Structural Differences in Random and Small World, Scale-Free Graphs

Thomas Draycott

08/02/2023

Contents

1 Introduction

This project seeks to investigate the structural differences between random graphs and small-world, scale-free graphs and Albert-Barabási graphs. Then, these properties will be tested by how they affect an SIRD model ran on both types of graph. The motivation for this project was from a deep interest in graph theory and with the recent COVID-19 pandemic how the ways human relations may affect the spread of infectious disease.

2 Literature Review

2.1 Graph Theory

Graph Theory is the study of networks where vertices are connected by edges, see APPENDIX for a further explanation.

2.2 Random Graphs

TO DO

2.3 Small World Graphs

In May 1967 Professor of Psychology at the Graduate School and University Center of the City University of New York, Stanley Milgram ran an experiment to see if a person living in Omaha, Nebraska could get a parcel to a stockbroker in Boston, Massachusetts (Milgram 1967). In his experiment he found the average path length to reach the stockbroker was 5.5, which created the term six degrees of separation (however Milgram's experiment had flaws which puts the exact number into doubt). This idea of having such a small average path length for such numerous nodes is a hallmark of a small world graph.
A Small world graph is formally defined by the following property: $L \propto \log N$ where L is the average shortest path length of the network and N is the total number of nodes (Watts and Strogatz 1998). Several models exist to generate small world graphs such as the Watts-Strogatz Model.

2.4 Scale Free Graphs

In networks that appear in the real world such as the internet and social groups, there exists nodes known as "hubs" (a node that has a higher degree then the average of the graph). This is an important property encapsulated in Scale-Free Graphs.
Scale-free Graphs are formally defined by the following power law: $P(k) \sim k^{-\gamma}$, k is the degree of a vertex, $P(k)$ is the probability of a node having degree k and γ is a parameter determined by the graph typically $2 < \gamma < 3$ (Onnela et al. 2007).

2.5 Barabasi-Albert Model

In 1999, Albert-László Barabási and Réka Albert developed the Albert-Barabási Model which generates small world, scale free graphs by a process of preferential attachment (Barabási and Albert 1999). The model works as such: define two parameters n (The number nodes the graph at the end of the process will have) and e (the number of edges added for each new node) take a seed graph, add one new node to the graph, using preferential attachment add e edges from the new node to nodes on the seed graph, continue till there are n nodes on the graph.
Preferential attachment describes a 'rich get richer effect' that is the higher the degree of the node the more likely it will gain a new edge, the following formula describes it $\prod(k_i) = \frac{k_i}{\sum_j k_j}$ where k_i is the degree of node i.

2.6 SIRD Model

SIRD stands for Susceptible, Infected, Recovered and Dead which is the states each individual in the model can take. There are many was of implementing a model like this such as a virus having specific infection 'power' and mortality rates or having the infection be determined entirely by the individual.

3 Mathematical Methods

3.1 Python

This project is written in the Python programming language as it is the language I am most familiar with and has very useful modules for graph analysis. All the code to create these graphs is completely doable without 3rd party libraries however as this project focuses on more complicated topics than just graph creation I will be using a 3rd party library to simplify creating graphs and manipulating them as detailed below.

3.2 Networkx

To begin we should explain the software that this project is based most heavily on. Networkx is a module for the Python programming language that allows for the creation and manipulation of graphs (Hagberg, Schult, and Swart 2008).

3.3 Creating Graphs

First we need to show how to create a graph using the software.

```
1  import networkx
2  import numpy
3  G = networkx.Graph()
4
5  G.add_node('A')
6  G.add_node('B')
7  G.add_node('C')
8  G.add_edge('A', 'B')
9  G.add_edge('C', 'B')
```

The above code does the following: it imports the Networkx package for us to use, creates a 'graph' object called G, creates nodes A, B, C, then adds two edges between A and B and B and C.
However that is too clunky for normal use so the Networkx package provides functions such as:

```
networkx.complete_graph(5)
```

to generate a complete graph with 5 nodes in one line, but we still have the atomic control of the graph as in the first example.

3.4 Creating Random Graphs

To create a random graph on a piece of paper it is quite simple: Select N nodes to add to the graph and p probability, draw N nodes on the paper, then go through every possible pair of nodes in the graph and draw an edge between them if when picking a random real number from [0,1] it is less than p. Networkx implements a function `networkx.gnp_random_graph` to create a random graph using the following code:

```python
import networkx
import itertools
def gnp_random_graph(n, p, seed=None):
    edges = itertools.combinations(range(n), 2)
    G = networkx.Graph()
    G.add_nodes_from(range(n))
    if p <= 0:
        return G
    if p >= 1:
        return networkx.complete_graph(n, create_using=G)

    for e in edges:
        if seed.random() < p:
            G.add_edge(*e)
    return G
```

Figure 1: The Random Graph Function

Let's explain, the function takes n (The total number of nodes) and p (The probability of drawing an edge between a pair of nodes) as parameters. The `itertools.combinations(range(n),2)` creates s list of every combination of 2 numbers up to n i.e. (0,1), (0,2), (1,2) and so on, then to graph G we add n nodes, if $p = 0$ then there are 0 edges in the graph, so it stays empty and if $p = 1$ then every node is attached to every other node, so it becomes a complete graph, if neither of those are true then we loop through all the possible combinations of nodes and choose a random number from $[0, 1]$ and if that is less than p we add an edge between those nodes.

3.5 Creating Barabasi-Albert Graphs

First let us describe how to create Barabasi-Albert Graphs on a piece of paper. To create a Barabasi-Albert Graph we must first start with a "seed" graph (An already drawn graph that we will grow using the Barabasi-Albert Model) then we will define two more parameters n for the total number of nodes we want this new graph to have and m the number of edges we will add each iteration of the model. The algorithm works in the following way: take your "seed" graph and assign each node on the graph a probability p according to its degree k using the formula $p(k_i) = \frac{k_i}{\sum_j k_j}$ we then randomly select m unique nodes according to the probabilities calculated (this can be done by label each node 1 to n then write on a piece of paper that label for each node, then fold the paper a number of times proportional to its probability for example folding 0.2 twice but 0.2 five times then randomly choosing the papers), add 1 new node to the graph and draw an edge from this new node to each of the randomly preexisting nodes, repeat this until we have n nodes on the graph.
As Barabasi-Albert graphs are the main focus of this project we will now show how they are created in networkx.

```
def barabasi_albert_graph(n, m, seed=None, initial_graph=None):

    if m < 1 or m >= n:
        raise nx.NetworkXError(
            f"Barabási-Albert network must have m >= 1 and m < n, m = {m}, n = {n}"
        )

    if initial_graph is None:
        # Default initial graph : star graph on (m + 1) nodes
        G = star_graph(m)
    else:
        if len(initial_graph) < m or len(initial_graph) > n:
            raise nx.NetworkXError(
                f"Barabási-Albert initial graph needs between m={m} and n={n} nodes"
            )
        G = initial_graph.copy()

    # List of existing nodes, with nodes repeated once for each adjacent edge
    repeated_nodes = [n for n, d in G.degree() for _ in range(d)]
    # Start adding the other n - m0 nodes.
    source = len(G)
    while source < n:
        # Now choose m unique nodes from the existing nodes
        # Pick uniformly from repeated_nodes (preferential attachment)
        targets = _random_subset(repeated_nodes, m, seed)
        # Add edges to m nodes from the source.
        G.add_edges_from(zip([source] * m, targets))
        # Add one node to the list for each new edge just created.
        repeated_nodes.extend(targets)
        # And the new node "source" has m edges to add to the list.
        repeated_nodes.extend([source] * m)

        source += 1
    return G
```

Figure 2: The Barabasi-Albert Graph Function

This is the Barabási-Albert function from Networkx. It takes 4 parameters, n,m, `seed`, and `initial_graph`. n and m are simply the same as explained above, `seed` is for the random functions later so we can control the behavior, `initial_graph` is where we input our 'seed graph' for the model to build on (If no graph is provided then it uses a Star graph with $(m+1)$ nodes). The function then creates a list of nodes: `repeated_nodes` where each node is repeated equal to its degree (A node with degree 5 is repeated 5 times) then takes random samples from this list to create the preferential attachment then finally returns the Barabási-Albert graph.

3.6 Gaining Information From Graphs

Methods in networkx allow us to extract key information from the graph objects such as `G.number_of_edges` which returns the number of edges that are contained in graph G or `networkx.all_neighbors(G,n)` which returns all the neighbors of node n in graph G.

4 Analysis

TODO

5 Evaluation

TODO

6 APPENDIX

6.1 Python Code

ADD CODE HERE

References

Barabási, Albert-László and Réka Albert (1999). "Emergence of scaling in random networks". In: *science* 286.5439, pp. 509–512.

Hagberg, Aric A., Daniel A. Schult, and Pieter J. Swart (2008). "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, pp. 11–15.

Milgram, Stanley (1967). "The small world problem". In: *Psychology today* 2.1, pp. 60–67.

Onnela, J-P et al. (2007). "Structure and tie strengths in mobile communication networks". In: *Proceedings of the national academy of sciences* 104.18, pp. 7332–7336.

Watts, Duncan J. and Steven H. Strogatz (June 1998). "Collective dynamics of 'small-world' networks". In: *Nature* 393.6684, pp. 440–442. ISSN: 1476-4687. DOI: 10.1038/30918. URL: https://doi.org/10.1038/30918.