MH 2401 - Algorithms & Computing III

Billy Kurniawan, Charvin Fernanlim, Cindy Laurentia
Gilbert Khonstantine, Kevin Jonathan

20 November 2018

Contents

1 Introduction

In this report, we will review about properties of random network models and compare them with Facebook model (Stanford Large Network Dataset Collection) to find the best approximation for Facebook. Additionally, we will discuss on the model of information spread over a network. All codes in the report will be using Python programming language.

2 Overview of Random Networks

There are 7 models to compare with Facebook model:

1. Erdős–Rényi
2. Watts-Strogatz
3. Barabási–Albert
4. Mediation-Driven Attachment
5. Bianconi–Barabási
6. Power Law Clustering
7. Custom

The properties of models to focus on are clustering coefficient, average shortest path lengths and degree distributions which will be plotted in a histogram.

2.1 Erdős–Rényi Model

Erdős–Rényi is the simplest model among all of the models. It is an undirected network with 2 parameters: n and p. n indicates the number of vertices in the graph, while p is the probability of a pair of randomly edges are connected. Since clustering coefficient of a node is the probability of two randomly selected friends are friend themselves, henceforth, the value of p (the probability of an edge presents between nodes) is expected to be equal to the clustering coefficient. Nonetheless, the average clustering coefficient in Erdős–Rényi network is small, thus, it may not be suitable to represent real world network which will have a lot of clusters.

Erdős–Rényi's degree distribution will follow Binomial distribution. However, the fact that n is large and p is near to zero, the degree distribution will follow Poisson distribution [1].

2.2 Watts-Strogatz Model

With its high clustering coefficient and low average of shortest path, Watts-Strogatz is known to have a small-world property. This random graph has 3 parameters: n, which is the number of vertices; k, which is an even number indicating the mean degree of a vertex; and p, which is the probability of rewiring.

To implement WS Model, firstly, a graph with N nodes is created. Then pick a node n_0 and connects it to k neighbours, $\frac{k}{2}$ on the left and $\frac{k}{2}$ on the right. Do the same for $n_1, \ldots, n_M$. Then, for each edge (n_i, n_j) where $i < j$, it is rewired with probability p by replacing (n_i, n_j) with (n_i, n_m), with m is uniformly chosen without self-loops (i.e. $i = m$) and vertex duplication (i.e. $k = j$).

The clustering coefficient of this graph when p = 0 is expected to be $\frac{3(k-2)}{4(k-1)}$. When $p > 0$, the clustering coefficient is $\frac{3(k-2)}{4(k-1)}(1-p)^3$ [1].

2.3 Barabási–Albert Model

Compared to the previous two models, Barabási–Albert is a growing graph model with a power law degree distributions (scale free network). Besides growth, this model also incorporates preferential attachment mechanism, which means a node has the tendency to connect with nodes with more links (higher degree). Barabási–Albert network has 2 parameters, namely n (number of vertices) and k (degrees of each node). The degree distribution of Barabási–Albert follows the power law function $P(k) \sim k^{-3}$ [2].

The algorithm starts with a connected network with n_0 nodes. Then, the new node is added one at a time by connecting it to maximum n_0 nodes, with probability $p_i = \frac{k_i}{\sum_j k_j}$, where k_i indicates the degree of vertex i and k_j's are the sum of the degree of pre-existing nodes j [2]. Thus, from this probability formula, it can be seen that the larger the value of k_i, the larger the probability of a node connects to it.

2.4 Mediation-Driven Attachment Model

Like Barabási–Albert, MDA model also has the preferential attachment property. However, MDA starts with a new node choosing a random node as its mediator. Then, this new node will connect to m of the mediator's neighbour at random. The probability of this new node i connects with these new nodes is $\frac{1}{N}\left(\frac{1}{k_1} + \frac{1}{k_2} + \ldots + \frac{1}{k_{k_i}}\right) = \frac{\sum_{j=1}^{k_i} \frac{1}{k_j}}{N}$ which can be written as $\frac{k_i}{N} \cdot \frac{\sum_{j=1}^{k_i} \frac{1}{k_j}}{k_i}$ [3]. Unlike Barabási–Albert, the degree distribution of MDA follows the function $P(k) \sim k^{-\partial(m)}$ which is dependent to the factor of m(number of edges of the new nodes) [3]. For small m, it is found that MDA model has winner takes all effect but for larger value of m, this effect is replaced with *winner take some* effect with degree distribution similar to Barabási–Albert [3].

2.5 Bianconi–Barabási Model

This model comes from the development of Barabási–Albert Model, but there can be a scenario where the late nodes can never turn into the largest hubs. In reality, a node's growth does not depend on the node's age only. In Bianconi–Barabási Model, there are 3 major concepts. The first two concepts are inherited from Barabási–Albert Model (Growth and PA). Beside these two concepts, Bianconi–Barabási Model uses another new

concept called the fitness. Fitness can be defined as the ability of each node to attract new edges, where nodes with bigger fitness will connect to new nodes at a higher rate than nodes with less fitness.

Algorithm:

1. Begin with a graph consisting of a fixed number of interconnected nodes. Where each node has different fitness, symbolize by η_i (i is the node).
2. Add a new node j with m possible edges and a fitness η_j at every iteration where the probability that this node connect to an existing node p is defined as $\Pi_p = \frac{\eta_p k_p}{\sum_q \eta_q k_q}$.
3. Repeat procedure number 2 until the number of nodes in the graph match with the desired total nodes.

The fitness distribution of this random model affects the degree distribution. The first case is when the fitness distribution has a finite domain, here the degree distribution will follow Power Law. In the next case where the fitness distribution has an infinite domain, nodes with relatively high fitness will attract a large number of nodes which will bring "the rich get richer" phenomenon.

2.6 Power Law Clustering Model

In the real life network, it is believed that the degree distribution will be heavy tailed and mimic the power-law graph. This is mainly because a real life network will always grow and not be static by time, hence it will continuously gain additional nodes. Moreover, its nodes are linked based preferentially and not randomly, which deducts another reason why Erdős–Rényi and Watts-Strogatz will not be suitable to model the real network.

This power law clustering graph complies Holme-Kim growing network [4]. Start with an existing graph, assuming there exists nodes i with degree k_i. Firstly, a new node will choose the vertices inside the existing graph based on preferential attachment with probability: $\Pi_i = \frac{k_i}{\sum_j k_j}$ [4].

After node i is connected to node m (mediator node), the other edges of i will be connected to m's neighbours with probability $p.p$ is the probability of i will form new triangles which consequently increase the clustering coefficient. This addresses the reason why in this Facebook data modelling, a value of p equals to 1 is set. Moreover, this also indicates that when p =0, the algorithm will follow Barabási–Albert model.

2.7 Custom Model

The custom model is a scale free network model which tackles the issue of having low clustering coefficient in other models such as Barabási–Albert, Bianconi–Barabási and

MDA models. The custom model follows the algorithm of 'A model for social networks' by Riitta Toivonen [5].

The algorithm starts with an existing model with n_0 vertices and connect a new node with m edges. Firstly, pick $m_i \geqslant 1$ random nodes as initial nodes. Then select $m_s \geqslant 0$ $(m - m_i)$ neighbours of each initial nodes as secondary nodes. Connect the new node to all the selected initial and secondary nodes.

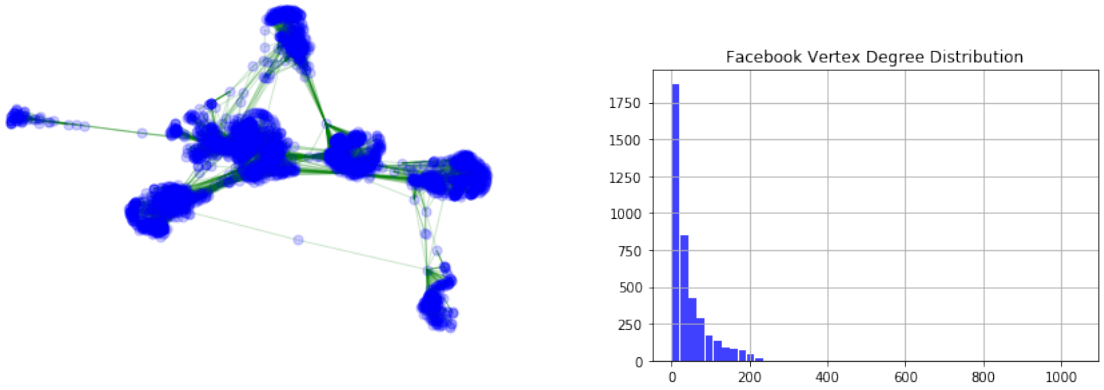
m_i nodes are chosen at random. However, m_s secondary nodes are chosen by preferential attachment (compare degrees among neighbours), neighbours with higher degrees have higher probability to form connections with the new node. The second process is the main reason high clustering coefficient can be achieved.

In addition, larger m_i results in lower clustering coefficient because there will be less triangle being formed. Since $m_s = m - m_i$, large m_i results in less edges being connected to secondary nodes.

3 Modelling Facebook

To determine which network models is the most similar to the sample of Facebook model given in the handout, we will go through these chronological steps:

1. Determine the value of the input parameters that suits the number of graphs and edges of the Facebook model the most
2. Compare the output parameters
3. Conclude the most suitable model



(a) Facebook Graph

(b) Facebook Degree Distribution

Figure 3: Facebook Model

3.1 Determine input parameters

Dataset statistics	
Nodes	4039
Edges	88234
Nodes in largest WCC	4039 (1.000)
Edges in largest WCC	88234 (1.000)
Nodes in largest SCC	4039 (1.000)
Edges in largest SCC	88234 (1.000)
Average clustering coefficient	0.6055
Number of triangles	1612010
Fraction of closed triangles	0.2647
Diameter (longest shortest path)	8
90-percentile effective diameter	4.7

Facebook Data from `https://snap.stanford.edu/data/ego-Facebook.html`

The sample given has 4039 vertices, 88234 edges, and average clustering coefficient (CC) of 0.6055. Our first objective is to create each of the test models so that it has 4039 vertices and approximately 88234 edges.

3.1.1 Erdős–Rényi Model

In the Erdős–Rényi random graph G(n, p), n = number of vertices = 4039 and each pair of edges is connected with probability p. That means expected number of edges is $\binom{4039}{2}p = 88234$. Then $p = \frac{88234}{\binom{4039}{2}} \approx 0.01082$

3.1.2 Watts-Strogatz Model

In the Watts-Strogatz random graph $G(n, k, p)$:
$n = 4039$
$k = \frac{2E}{n} \approx 44$ (rounded)
$p = 1 - \sqrt[3]{0.6055\frac{4(k-1)}{3(k-2)}} \approx 0.06147$

3.1.3 BA, MDA, BB, Custom Model

In the Barabási-Albert/MDA/Bianconi-Barabási/Custom random graph $G(n, k)$: n is the number of vertices, and k is close to the average degree of a vertex divided by 2.

3.1.4 Power Law Clustering Model

Similar to BA, MDA, BB and Custom model parameters for . However, addition of p is required which is the probability of adding a triangle after adding a random edge. From our tests and trials, we decided to set $p = 1$ as the best representation (it makes sense, as p goes higher, the number of triangles goes higher hence higher CC).

3.2 Comparisons with Facebook

To determine the most suitable, we will consider these factors:

- Average CC of Facebook (0.6055)
- Average shortest path of Facebook (3.6925)
- Facebook degree distribution (Long Tail)

Table of Comparison			
Model	Average CC	Average Shortest Path	Average Degree
Facebook	0.6055	3.69	43.69
Erdős–Rényi	0.0107	2.60	43.77
Watts-Strogatz	0.6058	3.13	44
Barabási-Albert	0.0373	2.51	43.76
MDA	0.0444	2.52	43.87
Custom	0.6923	1.99	43.87
Bianconi-Barabási	0.0994	2.26	43.87
Power Law Clustering	0.2597	2.70	43.75

Erdős–Rényi:

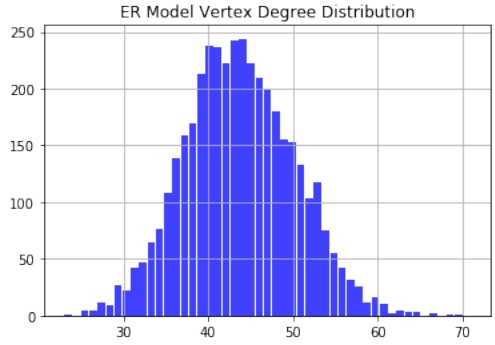


Figure 3.1: ER Degree Distribution

- Follows binomial distribution in its degree distribution
- Has far smaller average CC compared to Facebook
- Slightly smaller in average shortest path length compared to Facebook
- Does not follow scale-free degree distribution

Watts-Strogatz:

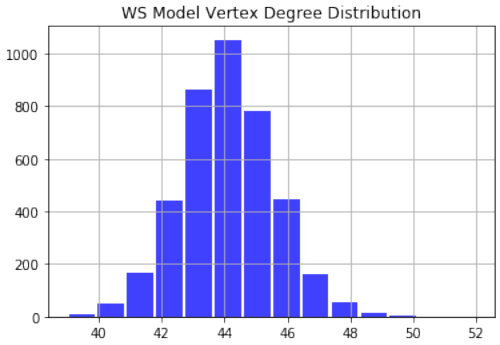


Figure 3.2: WS Degree Distribution

- Does not follow scale-free degree distribution
- Has the most similar average CC and average shortest path length among other models

Barabási-Albert:

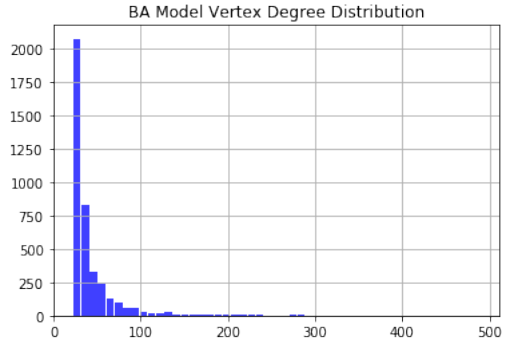


Figure 3.3: BA Degree Distribution

- Follows scale-free degree distribution
- Has far smaller average CC compared to Facebook
- Slightly smaller in average shortest path length compared to Facebook

Mediation-driven Attachment:

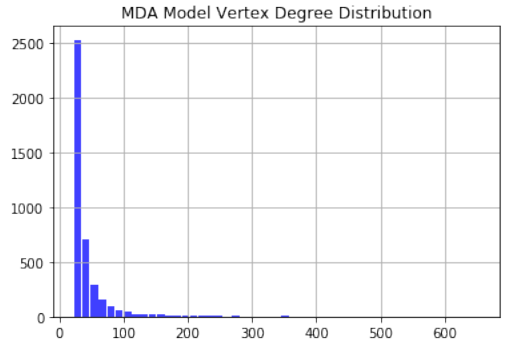


Figure 3.4: MDA Degree Distribution

- Follows scale-free degree distribution
- Has improved average CC compared to Barabási-Albert, but very small compared to Facebook
- Slightly smaller in average shortest path length compared to Facebook

Bianconi-Barabási:

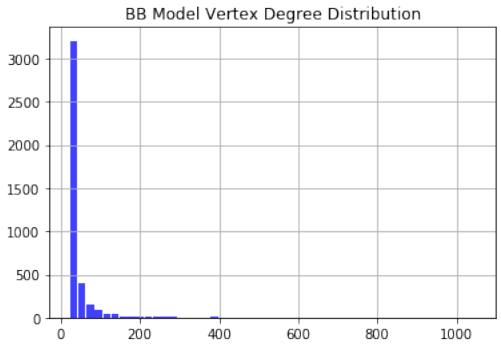


Figure 3.5: BB Degree Distribution

- Follows scale-free degree distribution
- Has improved average CC compared to MDA, but still very small compared to Facebook
- Has decreased in average shortest path length compared to MDA

Custom:

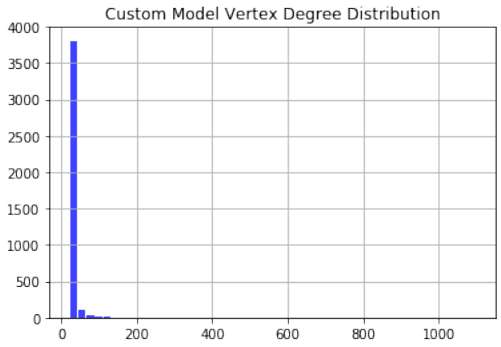
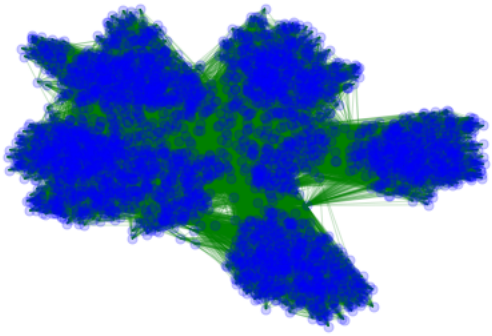


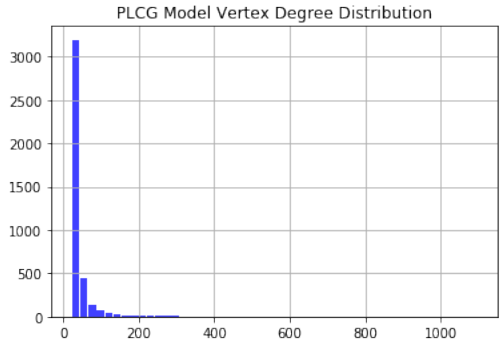
Figure 3.6: Custom Model Degree Distribution

- Follows scale-free degree distribution
- Has similar average CC to Facebook model (Major improvement)
- Has smallest average shortest path among all the models
- Has very small variance, does not resemble the histogram of Facebook model.

Power Law Cluster:



(a) PLCG Graph



(b) PLCG Degree Distribution

Figure 3.7: PLC Model

- Follows scale-free degree distribution
- Has smaller average CC compared to Facebook model (larger than most of the models nevertheless)
- Improvement in terms of average shortest path
- Has histogram visibly resembles Facebook even closer
- Shows existence of clusters and cliques, visibly seen in the graph (major improvement)

3.3 Conclusion

To sum up, it is decided that Power-Law Cluster Graph is the best representation of these models, due to the reasons below:

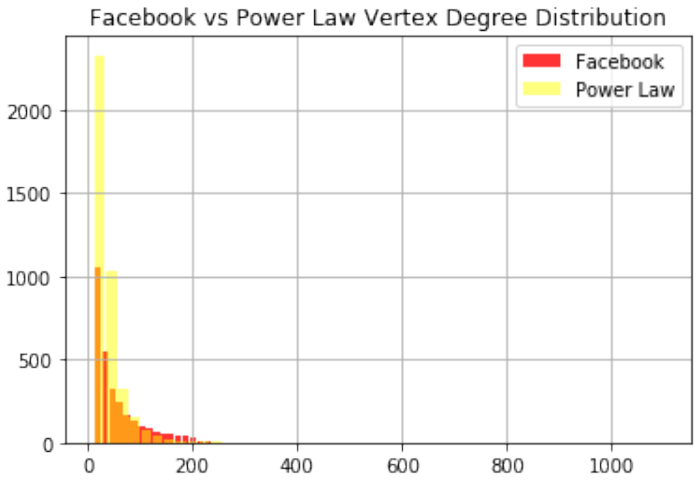


Figure 3.8: Facebook vs Power Law Cluster Degree Distribution

1. It follows scale-free degree distribution,
2. It has a considerably high average CC
3. It has a considerably similar average shortest path
4. The graph represents "communities" arguably similar to Facebook

4 Spread of Information

4.1 SIR Model

SIR Model is one of many models that explains the spread of disease over a network. It was usually used in medical fields to depict the spread of disease over time. However, this can also be depicted as spread of information over time in a social network, where disease is analogous to the spread of information and nodes are analogous to people in a sample space [1].

In SIR model, a social network is partitioned into 3 different groups, those are: Susceptible (S) – this group consist of people that do not know about the particular information that is being spread. Infected (I) – this group of people know about the particular information and they will pass on this information(s) to the susceptible group. The spread from this group of people to Susceptible group occurs with probability p. Recover (R) – this group consist of people that knew about the information and they have no interest to share that particular information. This group of people are added

from Infected groups after they had no interest to share the information. This transition occurs with probability q.

4.2 SIS Model

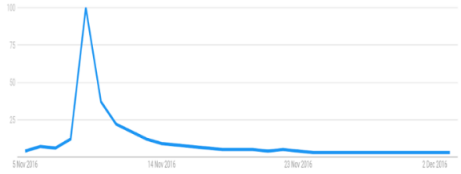
Not all google trend data can be modelled by SIR, especially the one with many peaks in it. Therefore, we introduce the SIS model. The difference between SIR and SIS lies on the recovery node. In SIR, once the node is recovered, it is no longer able to be infectious, however this is not necessarily true. In real life, sometimes the trend is able to reach the peak again. That is why, the enhancement made in SIS model is the recovery node is able to be infected again with probability q [1]. This might raise a question, is SIS better than SIR then? It depends on which google trend that we are trying to model. For a google trend with one peak, SIS is not capable of modelling such trend, which SIR is capable at. Similarly, for a google trend with some peaks, SIR is not capable of modelling it, which SIS is capable at. Therefore, each model has its own strength and weakness that covers each other.

4.3 Particle Swarm Optimization (PSO)

As we know, PSO will return the minimum or maximum value of an objective function with the constraints given. The objective function is $\sum(N(t) - S(t))^2$, where: $N(t)$ is obtained from the returned value of SIR or SIS model above. $S(t)$ is the actual google trend data. t is the time interval taken from google trend and the number of iterations occurs in SIR function [6]. Optimising the objective function by PSO to get the optimise p and q, one can try to model spread of information by using SIR or SIS.

5 Research of Event

Topic: Google Search on 'Donald Trump'



(a) "Donald Trump" search in 2016

(b) "Donald Trump" search in 2017

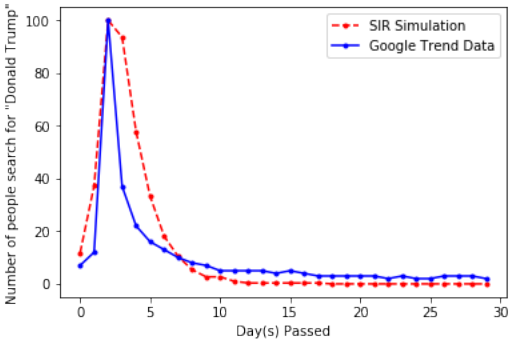
Figure 5.1: "Donald Trump" Google Trend

In this figure, two google trends are shown with the same topic of research but different time period. These models show an approximate spread of information over a network.

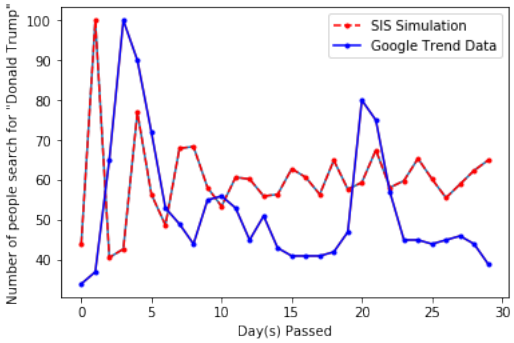
On the left, the graph went spike up and exponentially went down. This shows that the trend comes fast and went off quickly. The reason for that is on 2016 there is presidential election, where the current U.S. President, Donald Trump, made a sensational effect on this time period by winning the Presidential election, then after the election, people no longer search for "Donald Trump" since they have already know who he is. This kind of graph can be modelled by SIR model, which we will be explaining later on.

On the right, the graph went up and down in a more stable manner compared to the graph on the left. This might happen because there is no longer a sensational effect on the google searches, therefore it gets more and more stable. This kind of graph can be modelled by SIS model, which we will elaborate more later on.

Here we measure the precision of the SIR or SIS model with respect to the real google trend data, we want to minimize its objective function to estimate how accurate is SIR model in approximating the spread of information over a social network. In our case, we use the spread of information about "Donald Trump" for 30 days in two different time period.



(a) SIR Model



(b) SIS Model

Figure 4.2: SIR and SIS Model

From Particle Swarm Optimization by Power Law Clustering Graph model:
$p = 0.14403054$ and $q = 0.7199314$ for SIR
$p = 0.69977611$ and $q = 0.58111821$ for SIS

6 Code

```
In [ ]: import numpy as np
        import scipy as sp
        import networkx as nx
        import matplotlib.pyplot as plt
        import math
        import itertools as it
        import time
        import random
        from pyswarm import pso
```

```
In [ ]: def prop(G): #A function to print out the properties of graph
            vertex_degrees = list(dict(nx.degree(G)).values())
            #printing average degree
            print('Average degree: ',np.sum(vertex_degrees)/4039)
            edges = nx.number_of_edges(G)
            print('Number of edges: ',edges)
            print('Average cc: ',nx.average_clustering(G))
            print("Average Shortest Path: ",
                  nx.average_shortest_path_length(G))
            print("Diameter: ", nx.diameter(G))
```

6.1 Code for Facebook Model

```
In [ ]: #Facebook Model
        E = np.loadtxt("facebook_combined.txt")
```

```
E = E.astype(int)
E.shape
G1 = nx.Graph()
G1.add_edges_from(E)
nx.draw(G1, node_size = 50, node_color = 'blue',
        alpha = 0.2, edge_color = 'green')
plt.show()
vertex_degrees1 = list(dict(nx.degree(G1)).values())

#Plotting histogram
plt.hist(vertex_degrees1,
         bins = np.linspace(np.min(vertex_degrees),
                            1 + np.max(vertex_degrees), 50),
         label='Facebook',
         facecolor='red', alpha=0.8, rwidth = 0.9)
plt.title("Facebook Degree Distribution")
plt.grid(True)
plt.legend()
plt.show()
```

6.2 Code for Erdős–Rényi Model

```
In [ ]: #Erdos Renyi

n = 4039
p = 88234/(4039*4038*0.5)
g = nx.erdos_renyi_graph(n,p)
nx.draw(g, node_size = 50, node_color = 'blue',
        alpha = 0.2, edge_color = 'green')
plt.show()

prop(g)
vertex_degrees = list(dict(nx.degree(g)).values())
plt.hist(vertex_degrees,
         bins = np.linspace(np.min(vertex_degrees),
                            1 + np.max(vertex_degrees), 50),
         facecolor='blue', alpha=0.75, rwidth = 0.9)
plt.title("ER Model Vertex Degree Distribution")
plt.grid(True)
plt.show()
```

6.3 Code for Watts-Strogatz Model

```
In [ ]: #Watts-Strogatz
```

```
n=4039
k=44
p=1-(0.6055*4*43/(3*42))**(1/3)
G=nx.watts_strogatz_graph(n,k,p)

nx.draw(G, node_size = 50, node_color = 'blue',
        alpha = 0.2, edge_color = 'green')
plt.show()
prop(G)
vertex_degrees = list(dict(nx.degree(G)).values())

#Plotting histogram
plt.hist(vertex_degrees,
         bins = np.linspace(np.min(vertex_degrees),
                            1 + np.max(vertex_degrees), 15),
         facecolor='blue', alpha=0.75, rwidth = 0.9)
plt.title("WS Model Vertex Degree Distribution")
plt.grid(True)
plt.show()
```

6.4 Code for Barabási-Albert Model

In [ ]:
```
#Barabasi-Albert

n=4039
m=22
G=nx.barabasi_albert_graph(n,m)
nx.draw(G, node_size = 50, node_color = 'blue',
        alpha = 0.2, edge_color = 'green')
plt.show()
vertex_degrees = list(dict(nx.degree(G)).values())
prop(G)

#Plotting histogram
plt.hist(vertex_degrees,
         bins = np.linspace(np.min(vertex_degrees),
                            1100, 50),
         facecolor='blue', alpha=0.75, rwidth = 0.9)
plt.title("BA Model Vertex Degree Distribution")
plt.grid(True)
plt.show()
```

6.5 Code for Mediation-Driven Attachment Model

In [ ]:
```
#Mediation Driven Attachment

n = 4039
m = 22  #parameter m is average degree/2 (similar to BA model)
def MDA(n,m):
    G = nx.complete_graph(m+1)
    for v in np.arange(nx.number_of_nodes(G),n):
        v_exist=G.nodes() #extract nodes in graph G
        #List out degress of each node
        v_degrees= np.array(list(dict(nx.degree(G)).values()))
        #Choose a mediator
        v_mediator=np.random.choice(v_exist)
        #Finding the neighbors of mediator
        v_neighbors=G.neighbors(v_mediator)
        #Connect new node to m numbers of mediator's neighbors
        v_to_connect=np.random.choice(v_neighbors,
                                      size=m, replace=False).reshape(m,1)
        new_edges=np.concatenate((v*np.ones((m,1)),v_to_connect),axis=1)
        G.add_edges_from(new_edges)
    return G

G=MDA(n,m) #G is the graph with 4039 nodes
nx.draw(G, node_size = 50, node_color = 'blue',
        alpha = 0.2, edge_color = 'green')
plt.show()
v_degrees = list(dict(nx.degree(G)).values())
prop(G)

#Plotting histogram
plt.hist(v_degrees,
         bins = np.linspace(np.min(v_degrees),
                            1100, 50),
         facecolor='blue', alpha=0.75, rwidth = 0.9)
plt.title("MDA Model Vertex Degree Distribution")
plt.grid(True)
plt.show()
```

6.6 Code for Bianconi-Barabási Model

In [ ]:
```
#Bianconi-Barabasi

n = 4039
m = 22
```

```
def BB(n,m):
    G = nx.complete_graph(m+1)
    v_degrees = np.array(list(dict(nx.degree(G)).values()))
    #Assigning fitness level to every nodes
    v_fitness=np.random.random(nx.number_of_nodes(G))
    for v in np.arange(nx.number_of_nodes(G), n):
        v_exist = G.nodes()
        v_degrees = np.array(list(dict(nx.degree(G)).values()))
        v_fd=v_degrees*v_fitness
    #Connect probability based on degrees and fitness of individual node
        v_to_connect = np.random.choice(v_exist, size = m,
                                        p = v_fd / np.sum(v_fd),
                                        replace = False).reshape(m, 1)
        new_edges = np.concatenate((v * np.ones((m, 1)),
                                        v_to_connect), axis = 1)
        G.add_edges_from(new_edges)
        #Assign fitness level to new node
        v_fitness=np.append(v_fitness,np.random.random())
    return G

G=BB(n,m)
nx.draw(G, node_size = 50, node_color = 'blue',
        alpha = 0.2, edge_color = 'green')
plt.show()
v_degrees = list(dict(nx.degree(G)).values())
prop(G)

#Plotting histogram
plt.hist(v_degrees,
         bins = np.linspace(np.min(v_degrees),
                            1100, 50),
         facecolor='blue', alpha=0.75, rwidth = 0.9)
plt.title("BB Model Vertex Degree Distribution")
plt.grid(True)
plt.show()
```

6.7 Code for Custom Model

In [ ]:
```
#Custom Model

n = 4039
m = 22  #parameter m is average degree/2 (similar to BA model)
def Custom(n,m):
    G = nx.complete_graph(m+1)
    for v in np.arange(nx.number_of_nodes(G),n):
```

```
        v_exist=G.nodes()
        v_degrees= np.array(list(dict(nx.degree(G)).values()))
        #Choose initial node
        #Number of initial node to be connected
        #p can be adjusted accordingly to match facebook model
        mr=np.random.choice([1,2],p=[0.95,0.05])
        #Choose random node(s)
        v_ic=np.random.choice(v_exist,size=mr,replace=False)
        c1=v_ic.reshape(mr,1)
        #Choose secondary nodes
        v_neighbors=np.array([])
        #All the neighbours of initial node(s)
        for i in range(0,len(v_ic)):
            v_neighbors=np.append(v_neighbors,G.neighbors(v_ic[i]))

        v_sc=np.array(list(set(v_neighbors)))
        v_sc2=v_sc.astype(int)
        #Extracting degrees of neighbours
        v_sc_degrees= v_degrees[v_sc2]
        #Remaining edges will be connected to secondary nodes
        ms=m-mr
        c2=np.random.choice(v_sc, size=ms,
                            p=v_sc_degrees/np.sum(v_sc_degrees),
                            replace=False).reshape(ms,1)
        #connect new node to both inital and secondary nodes
        #Possibility which initial and secondary node chosen simultaneously
        #but with replace=False, it can only be connected once(as initial)
        v_to_connect=np.concatenate((c1,c2),axis=0)
        new_edges=np.concatenate((v*np.ones((m,1)),v_to_connect),axis=1)
        G.add_edges_from(new_edges)
    return G

G=Custom(n,m) #G is the graph with 4039 nodes
nx.draw(G, node_size = 50, node_color = 'blue',
        alpha = 0.2, edge_color = 'green')
plt.show()
v_degrees = list(dict(nx.degree(G)).values())
prop(G)

#Plotting histogram
plt.hist(v_degrees,
         bins = np.linspace(np.min(v_degrees),
                            1100, 50),
         facecolor='blue', alpha=0.75, rwidth = 0.9)
plt.title("Custom Model Vertex Degree Distribution")
```

```
plt.grid(True)
plt.show()
```

6.8 Code for Power Law Clustering Model

```
In [ ]: #Power Law CG

        n=4039
        m=22
        p=1
        #Using built in function
        G2=nx.random_graphs.powerlaw_cluster_graph(n,m,p)
        nx.draw(G2, node_size = 50, node_color = 'blue',
                alpha = 0.2, edge_color = 'green')
        plt.show()
        prop(G2)
        vertex_degrees2 = list(dict(nx.degree(G2)).values())

        #Plotting histogram
        plt.hist(vertex_degrees2,
                 bins = np.linspace(np.min(vertex_degrees),
                                    1100,50),
                 facecolor='blue', alpha=0.75, rwidth = 0.9)
        plt.title("PLCG Model Vertex Degree Distribution")
        plt.grid(True)
        plt.show()
```

6.9 Code for Facebook vs Power Law Model

```
In [ ]: #Comparing Power Law Model vs Facebook
        E = np.loadtxt("facebook_combined.txt")
        E = E.astype(int)
        E.shape
        G1 = nx.Graph()
        G1.add_edges_from(E)
        nx.draw(G1, node_size = 50, node_color = 'blue',
                alpha = 0.2, edge_color = 'green')
        plt.show()
        vertex_degrees1 = list(dict(nx.degree(G1)).values())

        #Plotting histogram comparing degree distribution
        plt.hist(vertex_degrees1,
                 bins = np.linspace(np.min(vertex_degrees),
                                    1 + np.max(vertex_degrees), 50),
```

```
            label='Facebook',
            facecolor='red', alpha=0.8, rwidth = 0.9)
    plt.hist(vertex_degrees2,
            bins = np.linspace(np.min(vertex_degrees),
                               1100,  50), label='Power Law',
            facecolor='yellow', alpha=0.5, rwidth = 0.9)
    plt.title("Facebook vs Power Law Vertex Degree Distribution")

    plt.grid(True)
    plt.legend()
    plt.show()
```

6.10 Code for SIR Model

```
In [ ]: def sir(G,p,q,num_infected):
    #first, we initialize the variables that we will use in this function
    # n is number of nodes
        n = nx.number_of_nodes(G)
    # I is the array of infected nodes
        I = np.unique(np.random.choice(range(n), num_infected, replace=False))
    # A is the adjacency matrix of G
        A = np.array(nx.to_numpy_matrix(G))
    # S is the array of Susceptible nodes
        S = np.unique(np.delete(np.arange(n),I))
        R = np.array([]) # R is the array of recovered nodes
    # numberofI is the array that will stored the number of infected people
    #in a particular day
        numberofI = np.array([num_infected])
    # number of days of a google trend (chosen google trend)
        for i in range(29):
            # 2 conditions, so that no error occurs
            if len(R) == 0:
                prob = np.triu((np.random.rand(n,n)), 1)
                prob = np.mat(prob + prob.T)
    #B : adjacency matrix of a sub-graph of G, where only infected nodes
    #and their neighbour is included
                B = A+0
    # we wanted to form this adjacency matrix so that each rows
    #and coloumns indexes represent node, where:
                B[S.astype(int),:] = 0
    #row's indexes are infected nodes, columns are susceptible or recover
                B[:,I.astype(int)] = 0
    # the entries shows the probability theyre infected
                prob_infected = np.array(B)*np.array(prob)
    # comparing to get boolean value
```

```
            infct = 1*(prob_infected>1-p)
# the nodes which are(is) newly infected, consist of neighbours of I
            newly_infected = np.array(np.where(np.sum(infct,axis=0)>0)[0])
#infected for temporary before eliminating the nodes which are(is) recovered
            I1 = np.unique(np.concatenate((np.array(I),newly_infected),
                                          axis = None))
# updating the array S, since I has changed
            S = np.delete(np.arange(n),I1)
#this is done for Infected nodes, infecting their neighbors
#next, is infected nodes that recover
#we create a 1 Dimensional array with the number of entries
#the same as no of infected (updated)
#the input for this array is random number from 0 to 1
#when it exceeds 1-q, the node is recovered and no longer infecting others
            recover_array = np.random.rand(I.size)
            R = I[(recover_array>1-q)]
            I = np.array(list(set(I1.tolist()) - set(R.tolist())))
            S = S
            numberofI = np.append(numberofI,I.size)
        #same explaination to the first condition
        else:
            prob = np.triu((np.random.rand(n,n)), 1)
            prob = np.mat(prob + prob.T)
# this matrix is adjacency matrix for nodes in I with their neighbours
            B = A+0
#row's indexes are infected nodes, columns are susceptible or recover
            B[S.astype(int),:] = 0
            B[:,I.astype(int)] = 0
            B[R.astype(int),:] = 0
            B[:,R.astype(int)] = 0
# entries shows the probability theyre infected
            prob_infected = np.array(B)*np.array(prob)
            infct = 1*(prob_infected>1-p)
            newly_infected = np.array(np.where(np.sum(infct,axis=0)>0)[0])
            #infected for temp before - R
            I1 = np.unique(np.concatenate((np.array(I),newly_infected),
                                          axis = None))
            S = np.delete(np.arange(n),I1)
            recover_array = np.random.rand(I.size)
            R = np.append(R,I[(recover_array>1-q)])
            I = np.array(list(set(I1.tolist()) - set(R.tolist())))
            S = S
            numberofI = np.append(numberofI,I.size)
#because we wanted to compare our result to google trend
#we need to normalize our result.
```

```
        normalized_I = (numberofI)/max(numberofI)*100
        return normalized_I
```

6.11 Code for SIR PSO

```
In [ ]: G = nx.random_graphs.powerlaw_cluster_graph(505, 22, 1)
        #30 days of Donald Trump search
        google_trend = [7,12,100,37,22,16,13,10,8,7,5,5,5,5,
                        4,5,4,3,3,3,3,3,2,3,2,2,3,3,3,2]
        def f(X):
            p = X[0]
            q = X[1]
            return np.sum((sir(G,p,q,int(0.07*505))-google_trend)**2)

        lb = [0, 0] # lower bounds for p and q
        ub = [1, 1] # upper bounds for p and q

        xopt, fopt = pso(f, lb, ub)
        #running pso to get the minimum p and q that makes
        #SIR similar to the chosen google trend


        print("Minimum p,q = ", xopt)
        print("error = ", fopt)
        a = sir(G,0.10719,0.46393,int(0.07*505))
        x = range(30)
        plt.plot(x,a,linestyle='--',marker='.',color='red',label='SIR Simulation')
        plt.plot(x,google_trend,marker='.',color='blue',label='Google Trend Data')
        plt.xlabel("Day(s) Passed")
        plt.ylabel('Number of people search for "Donald Trump"')
        plt.legend()
        plt.show()
```

6.12 Code for SIS Model

```
In [ ]: def sis(G,p,q,num_infected):
        #first we initialize the variable that we will be using, that is:
            n = nx.number_of_nodes(G) #number of nodes in G
        #an array of infected nodes
            I = np.unique(np.random.choice(range(n), num_infected, replace=False))
            A = np.array(nx.to_numpy_matrix(G)) #adjacency matrix of G
        #an array of susceptible nodes
            S = np.unique(np.delete(np.arange(n),I))
        #array of newly susceptible nodes
```

```
    S1 = np.array([])
#array of total people infected in each day
    numberofI = np.array([num_infected])
    for i in range(29):# number of days in google trend
        #we seperate into 2 conditions here to avoid error
        if len(S1) == 0:
            prob = np.triu((np.random.rand(n,n)), 1)
# we generate a matrix to resemble its probability for each edges
            prob = np.mat(prob + prob.T)
# this matrix is adjacency matrix for nodes in I with their neighbours
            B = A+0
#row's indexes are infected nodes, columns are susceptible
            B[S.astype(int),:] = 0
            B[:,I.astype(int)] = 0
# entries shows the probability they are infected
            prob_infected = np.array(B)*np.array(prob)
#we compare thid to probability p to determine which node is infected
            infct = 1*(prob_infected>1-p)
#here are the newly infected nodes
            newly_infected = np.array(np.where(np.sum(infct,axis=0)>0)[0])
#temporarily infected nodes before eliminating ndoes from S1
            I1 = np.unique(np.concatenate((np.array(I),newly_infected),
                                          axis = None))
            S = np.delete(np.arange(n),I1) #updating the S array
#this is done for Infected nodes, infecting their neighbors
#next, is infected nodes that back to susceptible
#we create a 1 Dimensional array with the number of entries
#the same as no of infected (updated)
#the input for this array is random number from 0 to 1
#when it exceeds q, the node is back to susceptible and be able to get infected
            recover_array = np.random.rand(I.size)
            S1 = I[(recover_array>1-q)]
            I = np.array(list(set(I1.tolist()) - set(S1.tolist())))
            S = np.unique(np.concatenate((S1,S)))
            numberofI = np.append(numberofI,I.size)
        else:
#the second condition is very similar to the first condition.
            prob = np.triu((np.random.rand(n,n)), 1)
            prob = np.mat(prob + prob.T)
# this matrix is adjacency matrix for nodes in I with their neighbours
            B = A+0
#row's indexes are infected nodes, columns are susceptible
            B[S.astype(int),:] = 0
            B[:,I.astype(int)] = 0
            B[S1.astype(int),:] = 0
```

```
                B[:,S1.astype(int)] = 0
        # entries shows the probability theyre infected
                prob_infected = np.array(B)*np.array(prob)
                infct = 1*(prob_infected>1-p)
                newly_infected = np.array(np.where(np.sum(infct,axis=0)>0)[0])
        #infected for temp before - R
                I1 = np.unique(np.concatenate((np.array(I),newly_infected),
                                              axis = None))
                S = np.delete(np.arange(n),I1)
                recover_array = np.random.rand(I.size)
                S1 = I[(recover_array>1-q)]
                I = np.array(list(set(I1.tolist()) - set(S1.tolist())))
                S = np.unique(np.concatenate((S1,S)))
                numberofI = np.append(numberofI,I.size)
            normalized_I = (numberofI)/max(numberofI)*100
            return normalized_I
```

6.13 Code for SIS PSO

```
In [ ]: G = nx.random_graphs.powerlaw_cluster_graph(303,22,1)
        #30 days of Donald Trump search
        google_trend = [34,37,65,100,90,72,53,49,44,55,56,45,51,43,41,41,
                        41,42,47,80,75,57,45,45,44,45,46,44,39 ]
        def f(X):
            p = X[0]
            q = X[1]
            return np.sum((sis(G,p,q,int(0.34*303))-google_trend)**2)

        lb = [0, 0] # lower bounds for p and q
        ub = [1, 1] # upper bounds for p and q

        xopt, fopt = pso(f, lb, ub)

        print("Minimum p,q = ", xopt)
        print("error = ", fopt)
        x = range(30)
        a = sis(G,xopt[0],xopt[1],int(0.34*303))
        plt.plot(x,a)
        plt.plot(x,google_trend)
        plt.plot(x,a,linestyle='--',marker='.',color='red',label='SIS Simulation')
        plt.plot(x,google_trend,marker='.',color='blue',label='Google Trend Data')
        plt.xlabel("Day(s) Passed")
        plt.ylabel('Number of people search for "Donald Trump"')
        plt.legend()
        plt.show()
```

References

[1] M.E.J. Newman, The structure and function of complex networks, SIAM Rev. 45 (2) (2003) 167–256.

[2] Saramäki, J., Kaski, K. (2004). Scale-free networks generated by random walkers. Physica A: Statistical Mechanics and Its Applications, 341, 80-86.

[3] Hassan, M. K., Islam, L., Haque, S. A. (2017). Degree distribution, rank-size distribution, and leadership persistence in mediation-driven attachment networks. Physica A: Statistical Mechanics and Its Applications, 469, 23-30.

[4] Oliveira, R. I., Ribeiro, R., Sanchis, R. (2018). Disparity of clustering coefficients in the Holme[U+2012]Kim network model. Advances in Applied Probability, 50(3), 918-943.

[5] Toivonen, R., Onnela, J., Saramäki, J., Hyvönen, J., Kaski, K. (2006). A model for social networks. Physica A: Statistical Mechanics and Its Applications, 371(2), 851-860.

[6] Poli, R., Kennedy, J., Blackwell, T. (2007). Particle swarm optimization. Swarm Intelligence, 1(1), 33-57.