Case Study 2: Power Law Models for Directed Graphs

You will implement two growing network models that create networks with power law degree distributions (when you use the right parameters and the networks that you generate are very large).

- **The Vertex Copying Model.** New vertices create their links via a combination of copying links from existing vertices, and creating links uniformly at random. The ratio of copying versus random determines the power law.
- **The Fitness Model** . Rather than using the degree of a vertex as a measure of attractiveness, the vertices in this model are born with a fitness score. Vertices with high fitness attract future links. The probability distribution of the fitness score determines whether the resulting graph adheres to a power law (or not).

You will implement these network models using Python and NetworkX, run some simulations, and reflect on what you observe.

You will submit two files:

- `abcd_powerlaw.py`: Your python code.
- `abcd_powerlaw_report.pdf`: Your report, including network visualizations and your answers to the four questions found below.

Here, you should replace "abcd" by the first initials of your group members. For example, Ang, Katara, Sokka and Toph would turn in `akst_powerlaw.py`.

I have provided a template python file `template_powerlaw.py`. This has some helper functions that you can use, as well as two functions that you must implement. See Appendix A below for a description of the helper functions.

Vertex Copying Model

In this model, the "success breeds success" dynamic is produced by copying part the bibliography of an existing paper, and then filling out the rest with randomly chosen citations. There are three parameters:

- c = out degree of each vertex
- γ = the probability that we copy a link (instead of adding a random link)
- *numsteps* = the number of times to run the process.

We grow the network as follows.

- Start with a complete directed network on $c+1$ papers, so each initial vertex has a directed link to all of the other initial vertices.
- We add *numsteps* papers sequentially. When paper $n+1$ appears, it makes c citations as follows
  - Pick an existing paper i uniformly at random. (Note that paper i cites c others.)
  - For each paper j cited by i:
- With probability $0 < \gamma < 1$, copy that citation: add an arc from $n+1$ to j
- Otherwise (with probability $1-\gamma$), cite paper chosen uniformly at random.

We proved in class that for large enough k, we have

$$\Pr(k) \sim k^{-\alpha} \qquad \text{where} \qquad \alpha = 1 + 1/\gamma.$$

Log Normal Fitness Attachment (LFNA) Model

What if vertices have an attractiveness, independent of their degree? Certainly there are situations where vertices are popular due to their inherent worth, and not just because other vertices connect to them. This brings us to the idea of the *fitness* of a vertex, which is its propensity to attract links (regardless of the number of current links to the vertex).

The LNFA model has two parameters

- c is the number of links that a new vertex makes to existing vertices
- σ is the standard deviation σ for the fitness distribution

In this model, we assume that the fitness of a vertex is drawn from a **standard log-normal distribution**. You can find a description of the standard log-normal distribution in Appendix B below. Python's `numpy` package gives an easy wayto get a sample from the standard log-normal distribution . The function call is `numpy.random.lognormal(0,sigma)` where `sigma` is your desired standard deviation. For your convenience, the `template_powerlaw.py` file includes a wrapper function `get_fitness(sigma)` for you to call instead.

We grow the network as follows.

- Start with a complete directed network on $c + 1$ papers, so each initial vertex has a directed link to all of the other initial vertices. The fitness of each vertex is drawn from the log-normal distribution.
- We add vertices sequentially. When vertex $n + 1$ appears, it makes c links as follows
  - Assign a random fitness Φ_{n+1} to the new vertex according to the log-normal distribution.
  - Add c independent links from the new vertex to the existing vertices so that the probability of connecting to vertex i is
$$\frac{\Phi_i}{\sum_{j=1}^{n} \Phi_j}.$$

This model produces networks with a wide variety of structure, depending on the standard deviation σ^2 of the vertex fitness values. This model produces a network with a power law degree distribution when the standard deviation is "just right," meaning that it is neither too small or two big.

The Assignment

Implement the models

I have provided a template python file `template_powerlaw.py`. This has some helper functions that you can use, as well as two functions that you must implement.

Rename this file as `abcd_powerlaw.py`, where a,b,c,d are the first initials of your group members. For example, Alice, Bob, Clarice and David would turn in `abcd_powerlaw.py`.

- Implement the `vertex_copy_model(c, gamma, num_steps)` method. This method will return a directed graph with `c + 1 + num_steps` vertices that is generated using the vertex copy model.

- Implement the `lfna_model(c, sigma, num_steps)` method in `template_powerlaw.py`. This method will return a directed graph with `c + 1 + num_steps` vertices that is generated using the LNFA model.

Simulation Report

You will generate 3 directed tree networks using each model, create visualizations using Gephi, and then write a short report on your findings. This report will be named `abcd_powerlaw_report.pdf`.

In all of your simulations, you will set

- `c = 1`,
- `num_steps = 100`,

which means that each of the networks that you generate will be a directed tree on 102 vertices. Here are the values that you should use for the final parameters.

- Vertex Copy Model: create a network for $\gamma = \frac{1}{9}, \frac{1}{2}, \frac{8}{9}$.
- LNFA Model: create a network for $\sigma = \frac{1}{4}, 2, 16$.

For each of these 6 networks, create a visualization using Gephi, using the following specifications.

- Vertices are colored by modularity.
- Vertices are sized by in degree.
- Vertex Labels are sized by in degree.
- Layout: use Force Atlas 2, LinLog mode, Prevent Overlap.

Each of these graphs is a tree, so your layout must not have any crossing edges. You can move hub vertices around using the dragging tool, and then re-run the layout to fix any leaf crossings.

Finally, answer the following questions.

1. Compare the three Vertex Copy Model networks to one another. What happens are you change γ? Which ones look like classic power law networks? Which ones don't? Why not?
2. Compare the three LNFA Model networks to one another. What happens are you change σ? Which ones look like classic power law networks? Which ones don't? Why not?
3. Compare the Vertex Copy Model networks with the LNFA networks. There are some (sometimes subtle) difference between them. Find as many distinguishing features that you can, and try to explain why the models lead to these structural differences. Calculate other centrality measures (PageRank, Eigenvector, Betweenness) and compare them across the models. Their behavior may help you to see some structural differences.

Appendix A: Helpful Functions in the `template_powerlaw.py` file

The template file for this assignment lets you focus on writing the code for the network growth. You must have the following packages installed:

- `networkx`

- `numpy`
- `matplotlib`

Here are the functions in the `template_powerlaw.py` file.

Helper Functions. Your code will call these functions.

- `get_seed_multidigraph(c)`: This method creates the initial graph that serves as the seed for the network growth process. The seed graph is a complete digraph on $c+1$ vertices. There is a directed arc between every pair of vertices. Moreover, this is a multigraph so you can have multiple edges between vertices. (This comes into play when $c > 1$, where the growth process might add multiple links between vertices.)
- `get_fitness(sigma)`: This is a wrapper function for sampling from the log-normal distribution.
- `run_vertex_copy(out_dir_name)`: This function calls `vertex_copy_model` four times (for the desired γ values, and exports the networks to the given directory.
- `run_lnfa(out_dir_name)`: This function calls `lnfa_model` four times (for the desired σ values, and exports the networks to the given directory.

Testing Functions. These functions might be helpful while you are writing your code.

- `get_alpha(G, k_min)`: Finds the best fit power law exponent for network G. Note that the networks we are generating are too small for the theoretical limiting values to take hold. So your power law exponents won't match the predicted values. However, α will change when you vary the parameters of the models.
- `get_alpha_from_data(data, k_min)`: Finds the best fit power law exponent for the array `data`. This function is called by `get_alpha(G, k_min)`. You don't need to call it directly.
- `plot_degrees(graph)`: Creates a log-log plot of the degree distribution of the graph.

Appendix B: The Standard Log-Normal Distribution

A standard log-normal distribution has mean value zero (that makes it "standard") and the logarithm of the values is normally distributed (that makes it "log normal"). The latter happens when your function is the **product of lots of independent random variables**. In other words, the LNFA supposes that factors $\phi_1, \phi_2, \ldots, \phi_L$ contribute to a vertex's attractiveness. Furthermore, these factors have a multiplicative effect, so that the fitness Φ_i of vertex i is

$$\Phi_i = \prod_{\ell=1}^{L} \phi_\ell.$$

Digging into the formulas, the PDF of a standard log-normal distribution is

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma x} \exp\left(-\frac{(\ln x)^2}{2\sigma^2}\right).$$

Different values of σ lead to very different PDF functions, as shown to the right. This means that the LNFA model creates a wide range of network topologies, depending on the standard deviation σ. Some of these standard deviations lead to a power law in the LNFA network, and others don't.