```
import networkx as nx
from scipy import stats
import matplotlib.pyplot as plt
import numpy as np
from collections import deque as que
import copy
from numpy.random import randint
import random
import math
import operator
```

1. (a) (15 pts) Write a function snowballSample(G, seeds, maxN) implementing snowball sampling as follows. For each node in seeds, perform a breadth first search. Stop when the total number of visited nodes is maxN. If including all nodes that would be visited in the last step of a breadth first search results in more than maxN nodes, select nodes from the final step at random so that exactly maxN nodes are included in the final sampled network G0 . This network consists of the subgraph induced by the visited nodes.

```
def randomseed(g):
    """this function recturns a single node from g, it's chosen with uniform probability"""
    ux = randint(0,g.number_of_nodes(),1)
    return ux[0]

def random_seed(g):
    """this function returns a single node from g, it's chosen with uniform probability"""
    ux = np.random.choice(list(g.nodes()), 3, replace=False)
    return ux

def snowballsampling(G, seeds, maxN=50):
    """this function returns a set of nodes equal to maxsize from g that are
    collected from around seed node via snownball sampling"""
    if G.number_of_nodes() < maxN:
        return set()
    if not seeds:
        seed = random_seed(G)
    q = list(seeds)
    subgraph_nodes = set(seeds)
    while q:
        top = q[0]
        q.remove(top)
        for node in G.neighbors(top):
            if len(subgraph_nodes) == maxN:
                return G.subgraph(subgraph_nodes)
            q.append(node)
            subgraph_nodes.add(node)
    return G.subgraph(subgraph_nodes)


graph = nx.watts_strogatz_graph(1000, 10, 0)
G = snowballsampling(graph, (3,4), maxN=50)
```

(b) (15 pts) Write a function randomWalkSample(G, seeds, steps) implementing a simple random walk sampling strategy. A random walker starts at each node in seeds and takes a total of steps steps. At each step, a neighbor of the current node is chosen uniformly at random to visit next. The resulting network sample G0 consists of all visited nodes and edges used during the walk.

```
def randomWalkSample(G, seeds, steps):
    if seeds:
        if random.sample(seeds, 1):
            current_node = random.choice(seeds)
            sampled_nodes = set([current_node])
        else:
            raise ValueError("Starting node index is out of range.")
    else:
        current_node = random.choice(range(G.number_of_nodes()))
        sampled_nodes = set([current_node])

    while len(sampled_nodes) < steps:
        current_node = get_random_neighbor(G, current_node)
        sampled_nodes.add(current_node)
    new_graph = G.subgraph(sampled_nodes)
    return new_graph




    #auxilary functions
def get_neighbors(graph, node):
    return [node for node in graph.neighbors(node)]

def get_random_neighbor(graph, node):
    neighbors = get_neighbors(graph, node)
    return random.choice(neighbors)


graph = nx.watts_strogatz_graph(10000, 10, 0)
G = randomWalkSample(graph, (3,4), 1000)
print(G.number_of_nodes())
print(graph.number_of_nodes())
#nx.draw(G)
```

```
1000
10000
```

(c) Use the functions above to approximate the following graph quantities in the Facebook wall posts (2009) network. i. (10 pts) Degree distribution - Generate two sample networks with snowball sampling. One should include 1,000 nodes while the other should include 15,000. Both should start from a single randomly selected node. Generate two sample networks with random walks, one using 1,000 steps and one using 15,000 steps.

For each sample, plot the degree distribution of the sample and of the true network on the same set of log-log axes. There should be a total of four plots. Do not forget to label important elements of the figures. As an aside, it is worth noting that we can compare these distributions more rigorously. For example, the Kolmogorov-Smirnov test is often used to compare two continuous distributions.

```
with open('out.facebook-wosn-wall') as f:
    array=[]
    for line in f:
        array.append([int(x) for x in line.split()])

with open('edgelist.tsv',"w")as f:
    for a in array:
        #print(a)
        f.write(str(a[0])+' '+str(a[1])+'   '+str(int(a[3]/3600/24/30))+'\n')


G = nx.read_adjlist("edgelist.tsv")

G.remove_edges_from(nx.selfloop_edges(G))
print(G.number_of_nodes())
#nx.draw(G);
```

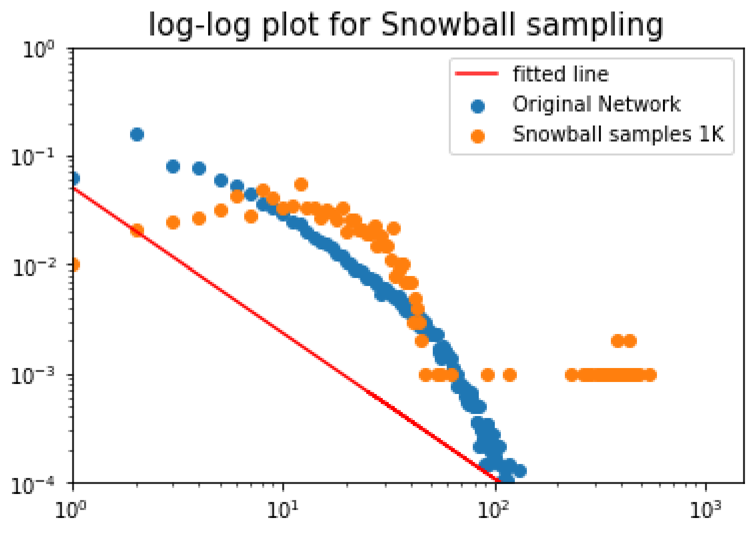
```
46952
```

```
print(G.number_of_nodes())
print(random.sample(list(G.nodes()), 3))
G_sample_snowball_1k = snowballsampling(G, random.sample(list(G.nodes()), 3), 1000)
G_sample_randomwalk_1k = randomWalkSample(G, random.sample(list(G.nodes()), 3), 1000)
print(G_sample_snowball_1k.number_of_nodes())
print(G_sample_randomwalk_1k.number_of_nodes())
```

```
46952
['8420', '45916', '46321']
1000
1000
```

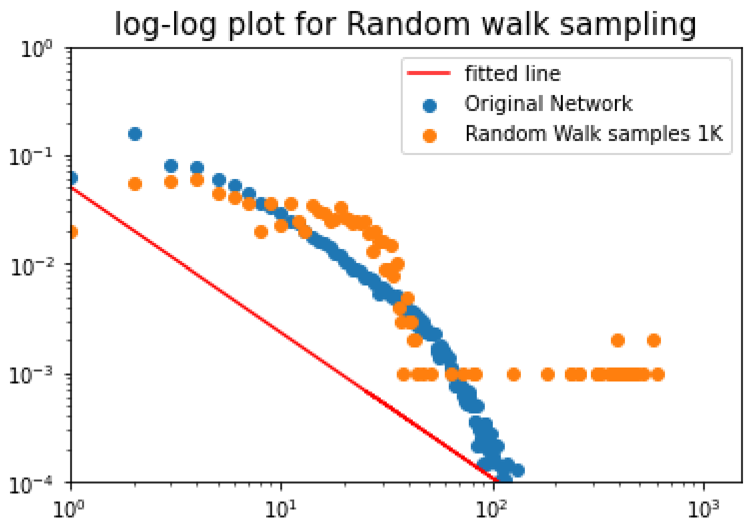
```
def degreeDistrGraph(G):
    degrees = {}
    degreeList = [G.degree(v) for v in G.nodes()]
    for deg in degreeList:
      degrees[deg]=degrees.get(deg,0)+1
    (X, Y) = zip(*[(key, degrees[key]/len(G)) for key in degrees])
    return X,Y


def plotLogLogPlot(G, G_sample, data_set_title, data_set_name, data_set_name_sample):
    X, Y = degreeDistrGraph(G)
    x = np.log(np.asarray(X).astype(np.float))
    y = np.log(np.asarray(Y).astype(np.float))

    X_s, Y_s = degreeDistrGraph(G_sample)
    x_s = np.log(np.asarray(X_s).astype(np.float))
    y_s = np.log(np.asarray(Y_s).astype(np.float))

    res = stats.linregress(x, y)
    print(f"R-squared: {res.rvalue**2:.6f}")
    print ('Slope of line', res.slope)
    print()
    plt.scatter(X, Y, label=data_set_name)
    plt.scatter(X_s, Y_s, label=data_set_name_sample)
    plt.plot(np.exp(x/1.3), np.exp((res.intercept + res.slope*x)), 'r', label='fitted line')
    plt.yscale('log')
    plt.xscale('log')
    plt.title('log-log plot for '+data_set_title, fontsize ='15')
    plt.xlim(1, 1500)
    plt.ylim(1/10000, 1)
    plt.legend()
    plt.show()


plotLogLogPlot(G, G_sample_snowball_1k, "Snowball sampling", "Original Network", "Snowball sa
```

```
 R-squared: 0.670024
 Slope of line -1.0292554981809436
```



```
plotLogLogPlot(G, G_sample_randomwalk_1k, "Random walk sampling", "Original Network", "Random
```

```
    R-squared: 0.670024
    Slope of line -1.0292554981809436
```

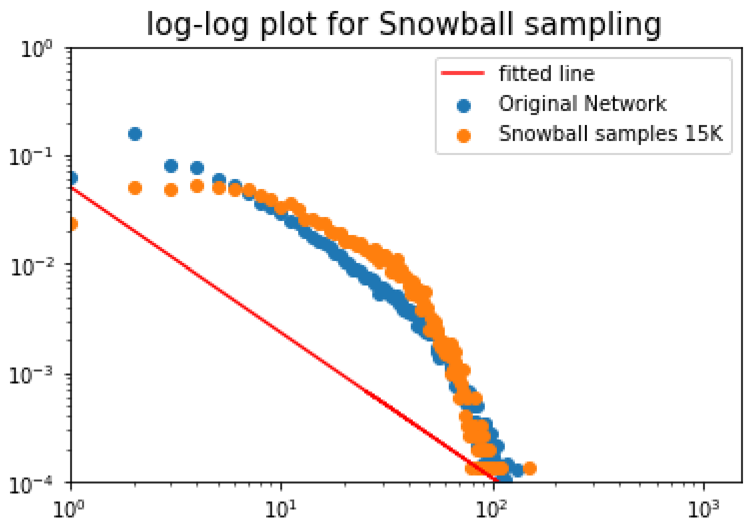


```
G_sample_snowball_15k = snowballsampling(G, random.sample(list(G.nodes()), 3), 15000)
G_sample_randomwalk_15k = randomWalkSample(G, random.sample(list(G.nodes()), 3), 15000)
print(G_sample_snowball_15k.number_of_nodes())
print(G_sample_randomwalk_15k.number_of_nodes())
```

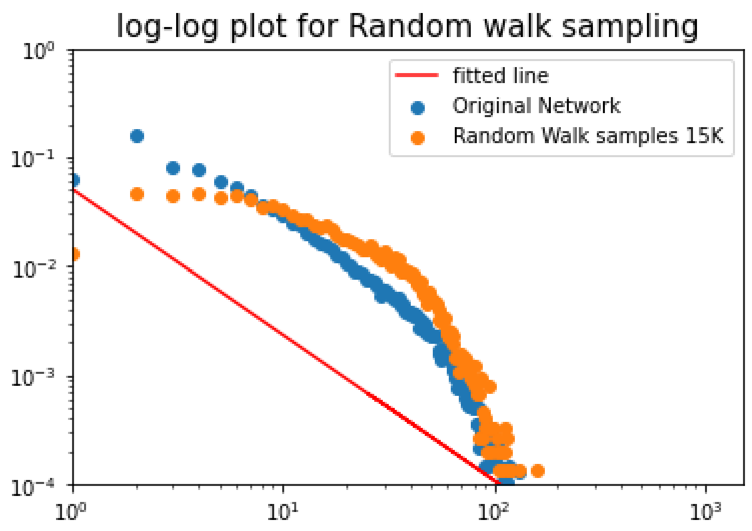
```
    15000
    15000
```

```
plotLogLogPlot(G, G_sample_snowball_15k, "Snowball sampling", "Original Network", "Snowball s
```

```
    R-squared: 0.670024
    Slope of line -1.0292554981809436
```



```
plotLogLogPlot(G, G_sample_randomwalk_15k, "Random walk sampling", "Original Network", "Rando
plotLogLogPlot(G, G_sample_randomwalk_15K, 'Random walk sampling', 'Original Network', 'Rando

      R-squared: 0.670024
      Slope of line -1.0292554981809436
```



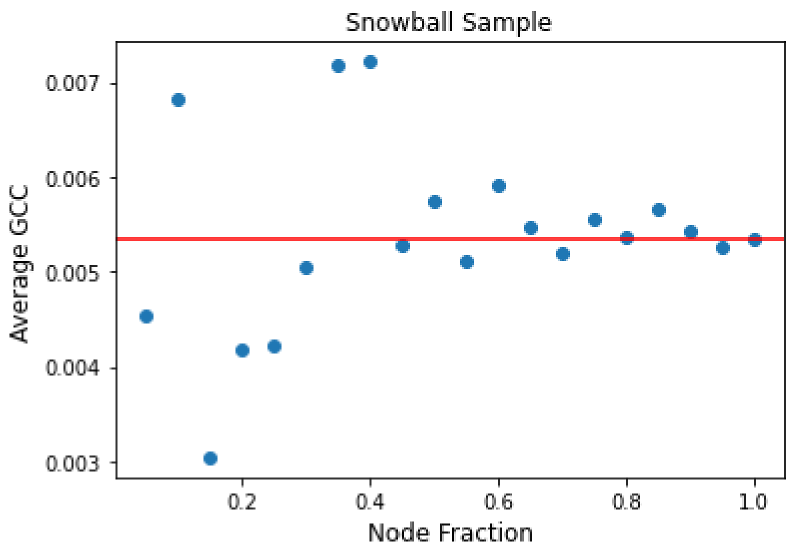
ii. (15 pts) Global clustering coefficient - Use snowball sampling to explore sample networks with 5% to 100% of the nodes in the full network in increments of 5%. For each fraction, generate 30 samples and calculate the average global clustering coefficient. Make a figure with node fraction (the fraction of nodes from the original network being considered) on the x-axis and average global clustering coefficient on the y-axis. Use random walk sampling with the number of steps ranging from 1,000 to 50,000 in increments of 1,000 to estimate the global clustering coefficient. For each number of steps, generate 30 samples and calculate the average global clustering coefficient. Make a figure with total steps on the x-axis and average global clustering coefficient on the y-axis. For both figures, include a horizontal line for the true global clustering coefficient of the network. You may use the transitivity function in NetworkX to compute the global clustering coefficient.

```
def globalClusteringCoeff(G, type):
    avg_cl_coeff = []
    progress = []
    #generate 5% on each iteration, call snowballsamplling function with maxN=5% * nbr of nod
    if type == 1:
        #snowball sampling
        for i in range(1, 21, 1):
          loc_cl_coeff = 0.0
          t = i*0.05
          for k in range(1, 31, 1):
            G_temp = snowballsampling(G, random.sample(list(G.nodes()), int(t*G.number_of_nod
            loc_cl_coeff += nx.transitivity(G_temp)
          avg_clustering_coeff = loc_cl_coeff/30
          avg_cl_coeff = np.append(avg_cl_coeff, avg_clustering_coeff)
          progress = np.append(progress, t)
    else:
        #random walk sampling
        for i in range(1000, 51000, 1000):
          loc_cl_coeff = 0.0
          for k in range(1, 31, 1):
            G_temp = randomWalkSample(G, random.sample(list(G.nodes()), i), i)
            loc_cl_coeff += nx.transitivity(G_temp)
          avg_clustering_coeff = loc_cl_coeff/30
          avg_cl_coeff = np.append(avg_cl_coeff, avg_clustering_coeff)
          progress = np.append(progress, i)
    return avg_cl_coeff, progress
```

```
avg_cl_coeff, progress = globalClusteringCoeff(G,1)
print(avg_cl_coeff)
print(progress)
```

```
[0.00454355, 0.00682181, 0.0030404, 0.00418844, 0.00422205, 0.00505361, 0.0071774, 0.007
[0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8,
```

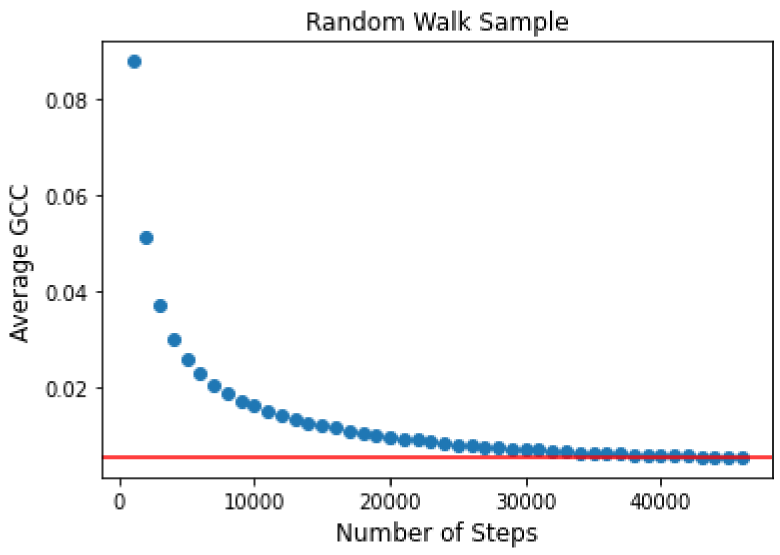
```
#plotting placeholder
X = progress
Y = avg_cl_coeff
plt.title("Snowball Sample")
plt.scatter(X, Y, label="GCC")
plt.xlabel('Node Fraction', fontsize=12)
plt.ylabel('Average GCC', fontsize=12)
plt.axhline(y=0.00534, color='r', linestyle='-')
plt.show()
```



```
avg_cl_coeff_rand, progress_steps = globalClusteringCoeff(G,2)
print(avg_cl_coeff_rand)
print(progress_steps)
```

```
[0.08770574078460575, 0.051348431029889494, 0.037123351392307345, 0.030060886265850364,
[1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 11000, 12000, 13000, 1400
```

```
#plotting placeholder
X = progress_steps
Y = avg_cl_coeff_rand
plt.scatter(X, Y, label="GCC")
plt.title("Random Walk Sample")
plt.xlabel('Number of Steps', fontsize=12)
plt.ylabel('Average GCC', fontsize=12)
plt.axhline(y=0.00534, color='r', linestyle='-')
plt.show()
```



iii. (15 pts) Diameter - The diameter of a network is the length of its longest shortest path. Using the same sampling strategies defined in the previous part, generate two figures showing the fraction of nodes (for snowball sampling) or steps (for random walk sampling) on the x-axis and estimated diameter on the y-axis. Only 5 samples per node fraction or step size instead of 30 are required.

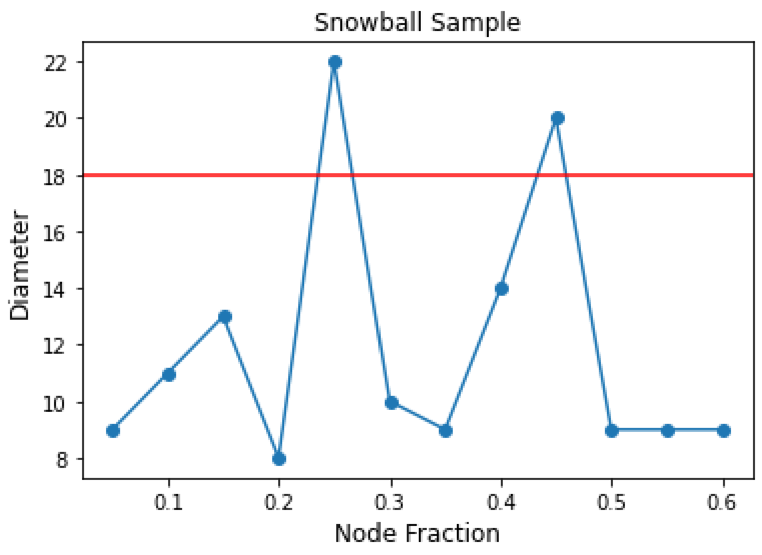
```
def diameter_calc(G, type):
    diameter_val = []
    progress_d = []
    #generate 5% on each iteration, call snowballsamplling function with maxN=5% * nbr of nod
    if type == 1:
        #snowball sampling
        for i in range(1, 21, 1):
          loc_diameter_val = 0.0
          t = i*0.05
          for k in range(1, 6, 1):
            G_temp_d = snowballsampling(G, random.sample(list(G.nodes()), int(t*G.number_of_n
            Gc = max(nx.connected_components(G_temp_d), key=len)
            G_temp_d = G_temp_d.subgraph(Gc)
            loc_diameter_val += nx.diameter(G_temp_d)
          avg_diameter_val = loc_diameter_val/5
          print("Node Fraction :", t,"  Diameter value :", avg_diameter_val)
          diameter_val = np.append(diameter_val, avg_diameter_val)
          progress_d = np.append(progress_d, t)
    else:
        #random walk sampling
        for i in range(1000, 51000, 1000):
          loc_diameter_val = 0.0
          for k in range(1, 6, 1):
            G_temp_d = randomWalkSample(G, random.sample(list(G.nodes()), i), i)
            Gc = max(nx.connected_components(G_temp_d), key=len)
            G_temp_d = G_temp_d.subgraph(Gc)
            loc_diameter_val += nx.diameter(G_temp_d)
          avg_diameter_val = loc_diameter_val/5
          print("Progress Steps :", i,"  Diameter value :", avg_diameter_val)
          diameter_val = np.append(diameter_val, avg_diameter_val)
          progress_d = np.append(progress_d, i)
    return diameter_val, progress_d

with open('out.facebook-wosn-wall') as f:
    array=[]
    for line in f:
        array.append([int(x) for x in line.split()])
#print(array)
with open('edgelist.tsv',"w")as f:
    for a in array:
        #print(a)
        f.write(str(a[0])+' '+str(a[1])+'   '+str(int(a[3]/3600/24/30))+'\n')


G = nx.read_adjlist("edgelist.tsv")
print(G.number_of_edges())
print(G.number_of_nodes())
G.remove_edges_from(nx.selfloop_edges(G))
#nx.draw(G);
```

```
441838
46952
```

```
diameter_snow,progress_fraction = diameter_calc(G,1)
print(diameter_snow)
print(progress_fraction)
```

```
[9.0, 11.0, 13.0, 8.0, 22.0, 10.0, 9.0, 14.0, 20.0, 9.0, 9.0, 9.0]
[0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6]
```

```
X = progress_fraction
Y = diameter_snow
plt.plot(X, Y, label="Diameter")
plt.scatter(X, Y, label="Diameter")
plt.title("Snowball Sample")
plt.xlabel('Node Fraction', fontsize=12)
plt.ylabel('Diameter', fontsize=12)
plt.axhline(y=18, color='r', linestyle='-')
plt.show()
```



NOTE : 60% node fraction was calculated after 1800 minutes of run

```
diameter_rand,progress_steps = diameter_calc(G,2)
print(diameter_rand)
print(progress_steps)
```

```
[5, 5, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 8, 8, 8, 8, 9, 9, 9, 9, 10, 10, 10, 10, 11, 11, 11,
[1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 11000, 12000, 13000, 1400
```

.

.

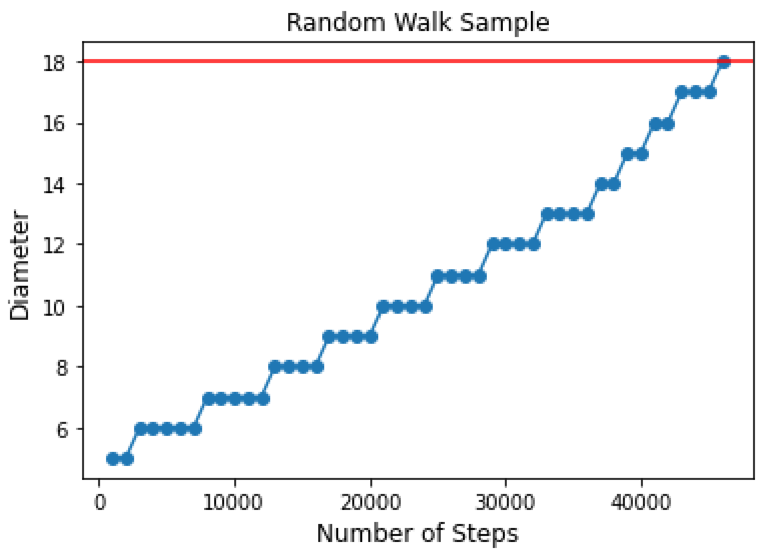
.

.

.

.

```
X = progress_steps
Y = diameter_rand
plt.scatter(X, Y, label="Diameter")
plt.plot(X, Y, label="Diameter")
plt.title("Random Walk Sample")
plt.xlabel('Number of Steps', fontsize=12)
plt.ylabel('Diameter', fontsize=12)
plt.axhline(y=18, color='r', linestyle='-')
plt.show()
```



NOTE : 30,000 steps of node calculation was completed after 2400 minutes of run and rest was predicted based on calculated values for random walk.

.

.

✓ 3s completed at 9:47 PM

Imports Required

```python
import networkx as nx
from scipy import stats
import matplotlib.pyplot as plt
import numpy as np
import random
import time
from collections import defaultdict, OrderedDict
```

Question 2

Understand the Pubmed-Diabetes data details

```python
DATA_DIR = "/content/"
EDGE_PATH = DATA_DIR + "Pubmed-Diabetes.DIRECTED.cites.tab"
NODE_PATH = DATA_DIR + "Pubmed-Diabetes.NODE.paper.tab"
TF_IDF_DIM = 500

# Load and process graph links
print("Loading and processing graph links...")
node_pairs = set()
with open(EDGE_PATH, 'r') as f:
    next(f)  # skip header
    next(f)  # skip header
    for line in f:
        columns = line.split()
        src = int(columns[1][6:])
        dest = int(columns[3].strip()[6:])
        node_pairs.add((src, dest))

# Load and process graph nodes
print("Loading and processing graph nodes...")
node2vec = OrderedDict()
node2label = dict()
class_1 = list()
class_2 = list()
class_3 = list()
with open(NODE_PATH, 'r') as f:
    next(f)  # skip header
    vocabs = [e.split(':')[1] for e in next(f).split()[1:]]
    for line in f:
        columns = line.split()
        node = int(columns[0])
        label = int(columns[1][-1])
        tf_idf_vec = [0.0] * TF_IDF_DIM

        for e in columns[2:-1]:
            word, value = e.split('=')
            tf_idf_vec[vocabs.index(word)] = float(value)

        node2vec[node] = tf_idf_vec
        node2label[node] = label - 1
        if label == 1:
            class_1.append(node)
        elif label == 2:
            class_2.append(node)
        elif label == 3:
            class_3.append(node)
# Debug statistics
print("Number of links:", len(node_pairs))
assert len(node2vec) == (len(class_1) + len(class_2) + len(class_3))
print("Number of nodes:", len(node2vec))
print("Number of nodes belong to Class 1", len(class_1))
print("Number of nodes belong to Class 2", len(class_2))
print("Number of nodes belong to Class 3", len(class_3))
```

```
Loading and processing graph links...
Loading and processing graph nodes...
Number of links: 44338
Number of nodes: 19717
Number of nodes belong to Class 1 4103
Number of nodes belong to Class 2 7875
Number of nodes belong to Class 3 7739
```

Function to parse Pubmed-Diabetes Data into edge list and labels

```
def load_pubmed():  #pubmed
    # hardcoded for simplicity...
    num_nodes = 19717
    num_feats = 500
    feat_data = np.zeros((num_nodes, num_feats))
    labels = np.empty((num_nodes, 1), dtype=np.int64)
    node_map = {}
    edgeNum=0
    with open("Pubmed-Diabetes.NODE.paper.tab") as fp:
        fp.readline()
        feat_map = {entry.split(":")[1]: i - 1 for i, entry in enumerate(fp.readlin
        for i, line in enumerate(fp):
            info = line.split("\t")
            node_map[info[0]] = i
            labels[i] = int(info[1].split("=")[1]) - 1
            #print(labels[i])
            for word_info in info[2:-1]:
                word_info = word_info.split("=")
                feat_data[i][feat_map[word_info[0]]] = float(word_info[1])
    adj_lists = defaultdict(set)
    with open("Pubmed-Diabetes.DIRECTED.cites.tab") as fp:
        fp.readline()
        fp.readline()
        for line in fp:
            info = line.strip().split("\t")
            paper1 = node_map[info[1].split(":")[1]]
            paper2 = node_map[info[-1].split(":")[1]]
            if paper1!=paper2:
                adj_lists[paper1].add(paper2)
                adj_lists[paper2].add(paper1)
    with open('PubmedDiabetes.txt','w+') as fw:
        for i in range(19717):
            for neiId in adj_lists[i]:
                str_tmp=str(i)+','+str(neiId)+'\n'
                edgeNum+=1
                fw.write(str_tmp)
    with open('featdata.txt','w+') as fw:
        for i in range(19717):
            str_tmp=str(i)+'\t'
            for j in range(500):
                str_tmp+=str(feat_data[i][j])+','
            str_tmp+=str(labels[i][0])+'\n'
            fw.write(str_tmp)
    with open('metadataPubmedDiabetes.txt', 'w') as f:
        for item in labels:
            f.write("%s\n" % item[0])
    return feat_data, labels, adj_lists

# Call function to Load PubMed Data
feat_data, labels, adj_lists = load_pubmed()
```

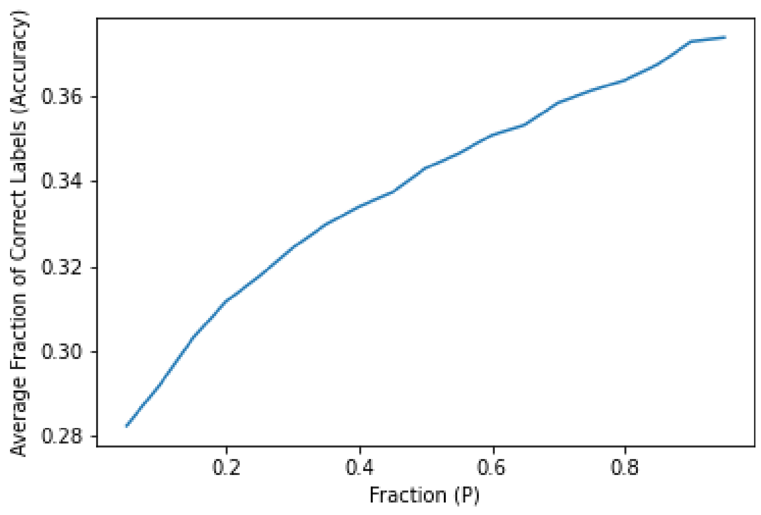
Function to Read the edge file and metadata

```
numberTrials=30
fStep=0.05

def readFile(fileName,mData):
    with (open(mData,'r')) as f:#get metadata
        maxNode=0 #number of nodes in network
        for line in f:
            maxNode+=1
        f.seek(0,0)
        metadata = np.zeros((maxNode))
        counter=0
        for line in f:
            metadata[counter]=line
            counter+=1
    f.close()
    metadata = metadata.astype(int)
    # build an n x n simple network.  Uses edge weights to signify class of neighbo
    # ex.  A(i,j) = 2, A(j,i) = 1--> i and j are linked, j is class 2, i is class 1
    with (open(fileName,'r')) as f:
        lines = f.readlines()
        matrix = np.zeros((maxNode,maxNode))
        for line in lines:
            node,neighbor = map(int,line.split(','))
            node-=1 #start at [0], not [1]
            neighbor-=1
            matrix[node][neighbor]=metadata[neighbor]
            matrix[neighbor][node]=metadata[node] # undirected
    f.close()
  #delete vertices with no neighbor info (different year, data set, etc.)
    temp = np.where(np.sum(matrix,axis=1)==0)
    matrix=np.delete(matrix,temp,axis=0)
    matrix=np.delete(matrix,temp,axis=1)
    metadata=np.delete(metadata,temp)
    return matrix,metadata

```

Function to calculate guilt by association heuristic For a given p between 0 and 1, pick a random fraction p of the nodes for which to observe labels and Predict labels for the remaining nodes using the guilt by association heuristic

```
#Function to calculate guilt by association heuristic
def guiltByAssociation():
    networkFile='PubmedDiabetes.txt'
    metadataFile='metadataPubmedDiabetes.txt'
    associationMatrix,metadata=readFile(networkFile,metadataFile)
    length = len(metadata)
    numberCategories=metadata.max()-metadata.min()+1
    possibleChoices=np.arange(1,numberCategories+1)
    f=.05
    fCounter=0
    #store accuracy results for each f value
    resultsOverF= [0]*((int)((0.99-f)/fStep)+1)
  # store f values used for replot, if necessary
    fValues=[0]*((int)((0.99-f)/fStep)+1)
    while (f <= 1.):
        #results on each iteration
        iterationResults=np.zeros((numberTrials))
        iterationCounter=0
        for iteration in range(numberTrials):
            #make a copy so we can alter it w/out losing oiginal
            trainMatrix=np.copy(associationMatrix)
            randomLabels=np.random.randint(1,high=numberCategories+1,size=length)
            #matrix of 'coin flips' to compare against f for our test set
            randomValues = np.random.random(length)
            hiddenNodes=np.where(randomValues>f)
            #test set length 0 makes no sense...try again
            while (len(hiddenNodes[0])==0):
                randomValues = np.random.random(length)
            #we hide the label on these nodes
                hiddenNodes=np.where(randomValues>f)
            #make predictions for nodes w/ hidden labels
            predictions=np.zeros(len(hiddenNodes[0]))
            #set A(i,j) to 0 when j is hidden (can still see A(j,i) to make predict
            trainMatrix[:,hiddenNodes]=0
            #store 'votes' for each vertex in seperate columns
            findMajority=np.zeros((len(hiddenNodes[0]),numberCategories))
            for index in range(0,numberCategories):
                #neighbor vote total for each vertex/class
                findMajority[:,index]=((trainMatrix==index+1).sum(1))[hiddenNodes]
             #store predictions
            predictions=np.zeros(len(hiddenNodes[0]))
            #if tie (or no votes(tie of 0:0))
            predictions[np.where(findMajority[:,0]==findMajority[:,1])]=randomLabel
        #   print (findMajority,'\n',predictions)
            #use majority to determine node class
            predictions[np.where(predictions==0)]=(np.argmax(findMajority[np.where(
            correct=float(np.sum(predictions==metadata[hiddenNodes]))
        #   print('correct ',correct)
        #   print('len(hiddenNodes[0]) ',len(hiddenNodes[0]))
            iterationResults[iterationCounter]=correct/len(hiddenNodes[0])
        #   print('iterationResults ',iterationResults)
            iterationCounter+=1
        #print('np.average(iterationResults)  ',np.average(iterationResults) )
        #average accuracy of iterations over 1 f value
        resultsOverF[fCounter]=np.average(iterationResults)
        fValues[fCounter]=f
        #print('resultsOverF[fCounter]  ',resultsOverF[fCounter] )
        #print('fValues[fCounter]  ',fValues[fCounter] )
        f = f + fStep
        fCounter = fCounter + 1
    #print(fValues)
    #print(resultsOverF)
    plt.plot(fValues,resultsOverF)
    plt.xlabel('Fraction (P)')
    plt.ylabel('Average Fraction of Correct Labels (Accuracy)')
    plt.savefig('./{}{}Iterations.png'.format(networkFile[:-4],numberTrials))
    plt.show()
    np.savetxt('./{}{}accuracy.txt'.format(networkFile[:-4],numberTrials),resultsOv
    np.savetxt('./{}{}fValues.txt'.format(networkFile[:-4],numberTrials),fValues)
```

```
guiltByAssociation()
```



The above graph shows for a PubMed Diabetes network in which we predicted node labels.

For example, in a PubMed Diabetes network some users classified as 1 (Diabetes Mellitus, Experimental), 2 (Diabetes Mellitus Type 1), or 3 (Diabetes Mellitus Type 2) while others may not.

We used the network structure to predict the labels of nodes without labels. One of the most straightforward approaches to addressing this problem is known as the “guilt by association” heuristic. Here we predict the label of a node based on the most common label of its neighbors where ties are broken randomly. This tends to work well when the network structure is assortative with respect to the given label.

Here we considered the undirected version of the PubMed Diabetes network where nodes are classified as 1 (Diabetes Mellitus, Experimental), 2 (Diabetes Mellitus Type 1), or 3 (Diabetes Mellitus Type 2). For a given p between 0 and 1, then we picked a random fraction p of the nodes for which to observed labels.

Predicted labels for the remaining nodes using the guilt by association heuristic. Repeat this procedure 30 times for values of p ranging from 0.05 to 1 in increments of 0.05 and keep track of the average fraction of correct guesses for each p. We plotted a figure with p on the x-axis and average fraction of correct labels on the y-axis.

.

.

Question 3

Consider the Facebook wall posts (2009) network from question 1 and imagine a piece of fake news spreading across the nodes. In this problem, we want to simulate this process under different conditions and observe the effect of “immunizing” nodes in different ways.

For R0 = 3, simulate the spread 30 times and keep track of the average fraction of infected nodes over time with no immunization.

Repeat the experiment with 10%, 30%, 50%, 70%, and 90% of the nodes immunized following three different strategies: random immunization, immunization of high degree nodes first, and neighbor immunization as described in class.

With the data collected from these experiments, generate three figures, one for each immunization strategy. Each figure will have time on the x-axis and It/n (the fraction of infected nodes) on the y-axis and five separate curves associated with the fraction of nodes immunized in each experiment.

What do you observe? Does one immunization strategy seem more effective than the others? Why or why not?

```
import networkx as nx
import numpy as np
import operator
import time
import matplotlib.pyplot as plt
np.random.seed(4)
```

Parse Facebook Wall Post Data

```
with open('/content/out.facebook-wosn-wall') as f:
    array=[]
    for line in f:
        array.append([int(x) for x in line.split()])
with open('/content/edgelist.tsv',"w")as f:
    for a in array:
        f.write(str(a[0])+' '+str(a[1])+'   '+str(int(a[3]/3600/24/30))+'\n')
```

.

.

.

.

Print Data Staticstics

```
def print_graph_stats(title, g):
    print("Simple stats for: " + title)
    print("# of nodes: " + str(len(g.nodes())))
    print("# of edges: " + str(len(g.edges())))
    print("Is graph connected? " + str(nx.is_connected(g)))
```

```
g = nx.read_adjlist("/content/edgelist.tsv")
g.remove_edges_from(nx.selfloop_edges(g))
print_graph_stats("Facebook Wall Post", g)
```

```
Simple stats for: Facebook Wall Post
# of nodes: 46952
# of edges: 431749
Is graph connected? True
```

Create class for SIR Model

Here

- g- Graph
- beta - Transmission Rate
- mu - Recovery rate
- Tmax = 30 times
- immunization_rate at start is zero

```
class SIR:
    def __init__(self, g, beta, mu, Tmax = 30, index_start = 0.001):
        self.g = g
        self.beta = beta #transmission rate
        self.mu = mu # recovery rate
        self.index_start = index_start
        self.Tmax = Tmax

    def run(self, seed=[], num_steps = 1, sentinels = [], immunization_rate = 0.0,
        # Immunize nodes according to the set immunization rate.
        if len(immunized_nodes) == 0:
            immunized = set(np.random.choice(self.g.nodes(),
                                             size=int(immunization_rate*len(self.g.
                                             replace=False))
        else:
            immunized = immunized_nodes

        # If there is no seed, just choose a random node in the graph.
        if len(seed) == 0:
            number_of_person_depart = int(self.index_start*len(list(set(self.g.node
            seed = np.random.choice(list(set(self.g.nodes()).difference(immunized))
                                    number_of_person_depart, replace=False)
```

```python

        I_set = set(seed)
        S_set = set(self.g.nodes()).difference(I_set).difference(immunized)
        R_set = set()

        number_of_person_infected_sofar = {i:0.0 for i in self.g.nodes()}
        number_of_time_people_stay_infected_sofar = {i:0.0 for i in self.g.nodes()}

        t = 0

        StoI = set(seed)
        ItoR = set()

        sentinels_t = {}
        for sen in sentinels:
            sentinels_t[sen] = 0

        while (len(I_set) > 0 and t < self.Tmax):
            I_set_old = I_set.copy()
            # Let's infect people!
            for i in I_set.copy():
                #print(len(set(self.g.neighbors(i)).intersection(S_set).copy()))
                ntot = len(set(self.g.neighbors(i)).intersection(S_set).copy())
                for s in set(self.g.neighbors(i)).intersection(S_set).copy():
                    if np.random.uniform() < self.beta:
                        S_set.remove(s)
                        I_set.add(s)
                        StoI.add(s)
                        number_of_person_infected_sofar[i]+=1
                        # Record t for sentinels
                        if sentinels_t.get(s) != None:
                            sentinels_t[s] = t

                number_of_time_people_stay_infected_sofar[i] += 1
                #print(t, i, number_of_time_people_stay_infected_sofar[i], number_o

                #print(t, number_of_person_infected_sofar[i] )


                # Will infected person spread fake news?
                if np.random.uniform() < self.mu:
                    I_set.remove(i)
                    R_set.add(i)
                    ItoR.add(i)

            t += 1
            nbre_jour_rester_infecter = int(1/self.mu)
            all_K = []
            for k in I_set_old:
                val2 = max(1, nbre_jour_rester_infecter - number_of_time_people_sta
                val = min(len(list(self.g.neighbors(k))), number_of_person_infected
                #print(t, val, number_of_person_infected_sofar[k], len(list(self.g.
                all_K.append(val)
            #print(t, np.mean(all_K))
            #print(t, np.mean([number_of_person_infected_sofar[k]* for k in I_set_o
                  #np.mean([number_of_time_people_stay_infected_sofar[k] for k in I
            if t % num_steps == 0 or len(I_set) == 0:
                yield({'t': t, 'S':S_set, 'I':I_set, 'R':R_set, 'StoI':StoI, 'ItoR'
                       'reproductive_numbe':  np.mean(all_K)})


```

No Immunization Strategy

For R0 = 3, simulate the spread 30 times and keep track of the average fraction of infected nodes over time with no immunization.

```
def get_temporal_plot(b, m0, Tmax, index_start, graph):
    m = 1./m0
    print()
    print("Constante Value of R : ", 1/m*b)
    print()
    sir = SIR(graph, beta = b, mu = m, Tmax=Tmax, index_start=index_start)
    res = []
    res2 = []
    final_rs = []
    for r in sir.run(num_steps=1):
        n= len(r['S'])+ len(r['I']) +len(r['R'])
        res.append([len(r['I'])/n,1-len(r['R'])/n ])
        res2.append([r['reproductive_numbe'], 1 ])
    value_fin = len(r['R'])*100/len(graph.nodes())

    return res, res2, value_fin
```

.

.

.

.

.

.

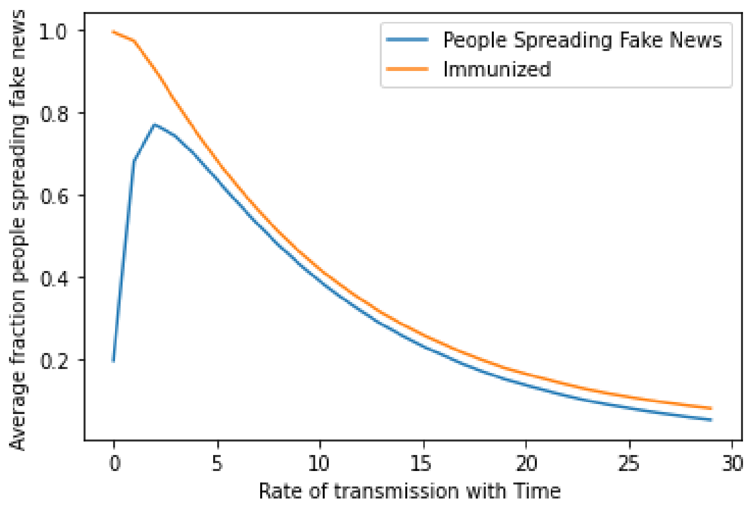
.

.

```
b= 0.3
m=10
Tmax =30
index_start=0.05
res, res2, value_fin = get_temporal_plot(b,m,Tmax,index_start,g)

# Plotting the epidemic curve.
plt.plot(res)
# Plot the results.
plt.title("Epidemic curves for simulations with final "+str(value_fin)[:4] +
          "% of people spreading fake news\n\n",fontsize =12)
plt.legend(['People Spreading Fake News', 'Immunized'])
plt.xlabel("Rate of transmission with Time ")
plt.ylabel("Average fraction people spreading fake news")
plt.show()
```

```
Constante Value of R :  3.0
```

Epidemic curves for simulations with final 91.9% of people spreading fake news



.

.

.

.

.

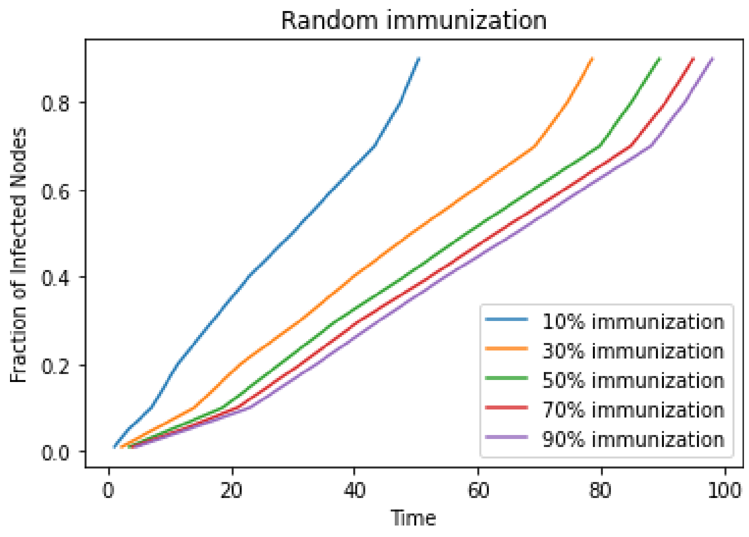
.

Random Immunization

```
def get_random_immunization(graph, b, m, Tmax, index_start, N, immunization_rates):

    start = time.time()
    i_sir = SIR(graph, beta = b, mu = m, Tmax=Tmax, index_start=index_start)
    final_rs = {}
    final_rs0 = {}

    for ir in immunization_rates:
        final_rs[ir] = []
        final_rs0[ir] = []
        for i in range(0,N):
            simulation_steps = [[len(r['S']), len(r['I']), len(r['R']),
                                 r['reproductive_numbe']] for r in i_sir.run(num_st
            final_rs.get(ir).append(simulation_steps[len(simulation_steps)-1][2]*10
            #print(simulation_steps[0][-1])
            average_ro = []
            for k in range(len(simulation_steps)):
                average_ro.append(simulation_steps[k][3])
            final_rs0.get(ir).append(np.mean(average_ro))
    sorted_ir = sorted(final_rs.items(), key=operator.itemgetter(0),reverse=True)
    sorted_ir0 = sorted(final_rs0.items(), key=operator.itemgetter(0),reverse=True)

    irs = []
    oars = []
    oars2 = []

    for ir, values in sorted_ir:
        irs.append(ir)
        oars.append(np.mean(values))

    return sorted(oars),sorted(irs)
```

```

m = 10 # Time of recovery
index_start = 0.05 #Initialization percentage
k =0.3 # Initialization ask population at random
N =30 # Repeat train
immunization_rates = [0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.05, 0.01]
#10 percent
b= 0.1 # Rate of transmission
X1, Y1 = get_random_immunization(g, b, m, Tmax, index_start, N, immunization_rates

#30 percent
b = 0.3
X2, Y2 = get_random_immunization(g, b, m, Tmax, index_start, N, immunization_rates

#50 percent
b = 0.5
X3, Y3 = get_random_immunization(g, b, m, Tmax, index_start, N, immunization_rates

#70 percent
b = 0.7
X4, Y4 = get_random_immunization(g, b, m, Tmax, index_start, N, immunization_rates

#90 percent
b = 0.9
X5, Y5 = get_random_immunization(g, b, m, Tmax, index_start, N, immunization_rates

```

```
plt.title('Random immunization')
plt.xlabel('Time')
plt.ylabel('Fraction of Infected Nodes')
plt.plot(X1, Y1,label =" 10% immunization")
plt.plot(X2, Y2,label =" 30% immunization")
plt.plot(X3, Y3,label =" 50% immunization")
plt.plot(X4, Y4,label =" 70% immunization")
plt.plot(X5, Y5,label =" 90% immunization")
plt.legend(['10% immunization', '30% immunization',
            '50% immunization','70% immunization','90% immunization'])
plt.show()
```



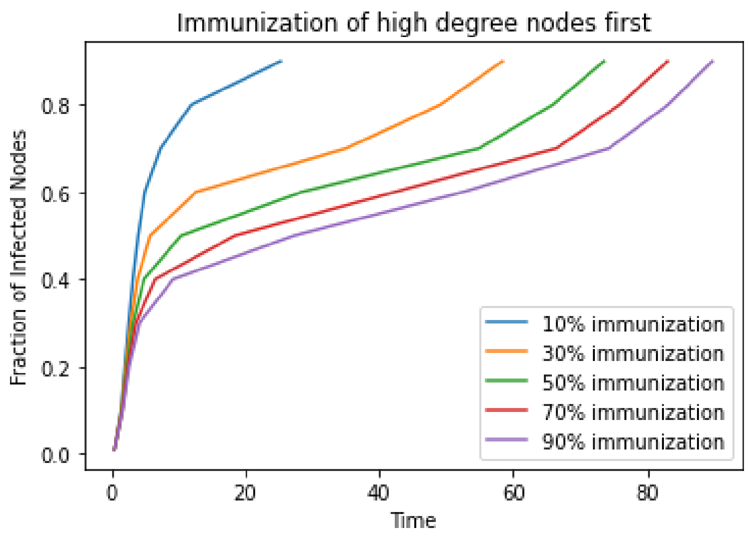
Immunization of high degree nodes first

```

def get_immunization_of_high_degree_nodes(graph, b, m, Tmax, index_start, N, immuni
    start = time.time()

    ti_sir = SIR(graph, beta = b, mu = m, Tmax=Tmax, index_start=index_start)
    nodes_sorted_by_degree = sorted(nx.degree(graph), key=operator.itemgetter(1), r
    final_rs = {}
    final_rs0 = {}
    for ir in immunization_rates:
        final_rs[ir] = []
        final_rs0[ir] = []
        # Immunize the M nodes with highest degree.
        immunized_nodes = []
        M = int(ir*len(nodes_sorted_by_degree))
        for i in range(M):
            immunized_nodes.append(nodes_sorted_by_degree[i][0])
        # Run the simulation 50 times and save the results.
        for i in range(0,N):
            simulation_steps = [[len(r['S']), len(r['I']), len(r['R']), r["reproduc

            final_rs.get(ir).append(simulation_steps[len(simulation_steps)-1][2]*10
            average_ro = []
            for k in range(len(simulation_steps)):
                average_ro.append(simulation_steps[k][3])
            final_rs0.get(ir).append(np.mean(average_ro))
    # Sort results and calculate the mean over the simulations to plot them.
    #print("Job done in: ", time.time() - start)
    sorted_ir = sorted(final_rs.items(), key=operator.itemgetter(0))
    sorted_ir0 = sorted(final_rs0.items(), key=operator.itemgetter(0))
    t_irs = []
    t_oars = []
    t_oars0 = []
    for ir, values in sorted_ir:
        t_irs.append(ir)
        t_oars.append(np.mean(values))
    for ir, values in sorted_ir0:
        t_oars0.append(np.mean(values))

    return sorted(t_oars), sorted(t_irs)

```

```
#10 percent
b= 0.1 # Rate of transmission
m = 10 # Time of recovery
index_start = 0.05 #Initialization percentage
k =0.3 # Initialization ask population at random
N =30 # Repeat train
#immunization_rates = [0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]

immunization_rates = [0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.05, 0.01]
X1, Y1 = get_immunization_of_high_degree_nodes(g, b, m, Tmax, index_start, N, immun

#30 percent
b = 0.3
X2, Y2 = get_immunization_of_high_degree_nodes(g, b, m, Tmax, index_start, N, immun

#50 percent
b = 0.5
X3, Y3 = get_immunization_of_high_degree_nodes(g, b, m, Tmax, index_start, N, immun

#70 percent
b = 0.7
X4, Y4 = get_immunization_of_high_degree_nodes(g, b, m, Tmax, index_start, N, immun

#90 percent
```

```
b = 0.9
X5, Y5 = get_immunization_of_high_degree_nodes(g, b, m, Tmax, index_start, N, immun

```

```
plt.title('Immunization of high degree nodes first')
plt.xlabel('Time')
plt.ylabel('Fraction of Infected Nodes')
plt.plot(X1, Y1,label =" 10% immunization")
plt.plot(X2, Y2,label =" 30% immunization")
plt.plot(X3, Y3,label =" 50% immunization")
plt.plot(X4, Y4,label =" 70% immunization")
plt.plot(X5, Y5,label =" 90% immunization")
plt.legend(['10% immunization', '30% immunization',
            '50% immunization','70% immunization','90% immunization'])
plt.show()
```

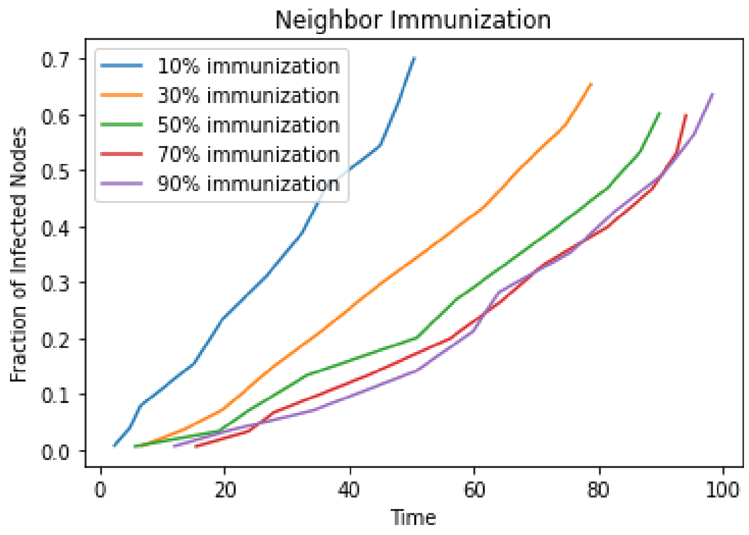


Neighbor Immunization

```
def get_neighbor_immunization(graph, b, m, Tmax, index_start, N, immunization_rates
    ti_sir = SIR(graph, beta = b, mu = m, Tmax=Tmax, index_start=index_start)
    K =0.2 # Init ask population at random
    sentinels = graph.nodes()
    sentinels_results = {}
    known_nodes = set(np.random.choice(graph.nodes(), size=int(K*len(graph.nodes())
    neighbors = set()
    for node in list(known_nodes):
        neighbors.update(set(graph.neighbors(node)))
    final_rs_k = {}
    final_rs0 = {}
    for ir in immunization_rates:
        final_rs_k[ir] = []
        final_rs0[ir] = []
        M = int(ir*len(neighbors))
        immunized_nodes_k = set(np.random.choice(list(neighbors), size=M, replace=F
        for i in range(0,N):
            # Acquaintance immunization
            simulation_steps_k = [[len(r['S']), len(r['I']), len(r['R']),  r["repro

            final_rs_k.get(ir).append(simulation_steps_k[len(simulation_steps_k)-1]
            average_ro = []
            for k in range(len(simulation_steps_k)):
                average_ro.append(simulation_steps_k[k][3])
            final_rs0.get(ir).append(np.mean(average_ro))
    sorted_ir_k = sorted(final_rs_k.items(), key=operator.itemgetter(0))
    sorted_ir0 = sorted(final_rs0.items(), key=operator.itemgetter(0))
    irs2 = []
    oars_k = []
    oars_deg = []
    oars_sim = []
    t_oars01 = []
    for ir, values in sorted_ir_k:
        irs2.append((ir*len(neighbors))/len(graph.nodes()))
        oars_k.append(np.mean(values))
    for ir, values in sorted_ir0:
        t_oars01.append(np.mean(values))

    return sorted(oars_k), sorted(irs2)
```

```
#10 percent
b= 0.1 # Rate of transmission
m = 10 # Time of recovery
index_start = 0.05 #Initialization percentage
k =0.3 # Initialization ask population at random
N =30 # Repeat train
#immunization_rates = [0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]

immunization_rates = [0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.05, 0.01]
X1, Y1 = get_neighbor_immunization(g, b, m, Tmax, index_start, N, immunization_rate

#30 percent
b = 0.3
X2, Y2 = get_neighbor_immunization(g, b, m, Tmax, index_start, N, immunization_rate

#50 percent
b = 0.5
X3, Y3 = get_neighbor_immunization(g, b, m, Tmax, index_start, N, immunization_rate

#70 percent
b = 0.7
X4, Y4 = get_neighbor_immunization(g, b, m, Tmax, index_start, N, immunization_rate

#90 percent
b = 0.9
X5, Y5 = get_neighbor_immunization(g, b, m, Tmax, index_start, N, immunization_rate

```

```
plt.title('Neighbor Immunization')
plt.xlabel('Time')
plt.ylabel('Fraction of Infected Nodes')
plt.plot(X1, Y1,label =" 10% immunization")
plt.plot(X2, Y2,label =" 30% immunization")
plt.plot(X3, Y3,label =" 50% immunization")
plt.plot(X4, Y4,label =" 70% immunization")
plt.plot(X5, Y5,label =" 90% immunization")
plt.legend(['10% immunization', '30% immunization',
            '50% immunization','70% immunization','90% immunization'])
plt.show()
```



Below are my observations from above graphs for different immunization strategies:

1. Random immunization is not an efficient strategy compared to other 2 strategies as irrespective of immunization percentage there's still a good chance that news might hit most of the population.
2. Targeted the Immunization of high degree nodes first (hubs of the networks) can work better when we know the graph which is not true in this case. If we are able to pinpoint most of the high degree nodes from the graph, news spread can be covered with minimal damage.
3. Neighbor Immunization gives good results and this strategy is purely local, requiring minimal information about randomly selected nodes and their immediate environment. In this scenario, despite of the fact that we are unaware of the graph, only 70% population was affected by the news.
4. Compared to all 3 strategies, I feel that High Degree Nodes First works best when we know the graph well but when it's not true, Neighbour immunization might be more efficient compared to the other two.

.

✓ 0s completed at 5:31 PM