

Finally we analysed the connectivity of the graph. A graph is connected if there exists at least a path between any pair of nodes. Again from the table, we can notice that the graph is not connected up to the function $p(n) = 0.9log(n)/n$ where we have a transition. $log(n)/n$ is a threshold function for the connectivity of the graph. We can also notice that approaching the threshold, the last nodes that remain outside the giant component and prevent the connectivity are isolated nodes (looking at *2nd cc* we have it 1 for $0.9log(n)/n$ and $1.1log(n)/n$).

p(n)	connected	giant	diameter	1st cc	2nd cc	deg_min	deg_max
0.5/n	0.00	0.0	12.55	25.90	22.47	0	8
0.9/n	0.00	0.0	50.52	301.06	211.81	0	9
1.1/n	0.00	1.0	160.08	17384.43	319.00	0	10
1.5/n	0.00	1.0	52.59	58202.49	38.24	0	11
2/n	0.00	1.0	31.90	79673.69	14.96	0	13
0.5 log(n)/n	0.00	1.0	11.08	99678.32	2.02	0	22
0.9 log(n)/n	0.03	1.0	7.60	99996.89	1.00	0	31
1.1 log(n)/n	0.76	1.0	7.00	99999.74	1.00	0	38
1.5 log(n)/n	1.00	1.0	6.00	100000.00	-	1	44
2 log(n)/n	1.00	1.0	5.00	100000.00	-	2	54
0.01	1.00	1.0	3.00	100000.00	-	153	490

Table 1: Results from simulations of G(n,p), focus on $n = 100000$

n	p(n)	connected	giant	diameter	1st cc	2nd cc	deg_min	deg_max
100	0.5/n	0.00	0.05	3.89	5.90	4.310000	0	4
100	0.9/n	0.00	0.71	8.07	14.88	7.990000	0	6
100	1.1/n	0.00	0.92	11.76	27.47	9.270000	0	8
100	1.5/n	0.00	1.00	15.03	53.39	7.190000	0	7
100	2/n	0.00	1.00	13.68	78.19	3.710000	0	9
100	0.5 log(n)/n	0.00	1.00	11.44	86.58	2.210000	0	9
100	0.9 log(n)/n	0.21	1.00	6.69	98.46	1.050633	0	12
100	1.1 log(n)/n	0.59	1.00	5.83	99.44	1.000000	0	14
100	1.5 log(n)/n	0.85	1.00	4.43	99.84	1.000000	0	19
100	2 log(n)/n	0.98	1.00	4.03	99.98	1.000000	0	23
100	0.01	0.00	0.82	9.34	19.72	8.430000	0	6
1000	0.5/n	0.00	0.00	6.85	11.58	8.880000	0	5
1000	0.9/n	0.00	0.09	19.13	56.58	28.490000	0	7
1000	1.1/n	0.00	0.72	31.08	158.04	48.360000	0	8
1000	1.5/n	0.00	1.00	30.76	579.34	14.720000	0	9
1000	2/n	0.00	1.00	20.10	799.31	6.220000	0	10
1000	0.5 log(n)/n	0.00	1.00	10.94	965.21	2.090000	0	14
1000	0.9 log(n)/n	0.13	1.00	6.91	998.03	1.022989	0	18
1000	1.1 log(n)/n	0.63	1.00	6.05	999.46	1.027027	0	21
1000	1.5 log(n)/n	0.95	1.00	5.01	999.95	1.000000	0	29
1000	2 log(n)/n	1.00	1.00	4.15	1000.00	-	1	32
1000	0.01	0.96	1.00	5.04	999.96	1.000000	0	27
10000	0.5/n	0.00	0.00	9.49	17.55	14.540000	0	7
10000	0.9/n	0.00	0.00	30.81	134.35	89.460000	0	8
10000	1.1/n	0.00	0.87	94.85	1676.05	169.850000	0	9
10000	1.5/n	0.00	1.00	42.65	5825.63	24.420000	0	10
10000	2/n	0.00	1.00	26.21	7967.78	10.300000	0	12
10000	0.5 log(n)/n	0.00	1.00	10.90	9896.13	2.040000	0	18
10000	0.9 log(n)/n	0.11	1.00	7.09	9997.51	1.000000	0	26
10000	1.1 log(n)/n	0.71	1.00	6.18	9999.65	1.000000	0	30
10000	1.5 log(n)/n	0.99	1.00	5.31	9999.99	1.000000	0	35
10000	2 log(n)/n	1.00	1.00	5.00	10000.00	-	2	42
10000	0.01	1.00	1.00	3.00	10000.00	-	55	150
100000	0.5/n	0.00	0.00	12.55	25.90	22.470000	0	8
100000	0.9/n	0.00	0.00	50.52	301.06	211.810000	0	9
100000	1.1/n	0.00	1.00	160.08	17384.43	319.000000	0	10
100000	1.5/n	0.00	1.00	52.59	58202.49	38.240000	0	11
100000	2/n	0.00	1.00	31.90	79673.69	14.960000	0	13
100000	0.5 log(n)/n	0.00	1.00	11.08	99678.32	2.020000	0	22
100000	0.9 log(n)/n	0.03	1.00	7.60	99996.89	1.000000	0	31
100000	1.1 log(n)/n	0.76	1.00	7.00	99999.74	1.000000	0	38
100000	1.5 log(n)/n	1.00	1.00	6.00	100000.00	-	1	44
100000	2 log(n)/n	1.00	1.00	5.00	100000.00	-	2	54
100000	0.01	1.00	1.00	3.00	100000.00	-	153	490

Table 2: Results from simulations of G(n,p)

Assignment 4: Barabási–Albert model

4.1 Introduction

In this last assignment, we are required to implement the Barabási–Albert model and to test its properties. BA is a generative model, which means that the graph is defined through a process/algorithm and cannot be defined mathematically by associating a probability space (which can be done for instance for G(n,p)). Simulations are therefore even more essential to study the properties of the model.

The main concept behind this model is the preferential attachment rule, according to which a new node that joins the network will be connected with higher probability to most popular nodes. Indeed, we start from an initial graph and we add iteratively one node at a time and connect it to m_{BA} nodes already in the graph (we didn't use the notation m to avoid confusions with the number of edges). The probability p_u for a new coming node to connect to a node u in the graph is given by eq. (6)

$$p_u = \frac{d_u^\gamma}{\sum_{v \in V} d_v^\gamma} \tag{6}$$

where d_u is the current degree of the node u and γ is the exponent that defines the proportion between p_u and d_u.

4.2 Algorithms

We now present two different algorithms to create a BA graph. The first approach adopted, which is presented in algorithm 4, is able to generate very efficiently a BA graph for the case $p_u \propto d_u$. It's employing essentially an additional vector (urn) which is constructed so that the number of times each node i of the graph appears in the vector is equal to the $deg(i)$. Picking a random element from urn corresponds then to sampling the discrete distribution described in eq. (6) with $\gamma = 1$.

The second approach (algorithm 5) instead computes the distribution each time and samples it, instead of employing an auxiliary vector. This method is more computationally expensive, but it's the only viable for the case $p_u \not\propto d_u$. To further optimize it, the cumulative distribution to perform the sampling from is not computed entirely, but just for the portion needed. Notice indeed in the function *random_choice* described in algorithm 6 the use of $sum\_p_u = \sum_u p_u$, so that we don't have to loop over all nodes in p_u to compute the sum and the *while* in line 14 that stops as soon as a new value is picked.
With this optimisation, especially for the case $p_u \propto d_u^\gamma$ with $\gamma > 1$, the sampling can be computed very fast, because the initial nodes are big hubs, usually chosen for the new edges. The computation of the distribution is therefore stopped after few iterations. We reduced in this way the average case complexity, while the worst case one remains still the same.

4.3 Results

In fig. 14, we report the sparsity of the adjacency matrix of a Barabási–Albert graph generated with $n = 1000$ and $m_{BA} = 3$. As expected, older nodes (first rows and columns) have more connections (more dense points), since they had more chances to be chosen from new nodes.

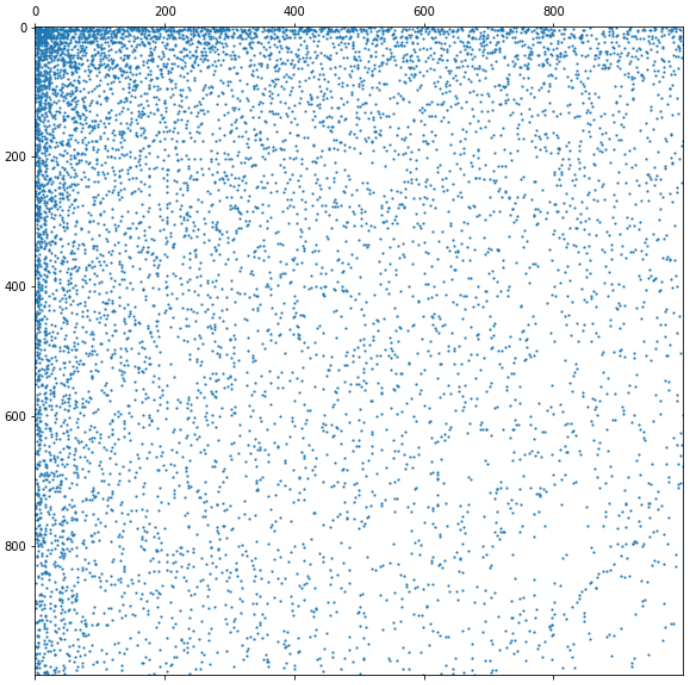


Figure 14: Sparsity of Barabási–Albert graph with $n = 1000$, $m_{BA} = 3$ and $m = 3990$ (number of edges)

The model was then run with $n = 100000$ and $m_{BA} = 3$, for the case $p_u \propto d_u$, $p_u \propto \sqrt{d_u}$ and $p_u \propto d_u^{1.5}$ (algorithm 4 was used for the case $p_u \propto d_u$, while algorithm 5 for the other two). For each of these functions of the degree, 100 graphs were generated. Following a Monte Carlo approach, the results shown in table 3 and figs. 15 and 16 are an average on those realisations.

gamma γ	diameter	clustering	max degree
1	7.54	0.000886	1540
0.5	8.52	0.000120	110
1.5	3.50	0.845655	97321

Table 3: Results for BA, $n = 100000$, $m_{BA} = 3$, average from 100 graphs for each γ

The clustering coefficient is computed approximately by taking at random 50000 nodes, and for each node picking 2 random neighbours (without repetition) and check if they are connected. The clustering coefficient is then simply the number of connected triples found divided by the number of trials (50000 in our case). The approximated diameter is found instead by computing, starting from 20 random nodes (without repetition), the maximum of distances toward all the other nodes.

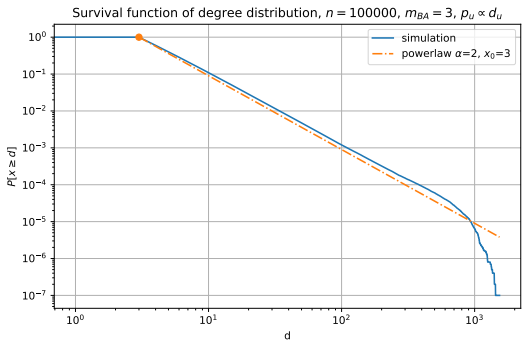


Figure 15: Survival function of degree distribution, BA with $n = 100000$, $m_{BA} = 3$, $\gamma = 1$

The result we obtain for $\gamma = 1$ are compatible with the theory for BA. Indeed we get a powerlaw for the degree distribution with coefficient 3 for the pdf

$$P[d_u = k] \propto \frac{1}{k^3} \tag{7}$$

which correspond to a coefficient of 2 for the cumulative as plotted in fig. 15.
As expected, the average diameter is small $diam = 7.54$. Also the clustering coefficient is very small, $C = 0.000886$.

Different results are obtained for the case in which $\gamma \neq 1$. In fig. 16 we compare the distribution of degree for different γ. For the case $p_u \propto \sqrt{d_u}$ the distribution decreases faster than the previous case and it's more concentrated for lower degrees.
The clustering is lower and the diameter is larger with respect to the simulations for $\gamma = 1$. This is producing a weaker preferential attachment.
For $p_u \propto d_u^{1.5}$, on the other hand, the distribution indicates the presence of very high degree nodes. Notice from table 3 that the maximum degree is 3 order of magnitude larger than the $\gamma = 1$ case. What happens here is that few nodes start to become larger and larger and this trend is accelerated by the superlinear relation of p_u and d_u. This is a stronger version of the preferential attachment. Since the presence of these big hubs, the diameter is reduced and the clustering is increased.

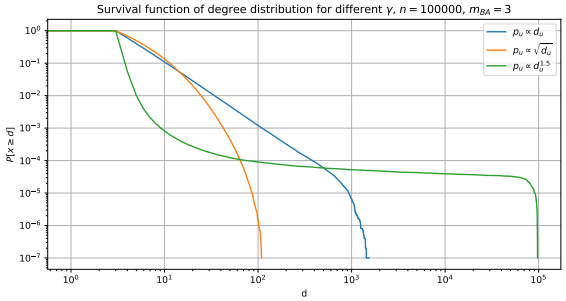


Figure 16: Comparison of degree distributions for different values of gamma

Algorithm 4 Optimized algorithm to generate BA for the case $p_u \propto d_u$

function GENERATEBA(n, m_{BA})
    $graph \leftarrow$ sparse matrix, size $n \times n$
    $urn \leftarrow$ vector, size $2nm_{BA}$

    $u \leftarrow 1$
    while $u < m_{BA} + 1$ **do** ▷ generate a complete graph of $m_{BA} + 1$ nodes
        $graph[u][0...u] \leftarrow 1$
        $urn[len(urn)...len(urn) + m_{BA}] \leftarrow u$
        $u \leftarrow u + 1$
    end while

    while $u < n$ **do** ▷ add a new node at each iteration

        $choices \leftarrow$ pick m **random** elements from urn (no repetitions)

        $graph[u][choices] \leftarrow 1$ ▷ connect node u to all nodes in *choices*

        $urn[len(urn)...len(urn) + m_{BA}] \leftarrow u$
        $urn[len(urn)...len(urn) + m_{BA}] \leftarrow choices$
        $u \leftarrow u + 1$
    end while

    return $graph + transpose(graph)$
end function

Algorithm 5 Generate BA, recomputing each time the distribution

function GENERATEBA($n, m_{BA}, f(x) = x^{\gamma}$)
  $graph \leftarrow$ sparse matrix, size $n \times n$
  $d_u \leftarrow$ vector, size n, full of m ▷ d_u is the degree of each node, initially set to m_{BA}
  $p_u \leftarrow$ vector, size n, full of $f(m_{BA})$
  ▷ p_u is the probability not normalized to pick a node

  $sum\_p_u \leftarrow 0$ ▷ sum_p_u is the $\sum_u p_u$

  $u \leftarrow 1$
  while $u < m_{BA} + 1$ **do** ▷ generate a complete graph of $m_{BA} + 1$ nodes
    $graph[u][0...u] \leftarrow 1$
    $u \leftarrow u + 1$
  end while

  $sum\_p_u \leftarrow (m_{BA} + 1) * f(m_{BA})$
  ▷ update the sum: added m_{BA} nodes with degree m_{BA}

  while $u < n$ **do** ▷ add a new node at each iteration

    $choices \leftarrow random\_choice(size = m_{BA}, p = p_u[0...u], sum\_p = sum\_p_u)$

    $graph[u][choices] \leftarrow 1$ ▷ connect node u to all nodes in $choices$

    $sum\_p_u -= p_u[choices]$ ▷ update the d_u, p_u and sum_p_u after the addition
    $d_u[choices] += 1$
    $p_u[choices] = f(d_u[choices])$
    $sum\_p_u += p_u[choices]$

    $u \leftarrow u + 1$
  end while

  return $graph + transpose(graph)$
end function

Algorithm 6 Function *random_choice* to perform sampling from a discrete distribution

```
1: function RANDOM_CHOICE(size, p, sum_p)
2:     cdf ← vector, size length(p)
3:     choices ← vector, size size
4:     cdf[0] = p[0]
5:
6:     i ← 0
7:     while i < size do                          ▷ loop on the number of choices to make
8:         r ← random() * sum_p_u                 ▷ random() returns a Uniform[0, 1]
9:
10:        if r < cdf[j] then                     ▷ if r was in the part of cdf already computed
11:
12:            new ← binary search in cdf[0...j + 1], such that cdf[new] ≤ r < cdf[new + 1]
13:        else
14:            while cdf[j] < r do                ▷ continue to compute the cdf
15:                j+ = 1
16:                cdf[j] ← cdf[j − 1] + p[j]
17:            end while
18:
19:            new ← j                            ▷ since the while stopped, j is inside the block
20:        end if
21:
22:        if new is not already in choices then
23:            choices[i] ← new
24:            i+ = 1
25:        end if
26:    end while
27:
28:    return choices
29: end function
```

References

[1] KONECT. *Reactome network dataset*. URL: http://konect.uni-koblenz.de/networks/reactome.

[2] Andrea Marino. *Centrality Measures - Computing Closeness and Betweenness*. URL: http://pages.di.unipi.it/marino/centrality18.pdf.

[3] Ulrik Brandes. "A faster algorithm for betweenness centrality". In: *Journal of mathematical sociology* 25.2 (2001), pp. 163–177.

[4] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection - Social circles: Facebook*. http://snap.stanford.edu/data/ego-Facebook.html.