Tackling Scale Free Networks

Sathyasri Sudhakar

July 2024

1 Introduction

This project focuses on understanding why various networks (social, communication, technological, biological) display similar features. The goal is to explore if these features can be explained by basic principles that govern the construction of these networks.
Typically, these networks are built using rules that determine the probability of connections between the vertices – which is based on their location and influence.
Scale Free networks, are characterized by vertex degrees following a power-law distribution is what we will be looking into here. That states that the fraction P(k) of nodes in the network having k connections to other nodes goes for large values of k as

$$P(k) \sim k^{-\gamma}$$

where γ is the scaling parameter. This means that few nodes have many connections whereas many nodes have little to no connections.

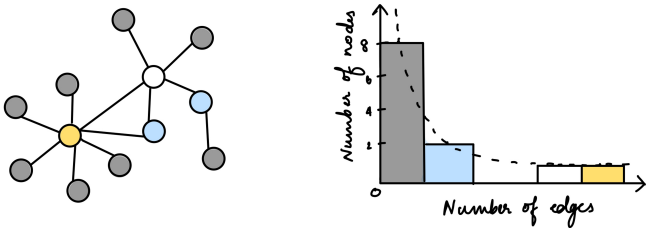

Figure 1: Example of Scale Free Network and Power Law Distribution.

2 Methodology

The main steps in this project are

- Efficiently generate scale free networks.
- Visualising the Network.
- Parameter Estimation.
- Classification of Networks in Categories.
- Applying Path finding algorithms.

Figure 2: Main process Flow

The Generation of Scale Free Networks is further divided into these steps:

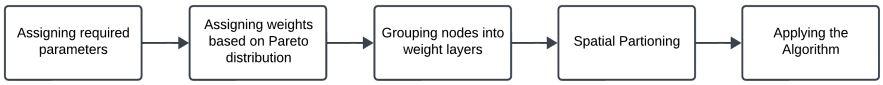

Figure 3: Process flow for generating scale free networks

The flowcharts above provide a visual representation of the sequence of outcomes associated with the project. Understanding this flowchart will help in comprehending the overall workflow and the inter-dependencies of different stages within the process

3 Work Done so far - As of 8th July 2024

3.1 Generating Scale Free Networks

3.1.1 Step 1- Importing the Required Libraries and Constants

Taking V=25, as the first example here to understand the code and algorithm flow and then giving an example for a larger node set.

This algorithm that has been implemented in python is designed for sampling a graph from a Generalized Inhomogeneous Random Graph (GIRG) by Bringmann et al. (2015) model in expected time O(n). GIRGs are a type of random graph model where nodes have positions and weights, and the probability of an

edge between two nodes depends on these attributes.
In the first step imported all the required libraries and assigned values to different nodes in the 2D plane.
All the constants that are assigned are:

- V - set of nodes
- c - constant
- α - constant
- dim - dimension
- Max 1 - the maximum possible position on both x and y axis
- X - V number of positions on the x axis
- Y - V number of positions on the y axis

The code for this is:

```
import random
import matplotlib.pyplot as plt
import math
import numpy as np
import networkx as nx
from itertools import combinations
plt.style.use('ggplot')

V=25 # Set of Nodes

c=5 # constant
alpha=2 # alpha value for type 2 calculation
dim=2 # dimension for type 2 calculation

# Node Positions
Max1=100
x=[random.randint(0,Max1) for i in range (V)]
y=[random.randint(0,Max1) for i in range (V)]
```

3.1.2 Step 2 - Assigning Weights to the nodes

Used Pareto Distribution - here took the alpha value as 3, and generated V number of weights. Post this the overall weight (sum of all the weights) - which is used in further calculations. Further plotting the weights to understand how the distribution is.

```
weights=(np.random.pareto(3, V))
weights=list(weights)
```

```
W=sum(weights)

sorted_weights=sorted(weights)
x_range = range(len(sorted_weights))
plt.figure(figsize=(5,4))
plt.plot(sorted(weights,reverse=True),x_range,marker='*', linestyle='-', col
plt.xlabel('Weight', fontsize=10)
plt.ylabel('Range of the Nodes', fontsize=10)
plt.title("Weight Distribution", fontsize=10)
plt.grid(True)
```

The weight distribution plot looks like this -

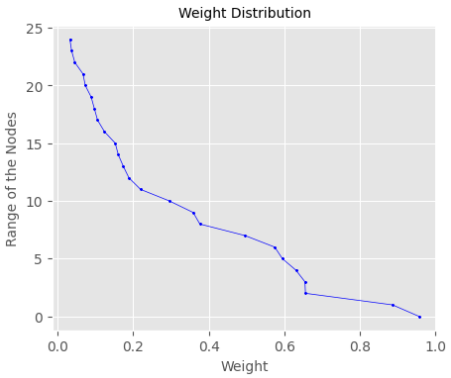

Figure 4: Weight Distribution based on Pareto Distribution

3.1.3 Step 3 - Dividing the weights into different weight layers

Making different layer ranges and how based on the weights they fall into each layer. The steps followed into dividing them into different weight layers are:

- Assigned N as the total number of weight layers that are possible.
- Calculated the quantiles for weights from the previous step.
- Based on the quantiles values, placing the different weights into different weight layers.

Why the Quantiles are taken ?
In order to divide the area under the Weight distribution curve, the quantiles are taken into consideration.

The code to achieve the above steps is as follows

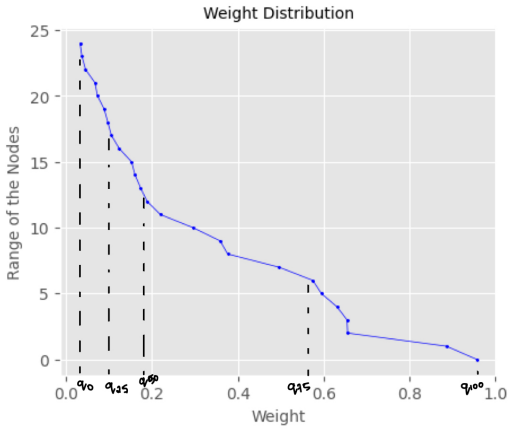

Figure 5: Weight Layers Classification based on Quantiles

```
N=int(math.log2(V))
quantiles = np.linspace(0, 1, N + 1)
quantile_values = np.quantile(sorted_weights, quantiles)

weight_layers = [[] for _ in range(N)]
for k, weight in enumerate(weights):
    for i in range(N):
        if i < N - 1:
            if quantile_values[i] <= weight < quantile_values[i + 1]:
                weight_layers[i].append(k)
                break
        else:
            if quantile_values[i] <= weight:
                weight_layers[i].append(k)
                break
```

3.1.4 Step 4 - Spatial Partitioning

Creation of Spatial boxes - helps make different combinations of A, B boxes which is nothing but different sized squares across the plane that's used to determine whether the points fall into Type 1 category or Type 2 category. Example of the working for 5 nodes in a (100x100) plane.

A few functions that help make the visaulization easier and plus help determine the spatial boxes are as follows:

```
def find_connections(i, j, x_values, y_values):
```

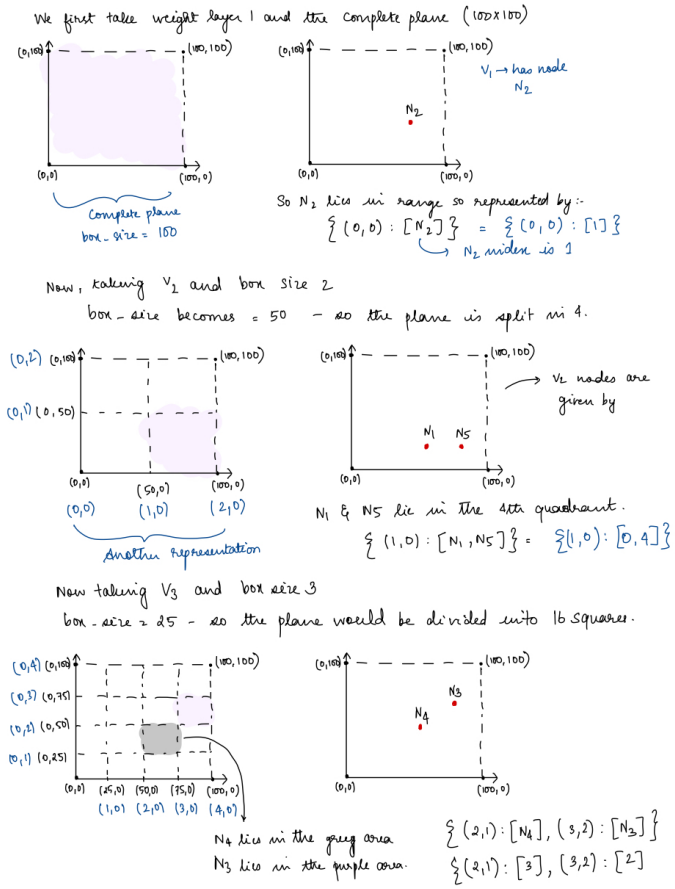

Figure 6: Example of Spatial Partitioning for 5 nodes

```
    return [[x_values[i],y_values[i]],[x_values[j], y_values[j]]]
def get_box_index(x, y, box_size):
    return (x//box_size, y//box_size)
def create_spatial_boxes(nodes, box_size):
    boxes = {}
    for i in nodes:Crea
        box_index = get_box_index(x[i], y[i], box_size)
        if box_index not in boxes:
            boxes[box_index] = []
        boxes[box_index].append(i)
    return boxes
box_sizes = [Max1 // (2 ** i) for i in range(N)]
boxes=[create_spatial_boxes(weight_layers[i],box_sizes[i])for i in range(N)]
```

3.1.5 Step 5 - Applying the Algorithm

Part 1: Classification

We consider different combinations of these boxes, classifying them into two categories: Type I or Type II.
A pair is classified as Type I if the two boxes have a distance of zero or pass a specific check. If either condition is met, the pair is Type I. Otherwise, the pair is classified as Type II, indicating that the boxes are farther apart.
Process Flow to understand the check for Type I:

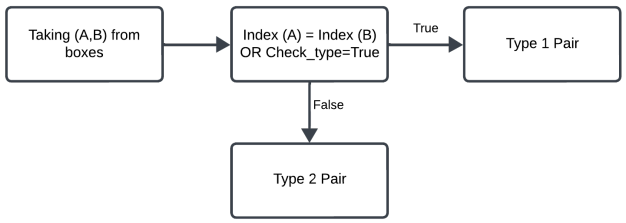

Figure 7: How the checks for Type 1 and Type 2 takes place

Further understanding the check for type 1:

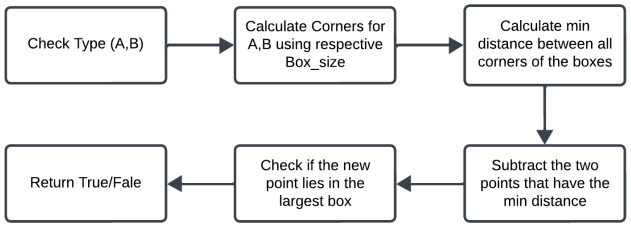

Figure 8: How the checks for Type 1 takes place

The functions used to achieve this is as follows:

```
def distance_between_2(x1, y1, x2, y2):
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
def geom_dist(p):
    return (math.log(random.random())/math.log(1-p))
def is_point_in_box(x, y, box):
```

```
    x_min = min(p[0] for p in box)
    x_max = max(p[0] for p in box)
    y_min = min(p[1] for p in box)
    y_max = max(p[1] for p in box)
    return x_min <= x <= x_max and y_min <= y <= y_max
def small_d(box_A_index,box_A_size,box_B_index,box_B_size):
    box_A_index=list(box_A_index)
    box_B_index=list(box_B_index)
    # All the edge points of box A are , if size is 100 - (0,0),(0,100),(100,100
    box_A_corners = [
        (box_A_index[0], box_A_index[1]),
        (box_A_index[0], box_A_index[1] + box_A_size),
        (box_A_index[0] + box_A_size, box_A_index[1] + box_A_size),
        (box_A_index[0] + box_A_size, box_A_index[1])
    ]
    print("Box A Corners :", box_A_corners)
    # All the edge points of box b are, if the size is 50 - (0,50),(0,100),(50,1
    box_B_corners = [
        (box_B_index[0], box_B_index[1]),
        (box_B_index[0], box_B_index[1] + box_B_size),
        (box_B_index[0] + box_B_size, box_B_index[1] + box_B_size),
        (box_B_index[0] + box_B_size, box_B_index[1])
    ]
    print("Box B Corners :", box_B_corners)
    distances = []
    for a in box_A_corners:
        for b in box_B_corners:
            distance = ((a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2) ** 0.5
            distances.append((distance, a, b))
    min_distance = min(distances, key=lambda x: x[0])
    print("Minimum distance is between the points:", min_distance[1], "and", mi
    return min_distance
def check_type1(box_A_index,box_A_size,box_B_index,box_B_size):
    box_A_index=list(box_A_index)
    box_A_corners = [
        (box_A_index[0], box_A_index[1]),
        (box_A_index[0], box_A_index[1] + box_A_size),
        (box_A_index[0] + box_A_size, box_A_index[1] + box_A_size),
        (box_A_index[0] + box_A_size, box_A_index[1])
    ]
    box_B_corners = [
        (box_B_index[0], box_B_index[1]),
        (box_B_index[0], box_B_index[1] + box_B_size),
        (box_B_index[0] + box_B_size, box_B_index[1] + box_B_size),
        (box_B_index[0] + box_B_size, box_B_index[1])
    ]
```

```
d=small_d(box_A_index, box_A_size, box_B_index, box_B_size)
a=d[1]
b=d[2]
new_point = [a_i - b_i for a_i, b_i in zip(a, b)]
#print("New point for comparison :", new_point)
if box_A_size>=box_B_size:
    result = is_point_in_box(new_point[0], new_point[1], box_A_corners)
else:
    result = is_point_in_box(new_point[0], new_point[1], box_B_corners)
print(f"The point ({new_point[0]}, {new_point[1]}) lies within the box: {res
return result
```

Part 2: Implementation of the algorithm

Here we check the pairs present in Type 1 and Type 2 - to see if edges are possible between them.
How the flow of algorithm goes for checking for edges between Type 1 and Type 2 are as follows:
Type 1 based Taking each node in box A and making combination with every node in box B which are of Type 1 we check the puv which is given as the weight of node in a * weight of node in b divided by the total weight of all the nodes overall. If this puv is greater than the sampled value then there's an Edge present between the nodes taken in consideration above.

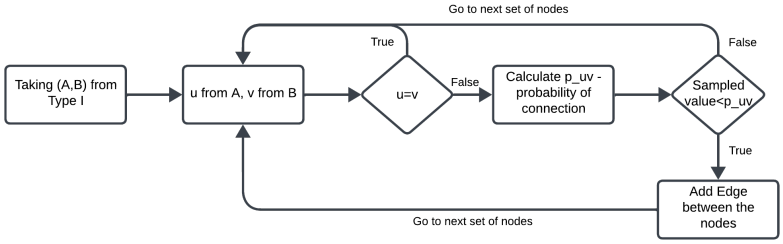

Figure 9: Edges check for Type I pairs

Type 2 based Taking each node in box A and making combination with every node in box B which are of Type 2 we first check the distance between the two nodes, then calculate the pbar value and then calculate the r (geom dist value for pbar). If this r is less than the overall length(A)*length(B) then we do another check where the sampled random is smaller than the weight of node in A * weight of node in B divided by the total weight W which is whole divided by pbar. If this condition is true then we add an edge between the two nodes.

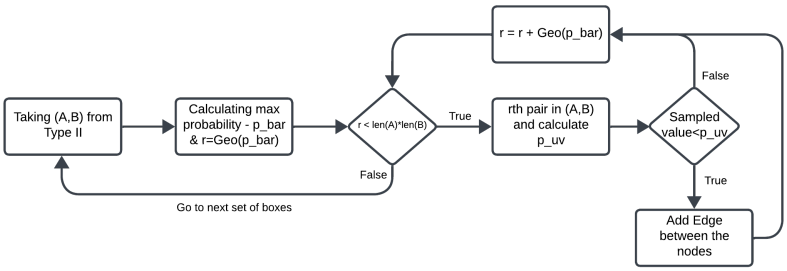

Figure 10: Edges check for Type II pairs

Code used to implementation of the main algorithm

```
E1 = set()
E2 = set()
for i in range(N):
    for j in range(i, N):
        pairs_type_I = []
        pairs_type_II = []
        for box_A_index, box_A_data in boxes[i].items():
            for box_B_index, box_B_data in boxes[j].items():
                box_A_nodes = box_A_data['nodes']
                box_A_size = box_A_data['box_size']
                box_B_nodes = box_B_data['nodes']
                box_B_size = box_B_data['box_size']
                print("Box-A-index",box_A_index)
                print("Box-B-index", box_B_index)
                print("Nodes-of-A-are", box_A_nodes)
                print("Nodes-of-B-are",box_B_nodes)
                print("Box-A-size", box_A_size)
                print("Box-B-size", box_B_size)
                if box_A_index == box_B_index or check_type1(box_A_index,box_A_s
                    pairs_type_I.append((box_A_nodes, box_B_nodes))
                    print("Updated-Pairs-type-1-:",pairs_type_I)
                    print("\n")
                else:
                    pairs_type_II.append((box_A_nodes, box_B_nodes))
                    print("Updated-Pairs-type-2-:",pairs_type_II)
                    print("\n")

                for (A, B) in pairs_type_I:
                    for u in A:
                        for v in B:
                            print("u-:",u)
```

```
                print("v :",v)
                if u!=v:
                    p_uv=(weights[u]* weights[v])/W
                    sample_random=random.random()
                    print(" p_uv ::",p_uv)
                    print( "sample_random :",sample_random)
                    print("\n")
                    if sample_random<p_uv:
                        E1.add((u,v))
                        print("Edge 1 has been added for this")
print("\n")
for (A, B) in pairs_type_II:
    #determing the smallest distance
    d = small_d(box_A_index, box_A_size, box_B_index, box_B_size
    print("The min distance between the two boxes is :",d[0])
    #Determining the largest weights in both the boxes
    w1 = max(weights[u] for u in A)
    w2 = max(weights[v] for v in B)
    print("Max weigh in A :",w1)
    print("Max weight in B :",w2)
    #Calculating the maximum proabability
    if d[0]==0:
        p_bar=0.01# if 0 then log(1-p)-> 0
    else:
        p_bar = min(c * (1/d[0])**(alpha*dim) * (w1*w2/W)**alpha
    print("The proability of there being a connection is :",p_ba

    r = geom_dist(p_bar)

    print("The geom_dist is given by :",r)
    #While loop
    while r < len(A) * len(B):
        # Get the r-th pair of vertices
        u = A[int((r-1) % len(A))]
        v = B[int((r-1) // len(A))]
        print("rth pair u is:",u)
        print("rth pair v is:",v)
        dis = distance_between_2(x[u], y[u], x[v], y[v])
        p_uv = min(c * (1/dis)**(alpha*dim) * (weights[u]*weight
        print("The distance between the 2 :",dis)

        print("the true probaility :",p_uv)
        if random.random() < p_uv / p_bar:
            E2.add((u, v))
            print("Edge 2 has been added for this")
        r=r+geom_dist(p_bar)
```

3.2 Visualising the Network

3.2.1 Step 1 - Basic Plot

Plotting all possible connections between the given nodes, just to have an idea on how it would look if all the nodes were able to make connections with every other node.
The code is as follows -

```
plt.figure(figsize=(5,4))
plt.scatter(x, y, color='black')
for (x1, y1), (x2, y2) in combinations(zip(x, y), 2):
    plt.plot([x1, x2], [y1, y2], 'b—', linewidth='0.2')
plt.xlabel('X-Values', fontsize=10)
plt.ylabel('Y-Values', fontsize=10)
plt.title("Possible-Connections-between-nodes", fontsize=10)
plt.grid(True)
plt.tight_layout()
plt.show()
```

The plot would look like this -

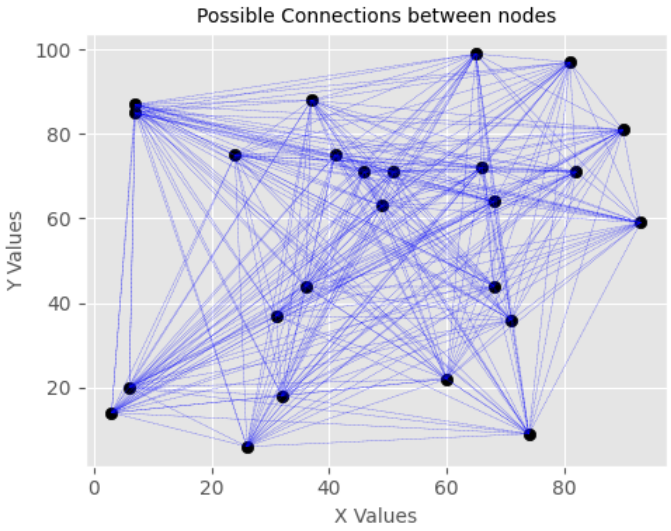

Figure 11: 2D Plane with nodes showing all the possible connections between them and the other nodes.

3.2.2 Step 2 - Scale Free Network post the implementation of the code

After applying the algorithm to the network that we took, this is what a scale free network looks like.

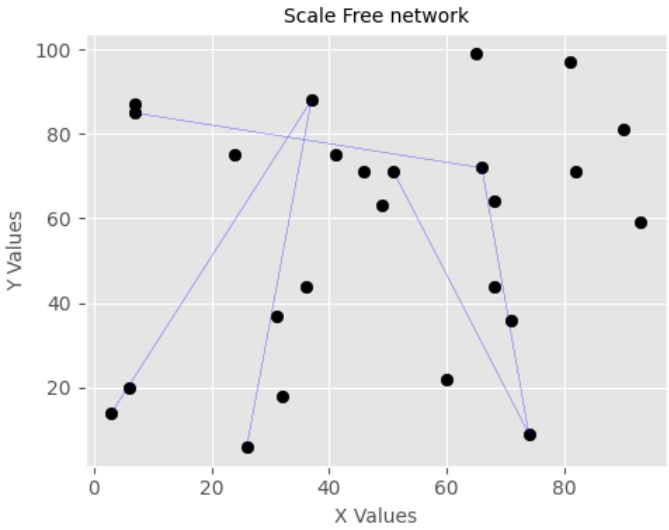

Figure 12: Scale Free Network got from applying the algorithm

3.2.3 Step 3 - Understanding the relation between the number of nodes and the number of edges

In order to understand the whether the network got, is a scale free network or not it must follow power law this can be done by checking the degree of number of nodes and the number of links each have.
The code to do this is as follows:

```
connections = E1.union(E2)
G = nx.Graph(connections)
node_degrees = G.degree()

degree_counts = {}
for node, degree in node_degrees:
    if degree not in degree_counts:
        degree_counts[degree] = 1
    else:
        degree_counts[degree] += 1
sorted_degree_counts = dict(sorted(degree_counts.items()))
```

```
V = max(G.nodes)
sum1 = sum(sorted_degree_counts.values())

if sum1 != V:
    zero_degree_count = V - sum1
    sorted_degree_counts[0] = zero_degree_count

for degree, count in sorted_degree_counts.items():
    print(f"{count} nodes have {degree} edges each.")

degrees = list(sorted(sorted_degree_counts.keys()))  # Ensure sorting starts
counts = [sorted_degree_counts[degree] for degree in degrees]
plt.figure(figsize=(5, 4))
plt.plot(degrees, counts, marker='o', linestyle='-', color='b')
plt.xlabel('Number of Edges/Links', fontsize=10)
plt.ylabel('Number of Nodes', fontsize=10)
plt.title("Node Degree Distribution", fontsize=10)
plt.grid(True)
plt.show()
```

The plot for the above code looks something like this:

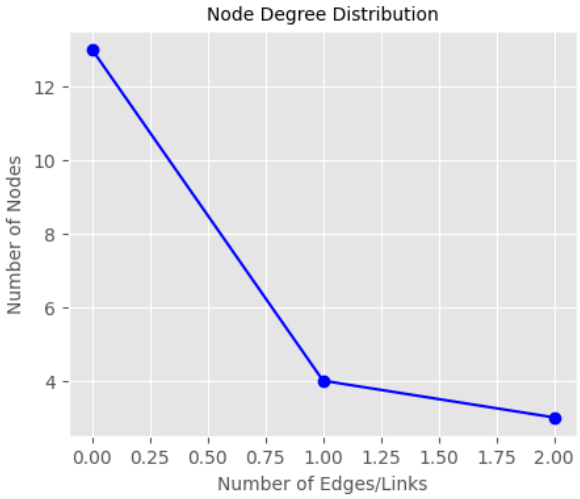

Figure 13: Power law distribution for the scale free network got above

References

Bringmann, K., Keusch, R. & Lengler, J. (2015), 'Sampling geometric inhomogeneous random graphs in linear time', *arXiv preprint arXiv:1511.00576* .