Complex Networks - Homework 6

Group 1: Iwan Pasveer, Flavia Leotta, Noah Frinking, Dylan Gavron

22nd October 2024

1 Introduction

This document is supported by markdown file "CN_HW6_Group1.ipynb", refer to it for the code. For more information about the libraries used, see section "Softwares and libraries".

2 Homework 6.1

We implemented the function:

`Link_Stub_Reconnection(degree_seq)` (1)

that takes a specific degree sequence (provided as a list) as argument and performs random link stub reconnection to randomly generate a Configuration Model graph. It includes a warning if a degree sequence with non-even sum is provided. By using $seed = 5$, we show an example of its implementation on `degree_seq = [3,2,2,1,2,3,1,4]` in Figure 1.

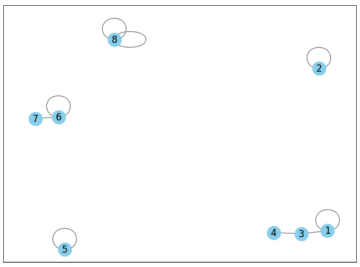

Figure 1: Implementation of function (1) on degree sequence = [3,2,2,1,2,3,1,4].

3 Homework 6.2

To make the algorithm more time efficient, we used a Fisher Yates shuffle and implemented a new function:

`Link_Stub_Reconnection_shuffle(degree_seq)` (2)

3.1 Time complexity of function (2)

We initialize the code by defining a condition (`if`): if this condition is not met, the algorithm is terminated, but since we're interested in the worst case analysis, we will assume that our algorithm gets explored. The conditions are:

1. Is the list provided an instance of class list? This has complexity $O(1)$.
2. Does the list provided have an even sum of the degrees? This has complexity $O(n)$ because it has to sum all of the n degrees.

All together we say that the `if` **condition has complexity** $O(n)$, where n is the total number of degrees in the degree sequence, so the total number of nodes in the graph.

When we pass the condition, we initialize, only once:

1. An empty graph. Complexity is constant, $O(1)$.
2. An empty list of stubs, $O(1)$.

Then the `for` loop iterates n times and:

1. adds a node to the graph. Time complexity is constant, $O(1)$, and since it's iterated n times becomes $O(n)$;
2. extends the list of stubs according to the node degree. By the end of the loop this has complexity $O(2m)$ where m is the total numnber of edges in the graph.

The total complexity of the `for` **loop will be O(n + 2m)**.

The `random.shuffle(stubs)` **has complexity O(2m)**, which is the total number of stubs.

Then there's a `while` loop that is broken when there are less than 2 stubs in the list. Since there are $2m$ stubs, the operation is repeated m times (since 2 stubs are removed each time). Removing two stubs and adding a stub all have a constant complexity of $O(1)$. The `while` **loop will have complexity O(3m)**.

The total complexity is given by:

$$O(n) + O(1) + O(1) + O(n + 2m) + O(2m) + O(3m) = O(n + m) \tag{3}$$

The worst case scenario is a complete regular graph in which each vertex is connected to all the other $(n - 1)$ vertices. That way, the total number of edges is $m = n(n - 1)/2$, and $m = 0(n^2)$. This gives a total complexity of $O(n^2)$.

In the more probable case of a sparse graph, though, we can assume that $m = o(n^2)$, and we can approximate the complexity as $O(n + m)$.

3.2 Comparison with function (1)

The previous function (without the random shuffle), starts similarly. It has the same condition, initialization and stub list creation, so, to summarize:

- `if` condition = $O(n)$;
- Empty graph = $O(1)$;
- Empty stub list = $O(1)$;
- `for` loop = $O(n + 2m)$.

The difference starts with the `while` statement. Again, we repeat the operation until we have at least 2 stubs, so the time complexity will be $O(m)$, but the content of the `while` loop is different. Every time we randomly choose a stub, the time complexity is $O(1)$, but removing it from the list has complexity $O(2m)$ because the algorithm has to scan through the list to remove it. Since we're doing worst case analysis, then the worst case scenario is that it requires going through all 2m stubs before finding the target node. So, the `while` loop has a total complexity of:

$$O(m(1 + 2m + 1 + 2m)) = 0(m(4m + 2)) = O(m^2) \tag{4}$$

The total complexity of the algorithm then will be:

$$O(n) + O(1) + O(1) + O(n + 2m) + O(m^2) = O(n + m^2) \tag{5}$$

As we said before, this is the complexity of a sparse graph, while for a regular one $m = O(n^2)$ and the complexity grows up until $O(n^4)$.

In both cases, the function that doesn't use the shuffle is less efficient and more time consuming than the one that does.

4 Homework 6.3

We run function (2) 60 times to produce graphs with the following degree sequences:

- $\vec{k}^{(1)} = [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]$
- $\vec{k}^{(2)} = [7, 4, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1]$

producing a total of 120 graphs. For each, we count the number of self loops and repeated edges: in 2 we report the average of the results.

Table 1: Average of the results for homework 6.3

Degree sequence	**Average number of self loops**	**Average number of multiple edges**
$\vec{k}^{(1)}$	0.300	0.267
$\vec{k}^{(2)}$	1.150	0.717

Table 2: Average number of self loops and multiple edges obtained by running function (2) 60 times for each degree sequence. To reproduce these results, we use `random.seed(5)` to randomly generate another 60 seeds used for the 60 iterations of function (2) that, we remind the reader, uses a Fisher Yates Shuffle to perform stub reconnection.

To prove that there is a significant difference in the number of self loops for graphs of the two degree sequences, we perform statistical analysis on RStudio. We used code:

```
t.test(tot_self_loops_k1,tot_self_loops_k2, paired = FALSE)
```
(6)

Since we have 60 values for each test, we use an unpaired t-student hypothesis testing. We set the null hypothesis H_0 as: £true difference in means is equal to 0", while the alternative hypothesis H_1 as: "true difference in means is not equal to 0". The threshold chosen for the p-value is 0.05.

We reject the null hypothesis for both tests (self loops test p-value: 7.094e-11, multiple edges test p-value: 0.000639), so we can say that there is statistical evidence that the two degree sequences produce different amounts of self loops and multiple edges.

This is probably because $\vec{k}^{(2)}$ has some nodes (especially the first and the second node) that have a significantly higher degree than the others in the same graph and surely more varied than the ones in $\vec{k}^{(1)}$ (since all nodes have degree 2). Nodes with an higher degree have an higher probability of forming self loops and multiple edges. For example, for a stub on node 1 of $\vec{k}^{(2)}$, over one third of the available stubs for it to connect to are also on node 1. For a stub on any node of $\vec{k}^{(1)}$, only 1 out of the 24 remaining stubs are on the same node, so a self-loop is much less likely.

5 Homework 6.4

We implemented the repeated configuration model by slightly changing function (2), so that when a graph with self-loops and multi-edges is discovered, it is discarded and the function is recursively called. Our resulting function is:

```
repeated_configuration_model(degree_seq, attempt = 1)
```
(7)

where attempt keeps track of the number of the attempts required to produce the desired graph. Another difference from function (2) is that this function not only returns the graph created, but also the number of attempts that were made to create it.

As before, we performed an unpaired t-student hypothesis testing on RStudio, to see if the average number of attempts for $\vec{k}^{(1)}$ (1.85) is different from the one for $\vec{k}^{(2)}$ (40.73).

- H_0: there is no difference between $\vec{k}^{(1)}$ and $\vec{k}^{(2)}$
- H_1: difference between $\vec{k}^{(1)}$ and $\vec{k}^{(2)}$
- p-value: 2.98e-10

We rejected the hypothesis, so the two populations were different. Furthermore, we tested if $\vec{k}^{(2)} > \vec{k}^{(1)}$ by using the test:

```
t.test(attempts_k1, attempts_k2, paired = FALSE, alternative = "less")
```
(8)

and the resulting p-value was 1.49e-10, so we can say that there's statistical significance that $\vec{k}^{(1)}$ requires less attempts than $\vec{k}^{(2)}$.

This might be because $\vec{k}^{(2)}$ has a degree distribution that's not very balanced, and it requires more complex pairing within nodes. As we showed in homework 6.3 the probability to create self loops and multiple edges was higher in $\vec{k}^{(2)}$, so it follows that the algorithm will have to discard the graph generated with $\vec{k}^{(2)}$ more often, and thus will take more attempts to create a simple graph.

6 Homework 6.5

We implemented a function that performs local rewiring on a given graph G and, as a standard, performs 30 rewires, but this can be changed by the user:

```
local_rewiring(G, rewires = 30)
```
(9)

This function returns both the new graph after the rewiring, and a list containing the average local clustering coefficients at each iteration. We implemented it on graphs G_1 and G_2 in figure 6.6 of lecture notes (as reported in 2).

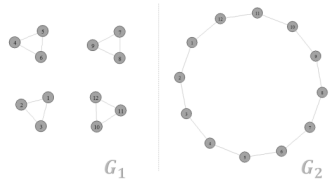

Figure 2: Figure 6.6 of lecture notes showing graphs G_1 and G_2 with the same degree sequence [2,2,2,2, 2,2,2,2, 2,2,2,2, 2,2,2,2].

For each iteration of the local rewiring algorithm, we report the average local clustering coefficient in Figure 3.

Figure 3: Average local clustering coefficient over time (as in number of rewires).

From NetworkX documentation, we know that the average clustering coefficient is calculated by the library as:

$$C = \frac{1}{n}\sum_{u \in G} c_u = \frac{1}{n}\sum_{v \in G} \frac{2T(u)}{k_u(k_u - 1)} \tag{10}$$

where T(u) is the number of triangles through node u and k_u is the degree of u.

G1 starts with average clustering coefficient equal to 1 because each node has clustering coefficient of 1. From Figure 2 we can appreciate that G1 is composed by four clusters composed by 3 nodes each: each node has degree 2 and is part of one triangle, so it follows that, according to equation 10, they all have clustering coefficient equal to 1. G2, on the other hand, starts with average clustering coefficient equal to 0 because none of the nodes is part of any triangle.

Over time, as rewiring operations are applied, the clustering coefficient of G1 gradually decreases while G2's increases, because more randomness is introduced in their nodes' edges. Over time, since both graphs share the same degree sequence, they tend to the same average clustering coefficient. This coefficient will be a middle ground between the two extreme cases of the beginning (1 and 0) since the formation of triangles through the rewiring algorithm is random and there's no deliberate pattern.

7 Homework 6.6

We modified twice the Barabási-Albert model in the source code given in the Lecture notes:

`BA_sublinear_PA(N, m0, m)` `BA_superlinear_PA(N, m0, m)`

to sample from a sub-linear probability distribution and a super-linear probability distribution, respectively. N is the total number of nodes while m0 are the initial nodes in the graph that are all connected to each other. It follows that N - m0 are the nodes that are added later, as required by the preferential attachment model, and m are the number of links that these nodes form when added to the graph, following the probabilities distributions shown below.

The sub-linear and super-linear preferential attachment models are given by, respectively:

$$Pr(k) = \frac{\sqrt{k}}{\sum_j \sqrt{k_j}} \qquad Pr(k) = \frac{(k)^2}{\sum_j (k_j)^2}$$

We implemented the functions on the test network generated in the Lecture notes, and we report the resulting graphs in Figure 4.

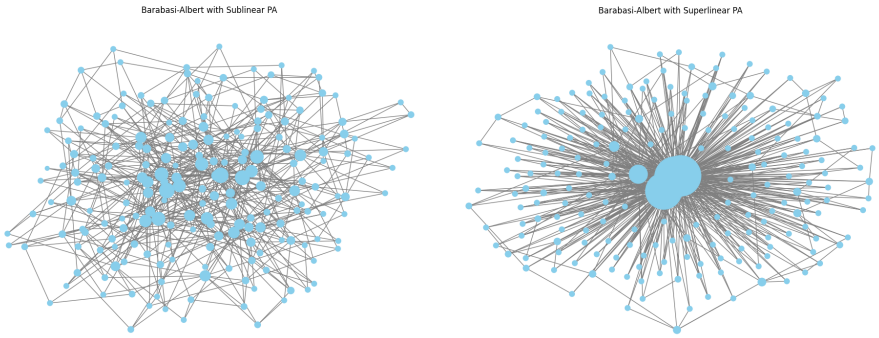

Figure 4: Networks generated using Barabási-Albert model with sublinear and superlinear probability distributions, respectively. The parameters used are N = 200, m0 = 5, m = 3, and `random.seed(5)`.

Using the same sample, we reported the degree distributions in a log-log scale (Figure 5). The sublinear preferential attachment model shows a decreasing trend, which means that hubs (nodes with an higher degree, thus connected to more vertices) are less frequent. Furthermore, the log-log plot doesn't show a straight line, so it produces a more evenly distributed network with no clear scale-free properties.In contrast, in the super-linear model there's a steeper slope, leading to a network that's different from a scale-free one due to the extreme degree of few nodes.

To further prove our point, we fitted our data on RStudio with two types of regression models: a linear one (to simulate a scale-free behaviour) and a polynomial one (with degree 4), and reported the R^2 measure to analyse the quality of the fits. The results in Figure 6 show that in both cases a polynomial model describes our results better (R^2 is closer to 1) than a linear one, that would be

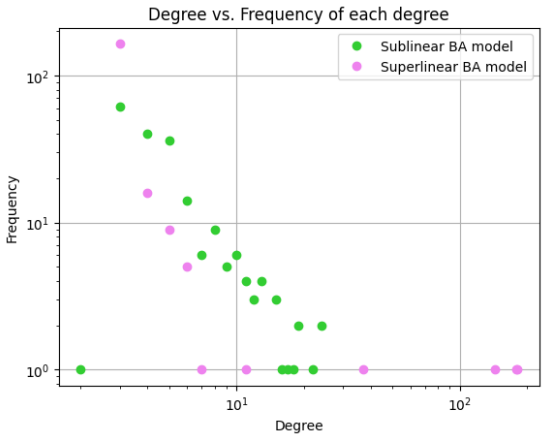

Figure 5: Log-log plot for the degree distribution of the sublinear and superlinear Barabási-Albert model, using the networks generated in Figure 4.

preferred for a scale-free network. We chose to report a polynomial of degree 4 because we found that in both cases $R^2 > 0.90$ and we wanted to avoid over-fitting, but even $degree = 3$ describes the curves with $R^2 > 0.80$, which is an acceptable threshold already. In any case, a linear model is simply not enough, and thus our networks cannot be called scale free.

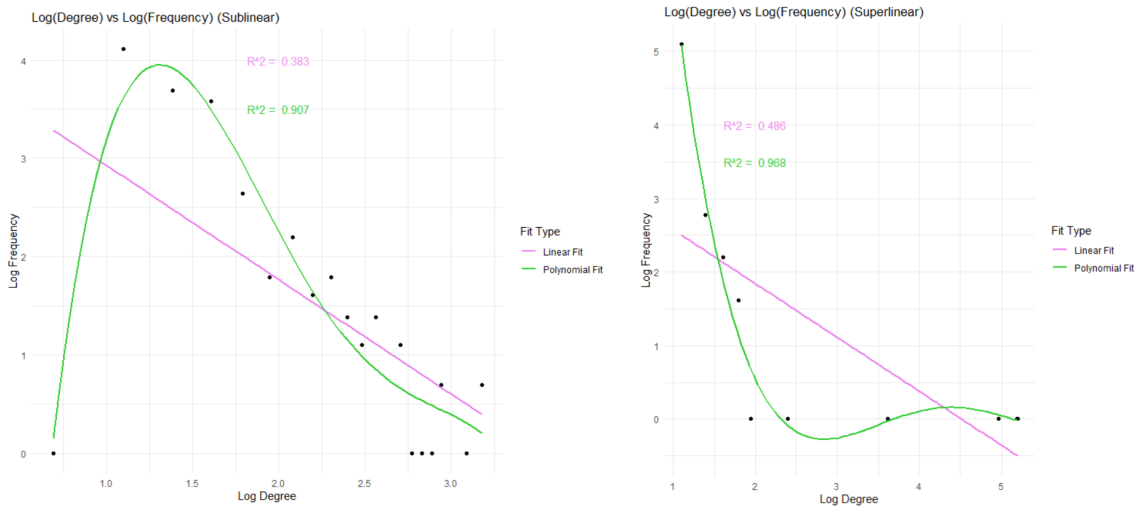

Figure 6: Statistical modeling of the log-log plot results, with R^2 measures for each type of regression.

8 Softwares and libraries

- Python version 3.12.7
- RStudio version 4.3.3 (2024-02-29 ucrt)
- Networkx version 3.3
- Matplotlib version 3.9.0
- Numpy version 1.26.4
- R package ggplot2: 3.5.1