Code

(Split into 4 programs – cascade_final.py, network_analysis_final.py, seed_selection_final.py & upgrades)

(New pages are started wherever necessary to keep sections of code on one page wherever possible, for clarity)

cascade_final.py

```
#  ALL NECESSARY IMPORTS ::

#product for generating all permutations for seed selection parameter fine-tuning
from itertools import product
#itemgetter for sorting lists of tuples / dicts by their second element / value
from operator import itemgetter
#math.log for calculating logs of probabilities in WC1 and WC2
from math import log
#copy.deepcopy for deepcopying graphs in seed selection models
from copy import deepcopy
#numpy to manipulate lists, calculate means, etc.
import numpy as np
#
import pandas as pd
#time.time to calculate and compare running time and efficiency
from time import time
#Counter to count frequencies in lists, for averaging edges in seed selection
models
from collections import Counter
#networkx to generate and handle networks from data
import networkx as nx
#csv to extract network data from .csv files
import csv
#winsound to alert me when propagation is complete for long proccessing periods
import winsound
#matplotlib.pyplot for plotting graphs and charts
import matplotlib.pyplot as plt
import matplotlib.cm
#matplotlib.offsetbox.AnchoredText for anchored textboxes on plotted figures
from matplotlib.offsetbox import AnchoredText

#Dataset dictionary
#Title : weighted, directed, filepath to .csv file
datasets = {
    #BitcoinOTC dataset (5881 nodes, 35592 edges)
    #(directed, weighted, signed)
    "BitcoinOTC": (True, True,
        r"D:\Sully\Documents\Computer Science BSc\Year 3\Term 2\Individual Pro-
ject\datasets\soc-sign-bitcoinotc.csv"),
    #Facebook dataset (4039 nodes, 88234 edges)
    #(undirected, unweighted, unsigned)
    "Facebook": (False, False,
        r"D:\Sully\Documents\Computer Science BSc\Year 3\Term 2\Individual Pro-
ject\datasets\facebook.csv")
}

#  CASCADE, ITERATION, PROPAGATION & SUCCESS FUNCTIONS ::
```

```
#
#  Functions to perform cascade & propagation on a given graph
#  with a given seed set and a given number of iterations of a
#  given cascade model.
#
#  Functions included:
#  1. Model-specific success functions
#  2. Propagation function
#  3. Iteration function
#  4. Cascade (ties everything together) function

#Determine propagation success for the various models
#(includes quality factor to differentiate positive/negative influence)
#(includes a switch penalty for nodes switching sign)

#Apply quality factor and switch factor variables
def successVars(sign, switch, qf, sf):
    if not switch:
        sf = 0
    if not sign:
        qf = (1-qf)
    return qf*(1-sf)

#Calculate whether propagation is successful (model-specific)
def success(successModel, sign, switch, timeDelay, g, target, targeting, pp, qf,
sf, a):
    if successModel == 'ICu':
        succ = (pp*successVars(sign, switch, qf, sf)*timeDelay)
    elif successModel == 'IC':
        succ = (pp*successVars(sign, switch, qf, sf)*g[targeting][tar-
get]['trust']*timeDelay)
    elif successModel == 'WC1':
        if a:
            recip = g.nodes[target]['degRecip']
        else:
            recip = (1 / g.in_degree(target))
        succ = (recip*successVars(sign, switch, qf, sf)*timeDelay*g[target-
ing][target]['trust'])
    elif successModel == 'WC2':
        if a:
            relDeg = g[targeting][target]['relDeg']
        else:
            snd = sum([(g.out_degree(neighbour)) for neighbour in g.predeces-
sors(target)])
            relDeg = (g.out_degree(targeting) / snd)
            #relDeg = mmNormalizeSingle(log(g.out_degree(targeting)/snd))
        succ = (relDeg*successVars(sign, switch, qf, sf)*timeDelay*g[target-
ing][target]['trust'])
    return np.random.uniform(0,1) < succ

#Returns probability with only the variables
#(no trust values, degree reciprocals or relational degrees)
def basicProb(weighted=False, *nodes):
    return pp * successVars(True, False)

#One complete turn of propagation from a given set of the newly
#  activated (positive & negative) nodes from the last turn.
#(1. new negative nodes attempt to negatively influence their neighbours)
#(2. new positive nodes attempt to positively influence their neighbours)
#(3. new positive nodes attempt to negatively influence their neighbours)
def propagateTurn(g, pn, pos, nn, neg, trv, td, successMod, pp, qf, sf, a):
```

```
    posCurrent, negCurrent = set(), set()
    for node in nn:
        for neighbour in g.neighbors(node):
            if (node, neighbour) not in trv:
                #Negative influence to neighbours of negative nodes
                if success(successMod, False, (neighbour in pos), td, g, neigh-
bour, node, pp, qf, sf, a):
                    negCurrent.add(neighbour)
                trv.add((node, neighbour))
    for node in pn:
        for neighbour in g.neighbors(node):
            if (node, neighbour) not in trv:
                #Positive influence to neighbours of positive nodes
                if neighbour not in negCurrent and success(successMod, True,
(neighbour in neg), td, g, neighbour, node, pp, qf, sf, a):
                    posCurrent.add(neighbour)
                #Negative influence to neighbours of positive nodes
                elif neighbour not in negCurrent and neighbour not in posCurrent
and success(successMod, False, (neighbour in pos), td, g, neighbour, node, pp,
qf, sf, a):
                    negCurrent.add(neighbour)
                trv.add((node, neighbour))
    return(posCurrent, negCurrent, trv)

#Calculate average positive spread over a given number of iterations
def iterate(g, s, its, successFunc, pp, qf, sf, tf, retNodes, a):
    #If no number of iterations is given, one is calculated based on the
    #  ratio of nodes to edges within the graph, capped at 2000.
    if not its:
        neRatio = (len(g)/(g.size()))
        if neRatio > 0.555:
            its = 2000
        else:
            its = ((neRatio/0.165)**(1/1.75))*1000
    influence = []
    for i in range(its):
        #Randomness seeded per iteration for repeatability & robustness
        np.random.seed(i)
        positive, posNew, negative, negNew, traversed, timeFactor = set(),
set(s), set(), set(), set(), 1
        #while there are newly influenced nodes from last turn...
        while posNew or negNew:
            #new nodes assigned to placeholder variables
            posLastTurn, negLastTurn = posNew, negNew
            #propagation turn is performed, returning positive&negative nodes and
traversed edges
            posNew, negNew, traversed = propagateTurn(g, posNew, positive, neg-
New, negative, traversed, timeFactor, successFunc, pp, qf, sf, a)
            #Positive and negative nodes are recalculated
            positive, negative = (positive.union(posNew, posLastTurn) - negNew),
(negative.union(negNew, negLastTurn) - posNew)
            #Time delay is taken away from the time factor
            if timeFactor < 0:
                timeFactor = 0
            else:
                timeFactor -= tf
        if retNodes:
            #Positive nodes added to list
            for p in positive:
                influence.append(p)
            #Number of nodes added to list
```

```
            infCount.append(len(positive))
        else:
            #Number of positive nodes added to list
            influence.append(len(positive))
    #If nodes are being returned
    if retNodes:
        #Average list of positive nodes are returned
        counts = Counter(influence)
        result = (sorted(counts, key=counts.get, re-
verse=True))[:int(np.mean(infCount))]
    #If nodes aren't being returned
    else:
        #Mean is returned
        result = np.mean(influence)
    return result

#Determine the cascade model and run the iteration function
#  with the appropriate success function
def cascade(g, s, its=0,
            model='IC', assign=1, ret=False,
            pp=0.2, qf=0.6, sf=0.7, tf=0.04):
    #g = graph, s = set of seed nodes, its = num of iterations
    #model = cascade model, #assign model, #return nodes?
    #pp = propagation probability, qf = quality factor
    #sf = switch factor, tf = time factor
    #Model is determined and appropriate success function is assigned
    #print(f'model = {model},  assign = {assign}  its = {its}\npp = {pp}, qf =
{qf}, sf = {sf}, tf = {tf} \n')
    if model != 'IC' and model != 'ICu' and assign:
        assignSelect(g, model, assign)
    success = model
    return iterate(g, s, its, success, pp, qf, sf, tf, ret, assign)

#Propagation models and their names are compiled into a list
propMods = [('IC', "Independent Cascade"),
            ('WC1', "Weighted Cascade 1"),
            ('WC2', "Weighted Cascade 2")]

#  NETWORK GRAPH SETUP ::
#
#  Functions to generate network graphs from various csv files,
#  and assign meaningful attributes to the nodes/edges to save
#  processing time during propagation.
#
#  Real datasets/graphs included:
#  1. soc-BitcoinOTC
#  2. ego-Facebook

#Removes any unconnected components of a given graph
def removeUnconnected(g):
    components = sorted(list(nx.weakly_connected_components(g)), key=len)
    while len(components)>1:
        component = components[0]
        for node in component:
            g.remove_node(node)
        components = components[1:]

#Generates NetworkX graph from given file path:
def generateNetwork(name, weighted, directed, path):
    #graph is initialized and named, dataframe is initialized
    newG = nx.DiGraph(Graph = name)
```

```python
    data = pd.DataFrame()
    #pandas dataframe is read from .csv file,
    #  with weight if weighted, without if not
    if weighted:
        data = pd.read_csv(path, header=None, usecols=[0,1,2],
                           names=['Node 1', 'Node 2', 'Weight'])
    else:
        data = pd.read_csv(path, header=None, usecols=[0,1],
                           names=['Node 1', 'Node 2'])
        data['Weight'] = 1
    #offset is calculated from minimum nodes
    offset = min(data[['Node 1', 'Node 2']].min())
    #each row of dataframe is added to graph as an edge
    for row in data.itertuples(False, None):
        #trust=weight, & distance=(1-trust)
        trustval = row[2]
        newG.add_edge(row[1]-offset, row[0]-offset,
                      trust=trustval, distance=(1-trustval))
        #if graph is undirected, edges are added again in reverse
        if not directed:
            newG.add_edge(row[0]-offset, row[1]-offset,
                          trust=trustval, distance=(1-trustval))
    #unconnected components are removed
    if directed:
        removeUnconnected(newG)
    return newG

#Generate graphs and compile into dictionaries:

#Dictionaries for groups of graphs are intialized
graphs, mockGraphs, rndmGraphs, diGraphs, allGraphs = {}, {}, {}, {}, {}

#Generate graphs from real datasets using the datasets dictionary
for g in datasets:
    realGraph = generateNetwork((g + " Network"),
                                datasets[g][0], datasets[g][1],
                                datasets[g][2])
    graphs[g], diGraphs[g], allGraphs[g] = realGraph, realGraph, realGraph


#Generate various mock graphs for testing and debugging:

#Custom, small directed, unweighted graph
mockG, name = nx.DiGraph(), "mock1: Custom, small"
testedges = [(1,2), (2,4), (2,5), (2,6), (3,5), (4,5), (5,9), (5,10), (6,8),
             (7,8), (8,9)]
mockG.add_edges_from(testedges)
nx.set_edge_attributes(mockG, 1, 'trust')
mockGraphs[name], diGraphs[name], allGraphs[name] = mockG, mockG, mockG

#Medium-sized path graph
#(each node only has edges to the node before and/or after it)
mockG, name = nx.path_graph(100), "mock2: Path graph, 100 nodes"
nx.set_edge_attributes(mockG, 1, 'trust')
mockGraphs[name], allGraphs[name] = mockG, mockG

#Medium-sized, randomly generated directed, unweighted graph
mockG, name = nx.DiGraph(), "mock3: Random, trustvals=1"
for i in range(50):
    for j in range(10):
        targ = np.random.randint(-40,50)
```

```
        if targ > -1:
            mockG.add_edge(i, targ)
mockGraphs[name], rndmGraphs[name], diGraphs[name], allGraphs[name] = mockG,
mockG, mockG, mockG

#Medium-sized, randomly generated directed, randomly-weighted graph
mockG, name = nx.DiGraph(), "mock4: Random, trustvals=random"
for i in range(50):
    for j in range(10):
        targ = np.random.randint(-40,50)
        if targ > -1:
            tru = np.random.uniform(0,1)
            mockG.add_edge(i, targ, trust=tru)
mockGraphs[name], rndmGraphs[name], diGraphs[name], allGraphs[name] = mockG,
mockG, mockG, mockG

#Functional testing for new graphing method
"""
for gl in [realGraphs, mockGraphs]:
    for g in gl:
        print(g + ": " + str(gl[g].size()))
        print(g + ": " + str(len(gl[g])) + "\n")
#"""
print("")

#Normalize (Min-Max) every value in a given dictionary (method 2 & 3)
def mmNormalizeDict(dic, elMax, elMin):
    #for key, value in dic.items():
    #    dic[key] = ((value - elMin) / (elMax - elMin))
    #printResults("Assigned", dic.values())
    #print("Assigned normalization:\nMax = " + str(elMax) + "\nMin = "
    #       + str(elMin) + "\nMean = "
    #       + str(np.mean(list(dic.values()))))
    #return dic
    return {key: ((val - elMin)/(elMax - elMin)) for key,val in dic.items()}

#Min-Max Normalize a given list (method 1)
def mmNormalizeLis(lis):
    elMax, elMin = max(lis), min(lis)
    return list(map(lambda x : ((x - elMin)/(elMax - elMin)), lis))

#Assign method 1 - manual

#Calculate all relational degrees for a graph
def allRelDegs1(g):
    allRds = []
    for target in g:
        if not g.in_degree(target):
            continue
        snd = 0
        for neighbour in g.predecessors(target):
            snd += g.out_degree(neighbour)
        for targeting in g.predecessors(target):
            allRds.append(log(g.out_degree(targeting) / snd))
    return allRds


"""
relDegsTest1 = allRelDegs(graphs['Facebook'])

elMax, elMin = max(relDegsTest1), min(relDegsTest1)
```

```
relDegsTest2 = mmNormalizeLis(relDegsTest1)

#Min-max normalize a single value, given the needed max & min
def mmNormalizeSingle(val, elMax, elMin):
    return ((val - elMin)/(elMax - elMin))
"""
print("")

"""

#AllRelDegs for dictionary, to check functionality after finding
#  errors -
#Normalizes return dict and compares it to dict from assign method 3.
def allRelDegs2(g):
    allRdsDict = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = 0
        for neighbour in g.predecessors(target):
            snd += g.out_degree(neighbour)
        for targeting in g.predecessors(target):
            rdval = log(g[targeting][target]['trust']*(g.out_degree(targeting) /
snd))
            allRdsDict[(targeting, target)] = log((g.out_degree(targeting) /
snd))
    return allRdsDict

#relDegsTest1 = allRelDegs(graphs['Facebook'])
relDegsTest1, relDegsTestDict = allRelDegs(graphs['Facebook'])
elMax, elMin = max(relDegsTest1), min(relDegsTest1)
relDegsTest2 = mmNormalizeLis(relDegsTest1)
#relDegsTest3 = mmNormalizeDict(relDegsTestDict,
 #                              max(relDegsTestDict.values()),
 #                              min(relDegsTestDict.values()))



#Printing averages, maximums & minimums to find the
#  error from the initial erroneous results
printResults("Test list: ", relDegsTest1)
printResults("Test normalized list: ", relDegsTest2)
printResults("Test normalized dict v1: ", list(relDegsTestDict.values()))
printResults("Test normalized dict v2: ", list(relDegsTest3)
"""
print("")

#Methods that assign probabilities for WC1 & WC2 to nodes or edges

#Calculate manipulated degree-reciprocals for all nodes in a graph, and
#  assign them as node attributes for the Weighted Cascade 1 model

#Selects and runs appropriate assigning method
def assignSelect(g, propMod, assignMod):
    if assignMod:
        assignMods[propMod][assignMod](g)

#Log-scaling method - default if not specified
def assignRecips1(g):
    drs = {}
    for target in g:
```

```
        if not g.in_degree(target):
            continue
        drs[target] = log(1 / g.in_degree(target))
    elMax = drs[max(drs, key=drs.get)]
    elMin = drs[min(drs, key=drs.get)]
    drs = mmNormalizeDict(drs, elMax, elMin)
    nx.set_node_attributes(g, drs, "degRecip")

#Square-rooting method
def assignRecips2(g):
    drs = {}
    for target in g:
        if not g.in_degree(target):
            continue
        drs[target] = ((1 / g.in_degree(target)) ** (1/2))
    nx.set_node_attributes(g, drs, "degRecip")

#Cube-rooting method
def assignRecips3(g):
    drs = {}
    for target in g:
        if not g.in_degree(target):
            continue
        drs[target] = ((1 / g.in_degree(target)) ** (1/3))
    nx.set_node_attributes(g, drs, "degRecip")

#Calculate manipulated relational-degrees for all edges in a graph, and
#  assign them as edge attributes for the Weighted Cascade 2 model

#Log-scaling method
def assignRelDegs1(g):
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = 0
        for targeting in g.predecessors(target):
            snd += g.out_degree(targeting)
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = log(g.out_degree(targeting) / snd)
    #elMax = rds[max(rds, key=rds.get)]
    #elMin = rds[min(rds, key=rds.get)]
    rds = mmNormalizeDict(rds, max(rds.values()), min(rds.values()))
    nx.set_edge_attributes(g, rds, "relDeg")

#Square-rooting method
def assignRelDegs2(g):
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = 0
        for targeting in g.predecessors(target):
            snd += g.out_degree(targeting)
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = (((g.out_degree(targeting)) / snd) **
(1/2))
    nx.set_edge_attributes(g, rds, "relDeg")

#Cube-rooting method
def assignRelDegs3(g):
```

```
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = sum([(g.out_degree(neighbour))
                   for neighbour in g.predecessors(target)])
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = (((g.out_degree(targeting)) / snd) **
(1/3))
    nx.set_edge_attributes(g, rds, "relDeg")

#Assign method dictionary for selection depending on parameters
assignMods = {'WC1': {1: assignRecips1, 2: assignRecips2, 3: assignRecips3},
              'WC2': {1: assignRelDegs1, 2: assignRelDegs2, 3:assignRelDegs3}}

#Assign method 1 - manual

#Calculate all relational degrees
def allRelDegs(g):
    #allRds = []
    allRds, allRdsDict = [], {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = sum([g.out_degree(neighbour)
                   for neighbour in g.predecessors(target)])
        #print("\nSND = " + str(snd) + "\n")
        for targeting in g.predecessors(target):
            rdval = log(g[targeting][target]['trust']*(g.out_degree(targeting) /
snd))
            allRds.append(rdval)
            allRdsDict[(targeting, target)] = log((g.out_degree(targeting) /
snd))
    return allRds, allRdsDict



#Functionality & quality testing of assignment functions
"""
for assignTest in [0,1]:
    print('assign method: ' + str(assignTest))
    measureTime1(cascade, graphs['Facebook'], [1], 15, 'WC2', assignTest,
                 0.5, 0.7, 0.7, 0.08)
    print("")
#"""
print("")

#Results: (S=[1], 75its, pp=0.5, qf=0.7, sf=0.7, tf=0.04)
#Manual log-scaling----------1.427 spread, 0.368secs
#Pre-assigned log-scaling----888.773 spread, 77.491secs
#Initially not equal --> typo in allRelDegs (return line indented so no loop)
#Due to the typo these results are erroneous

#Lowered iterations due to it taking so long to process the manual method
#Results: (S=[1], 75its, pp=0.5, qf=0.7, sf=0.7, tf=0.04)
#Manual log-scaling---------1188.533 spread, 328.085secs
#Pre-assigned log-scaling---1188.533 spread, 13.405secs


#  MISCALLANEOUS & UTILITY METHODS/FUNCTIONS ::
#
```

```python
#  Methods & functions for various purposes, that are either required
#  in other sections of the program or optimize their performance.
#
#  Methods/Functions included:
#  1. Measure time/speed of a given function
#  2. Min-max normalize a given dictionary, scaling between 0 and 1.
#  3. Draw a histogram from a given dictionary of probabilities

#Time measuring functions

#Measure the time taken to perform a given function
def measureTime1(func, *pars):
    startT = time()
    print(func(*pars))
    print(str(time() - startT) + "\n")

#Same as measureTime, but also returns the result from the given function
def measureTimeRet(func, *pars):
    startT = time()
    return func(*pars), round((time() - startT), 3)

#Same as measureTime1, but prints a given message initially
def measureTime2(msg, func, *pars):
    print(msg + ":")
    startT = time()
    print(func(*pars))
    print(str(time() - startT) + " secs\n")

#Given a seed selection model, and can also take parameters for that, selects a
seed set and
#  measures the time taken to do so. Checks this seed set hasn't already been
propagated to,
#  and if it hasn't performs a given propagation model on it and measures the
time it took.
#Also returns the seed set, so that it can be added to the set of propagated-to
seed sets.
def measureTime3(seedSel, propMod, oldSeeds, g, qty, its, *params):
    print(seedSel[1] + " Seed Selection:")
    startT = time()
    S = set(seedSel[0](g, qty, *params))
    print(str(S) + "\n" + str(time() - startT) + "\n")
    found = False
    for oldSeedSet in oldSeeds:
        if S in oldSeedSet:
            found = True
            print(seedSel[1] + "has the same seed set as " + oldSeedSet[1] +
                  ". No need for propagation, check previous results.\n")
    if found:
        return S
    print(propMod[1] + ": (" + str(its) + " iterations)")
    startT = time()
    print(str(cascade(g, S, its, propMod[0])))
    print(str(time() - startT) + "\n\n")
    return S

#Same as measureTime3 without the old seed checking
def measureTime4(seedSel, propMod, oldSeeds, g, qty, its, *params):
    print(seedSel[1] + " Seed Selection:")
    startT = time()
    S = set(seedSel[0](g, qty, *params))
    print(str(S) + "\n" + str(time() - startT) + "\n")
```

```python
    print(propMod[1] + ": (" + str(its) + " iterations)")
    startT = time()
    print(str(cascade(g, S, its, propMod[0])))
    print(str(time() - startT) + "\n\n")

#switch factor testing against quality factor
"""
for q in [0.2, 0.8]:
    for sw in [0, 0.2, 0.4, 0.6, 0.8, 1]:
        print("QualityFactor = " + str(q) + "\nSwitch factor = " + str(sw))
        print(cascade(graphs['BitcoinOTC'], [1], 25, qf=q, sf=sw))
        print("")
#"""

#  TIME-TESTING METHODS/FUNCTIONS ::
#
#  Methods & functions for testing processing times for cascade functions,
#  varying the number of iterations, variables used and the variables' values.


#Random seed selection model for functionality
#  testing & variable comparison testing
def randomSeeds(g, k):
    return set(np.random.choice(g, k, replace=False))

#Compare positive influence spreads for a given list of graphs with
#  a range of different parameter values, and plot a line graph to show.
def compareVars(g, gs, S, its, model, vss):
    values = []
    order = 0
    for c, vs in enumerate(vss):
        if vs[0] == 'pp':
            startTime = time()
            values.append([cascade(gs[g], S, its, model, pp=v) for v in vs[2]])
            print("Variable: " + vs[0] + "\n" + str(values[order]) + "\n" +
                  str(round((time() - startTime), 5)) + " secs\n")
            vs[1] = order
            order += 1
        elif vs[0] == 'qf':
            startTime = time()
            values.append([cascade(gs[g], S, its, model, qf=v) for v in vs[2]])
            print("Variable: " + vs[0] + "\n" + str(values[order]) + "\n" +
                  str(round((time() - startTime), 5)) + " secs\n")
            vs[1] = order
            order += 1
        elif vs[0] == 'sf':
            startTime = time()
            values.append([cascade(gs[g], S, its, model, sf=v) for v in vs[2]])
            print("Variable: " + vs[0] + "\n" + str(values[order]) + "\n" +
                  str(round((time() - startTime), 5)) + " secs\n")
            vs[1] = order
            order += 1
        else:
            startTime = time()
            values.append([cascade(gs[g], S, its, model, tf=v) for v in vs[2]])
            print("Variable: " + vs[0] + "\n" + str(values[order]) + "\n" +
                  str(round((time() - startTime), 5)) + " secs\n")
            vs[1] = order
            order += 1
    figs, axs = plt.subplots(figsize=(10,6))
    axs.set_xlabel("Variable Probabilities")
```

```
    axs.set_ylabel("Spread")
    axs.set_title(g + "\n" +
                  "Probabilitiy Comparison: " + model)
    for c, vs in enumerate(vss):
        axs.plot(vs[2], values[vs[1]], label=vs[0], marker='o', markersize=4)
    axs.legend()
    plt.show()

#Compare positive influence spreads for a given list of graphs with
#  a range of different models, and plot a line graph to show.
def compareVars2(g, gs, S, its, models, vss):
    values = []
    order = 0
    for model in models:
        for c, vs in enumerate(vss):
            if vs[0] == 'pp':
                startTime = time()
                values.append([cascade(gs[g], S, its, model, pp=v) for v in
vs[2]])
                print("Variable: " + vs[0] + "\n" + str(values[order]) + "\n" +
                      str(round((time() - startTime), 5)) + " secs\n")
                vs[1] = order
                order += 1
            elif vs[0] == 'qf':
                startTime = time()
                values.append([cascade(gs[g], S, its, model, qf=v) for v in
vs[2]])
                print("Variable: " + vs[0] + "\n" + str(values[order]) + "\n" +
                      str(round((time() - startTime), 5)) + " secs\n")
                vs[1] = order
                order += 1
            elif vs[0] == 'sf':
                startTime = time()
                values.append([cascade(gs[g], S, its, model, sf=v) for v in
vs[2]])
                print("Variable: " + vs[0] + "\n" + str(values[order]) + "\n" +
                      str(round((time() - startTime), 5)) + " secs\n")
                vs[1] = order
                order += 1
            else:
                startTime = time()
                values.append([cascade(gs[g], S, its, model, tf=v) for v in
vs[2]])
                print("Variable: " + vs[0] + "\n" + str(values[order]) + "\n" +
                      str(round((time() - startTime), 5)) + " secs\n")
                vs[1] = order
                order += 1
    figs, axs = plt.subplots(figsize=(10,6))
    axs.set_xlabel("Variable Probabilities")
    axs.set_ylabel("Spread")
    axs.set_title(g + "\n" +
                  "Probabilitiy Comparison: " + model)
    for c, vs in enumerate(values):
        axs.plot(vss[0][2], vs, label=(models[c] + " " + vss[0][0]), marker='o',
markersize=4)
    axs.legend()
    plt.show()

#pp experiment for IC
qty, its, model, gs = 5, 200, 'IC', [graphs]
vss = [['pp', 0, [x*0.1 for x in range(11)]]]
```

```
startTime = time()
s = randomSeeds(graphs['BitcoinOTC'], qty)
print(g + "\nseed set: " + str(s) + "\n" +
      str(round((time() - startTime), 5)) + " secs\n")
compareVars('BitcoinOTC', graphs, s, its, model, vss)
print(str(round((startTime - time()), 4)) + " secs")

#306 secs

#pp experiment for all models
qty, its, models, gs = 5, 50, ['IC','WC1','WC2'], [graphs]
vss = [['pp', 0, [x*0.1 for x in range(11)]]]
startTime = time()
s = randomSeeds(graphs['BitcoinOTC'], qty)
print(g + "\nseed set: " + str(s) + "\n" +
      str(round((time() - startTime), 5)) + " secs\n")
compareVars2('BitcoinOTC', graphs, s, its, models, vss)
print(str(round((startTime - time()), 4)) + " secs")

#306 secs

#qf experiment for IC
qty, its, model, gs = 5, 200, 'IC', [graphs]
vss = [['qf', 0, [x*0.1 for x in range(11)]]]
startTime = time()
s = randomSeeds(graphs['BitcoinOTC'], qty)
print(g + "\nseed set: " + str(s) + "\n" +
      str(round((time() - startTime), 5)) + " secs\n")
compareVars('BitcoinOTC', graphs, s, its, model, vss)
print(str(round((startTime - time()), 4)) + " secs")

#300 secs

#qf experiment for all models
qty, its, model, gs = 5, 50, ['IC','WC1','WC2'], [graphs]
vss = [['qf', 0, [x*0.1 for x in range(11)]]]
startTime = time()
s = randomSeeds(graphs['BitcoinOTC'], qty)
print(g + "\nseed set: " + str(s) + "\n" +
      str(round((time() - startTime), 5)) + " secs\n")
compareVars2('BitcoinOTC', graphs, s, its, model, vss)
print(str(round((startTime - time()), 4)) + " secs")

#300 secs

#switch factor experiment for IC
qty, its, model, gs = 5, 200, 'IC', [graphs]
vss = [['sf', 0, [x*0.1 for x in range(11)]]]

startTime = time()
s = randomSeeds(graphs['BitcoinOTC'], qty)
print(g + "\nseed set: " + str(s) + "\n" +
      str(round((time() - startTime), 5)) + " secs\n")
compareVars('BitcoinOTC', graphs, s, its, model, vss)
print(str(round((startTime - time()), 4)) + " secs")

#290 secs

#switch factor experiment for all models
qty, its, model, gs = 5, 50, ['IC','WC1','WC2'], [graphs]
vss = [['sf', 0, [x*0.1 for x in range(11)]]]
```

```
startTime = time()
s = randomSeeds(graphs['BitcoinOTC'], qty)
print(g + "\nseed set: " + str(s) + "\n" +
      str(round((time() - startTime), 5)) + " secs\n")
compareVars2('BitcoinOTC', graphs, s, its, model, vss)
print(str(round((startTime - time()), 4)) + " secs")

#290 secs

#time factor experiment for IC
qty, its, model, gs = 5, 200, 'IC', [graphs]
vss = [['tf', 0, [x*0.02 for x in range(5)] + [y*0.1 for y in range(1,11)]]]
startTime = time()
s = randomSeeds(graphs['BitcoinOTC'], qty)
print(g + "\nseed set: " + str(s) + "\n" +
      str(round((time() - startTime), 5)) + " secs\n")
compareVars('BitcoinOTC', graphs, s, its, model, vss)
print(str(round((startTime - time()), 4)) + " secs")

#280 secs

#time factor experiment for all models
qty, its, model, gs = 5, 50, ['IC','WC1','WC2'], [graphs]
vss = [['tf', 0, [x*0.02 for x in range(5)] + [y*0.1 for y in range(1,11)]]]
startTime = time()
s = randomSeeds(graphs['BitcoinOTC'], qty)
print(g + "\nseed set: " + str(s) + "\n" +
      str(round((time() - startTime), 5)) + " secs\n")
compareVars2('BitcoinOTC', graphs, s, its, model, vss)
print(str(round((startTime - time()), 4)) + " secs")

#280 secs

#all parameters experiment
qty, its, model = 5, 200, 'IC'
vss = [['qf', 0, [x*0.1 for x in range(11)]],
       ['sf', 0, [x*0.1 for x in range(11)]],
       ['pp', 0, [x*0.1 for x in range(11)]],
       ['tf', 0, [x*0.02 for x in range(5)] + [y*0.1 for y in range(1,11)]]]
startTime = time()
s = randomSeeds(graphs['BitcoinOTC'], qty)
print('BitcoinOTC' + "\nseed set: " + str(s) + "\n" +
      str(round((time() - startTime), 5)) + " secs\n")
compareVars('BitcoinOTC', graphs, s, its, model, vss)
print(str(round((startTime - time()), 4)) + " secs")


#Basic propagation probability & quality factor test
g, s, its = graphs['BitcoinOTC'], {1}, 10
def TestRun(g, s, its, p=0.2, q=0.6):
    t = time()
    print("Test IC " + str(cascade(g, s, its, pp=p, qf=q))) #Independent Cascade
is default
    print(str(round((time()-t),5)) + " secs\n")
    t = time()
    print("Test WC1 " + str(cascade(g, s, its, model='WC1', pp=p, qf=q)))
#Weighted Cascade 1
    print(str(round((time()-t),5)) + " secs\n")
    t = time()
```

```
    print("Test WC2 " + str(cascade(g, s, its, model='WC2', pp=p, qf=q)))
#Weighted Cascade 2
    print(str(round((time()-t),5)) + " secs\n")

#TestRun(g, s, its)

for vals in [(0.2, 0.2), (0.2, 0.8), (0.8, 0.2), (0.8, 0.8)]:
    print("PP = " + str(vals[0]) + "\nQF = " + str(vals[1]))
    TestRun(g, s, its, vals[0], vals[1])
```

network_analysis_final.py

```
#  ALL NECESSARY IMPORTS ::

#product for generating all permutations for seed selection parameter fine-tuning
from itertools import product
#itemgetter for sorting lists of tuples / dicts by their second element / value
from operator import itemgetter
#math.log for calculating logs of probabilities in WC1 and WC2
from math import log
#copy.deepcopy for deepcopying graphs in seed selection models
from copy import deepcopy
#numpy to manipulate lists, calculate means, etc.
import numpy as np
#
import pandas as pd
#time.time to calculate and compare running time and efficiency
from time import time
#Counter to count frequencies in lists, for averaging edges in seed selection
models
from collections import Counter
#networkx to generate and handle networks from data
import networkx as nx
#csv to extract network data from .csv files
import csv
#winsound to alert me when propagation is complete for long proccessing periods
import winsound
#matplotlib.pyplot for plotting graphs and charts
import matplotlib.pyplot as plt
#matplotlib.offsetbox.AnchoredText for anchored textboxes on plotted figures
from matplotlib.offsetbox import AnchoredText

#Dataset dictionary
#Title : weighted, directed, filepath to .csv file
datasets = {
    #BitcoinOTC dataset (5881 nodes, 35592 edges)
    #(directed, weighted, signed)
    "BitcoinOTC": (True, True,
         r"D:\Sully\Documents\Computer Science BSc\Year 3\Term 2\Individual Pro-
ject\datasets\soc-sign-bitcoinotc.csv"),
    #Facebook dataset (4039 nodes, 88234 edges)
    #(undirected, unweighted, unsigned)
    "Facebook": (False, False,
         r"D:\Sully\Documents\Computer Science BSc\Year 3\Term 2\Individual Pro-
ject\datasets\facebook.csv")
}

#  NETWORK GRAPH SETUP ::
#
#  Functions to generate network graphs from various csv files,
#  and assign meaningful attributes to the nodes/edges to save
#  processing time during propagation.
#
#  Datasets/graphs included:
#  1. soc-BitcoinOTC
#  2. ego-Facebook

#Removes any unconnected components of a given graph
def removeUnconnected(g):
    components = sorted(list(nx.weakly_connected_components(g)), key=len)
    while len(components)>1:
```

```
        component = components[0]
        for node in component:
            g.remove_node(node)
        components = components[1:]

#Generates NetworkX graph from given file path:
def generateNetwork(name, weighted, directed, path):
    #graph is initialized and named, dataframe is initialized
    newG = nx.DiGraph(Graph = name)
    data = pd.DataFrame()
    #pandas dataframe is read from .csv file,
    #  with weight if weighted, without if not
    if weighted:
        data = pd.read_csv(path, header=None, usecols=[0,1,2],
                           names=['Node 1', 'Node 2', 'Weight'])
        wMax, wMin = data[['Weight']].max().item(), data[['Weight']].min().item()
    else:
        data = pd.read_csv(path, header=None, usecols=[0,1],
                           names=['Node 1', 'Node 2'])
    #offset is calculated from minimum nodes
    nodeOffset = min(data[['Node 1', 'Node 2']].min())
    #each row of dataframe is added to graph as an edge
    for row in data.itertuples(False, None):
        #trust=weight, & distance=(1-trust)
        if weighted:
            trustval = ((row[2]-wMin)/(wMax-wMin))
        else:
            trustval = 1
        newG.add_edge(row[1]-nodeOffset, row[0]-nodeOffset,
                      trust=trustval, distance=(1-trustval))
        #if graph is undirected, edges are added again in reverse
        if not directed:
            newG.add_edge(row[0]-nodeOffset, row[1]-nodeOffset,
                          trust=trustval, distance=(1-trustval))
    #unconnected components are removed
    if directed:
        removeUnconnected(newG)
    return newG

#Generate graphs and compile into dictionaries:

#Dictionaries for groups of graphs are intialized
graphs, mockGraphs, rndmGraphs, diGraphs, allGraphs = {}, {}, {}, {}, {}

#Generate graphs from real datasets using the datasets dictionary
for g in datasets:
    realGraph = generateNetwork((g + " Network"),
                                datasets[g][0], datasets[g][1],
                                datasets[g][2])
    graphs[g], diGraphs[g], allGraphs[g] = realGraph, realGraph, realGraph


#Generate various mock graphs for testing and debugging:

#Custom, small directed, unweighted graph
mockG, name = nx.DiGraph(), "mock1"
testedges = [(1,2), (2,4), (2,5), (2,6), (3,5), (4,5), (5,9), (5,10), (6,8),
            (7,8), (8,9)]
mockG.add_edges_from(testedges)
nx.set_edge_attributes(mockG, 1, 'trust')
mockGraphs[name], diGraphs[name] = mockG, mockG
```

```
#Medium-sized, randomly generated directed, unweighted graph
mockG, name = nx.DiGraph(), "mock2"
for i in range(50):
    for j in range(10):
        targ = np.random.randint(-40,50)
        if targ > -1:
            mockG.add_edge(i, targ, trust=1)
mockGraphs[name], rndmGraphs[name], diGraphs[name] = mockG, mockG, mockG

#Medium-sized, randomly generated directed, randomly-weighted graph
mockG, name = nx.DiGraph(), "mock3"
for i in range(50):
    for j in range(10):
        targ = np.random.randint(-40,50)
        if targ > -1:
            tru = np.random.uniform(0,1)
            mockG.add_edge(i, targ, trust=tru)
mockGraphs[name], rndmGraphs[name], diGraphs[name] = mockG, mockG, mockG

#Functional testing for new graphing method
"""
for gl in [realGraphs, mockGraphs]:
    for g in gl:
        print(g + ": " + str(gl[g].size()))
        print(g + ": " + str(len(gl[g])) + "\n")
#"""
print("")

#Calculate the logs of the degree-reciprocals for all nodes in a graph,
#  and assign them as node attributes to that graph (WC1)
def assignRecips(g):
    drs = {}
    for target in g:
        if not g.in_degree(target):
            continue
        drs[target] = log(1 / g.in_degree(target))
    elMax = drs[max(drs, key=drs.get)]
    elMin = drs[min(drs, key=drs.get)]
    drs = mmNormalizeDict(drs, elMax, elMin)
    nx.set_node_attributes(g, drs, "degRecip")

def assignRecips2(g):
    drs = {}
    for target in g:
        if not g.in_degree(target):
            continue
        drs[target] = ((1 / g.in_degree(target)) ** (1/2))
    nx.set_node_attributes(g, drs, "degRecip")

def assignRecips3(g):
    drs = {}
    for target in g:
        if not g.in_degree(target):
            continue
        drs[target] = ((1 / g.in_degree(target)) ** (1/3))
    nx.set_node_attributes(g, drs, "degRecip")

#Calculate the logs of the relational-degrees for all edges in a graph,
#  and assign them as edge attributes to that graph (WC2)
def assignRelDegs(g):
```

```
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = 0
        for targeting in g.predecessors(target):
            snd += g.out_degree(targeting)
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = log(g.out_degree(targeting) / snd)
    elMax = rds[max(rds, key=rds.get)]
    elMin = rds[min(rds, key=rds.get)]
    rds = mmNormalizeDict(rds, elMax, elMin)
    nx.set_edge_attributes(g, rds, "relDeg")

def assignRelDegs2(g):
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = 0
        for targeting in g.predecessors(target):
            snd += g.out_degree(targeting)
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = ((g.out_degree(targeting) / snd) ** (1/2))
    nx.set_edge_attributes(g, rds, "relDeg")

def assignRelDegs3(g):
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = 0
        for targeting in g.predecessors(target):
            snd += g.out_degree(targeting)
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = ((g.out_degree(targeting) / snd) ** (1/3))
    nx.set_edge_attributes(g, rds, "relDeg")

#  MISCALLANEOUS & UTILITY METHODS/FUNCTIONS ::
#
#  Functions for printing some/all of the maximum, minimum, mean,
#  median and range of a given list. For comparison of normalization
#  techniques.

#Print the mean, median, maximum and minimum of a given list
def printResults(lis, msg, space):
    print(msg)
    print("Mean = " + str(round(np.mean(lis), 5)))
    print("Median = " + str(round(np.median(lis), 5)))
    print("Max = " + str(round(max(lis), 5)))
    print("Min = " + str(round(min(lis), 5)))
    print("Range = " + str(round((max(lis)-min(lis)), 5)))
    if space:
        print("")

#Print the maximum, minimum and averages of a given list
#(more concisely; for larger comparisons)
def printResults1(lis, msg, space):
    print(msg)
    print("Mean = " + str(round(np.mean(lis), 5))
          + ". Median = " + str(round(np.median(lis), 5))
```

```
          + "\nMax = " + str(round(max(lis), 5)) + ". Min = "
          + str(round(min(lis), 5)) + ". Range = "
          + str(round((max(lis)-min(lis)), 5))
          + "\n---------------------------------")
    if space:
        print("")

#Print the mean and median of a given list
#(more concisely; for more specific, larger comparisons)
def printResults2(lis, msg, space):
    print(msg)
    print("Mean = " + str(round(np.mean(lis), 5))
          + ". Median = " + str(round(np.median(lis), 5)))
    if space:
        print("")

#  NETWORK-WIDE ANALYSIS ::
#
#  Analyses:
#  1. Strongly/Weakly Connected components - these are subgraphs or sections of
the network where:
#     -every node can reach every other node (strongly connected)
#     -every node is reachable from some other node (weakly connected)
#  2. Mutuality percentage (fraction of edges that are bidirectional)
#  3. Density percentage (actual edges / possible edges)
#  4. Percentage of nodes with no incoming/outgoing edges

#Returns the mutuality-percentage of a given graph
#(how many of the edges are parallel/bi-directional)
def strongWeak(g):
    weak = len(list(nx.weakly_connected_components(g)))
    strong = len(list(nx.strongly_connected_components(g)))
    return round((weak/strong)*100, 5)

#Results are printed and compared for every directed graph
"""
for g in diGraphs:
    print(g + "\n# of weak components / # of weak components:\n"
          + str(strongWeak(diGraphs[g])) + "%\n")
#"""
print("")

#Returns the mutuality-percentage of a given graph
#(how many of the edges are parallel/bi-directional)
def mutuality(g):
    edgeSet = set(g.edges)
    count = 0
    for (u,v) in edgeSet:
        if (v,u) in edgeSet:
            count += 1
    return round((count/g.size())*100, 5)

#Results are printed and compared for every directed graph
"""
for g in diGraphs:
    print(g + ": mutuality: " + str(mutuality(diGraphs[g])) + "%")
#"""
print("")

#Returns the density-percentage of a given graph
#(how many possible edges are actually present)
```

```
def density(g):
    nodeCount = len(g)
    return round((g.size()/((nodeCount*(nodeCount-1))/2))*100, 5)

#Results are printed and compared for every graph
"""
for g in allGraphs:
    print(g + " density: " + str(density(allGraphs[g])) + "%")
#"""
print("")

#Returns percentage of nodes that have no incoming edges
def noIncoming(g):
    return round((len([node for node in g if not g.in_degree(node)])/len(g))*100,
5)

#Returns percentage of nodes that have no outgoing edges
def noOutgoing(g):
    return round((len([node for node in g if not g.out_de-
gree(node)])/len(g))*100, 5)

#Results are printed and compared for every directed graph
"""
for g in diGraphs:
    print(g + " nodes with no incoming edges: "
          + str(noIncoming(diGraphs[g])) + "%")
print("")
for g in diGraphs:
    print(g + " nodes with no outgoing edges: "
          + str(noOutgoing(diGraphs[g])) + "%")
#"""
print("")

#   DEGREE-RELATED ANALYSIS FUNCTIONS ::
#
#   These tests specifically analyse the networks' degrees, degree reciprocals
#   and relational degrees (used to calculate certain propagation probabilities),
#   as well as various normalization or scaling techniques applied to them.
#
#   This was to rectify the issue I encountered with my WC1 & WC2 models,
#   that the propagation probabilities were too low across the whole dataset,
#   due to the high number of edges and the wide range of nodes' degrees.
#
#   My aim was to find a spread of the probabilities whereby:
#   the mean falls between 0.25-0.75, but the key relationships are kept intact.
#
#   The key relationships being between the propagation probability and:
#   -the in_degree of the target node (WC1).
#   -the out_degree of the targeting node, relative to the out_degrees of all of
the target node's neighbours (WC2).
#
#   Functions:
#   1. In-degrees & Out-degrees
#   2. Degree reciprocals for WC1
#   3. Relational degrees for WC2 (incl. sum of all neighbours' degrees)
#   4. Incorporating propagation variables (pp & qf)
#   5. Root normalization/scaling
#   6. Min-Max normalization/scaling
#   7. Max-normalization/scaling
#   8. Z-score normalizations (incl. adjust-scaling)
#   9. Robust/interquartile normalization
```

```
#  10. Log-scaling

#Return the in_degrees and out_degrees for all nodes in a graph:
def degsList(g, weighted=False):
    inDegs = []
    outDegs = []
    for node in g:
        if weighted:
            inDegs.append(g.in_degree(node, weight='trust'))
            outDegs.append(g.out_degree(node, weight='trust'))
        else:
            inDegs.append(g.in_degree(node))
            outDegs.append(g.out_degree(node))
    return (inDegs, outDegs)

#Calculate and return the degree-reciprocals for all nodes in a graph (WC1):
def calcRecips(g):
    recips = []
    for node in g:
        if not g.in_degree(node):
            continue
        recips.append(1/(g.in_degree(node)))
    return recips

#Calculate and return the relational-degrees for all edges in a graph (WC2):
def calcRelDegs(g, weighted=False):
    relDegs = []
    for target in g:
        if not g.in_degree(target):
            continue
        #sum of target's neighbours' out_degrees
        snd = 0
        for neighbour in g.predecessors(target):
            snd += g.out_degree(neighbour)
        #relational degrees calculated
        for targeting in g.predecessors(target):
            if weighted:
                relDegs.append((g.out_degree(targeting)/snd)*g[targeting][tar-
get]['trust'])
            else:
                relDegs.append(g.out_degree(targeting)/snd)
    return relDegs

#Probability calculating functions are compiled into a list
probFuncs = [(calcRecips, "Degree Reciprocals"), (calcRelDegs, "Relational De-
grees")]

#Averages, maximums and minimums are printed
#"""
for g in graphs:
    print(str(g))
    #indegs, outdegs = degsList(diGraphs[g])
    #printResults1(indegs, "Unweighted in-degrees", False)
    #printResults1(outdegs, "Unweighted out-degrees", False)
    #indegs, outdegs = degsList(diGraphs[g], True)
    #printResults1(indegs, "Weighted in-degrees", False)
    #printResults1(outdegs, "Weighted out-degrees", False)
    printResults1(calcRecips(diGraphs[g]), "Degree reciprocals", False)
    printResults1(calcRelDegs(diGraphs[g]), "Unweighted relational degrees",
False)
```

```
    printResults1(calcRelDegs(diGraphs[g], True), "Weighted relational degrees",
True)
    print("")
#"""
print("")

#Multiply all values in a list by a quality factor qf
def varsList(lis, qf=0.7):
    return list(map(lambda x : x*qf, lis))

#Returns a list of all the trust values from all edges of a given graph,
#  multiplied by pp.
#For comparison with Independent Cascade probabilities
def icProb(g, pp=0.2):
    icprobs = []
    for (u,v,t) in g.edges.data('trust'):
        if not t:
            continue
        icprobs.append(t * pp)
    return icprobs

#Convert all elements in a list to a given root
def rootList(lis, root):
    return list(map(lambda x : x**root, lis))

#Direct rooting of degree-reciprocals in a given graph, up to a given number k:
def recipsRoots(g, k):
    recips = calcRecips(g)
    probs = []
    for i in range(1, k+1):
        probs.append(round(np.mean(rootList(recips, (1/i))), 5))
        #print("Average degRecip prob to the power of " + str(i) + " = " +
str(prob))
    return probs

#Direcct rooting of relational-degrees in a given graph, up to a given number k:
def relDegsRoots(g, k):
    relDegs = calcRelDegs(g)
    probs = []
    for i in range(1, k+1):
        probs.append(round(np.mean(rootList(relDegs, (1/i))), 5))
        #print("Average RelDeg prob to the power of " + str(i) + " = " +
str(prob))
    return probs

#Averages, maximums and minimums are printed
"""
for probFunc in probFuncs:
    print(probFunc[1] + ":\n")
    for g in diGraphs:
        print(g)
        probs = probFunc[0](diGraphs[g])
        for i in range(1,5):
            printResults1(rootList(probs, 1/i), ("Rooted by " + str(i)), True)
    print("")
"""
print("")

#Min-max normalization of a given list
#(a normalization technique itself, but can also be combined with
# other techniques to scale the values between 0 and 1.)
```

```
def mmNormalize(lis):
    elMax = max(lis)
    elMin = min(lis)
    return list(map(lambda x : ((x - elMin) / (elMax - elMin)), lis))

#Direct min-max normalization of degree-reciprocals:
def mmNormalizeDegRec(g):
    degRecs = calcRecips(g)
    norDegRecs = []
    for dr in degRecs:
        norDegRecs.append((dr - min(degRecs)) / (max(degRecs) - min(degRecs)))
    return (degRecs, norDegRecs)

#Direct min-max normalization of relational-degrees:
def mmNormalizeRelDeg(g):
    relDegs = calcRelDegs(g)
    norRelDegs = []
    for rd in relDegs:
        norRelDegs.append((rd - min(relDegs)) / (max(relDegs) - min(relDegs)))
    return (relDegs, norRelDegs)

#Averages, maximums and minimums are printed
"""
for probFunc in probFuncs:
    print(probFunc[1] + ":\n")
    for g in diGraphs:
        print(g)
        printResults1(mmNormalize(probFunc[0](diGraphs[g])), "Min-max normal-
ized", True)
    print("")
"""
print("")

#Normalization by dividing every element in a given list by the maximum value
def maxNormalize(lis):
    elMax = max(lis)
    if not elMax:
        elMax = 0.000000001
    return list(map(lambda x : x/elMax, lis))

#Averages, maximums and minimums are printed
"""
for probFunc in probFuncs:
    print(probFunc[1] + ":\n")
    for g in diGraphs:
        print(g)
        printResults1(maxNormalize(probFunc[0](diGraphs[g])),
                      "Max normalized", True)
    print("")
"""
print("")

#Z-score normalization of a given list
def zNormalize(lis):
    mean = np.mean(lis)
    meanSqs = list(map(lambda x : ((x - mean) ** 2), lis))
    stanDev = np.mean(meanSqs) ** (1/2)
    zScores = list(map(lambda x : ((x - mean) / stanDev), lis))
    return zScores

#Returns mean and standard deviation of a given list
```

```
def standardDev(lis):
    mean = np.mean(lis)
    meanSqs = list(map(lambda x : ((x - mean) ** 2), lis))
    stanDev = np.mean(meanSqs) ** (1/2)
    return mean, stanDev

#Scale a given list to between 0 and 1
def adjust(lis):
    elMax = max(lis)
    elMin = abs(min(lis))
    return list(map(lambda x : ((x + elMin) / (elMax + elMin)), lis))

#Averages, maximums and minimums are printed
"""
for probFunc in probFuncs:
    print(probFunc[1] + ":\n")
    for g in diGraphs:
        print(g)
        printResults1(zNormalize(probFunc[0](diGraphs[g])), "Z-score normalized",
True)
    print("")
"""
print("")

#Robust normalization using interquartile range
def robustNormalize(lis):
    median = np.median(lis)
    q75, q25 = np.percentile(lis, [75, 25])
    iqr = q75 - q25
    return list(map(lambda x : ((x - median) / iqr), lis))

#Averages, maximums and minimums are printed
"""
for probFunc in probFuncs:
    print(probFunc[1] + ":\n")
    for g in graphs:
        print(g)
        printResults1(robustNormalize(probFunc[0](diGraphs[g])), "Robust normal-
ized", True)
    print("")
#"""
print("")

#Log-scale a given list
def logList(lis):
    return list(map(lambda x : log(x), lis))

#Averages, maximums and minimums are printed
"""
for probFunc in probFuncs:
    print(probFunc[1] + ":\n")
    for g in diGraphs:
        print(g)
        printResults1(logList(probFunc[0](diGraphs[g])), "Log-scaled", True)
    print("")
"""
print("")

#   GRAPHING & COMPAING NETWORK-WIDE PROBABILITIES ::
#
#   Functions:
```

```
#   1. Plot pie chart
#   2. Plot histograms comparing probabilities from normalization techniques
#      for a single graph.
#   3. Plot histograms comparing probabilities for a given list of graphs,
#      comparing at each normalization technique.
#   4. Plot histograms comparing probabilities for a given list of graphs,
#      comparing at each normalization technique, with a given list of
#      normalization functions.

#Plot pie chart for probability distribution
def compareNetworksPie1(g, gname, func):
    probs, frac, explode = func[0](g), [[], [], [], [], []], (0.1, 0.1, 0.1, 0.1,
0.1)
    for prob in probs:
        if prob < 0.2:
            frac[0].append(round(prob, 1))
        elif prob < 0.4:
            frac[1].append(round(prob, 1))
        elif prob < 0.6:
            frac[2].append(round(prob, 1))
        elif prob < 0.8:
            frac[3].append(round(prob, 1))
        else:
            frac[4].append(round(prob, 1))
    fracnames, values = [(str(round(i*0.2, 1)) + " -- " +
str((round(((i*0.2)+0.2), 1)))) for i in range(5)], [len(p) for p in frac]
    fig, ax = plt.subplots(1, 1, figsize=(10, 6))
    ax.grid(zorder=0)
    ax.pie(values, labels=fracnames, explode=explode, autopct='%1f%%')
    fig.suptitle(gname + " - " + func[1] + " Probability Distribution:", font-
size=20, fontweight='bold')
    fig.subplots_adjust(top=0.88)

#"""
for gs in [graphs]:
    for g in gs:
        for probFunc in probFuncs:
            compareNetworksPie1(gs[g], g, probFunc)
#"""
print("")

#General function to plot a bar chart for a
#  list of names against a list of values
def generalBar(lis, vals):
    if len(lis[1]) != len(vals[1]):
        print("Error, not the same size")
        return
    fig, ax = plt.subplots(1, 1, figsize=((len(lis[1])*2.5),6))
    ax.grid(zorder=0)
    topVal = max(vals[1])
    ax.set_ylim([0, topVal*1.25])
    ax.bar(lis[1], vals[1], width=0.4, facecolor=colors,
           edgecolor='black', linewidth=2.5, zorder=3)
    for v in range(len(vals[1])):
        if vals[1][v] == max(vals[1]):
            try:
                label = AnchoredText(("Maximum = " + lis[1][v]) +
                                     "\nMean = " + str(round(np.mean(vals[1]),
3)))
                ax.add_artist(label)
            except Exception as e:
```

```
                print(e)
    for count, (xbar,ybar) in enumerate(zip(lis[1], vals[1])):
        if ybar/topVal < 0.3:
            y = ybar + (max(values) * 0.05)
        else:
            y = ybar*0.5
        ax.annotate(ybar, xy=((0.5)*(2*count), y),
                    rotation=90, ha='center', fontsize=16)
    #Subtitle, x-labels & y-labels are set for each axis
    ax.set_xlabel(lis[0], fontsize=15)
    ax.set_ylabel(func[0] + " (%)", fontsize=15)
    #Titles are set and the layout (incl. padding/gaps) is set and adjusted
    fig.tight_layout(pad=5)
    fig.suptitle(func[1] + " Comparison:", fontsize=24, fontweight='bold')
    fig.subplots_adjust(top=0.88)


#Plot bar charts for every metric for one list of graphs
def compareNetworksBar1(graphlist, func):
    #
    labels, values = [], []
    for g in graphlist:
        labels.append(g)
        values.append(func[0](graphlist[g]))
    fig, ax = plt.subplots(1, 1, figsize=(8,5))
    ax.grid(zorder=0)
    topVal = max(values)
    ax.set_ylim([0, topVal*1.38])
    ax.bar(labels, values, width=0.3,
           facecolor='lightsteelblue', edgecolor='black',
           linewidth=2.5, zorder=3)
    label = AnchoredText(("Mean = " + str(round(np.mean(values), 3)) +
                         "\nMedian = " + str(round(np.median(values), 3))),
                        loc=1, prop=dict(size=10))
    ax.add_artist(label)
            #An annotation displaying each bar's value is created and
            #  relatively positioned in each column
    for count, (xbar,ybar) in enumerate(zip(labels, values)):
        if ybar/topVal < 0.3:
            y = ybar + (max(values) * 0.05)
        else:
            y = ybar*0.5
        ax.annotate(round(ybar, 2), xy=((0.5)*(2*count), ybar+(topVal*0.05)),
                    #rotation=90,
                    ha='center', fontsize=14)
    #Subtitle, x-labels & y-labels are set for each axis
    ax.set_xlabel('Graphs', fontsize=15)
    ax.set_ylabel(func[1] + " (%)", fontsize=15)
    #Titles are set and the layout (incl. padding/gaps) is set and adjusted
    fig.tight_layout(pad=5)
    fig.suptitle(func[1] + " Comparison:", fontsize=20, fontweight='bold')
    fig.subplots_adjust(top=0.88)


#List of lists of graphs with labels are compiled
glistlist = [graphs, diGraphs]
#Lists of functions with labels are compiled
fs = [(mutuality, "Mutuality"), (density, "Density"),
      (strongWeak, "# of weak comps / # of strong comps"),
      (noIncoming, "Nodes with no incoming edges"),
      (noOutgoing, "Nodes with no outgoing edges")]
```

```
#Plotting graphs to compare network-wide metrics of all graphs
#"""
for gs in glistlist:
    for f in fs:
        compareNetworksBar1(gs, f)
#"""
print("")

#Plot and display bar charts of given network-wide metrics,
#  comparing a given list of lists of graphs
def compareNetworksBar2(graphlistlist, funclist):
    #Network metric values and labels are calculated for every graph in a given
    #  list of lists of graphs and compiled into 2 lists
    labels, values = [[g for g in graphlist] for graphlist in graphlistlist], []
    for graphlist in graphlistlist:
        values.append([[func[0](graphlist[g]) for g in graphlist] for func in 
funclist])
    #Subplots are created, as wide as the number of metrics and as tall as
    #  the number of different lists of graphs
    figs, axs = plt.subplots(len(funclist), len(graphlistlist), figsize=(15,30), 
sharey=False)
    for f in range(len(funclist)):
        for g in range(len(graphlistlist)):
            #Gridlines are drawn behind the graph
            axs[f, g].grid(zorder=0)
            #Width of bars is adjusted based on the length of the current graph 
list
            barwidth = (len(graphlistlist[g]))*0.1
            while barwidth > 1:
                barwidth *= 0.5
            #Values are assigned from the array to prevent repetitive nested ar-
ray access
            vals = values[g][f]
            #Maximum value is calculated and y-limits are adjusted to more than 
that,
            #  to reserve space for a text boxt in the upper-right
            topVal = max(vals)
            axs[f, g].set_ylim([0, topVal*1.38])
            #Bar chart is plotted with customised visual settings
            axs[f, g].bar(labels[g], vals, width=(barwidth),
                          facecolor='lightsteelblue', edgecolor='black',
                          linewidth=2.5, zorder=3)
            #Mean and median are calculated and displayed in a text-box
            label = AnchoredText(("Mean = " + str(round(np.mean(vals), 3)) +
                                  "\nMedian = " + str(round(np.median(vals), 
3))),
                                 loc=1, prop=dict(size=10))
            axs[f, g].add_artist(label)
            #An annotation displaying each bar's value is created and
            #  relatively positioned in each column
            for count, (xbar,ybar) in enumerate(zip(labels[g], vals)):
                if ybar/topVal < 0.3:
                    y = ybar + (max(vals) * 0.05)
                else:
                    y = ybar*0.5
                axs[f, g].annotate(round(ybar, 2), xy=((0.5)*(2*count), 
ybar+(topVal*0.05)),
                                   ha='center', fontsize=12)
            #Subtitle, x-labels & y-labels are set for each axis
            axs[f, g].set_title((funclist[f][1] + ": "), fontsize=20)
```

```
            axs[f, g].set_xlabel('Graphs', fontsize=15)
            axs[f, g].set_ylabel("Metric (%)", fontsize=15)
    #Titles are set and the layout (incl. padding/gaps) is set and adjusted
    figs.tight_layout(pad=5)
    figs.suptitle("Network-wide metrics", fontsize=24, fontweight='bold')
    figs.subplots_adjust(top=0.95)

#List of lists of graphs with labels are compiled
gs = [diGraphs, graphs]
#Lists of functions with labels are compiled
fs = [(mutuality, "Mutuality"), (density, "Density"),
      (strongWeak, "# of weak comps / # of strong comps"),
      (noIncoming, "Nodes with no incoming edges"),
      (noOutgoing, "Nodes with no outgoing edges")]

#Plotting graphs to compare network-wide metrics of all graphs
#"""
compareNetworksBar2(gs, fs)
#"""
print("")

rooted = []
for g in graphs:
    rooted.append(rootList(calcRelDegs(graphs[g]), (1/2)))

#Histograms of probabilties & normalization techniques #1
#Every normalization technique for one graph
def compareProbsHist1(g, gs, baseFunc):
    #Probabilities are calculated with every technique and compiled into a list
    baseProbs = baseFunc[0](gs[g])
    probs = [(baseProbs, "Base values"),
             (rootList(baseProbs, (1/2)), "Square rooted"),
             (rootList(baseProbs, (1/3)), "Cube rooted"),
             (rootList(baseProbs, (1/4)), "Fourth root"),
             (mmNormalize(baseProbs), "Min-Max normalized"),
             (maxNormalize(baseProbs), "Max normalized"),
             (zNormalize(baseProbs), "Z-score normalized"),
             (robustNormalize(baseProbs), "Interquartile normalized"),
             (logList(baseProbs), "Log-scaled")]
    #Subplots are created, 2 wide and as tall as the number of different proba-
bilities
    figs, axs = plt.subplots(len(probs), 2, figsize=(15, 50), sharey=False)
    for f in range(len(probs)):
        #Values are scaled to within 0 and 1, if they are not already
        if max(probs[f][0]) > 1 or min(probs[f][0]) < 0:
            probs[f] = (mmNormalize(probs[f][0]), probs[f][1])
        for i in range(2):
            #Values and label are assigned from the array to prevent repetitive
nested array access
            #Axes are plotted with two sets of probabilities - once for these
values and then
            #  these values multiplied by a quality factor, alternatively.
            if not i:
                vals, title = probs[f]
                axs[f, i].set_title(g + ": " + title, fontsize=20)
            else:
                vals, title = varsList(probs[f][0]), probs[f][1]
                axs[f, i].set_title(g + ": " + title +
                                    " (multiplied by qf)", fontsize=20)
            #Gridlines are drawn behind the graph
            axs[f, i].grid(zorder=0)
```

```
            #Histogram is plotted with customised visual settings
            bars, bins, _ = axs[f, i].hist(vals, bins=[num*0.1 for num in
range(11)],
                                         facecolor='lightsteelblue',
edgecolor='dimgrey',
                                         linewidth=2.5, rwidth=0.75, zorder=3)
            #Maximum y-value is calculated and y-limits are adjusted to more than
that,
            #  to reserve space for a text boxt in the upper-right
            maxBar = max(bars)
            axs[f, i].set_ylim([0, maxBar*1.25])
            #Mean and median are calculated and displayed in a text-box
            label = AnchoredText(("Mean = " + str(round(np.mean(vals), 3)) +
                                  "\nMedian = " + str(round(np.median(vals),
3))),
                                 loc=1, prop=dict(size=10))
            axs[f, i].add_artist(label)
            #An annotation displaying each histogram bar's value is created and
            #  relatively positioned in each column
            for count, (xbar,ybar) in enumerate(zip(bins, bars)):
                if ybar/maxBar < 0.3:
                    y = ybar + (maxBar * 0.05)
                elif ybar/maxBar > 0.7:
                    y = ybar*0.75
                else:
                    y = ybar*0.5
                axs[f, i].annotate(round(ybar, 0), xy=(xbar+0.05, y),
                                   rotation=90, ha='center', fontsize=12)
            #Histogram bins, x-labels & y-labels are set for each axis
            axs[f, i].set_xticks([num*0.1 for num in range(11)])
            axs[f, i].set_xlabel("Probabilities", fontsize=15)
            axs[f, i].set_ylabel("Frequencies", fontsize=15)
    #Titles are set and the layout (incl. padding/gaps) is set and adjusted
    figs.tight_layout(pad=5)
    figs.subplots_adjust(top=0.96, bottom=0.02)
    figs.suptitle(baseFunc[1] + ": Various Normalization Techniques",
                  fontsize=24, fontweight='bold')


#Plotting histograms to compare probabilities with various
#  normalization techniques for one graph
#"""
for gs in [graphs]:
    for g in gs:
        for probFunc in probFuncs:
            compareProbsHist1(g, gs, probFunc)
#compareProbsHist1(namedGraphs[0], probFuncs[0])
#"""
print("")

#Histograms of probabilities & normalization techniques #2a
#Every graph in list, one-after-another
def compareProbsHist2(graphlist, baseFunc):
    #Probabilities are calculated with every technique and compiled into a list
    probs = []
    for g in graphlist:
        baseProbs = baseFunc[0](graphlist[g])
        probs.append([(baseProbs, "Base values"),
                      (rootList(baseProbs, (1/2)), "Square rooted"),
                      (rootList(baseProbs, (1/3)), "Cube rooted"),
                      (rootList(baseProbs, (1/4)), "Fourth root"),
```

```
                        (mmNormalize(baseProbs), "Min-Max normalized"),
                        (maxNormalize(baseProbs), "Max normalized"),
                        (zNormalize(baseProbs), "Z-score normalized"),
                        (robustNormalize(baseProbs), "Interquartile normalized"),
                        (logList(baseProbs), "Log-scaled")])
    #Subplots are created, 2 wide and as tall as the number of different proba-
bilities
    figs, axs = plt.subplots(len(probs)*len(probs[0]), 2, figsize=(15, 130),
sharey=False)
    for f in range(len(probs[0])):
        for gc, g in enumerate(graphlist):
            index = (f*len(probs))+gc
            #Values are scaled to within 0 and 1, if they are not already
            if max(probs[gc][f][0]) > 1 or min(probs[gc][f][0]) < 0:
                probs[gc][f] = (mmNormalize(probs[gc][f][0]), (probs[gc][f][1] +
" (adjusted)"))
            #Values and label are assigned from the array to prevent repetitive
nested array access
            vals, title = probs[gc][f]
            #Axes are plotted twice - once for these values and then these values
            #  multiplied by a quality factor, alternatively.
            for i in range(2):
                if not i:
                    axs[index, i].set_title(g + ": " + title, fontsize=20)
                else:
                    vals = varsList(vals)
                    axs[index, i].set_title(g + ": " + title +
                                          " multiplied by qf", fontsize=20)
                #Gridlines are drawn behind the graph
                axs[index, i].grid(zorder=0)
                #Histogram is plotted with customised visual settings
                bars, bins, _ = axs[index, i].hist(vals, bins=[num*0.1 for num in
range(11)],
                                             facecolor='lightsteelblue',
edgecolor='dimgrey',
                                             linewidth=2.5, rwidth=0.75,
zorder=3)
                #Maximum y-value is calculated and y-limits are adjusted to more
than that,
                #  to reserve space for a text boxt in the upper-right
                maxBar = max(bars)
                axs[index, i].set_ylim([0, maxBar*1.25])
                #Mean and median are calculated and displayed in a text-box
                label = AnchoredText(("Mean probability = " +
str(round(np.mean(vals), 3)) +
                                    "\nMedian probability = " +
str(round(np.median(vals), 3))),
                                   loc=1, prop=dict(size=10))
                axs[index, i].add_artist(label)
                #An annotation displaying each histogram bar's value is created
and
                #  relatively positioned in each column
                for count, (xbar,ybar) in enumerate(zip(bins, bars)):
                    frac = ybar/maxBar
                    if frac < 0.3:
                        y = ybar + (maxBar * 0.05)
                    elif frac > 0.7:
                        y = ybar*0.75
                    else:
                        y = ybar*(frac)
                    axs[index, i].annotate(ybar, xy=(xbar+0.05, y),
```

```
                                    rotation=90, ha='center', fontsize=12)
                #Histogram bins, x-labels & y-labels are set for each axis
                axs[index, i].set_xticks([num*0.1 for num in range(11)])
                axs[index, i].set_xlabel("Probabilities", fontsize=15)
                axs[index, i].set_ylabel("Frequencies", fontsize=15)
    #Titles are set and the layout (incl. padding/gaps) is set and adjusted
    figs.tight_layout(pad=5)
    figs.subplots_adjust(top=0.97, bottom=0.3)
    figs.suptitle(baseFunc[1] + ": Various Normalization Techniques",
                  fontsize=24, fontweight='bold')

#Plotting histograms to compare probabilities with various
#  normalization techniques of all graphs
#"""
for probFunc in probFuncs:
    compareProbsHist2(graphs, probFunc)
#"""
print("")

#Histograms of probabilities & normalization techniques #2b
#Graphs alternating
def compareProbsHist2Alternate(graphlist, baseFunc, funclist):
    #Probabilities are calculated with every technique from a given list
    #  and compiled into a list
    probs = []
    for g in graphlist:
        graphProbs = []
        baseProbs = baseFunc[0](graphlist[g])
        graphProbs.append((baseProbs, "Base values"))
        for func in funclist:
            if len(func) > 2:
                graphProbs.append((func[0](baseProbs, func[1]), func[2]))
            else:
                graphProbs.append((func[0](baseProbs), func[1]))
        probs.append(graphProbs)

    #Subplots are created, 2 wide and as tall as the number of different proba-
bilities
    figs, axs = plt.subplots(len(probs)*len(probs[0]), 2, figsize=(15, 130),
sharey=False)
    for f in range(len(probs[0])):
        for g, graph in enumerate(graphlist):
            index = (f*len(probs))+g
            #Values are scaled to within 0 and 1, if they are not already
            if max(probs[g][f][0]) > 1 or min(probs[g][f][0]) < 0:
                probs[g][f] = (mmNormalize(probs[g][f][0]), (probs[g][f][1] + "
(adjusted)"))
            #Values and label are assigned from the array to prevent repetitive
nested array access
            vals, title = probs[g][f]
            #Axes are plotted with two sets of probabilities - once for these
values and then
            #  these values multiplied by a quality factor, alternatively.
            for i in range(2):
                if not i:
                    axs[index, i].set_title(graph + ": " + title, fontsize=20)
                else:
                    vals = varsList(vals)
                    axs[index, i].set_title(graph + ": " + title +
                                            " multiplied by qf", fontsize=20)
                #Gridlines are drawn behind the graph
```

```
                axs[index, i].grid(zorder=0)
                #Histogram is plotted with customised visual settings
                bars, bins, _ = axs[index, i].hist(vals, bins=[num*0.1 for num in
range(11)],
                                             facecolor='lightsteelblue',
edgecolor='dimgrey',
                                             linewidth=2.5, rwidth=0.75,
zorder=3)
                #Maximum y-value is calculated and y-limits are adjusted to more
than that,
                #  to reserve space for a text boxt in the upper-right
                maxBar = max(bars)
                axs[index, i].set_ylim([0, maxBar*1.25])
                #Mean and median are calculated and displayed in a text-box
                label = AnchoredText(("Mean probability = " +
str(round(np.mean(vals), 3)) +
                                      "\nMedian probability = " +
str(round(np.median(vals), 3))),
                                     loc=1, prop=dict(size=10))
                axs[index, i].add_artist(label)
                #An annotation displaying each histogram bar's value is created
and
                #  relatively positioned in each column
                for count, (xbar,ybar) in enumerate(zip(bins, bars)):
                    frac = ybar/maxBar
                    if frac < 0.3:
                        y = ybar + (maxBar * 0.05)
                    elif frac > 0.7:
                        y = ybar*0.8
                    else:
                        y = ybar*(frac)
                    axs[index, i].annotate(ybar, xy=(xbar+0.05, y),
                                           rotation=90, ha='center', fontsize=12)
                #Histogram bins, x-labels & y-labels are set for each axis
                axs[index, i].set_xticks([num*0.1 for num in range(11)])
                axs[index, i].set_xlabel("Probabilities", fontsize=15)
                axs[index, i].set_ylabel("Frequencies", fontsize=15)
    #Titles are set and the layout (incl. padding/gaps) is set and adjusted
    figs.tight_layout(pad=5)
    figs.subplots_adjust(top=0.97, bottom=0.3)
    figs.suptitle(baseFunc[1] + ": Various Normalization Techniques",
                  fontsize=24, fontweight='bold')

#List of functions with labels are compiled
fs = [(rootList, (1/2), "Square rooted"),
      (rootList, (1/3), "Cube rooted"),
      (rootList, (1/4), "Fourth root"),
      (mmNormalize, "Min-Max normalized"),
      (zNormalize, "Z-score normalized"),
      (robustNormalize, "Interquartile normalized"),
      (logList, "Log-scaled")]

#Plotting histograms of probabilities with various given
#  normalization techniques alternating between graphs
#"""
for probFunc in probFuncs:
    compareProbsHist2Alternate(graphs, probFunc, fs)
#"""
print("")

#Line-graphs of probabilty spreads & normalization techniques #3
```

```
#
def compareProbsLine1(g, gs, baseFunc, funclist, colours):
    #
    probs, ind = [(baseFunc[0](gs[g]), "Base values")], 0
    for c, func in enumerate(funclist):
        if len(func) > 2:
            for par in range(len(func[1])):
                probs.append((func[0](probs[0][0], func[1][par]), func[2][par]))
                ind += 1
        else:
            probs.append((func[0](probs[0][0]), func[1]))
            ind += 1
        #
        if max(probs[ind][0]) > 1 or min(probs[ind][0]) < 0:
            probs[ind] = (mmNormalize(probs[ind][0]),
                          (probs[ind][1] + " (adjusted)"))
    #
    fracs = [[[] for _ in range(12)] for _ in range(len(probs))]
    for f in range(len(probs)):
        for prob in probs[f][0]:
            added, i = False, 0
            while added == False:
                if prob <= i*0.1:
                    #try:
                    fracs[f][i].append(prob)
                    added = True
                else:
                    i+=1
        #
        for frac in range(len(fracs[f])):
            fracs[f][frac] = len(fracs[f][frac])
    #
    fig, ax = plt.subplots(1, 1, figsize=(10, 5), sharey=False)
    #
    ax.grid(zorder=0)
    xlabels = [i*0.1 for i in range(12)]
    #
    for i in range(len(probs)):
        ax.plot(xlabels, fracs[i], label=probs[i][1], marker='o',
                color=colours[i], linewidth=4, zorder=3)
    #
    ax.legend()
    fig.suptitle(g + " " + probFunc[1] +
                 " Probability Normalization Comparisons:",
                 fontsize=20, fontweight='bold')

#
fs = [(rootList, [(1/2), (1/3), (1/4)],
        ["Square rooted", "Cube rooted", "Fourth root"]),
       (mmNormalize, "Min-Max normalized"),
       (zNormalize, "Z-score normalized"),
       (robustNormalize, "Interquartile normalized"),
       (logList, "Log-scaled")]
cs = ['red', 'orange', 'lawngreen', 'green', 'aqua',
      'lightskyblue', 'blue', 'magenta', 'mediumpurple', 'darkkhaki']

#
#"""
for g in graphs:
    for probFunc in probFuncs:
        compareProbsLine1(g, graphs, probFunc, fs, cs)
```

```
#"""
print("")

def compareProbsLine2(graphlist, baseFunc, funclist, colours, lines):
    #
    allProbs = []
    for g in graphlist:
        gProbs, ind = [(baseFunc[0](graphlist[g]), "Base Values")], 0
        for func in funclist:
            if len(func) > 2:
                for par in range(len(func[1])):
                    gProbs.append((func[0](gProbs[0][0], func[1][par]),
func[2][par]))
                    ind += 1
            else:
                gProbs.append((func[0](gProbs[0][0]), func[1]))
                ind += 1
            #
            if max(gProbs[ind][0]) > 1 or min(gProbs[ind][0]) < 0:
                gProbs[ind] = (mmNormalize(gProbs[ind][0]),
                               (gProbs[ind][1] + " (adjusted)"))
        allProbs.append(gProbs)
    #
    fracs = [[[[] for _ in range(11)] for _ in range(len(allProbs[g]))]
             for g in range(len(allProbs))]
    for g in range(len(allProbs)):
        for f in range(len(allProbs[g])):
            for prob in allProbs[g][f][0]:
                added, i = False, 0
                while added == False:
                    if prob < 0 or prob > 1:
                        print("Error encountered with " + allProbs[g][f][1] +
"!")
                        return
                    elif prob <= i*0.1:
                        fracs[g][f][i].append(prob)
                        added = True
                    else:
                        i+=1
            #
            for frac in range(len(fracs[g][f])):
                fracs[g][f][frac] = len(fracs[g][f][frac])
    #
    figs, axs = plt.subplots(len(allProbs), 1, figsize=(10, 8), sharey=False)
    xlabels = [i*0.1 for i in range(11)]
    for c, g in enumerate(graphlist):
        #
        axs[c].grid(zorder=0, which='both')
        #
        maxProbFreq = 0
        for f in range(len(allProbs[c])):
            if maxProbFreq < max(fracs[c][f]):
                maxProbFreq = max(fracs[c][f])
            lstyle = f
            while lstyle > 2:
                lstyle -= 3
            axs[c].plot(xlabels, fracs[c][f], label=allProbs[c][f][1],
                        linestyle=lines[f], marker='o', markersize=7.5,
                        alpha=0.7, color=colours[f], linewidth=4, zorder=3)
        #
        axs[c].set_ylim([0, maxProbFreq*1.25])
```

```
        axs[c].set_title(probFunc[1] + " Comparisons:")
        axs[c].legend(ncol=2)
        figs.suptitle(g + " " + probFunc[1] + ":",
                      fontsize=20, fontweight='bold')

#
fs = [(rootList, [(1/2), (1/3), (1/4)],
       ["Square rooted", "Cube rooted", "Fourth root"]),
      (mmNormalize, "Min-Max normalized"),
      (zNormalize, "Z-score normalized"),
      (robustNormalize, "Interquartile normalized"),
      (logList, "Log-scaled")]
cs = ['red', 'orange', 'lawngreen', 'green', 'aqua',
      'lightskyblue', 'blue', 'magenta', 'mediumpurple', 'darkkhaki']
ls = ['-', (0, (5, 5)), (0, (5, 5)), (0, (5, 5)), '--', 'dotted', (0, (5, 10)),
(0, (5, 5)),]

#
#"""
for gs in [graphs]:
    for probFunc in probFuncs:
        compareProbsLine2(gs, probFunc, fs, cs, ls)
#"""
print("")

def compareProbsLine3(graphlist, baseFunc, funclist, colours, lines):
    #
    allProbs = []
    for g in graphlist:
        gProbs, ind = [(baseFunc[0](graphlist[g]), "Base Values")], 0
        for func in funclist:
            if len(func) > 2:
                for par in range(len(func[1])):
                    gProbs.append((func[0](gProbs[0][0], func[1][par]),
func[2][par]))
                    ind += 1
            else:
                gProbs.append((func[0](gProbs[0][0]), func[1]))
                ind += 1
            #
            if max(gProbs[ind][0]) > 1 or min(gProbs[ind][0]) < 0:
                gProbs[ind] = (mmNormalize(gProbs[ind][0]),
                               (gProbs[ind][1] + " (adjusted)"))
        allProbs.append(gProbs)
    #
    fracs = [[[[] for _ in range(11)] for _ in range(len(allProbs[g]))]
             for g in range(len(allProbs))]
    for g in range(len(allProbs)):
        for f in range(len(allProbs[g])):
            for prob in allProbs[g][f][0]:
                added, i = False, 0
                while added == False:
                    if prob < 0 or prob > 1:
                        print("Error encountered with " + allProbs[g][f][1] +
"!")
                        return
                    elif prob <= i*0.1:
                        fracs[g][f][i].append(prob)
                        added = True
                    else:
                        i+=1
```

```
            #
            for frac in range(len(fracs[g][f])):
                fracs[g][f][frac] = len(fracs[g][f][frac])
    #
    figs, axs = plt.subplots(len(allProbs), 1, figsize=(10, 8), sharey=False)
    xlabels = [i*0.1 for i in range(11)]
    for c, g in enumerate(graphlist):
        #
        axs[c].grid(zorder=0, which='both')
        #
        maxProbFreq = 0
        for f in range(len(allProbs[c])):
            if maxProbFreq < max(fracs[c][f]):
                maxProbFreq = max(fracs[c][f])
            lstyle = f
            while lstyle > 2:
                lstyle -= 3
            axs[c].plot(xlabels, fracs[c][f], label=allProbs[c][f][1],
                        linestyle=lines[f], marker='o', markersize=7.5,
                        alpha=0.7, color=colours[f], linewidth=4, zorder=3)
        #
        axs[c].set_ylim([0, maxProbFreq*1.25])
        axs[c].set_title(probFunc[1] + " Comparisons:")
        axs[c].legend(ncol=2)
        figs.suptitle(g + " " + probFunc[1] + ":",
                      fontsize=20, fontweight='bold')

#
fs = [(rootList, [(1/2), (1/3), (1/4)],
       ["Square rooted", "Cube rooted", "Fourth root"]),
      (mmNormalize, "Min-Max normalized"),
      (zNormalize, "Z-score normalized"),
      (robustNormalize, "Interquartile normalized"),
      (logList, "Log-scaled")]
cs = ['red', 'orange', 'lawngreen', 'green', 'aqua',
      'lightskyblue', 'blue', 'magenta', 'mediumpurple', 'darkkhaki']
ls = ['-', (0, (5, 5)), (0, (5, 5)), (0, (5, 5)), '-', '-', (0, (5, 5)), (0, (5,
5)),]

#
#"""
for gs in [graphs]:
    for probFunc in probFuncs:
        compareProbsLine3(gs, probFunc, fs, cs, ls)
#"""
print("")
```

seed_selection_final.py

```
#  ALL NECESSARY IMPORTS ::

#product for generating all permutations for seed selection parameter fine-tuning
from itertools import product
#itemgetter for sorting lists of tuples / dicts by their second element / value
from operator import itemgetter
#math.log for calculating logs of probabilities in WC1 and WC2
from math import log
#copy.deepcopy for deepcopying graphs in seed selection models
from copy import deepcopy
#numpy to manipulate lists, calculate means, etc.
import numpy as np
#
import pandas as pd
#time.time to calculate and compare running time and efficiency
from time import time
#Counter to count frequencies in lists, for averaging edges in seed selection
models
from collections import Counter
#networkx to generate and handle networks from data
import networkx as nx
#csv to extract network data from .csv files
import csv
#winsound to alert me when propagation is complete for long proccessing periods
import winsound
#matplotlib.pyplot for plotting graphs and charts
import matplotlib.pyplot as plt
#matplotlib.offsetbox.AnchoredText for anchored textboxes on plotted figures
from matplotlib.offsetbox import AnchoredText

#Dataset dictionary
#Title : weighted, directed, filepath to .csv file
datasets = {
    #BitcoinOTC dataset (5875 nodes, 35592 edges)
    #(directed, weighted, signed)
    "BitcoinOTC": (True, True,
        r"D:\Sully\Documents\Computer Science BSc\Year 3\Term 2\Individual Pro-
ject\datasets\soc-sign-bitcoinotc.csv"),
    #Facebook dataset (4039 nodes, 88234 edges)
    #(undirected, unweighted, unsigned)
    "Facebook": (False, False,
        r"D:\Sully\Documents\Computer Science BSc\Year 3\Term 2\Individual Pro-
ject\datasets\facebook.csv")
}

#  CASCADE, ITERATION, PROPAGATION & SUCCESS FUNCTIONS ::
#
#  Functions to perform cascade & propagation on a given graph
#  with a given seed set and a given number of iterations of a
#  given cascade model.
#
#  Functions included:
#  1. Model-specific success functions
#  2. Propagation function
#  3. Iteration function
#  4. Cascade (ties everything together) function

#Determine propagation success for the various models
#(includes quality factor to differentiate positive/negative influence)
```

```
#(includes a switch penalty for nodes switching sign)

#Apply quality factor and switch factor variables
def successVars(sign, switch, qf=1, sf=1):
    if not sign:
        qf = (1-qf)
    if not switch:
        sf = 0
    return qf*(1-sf)

#Calculate whether propagation is successful (model-specific)
def success(successModel, sign, switch, timeDelay, g, target, targeting, pp, qf,
sf, a):
    if successModel == 'IC':
        succ = (pp*successVars(sign, switch, qf, sf)*g[targeting][tar-
get]['trust']*timeDelay)
    elif successModel == 'WC1':
        if a:
            recip = g.nodes[target]['degRecip']
        else:
            recip = (1 / g.in_degree(target))
        succ = (recip*successVars(sign, switch, qf, sf)*g[targeting][tar-
get]['trust']*timeDelay)
    elif successModel == 'WC2':
        if a:
            relDeg = g[targeting][target]['relDeg']
        else:
            snd = sum([(g.out_degree(neighbour)) for neighbour in g.predeces-
sors(target)])
            relDeg = (g.out_degree(targeting) / snd)
        succ = (relDeg*successVars(sign, switch, qf, sf)*timeDelay*g[target-
ing][target]['trust'])
    return np.random.uniform(0,1) < succ

#Returns probability with only the variables
#(no trust values, degree reciprocals or relational degrees)
def basicProb(weighted=False, *nodes):
    return pp * successVars(True, False)

#One complete turn of propagation from a given set of the newly
#  activated (positive & negative) nodes from the last turn.
#(1. new negative nodes attempt to negatively influence their neighbours)
#(2. new positive nodes attempt to positively influence their neighbours)
#(3. new positive nodes attempt to negatively influence their neighbours)
def propagateTurn(g, pn, pos, nn, neg, trv, td, successMod, pp, qf, sf, a):
    posCurrent, negCurrent = set(), set()
    for node in nn:
        for neighbour in g.neighbors(node):
            if (node, neighbour) not in trv:
                #Negative influence to neighbours of negative nodes
                if success(successMod, False, (neighbour in pos), td, g, neigh-
bour, node, pp, qf, sf, a):
                    negCurrent.add(neighbour)
                trv.add((neighbour, node))
                trv.add((node, neighbour))

    for node in pn:
        for neighbour in g.neighbors(node):
            if (node, neighbour) not in trv:
                #Positive influence to neighbours of positive nodes
```

```
                if neighbour not in negCurrent and success(successMod, True,
(neighbour in neg), td, g, neighbour, node, pp, qf, sf, a):
                    posCurrent.add(neighbour)
                #Negative influence to neighbours of positive nodes
                elif neighbour not in negCurrent and neighbour not in posCurrent
and success(successMod, False, (neighbour in pos), td, g, neighbour, node, pp,
qf, sf, a):
                    negCurrent.add(neighbour)
                trv.add((neighbour, node))
                trv.add((node, neighbour))
    return(posCurrent, negCurrent, trv)

#Calculate average positive spread over a given number of iterations
def iterate(g, s, it, successFunc, pp, qf, sf, tf, ret, a):
    #If no number of iterations is given, one is calculated based on the
    #  ratio of nodes to edges within the graph, capped at 2000.
    if not it:
        neRatio = (len(g)/(g.size()))
        if neRatio > 0.555:
            it = 2000
        else:
            it = ((neRatio/0.165)**(1/1.75))*1000
    influence = []
    if ret:
        infCount = []
    for i in range(it):
        #Randomness seeded for repeatability & robustness
        np.random.seed(i)
        positive, posNew, negative, negNew, traversed, timeFactor = set(),
set(s), set(), set(), set(), 1
        #while there are newly influenced nodes from last turn...
        while posNew or negNew:
            #new nodes assigned to placeholder variables
            posLastTurn, negLastTurn = posNew, negNew
            #propagation turn is performed, returning positive&negative nodes and
traversed edges
            posNew, negNew, traversed = propagateTurn(g, posNew, positive, neg-
New, negative, traversed, timeFactor, successFunc, pp, qf, sf, a)
            #Positive and negative nodes are recalculated
            positive, negative = (positive.union(posNew, posLastTurn) - negNew),
(negative.union(negNew, negLastTurn) - posNew)
            #Time delay is taken away from the time factor
            timeFactor -= tf
        if ret:
            #Positive nodes added to list
            for p in positive:
                influence.append(p)
            #Number of nodes added to list
            infCount.append(len(positive))
        else:
            #Number of positive nodes added to list
            influence.append(len(positive))
    #If nodes are being returned
    if ret:
        #Average list of positive nodes are returned
        counts = Counter(influence)
        result = (sorted(counts, key=counts.get, re-
verse=True))[:int(np.mean(infCount))]
    #If nodes aren't being returned
    else:
        #Mean is returned
```

```
        result = np.mean(influence)
    return result

#Propagation probability declared outside the function, because
#  some seed selection models use it without the cascade function.
pp = 0.2
#Determine the cascade model and run the iteration function
#  with the appropriate success function
def cascade(g, s, its=0,
            model='IC', assign=1, ret=False,
            p=pp, qf=0.7, sf=0.8, tf=0.04):
    #g = graph, s = set of seed nodes, its = num of iterations
    #model = cascade model, #assign model, #return nodes?
    #p = propagation probability, qf = quality factor
    #sf = switch factor, tf = time factor
    #Model is determined and appropriate success function is assigned
    #print(f'model = {model},  assign = {assign}  its = {its}\npp = {p}, qf =
{qf}, sf = {sf}, tf = {tf} \n')
    if model != 'IC' and assign:
        assignSelect(g, model, assign)
    success = model
    return iterate(g, s, its, success, p, qf, sf, tf, ret, assign)

#Propagation models and their names are compiled into a list
propMods = [('IC', "Independent Cascade"),
            ('WC1', "Weighted Cascade 1"),
            ('WC2', "Weighted Cascade 2")]

#Return a set of all reachable nodes from a given node or set of nodes,
#  by recursive depth-first traversal.
#Used in improved greedy & mixed greedy seed selection models
def reach(g, node, reached, traversed):
    for neighbour in g.neighbors(node):
        if (node,neighbour) not in traversed and neighbour not in reached:
            reached.add(neighbour)
            traversed.add((node, neighbour))
            reached, traversed = reach(g, neighbour, reached, traversed)
    return reached, traversed

def reachable(g, s):
    reached, traversed = set(), set()
    for node in s:
        if node not in g:
            continue
        else:
            reached.add(node)
        reached, traversed = reach(g, node, reached, traversed)
    return reached

#  NETWORK GRAPH SETUP ::
#
#  Functions to generate network graphs from various csv files,
#  and assign meaningful attributes to the nodes/edges to save
#  processing time during propagation.
#
#  Datasets/graphs included:
#  1. soc-BitcoinOTC
#  2. ego-Facebook

#Removes any unconnected components of a given graph
def removeUnconnected(g):
```

```
    components = sorted(list(nx.weakly_connected_components(g)), key=len)
    while len(components)>1:
        component = components[0]
        for node in component:
            g.remove_node(node)
        components = components[1:]

#Generates NetworkX graph from given file path:
def generateNetwork(name, weighted, directed, path):
    #graph is initialized and named, dataframe is initialized
    newG = nx.DiGraph(Graph = name)
    data = pd.DataFrame()
    #pandas dataframe is read from .csv file,
    #  with weight if weighted, without if not
    if weighted:
        data = pd.read_csv(path, header=None, usecols=[0,1,2],
                           names=['Node 1', 'Node 2', 'Weight'])
        wMax, wMin = data[['Weight']].max().item(), data[['Weight']].min().item()
    else:
        data = pd.read_csv(path, header=None, usecols=[0,1],
                           names=['Node 1', 'Node 2'])
    #offset is calculated from minimum nodes
    nodeOffset = min(data[['Node 1', 'Node 2']].min())
    #each row of dataframe is added to graph as an edge
    for row in data.itertuples(False, None):
        #trust=weight, & distance=(1-trust)
        if weighted:
            trustval = ((row[2]-wMin)/(wMax-wMin))
        else:
            trustval = 1
        newG.add_edge(row[1]-nodeOffset, row[0]-nodeOffset,
                      trust=trustval, distance=(1-trustval))
        #if graph is undirected, edges are added again in reverse
        if not directed:
            newG.add_edge(row[0]-nodeOffset, row[1]-nodeOffset,
                          trust=trustval, distance=(1-trustval))
    #unconnected components are removed
    if directed:
        removeUnconnected(newG)
    return newG

#Generate graphs and compile into dictionaries:

#Dictionaries for groups of graphs are intialized
graphs, mockGraphs, rndmGraphs, diGraphs, allGraphs = {}, {}, {}, {}, {}

#Generate graphs from real datasets using the datasets dictionary
for g in datasets:
    realGraph = generateNetwork((g + " Network"),
                                datasets[g][0], datasets[g][1],
                                datasets[g][2])
    graphs[g], diGraphs[g], allGraphs[g] = realGraph, realGraph, realGraph


#Generate various mock graphs for testing and debugging:

#Custom, small directed, unweighted graph
mockG, name = nx.DiGraph(), "mock-custom"
testedges = [(1,2), (2,4), (2,5), (2,6), (3,5), (4,5), (5,9), (5,10), (6,8),
             (7,8), (8,9)]
mockG.add_edges_from(testedges)
```

```
nx.set_edge_attributes(mockG, 1, 'trust')
mockGraphs[name], diGraphs[name] = mockG, mockG

#Medium-sized path graph
#(each node only has edges to the node before and/or after it)
mockG, name = nx.path_graph(100), "mock-path"
nx.set_edge_attributes(mockG, 1, 'trust')
mockGraphs[name] = mockG

#Medium-sized, randomly generated directed, unweighted graph
mockG, name = nx.DiGraph(), "mock3-random1"
for i in range(50):
    for j in range(10):
        targ = np.random.randint(-40,50)
        if targ > -1:
            mockG.add_edge(i, targ, trust=1)
mockGraphs[name], rndmGraphs[name], diGraphs[name] = mockG, mockG, mockG

#Medium-sized, randomly generated directed, randomly-weighted graph
mockG, name = nx.DiGraph(), "mock4-random2"
for i in range(50):
    for j in range(10):
        targ = np.random.randint(-40,50)
        if targ > -1:
            tru = np.random.uniform(0,1)
            mockG.add_edge(i, targ, trust=tru)
mockGraphs[name], rndmGraphs[name], diGraphs[name] = mockG, mockG, mockG

#Functional testing for new graphing method
"""
for gl in [realGraphs, mockGraphs]:
    for g in gl:
        print(g + ": " + str(gl[g].size()))
        print(g + ": " + str(len(gl[g])) + "\n")
#"""
print("")

#Normalize (Min-Max) every value in a given dictionary
def mmNormalizeDict(dic, elMax, elMin):
    #for key, value in dic.items():
    #    dic[key] = ((value - elMin) / (elMax - elMin))
    #printResults("Assigned", dic.values())
    #print("Assigned normalization:\nMax = " + str(elMax) + "\nMin = "
    #       + str(elMin) + "\nMean = "
    #       + str(np.mean(list(dic.values()))))
    #return dic
    return {key: ((val - elMin)/(elMax - elMin)) for key,val in dic.items()}

#Methods that assign probabilities for WC1 & WC2 to nodes or edges

#Calculate manipulated degree-reciprocals for all nodes in a graph, and
#  assign them as node attributes for the Weighted Cascade 1 model

#Log-scaling method - default if not specified
def assignRecips1(g):
    drs = {}
    for target in g:
        if not g.in_degree(target):
            continue
        drs[target] = log(1 / g.in_degree(target))
    elMax = drs[max(drs, key=drs.get)]
```

```
    elMin = drs[min(drs, key=drs.get)]
    drs = mmNormalizeDict(drs, elMax, elMin)
    nx.set_node_attributes(g, drs, "degRecip")

#Square-rooting method
def assignRecips2(g):
    drs = {}
    for target in g:
        if not g.in_degree(target):
            continue
        drs[target] = ((1 / g.in_degree(target)) ** (1/2))
    nx.set_node_attributes(g, drs, "degRecip")

#Cube-rooting method
def assignRecips3(g):
    drs = {}
    for target in g:
        if not g.in_degree(target):
            continue
        drs[target] = ((1 / g.in_degree(target)) ** (1/3))
    nx.set_node_attributes(g, drs, "degRecip")

#Calculate manipulated relational-degrees for all edges in a graph, and
#  assign them as edge attributes for the Weighted Cascade 2 model

#Log-scaling method
def assignRelDegs1(g):
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = 0
        for targeting in g.predecessors(target):
            snd += g.out_degree(targeting)
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = log((g.out_degree(targeting) / snd))
    #elMax = rds[max(rds, key=rds.get)]
    #elMin = rds[min(rds, key=rds.get)]
    rds = mmNormalizeDict(rds, max(rds.values()), min(rds.values()))
    nx.set_edge_attributes(g, rds, "relDeg")

#Square-rooting method
def assignRelDegs2(g):
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = sum([(g.out_degree(neighbour))
                   for neighbour in g.predecessors(target)])
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = (((g.out_degree(targeting)) / snd) **
(1/2))
    nx.set_edge_attributes(g, rds, "relDeg")

#Cube-rooting method
def assignRelDegs3(g):
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = sum([(g.out_degree(neighbour))
```

```
                      for neighbour in g.predecessors(target)])
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = (((g.out_degree(targeting)) / snd) **
(1/3))
    nx.set_edge_attributes(g, rds, "relDeg")

#Assign method dictionary for selection depending on parameters
assignMods = {'WC1': {1: assignRecips1, 2: assignRecips2, 3: assignRecips3},
              'WC2': {1: assignRelDegs1, 2: assignRelDegs2, 3:assignRelDegs3}}

#Selects and runs appropriate assigning method
def assignSelect(g, propMod, assignMod):
    if assignMod:
        assignMods[propMod][assignMod](g)


#Manual method to assign relational degree for WC2 to edges

#Return list of relational degrees
def allRelDegs(g):
    #allRds = []
    allRds, allRdsDict = [], {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = sum([g.out_degree(neighbour)
                      for neighbour in g.predecessors(target)])
        for targeting in g.predecessors(target):
            rdval = log(g[targeting][target]['trust']*(g.out_degree(targeting) /
snd))
            allRds.append(rdval)
            allRdsDict[(targeting, target)] = log((g.out_degree(targeting) /
snd))
    return allRds

#List of relational degrees, maximum & minimums are assigned
relDegsTest1 = allRelDegs(graphs['Facebook'])
elMax, elMin = max(relDegsTest1), min(relDegsTest1)

#Normalize a single element, given the max and min elements
def mmNormalizeSingle(val):
    return ((val - elMin)/(elMax - elMin))

#  MISCALLANEOUS & UTILITY METHODS/FUNCTIONS ::
#
#  Methods & functions for various purposes, that are either required
#  in other sections of the program or optimize their performance.
#
#  Methods/Functions included:
#  1. Measure time/speed of a given function
#  2. Min-max normalize a given dictionary, scaling between 0 and 1.
#  3. Draw a histogram from a given dictionary of probabilities

#Normalize (Min-Max) every value in a given dictionary
def mmNormalizeDict(dic, elMax, elMin):
    for key, value in dic.items():
        dic[key] = ((value - elMin) / (elMax - elMin))
    return dic

#Time measuring functions
```

```
#Measure the time taken to perform a given function
def measureTime1(func, *pars):
    startT = time()
    print(func(*pars))
    print(str(round((time() - startT), 3)) + "\n")

#MeasureTime1 with no printing of function
def measureTime1NoPrint(func, *pars):
    startT = time()
    func(*pars)
    print(str(round((time() - startT), 3)) + " secs\n")

#Same as measureTime, but also returns the result from the given function
def measureTimeRet(func, *pars):
    startT = time()
    return func(*pars), round((time() - startT), 3)

#Same as measureTime1, but prints a given message initially
def measureTime2(msg, func, *pars):
    print(msg + ":")
    startT = time()
    print(func(*pars))
    print(str(round((time() - startT), 3)) + " secs\n")

#Given a seed selection model, and can also take parameters for that, selects a
seed set and
#  measures the time taken to do so. Checks this seed set hasn't already been
propagated to,
#  and if it hasnt't performs a given propagation model on it and measures the
time it took.
#Also returns the seed set, so that it can be added to the set of propagated-to
seed sets.
def measureTime3(seedSel, propMod, oldSeeds, g, qty, its, *params):
    print(seedSel[1] + " Seed Selection:")
    startT = time()
    S = set(seedSel[0](g, qty, *params))
    print(str(S) + "\n" + str(time() - startT) + "\n")
    found = False
    for oldSeedSet in oldSeeds:
        if S in oldSeedSet:
            found = True
            print("Same seed set as " + oldSeedSet[1] +
                  ".\nNo need for propagation, check previous results.\n")
    if found:
        return S
    print(propMod[1] + ": (" + str(its) + " iterations)")
    startT = time()
    print(str(cascade(g, S, its, propMod[0])))
    print(str(time() - startT) + "\n\n")
    return S

#Same as measureTime3, but doesn't print strings and returns results
def measureTime3Ret(seedSel, propMod, vals, gname, g, qty, its, *params):
    #found = False
    startT = time()
    Seed = (set(seedSel[0](g, qty, *params)), round((time()-startT), 5))
    #endTime = time() - startT
    #if vals and findNestedDictVal(vals, S):
    #    found = True
    #vals[gname][seedSel[1]][params]['Seed'] = (S, endTime)
    #if found:
```

```
    #    return vals

def measureRetTup1(func, g, qty, *params):
    startT = time()
    return (set(seedSel[0](g, qty, *params)), round((time()-startT), 5))

def measureRetTup2(func, g, S, its):
    startT = time()
    return (cascade(g, S, its, propMod[0]), round((time()-startT), 5))

#Same as measureTime3 without the old seed checking
def measureTime4(seedSel, propMod, oldSeeds, g, qty, its, *params):
    print(seedSel[1] + " Seed Selection:")
    startT = time()
    S = set(seedSel[0](g, qty, *params))
    print(str(S) + "\n" + str(time() - startT) + "\n")
    print(propMod[1] + ": (" + str(its) + " iterations)")
    startT = time()
    print(str(cascade(g, S, its, propMod[0])))
    print(str(time() - startT) + "\n\n")

#   SEED SELECTION MODEL PARAMETER FINE-TUNING ::
#
#   Seed selection functions with variable parameters, and
#   functions to compare the spreads of different parameter values.
#
#   Sections:
#   1. Seed selection models with variable parameters
#      -closeness centrality, -betweenness centrality,
#      -approximate current flow betweenness centrality
#      -load centrality, -eigenvector centrality,
#      -katz centrality, -harmonic centrality, -page rank
#   2. Printing resultant spreads using different permutations of
#      parameters to compare and choose the best.

#Seed selection functions with variable parameters, for testing different
#   value parameters and comparing their resultant spread.

#Closeness-centrality w/ variable parameters
def ccSeedsTune(g, qty, wf, dis=None):
    ccs = nx.closeness_centrality(g, distance=dis, wf_improved=wf)
    return sorted(ccs, key=ccs.get, reverse=True)[:qty]

#Betweenness-centrality w/ variable parameters
def btwnCSeedsTune(g, qty, kp, s, w=None):
    btwns = nx.betweenness_centrality(g, k=kp, normalized=False, weight=w,
seed=s)
    return sorted(btwns, key=btwns.get, reverse=True)[:qty]

#Approximate current-flow Betweenness-centrality w/ variable parameters
def approxCfBtwnCSeedsTune(g, qty, p):
    #A negative p indicates that k should be the graph's size
    #  divided by p, not p itself
    if p < 0:
        k = g.size()/abs(p)
    else:
        k = p
    acfBtwns = nx.approximate_current_flow_betweenness_centrality(nx.to_undi-
rected(g), normalized=False, kmax=k, seed=10)
    return sorted(acfBtwns, key=acfBtwns.get, reverse=True)[:qty]
```

```
#Load-centrality w/ variable parameters
def loadCSeedsTune(g, qty, cut, w=None):
    lcs = nx.load_centrality(g, normalized=False, weight=w, cutoff=cut)
    return sorted(lcs, key=lcs.get, reverse=True)[:qty]

#Eigenvector-centrality w/ variable parameters
def evCSeedsTune(g, qty, mi, t, w=None):
    evcs = nx.eigenvector_centrality(g, weight=w, tol=t, max_iter=mi)
    return sorted(evcs, key=evcs.get, reverse=True)[:qty]

#Katz-centrality w/ variable parameters
def kCSeedsTune(g, qty, b, a, w=None):
    kcs = nx.katz_centrality_numpy(g, alpha=a, beta=b, normalized=False,
weight=w)
    return sorted(kcs, key=kcs.get, reverse=True)[:qty]

#Harmonic-centrality w/ variable parameters
def harmCSeedsTune(g, qty, p=None):
    hcs = nx.harmonic_centrality(g, distance=p)
    return sorted(hcs, key=hcs.get, reverse=True)[:qty]

#PageRank w/ variable parameters
def pageRankSeedsTune(g, qty, al, w=None):
    prs = nx.pagerank(g, alpha=al, weight=w)
    return sorted(prs, key=prs.get, reverse=True)[:qty]

#Seed selection model parameter fine-tuning

#Runs seed selection models that can take different parameter values, with
#  every possible permutation of values and prints the results of each.
def paramFineTune(gs, qty, its, sModsParams):
    for g in graphs:
        print(g + ":\n")
        #Set of tried seed sets is kept, to avoid unneccesary repeated propaga-
tions.
        oldSeeds = set()
        for seedSel in sModsTune:
            #Generates tuples for every possible permutation from the given tuple
of variables
            #Special case for single parameters, as they need to be within an
iterable
            singleParam = False
            if len(seedSel[2]) > 1:
                params = list(product(*seedSel[2]))
            else:
                params = list(*seedSel[2])
                singleParam = True
            for paramPerm in params:
                print(paramPerm)
                if singleParam:
                    paramPerm = [paramPerm]
                try:
                    currentSeeds = measureTime3(seedSel, propMods[0],
                                                oldSeeds, graphs[g],
                                                qty, its, *paramPerm)
                    #Seeds obtained are added as frozen set to a set of tried
seeds,
                    #  to avoid propagating the same seed set twice.
                    oldSeeds.add((frozenset(currentSeeds),
                                  (seedSel[1] + ": " + str(paramPerm))))
                except Exception as e:
```

```
                    print(e)

#Seed selection models to be fine-tuned, with tuples
#  for each parameter containing every value to be tried.
sModsTune = [(btwnCSeedsTune, "BetweennessCentrality", ((25,50,100,200), (10,
None))),
             (approxCfBtwnCSeedsTune, "ApproxCF-BetweennessCentrality",
[(10000,500,-50,-200)]),
             (loadCSeedsTune, "LoadCentrality", [(1,2,3,4)]),
             (evCSeedsTune, "EigenvectorCentrality", ((100,500,1000),
(0.001,0.0025,0.005))),
             (kCSeedsTune, "KatzCentrality", ((0.75,1,1.25,1.5),
(0.05,0.1,0.15,0.2))),
             (harmCSeedsTune, "HarmonicCentrality", ([(None, 'trust')])),
             (pageRankSeedsTune, "PageRank", [(0.65,0.75,0.85,0.95)]),
             (ccSeedsTune, "ClosenessCentrality", [(True, False)])]
#Parameter fine-tuning
"""
paramFineTune(graphs, 8, 25, sModsTune)
#"""
print("")

#Text output from above function:
"""
BitcoinOTC:

(25, 10)
BetweennessCentrality Seed Selection:
{0, 34, 6, 904, 2027, 4558, 2641, 1809}
0.9559996128082275

Independent Cascade: (25 iterations)
579.48
3.3419723510742188


(25, None)
BetweennessCentrality Seed Selection:
{34, 6, 1351, 904, 2124, 1809, 2387, 3128}
0.9119899272918701

Independent Cascade: (25 iterations)
543.6
2.987001419067383


(50, 10)
BetweennessCentrality Seed Selection:
{0, 34, 6, 904, 4171, 2027, 2641, 1809}
1.7610361576080322

Independent Cascade: (25 iterations)
579.08
2.9539971351623535


(50, None)
BetweennessCentrality Seed Selection:
{0, 34, 6, 904, 12, 2641, 1809, 3734}
1.7709946632385254
```

```
Independent Cascade: (25 iterations)
578.68
3.026970386505127


(100, 10)
BetweennessCentrality Seed Selection:
{0, 34, 6, 904, 4171, 2027, 2641, 1809}
3.460965156555176

Same seed set as BetweennessCentrality: (50, 10).
No need for propagation, check previous results.

(100, None)
BetweennessCentrality Seed Selection:
{0, 34, 6, 904, 4171, 2027, 2641, 1809}
3.956001043319702

Same seed set as BetweennessCentrality: (50, 10).
No need for propagation, check previous results.

Same seed set as BetweennessCentrality: (100, 10).
No need for propagation, check previous results.

(200, 10)
BetweennessCentrality Seed Selection:
{0, 34, 6, 904, 4171, 2027, 1809, 2641}
7.684998512268066

Same seed set as BetweennessCentrality: (50, 10).
No need for propagation, check previous results.

Same seed set as BetweennessCentrality: (100, None).
No need for propagation, check previous results.

Same seed set as BetweennessCentrality: (100, 10).
No need for propagation, check previous results.

(200, None)
BetweennessCentrality Seed Selection:
{0, 34, 6, 904, 4171, 2124, 2641, 1809}
8.950001239776611

Independent Cascade: (25 iterations)
579.64
2.987016439437866


10000
ApproxCF-BetweennessCentrality Seed Selection:
{1952, 34, 904, 4171, 2124, 2027, 1809, 5851}
15.70596981048584

Independent Cascade: (25 iterations)
563.44
3.250000238418579


500
ApproxCF-BetweennessCentrality Seed Selection:
{1952, 34, 904, 4171, 2124, 2027, 1809, 5851}
```

```
14.256999969482422

Same seed set as ApproxCF-BetweennessCentrality: [10000].
No need for propagation, check previous results.

-50
ApproxCF-BetweennessCentrality Seed Selection:
{1952, 34, 904, 4171, 2124, 2027, 1809, 5851}
15.11800241470337

Same seed set as ApproxCF-BetweennessCentrality: [10000].
No need for propagation, check previous results.

Same seed set as ApproxCF-BetweennessCentrality: [500].
No need for propagation, check previous results.

-200
ApproxCF-BetweennessCentrality Seed Selection:
{1952, 34, 904, 4171, 2124, 2027, 1809, 5851}
14.81799840927124

Same seed set as ApproxCF-BetweennessCentrality: [10000].
No need for propagation, check previous results.

Same seed set as ApproxCF-BetweennessCentrality: [-50].
No need for propagation, check previous results.

Same seed set as ApproxCF-BetweennessCentrality: [500].
No need for propagation, check previous results.

1
LoadCentrality Seed Selection:
{0, 1, 2, 3, 4, 5, 14, 15}
0.08399629592895508

Independent Cascade: (25 iterations)
494.44
2.95900297164917


2
LoadCentrality Seed Selection:
{34, 6, 904, 2027, 2124, 4171, 2641, 1809}
5.055690288543701

Independent Cascade: (25 iterations)
572.0
3.5680010318756104


3
LoadCentrality Seed Selection:
{0, 34, 6, 904, 2027, 2124, 2641, 1809}
52.553258419036865

Independent Cascade: (25 iterations)
587.44
3.2359983921051025
#"""
print("")
```

```
#   SEED SELECTION MODELS (post parameter fine-tuning)::
#
#   Functions for selecting a seed set from a given graph, using
#   various different strategies. Split into 4 sections.
#
#   Sections:
#   1. Random seed selection model (for comparison)
#   2. Models from prior papers
#      -random, -original greedy, -CELF,
#      -improved greedy, degree discount
#   3. Models based on network analysis metrics
#      -degree centrality, -out_degree centrality, -closeness centrality,
#      -information centrality, -betweenness centrality,
#      -approximate current flow betweenness centrality,
#      -load centrality, -eigenvector centrality, -katz centrality
#      -subgraph centrality, -harmonic centrality,
#      -vote rank, -page rank, -HITS hubs, -HITS authorities
#   4. New models
#      -Mixed greedy 1.1, 1.2, 2.1 & 2.2
#      -customHeuristic
#      -disconnect (DOESN'T WORK)

#Random seed selection model
def randomSeeds(g, qty, *other):
    return set(np.random.choice(g, qty, replace=False))
randomTuple = (randomSeeds, 'random')

#Seed Selection Models from past research

#Original Greedy from Kempe et al 2003
#Calculates spread for every node not in the seed set, and adds the
#   highest to the seed set. Repeat until seed set is full.
def ogGreedySeeds(g, qty, its, propFunc='IC'):
    S = set()
    for _ in range(qty):
        inf = {node: cascade(g, S.union({node}), its, model=propFunc)
               for node in g if node not in S}
        S.add(max(inf, key=inf.get))
    return S

#Cost-effective Lazy-forward (CELF) from Leskovec 2007
#Uses submodularity of propagation to optimize - spread of every node
#   doesn't need to be calculated every time.
#Calculates the spread of every node and creates a sorted list,
#   and extracts the highest to seed set. Then the new highest is
#   recalculated and if it remains the highest is added to the
#   seed set, otherwise the list is resorted.
def celfSeeds(g, qty, its, propFunc='IC'):
    infs = sorted([(node, cascade(g, {node}, its, model=propFunc))
                   for node in g], key=itemgetter(1), reverse=True)
    S = {infs[0][0]}
    spread = infs[0][1]
    infs = infs[1:]
    for _ in range(qty-1):
        sameTop = False
        while not sameTop:
            check = infs[0][0]
            infs[0] = (check, cascade(g, S.union({check}), its,
                                      model=propFunc)- spread)
            infs = sorted(infs, key=itemgetter(1), reverse=True)
            sameTop = (infs[0][0] == check)
```

```
        S.add(infs[0][0])
        spread += infs[0][1]
        infs = infs[1:]
    return S

#Improved Greedy from Chen 2009
#Creates a simulated copy of the graph, removing edges with the probability
#  (1 - pp). Then the reach within that graph is calculated for each node,
#  and the highest is added to the seed set.
def impGreedySeeds(g, qty, its):
    S = set()
    for _ in range(qty):
        for i in range(its):
            np.random.seed(i)
            remove = [(u, v) for (u, v, t) in g.edges.data('trust')
                          if np.random.uniform(0, 1) > (pp*t)]
            newG = deepcopy(g)
            newG.remove_edges_from(remove)
            rSnewG = reachable(newG, S)
            infs = [(node, len(reachable(newG, {node}))) for node in newG
                        if node not in rSnewG]
        #infs = sorted([(node, (val/its)) for (node, val) in infs],
        #              key=lambda x:x[1], reverse=True)
        #S.add(infs[0][0])
        infs = [(node, val/its) for (node, val) in infs]
        S.add(max(infs, key=itemgetter(1))[0])
    return S

#Degree Discount from Chen 2009
#Calculates degree 'score' of each node, adds highest to seed set,
#  discounts score of every neighbour node of newly added seed,
#  and repeats until seed set is filled.
def degDiscSeeds(g, qty, *other):
    S = set()
    nodes = {}
    for node in g:
        deg = g.degree(node)
        nodes[node] = (deg, 0, deg)
    for _ in range(qty):
        ddvmax = 0
        for node in nodes:
            if node not in S:
                if nodes[node][2] > ddvmax:
                    ddvmax = nodes[node][2]
                    u = node
        S.add(u)
        for neighbour in g.neighbors(u):
            if neighbour not in S:
                dv, tv, ddv = nodes[neighbour]
                tv += 1
                ddv = dv - (2*tv) - ((dv - tv)*tv*pp)
                nodes[neighbour] = dv, tv, ddv
    return S

#seeds compiled into list
priorSeeds = [(ogGreedySeeds, 'ogGreedy'), (celfSeeds, 'celf'),
              (impGreedySeeds, 'impGreedy'), (degDiscSeeds,'degDisc'),
              (randomSeeds, 'random')]

#Network-Analysis-metric-based Seed Selection Models
```

```
#Degree-centrality
def degCSeeds(g, qty):
    dcs = nx.degree_centrality(g)
    return sorted(dcs, key=dcs.get, reverse=True)[:qty]

#In-degree-centrality
def inDegCSeeds(g, qty):
    dcs = nx.in_degree_centrality(g)
    return sorted(dcs, key=dcs.get, reverse=True)[:qty]

#Out-degree-centrality
def outDegCSeeds(g, qty):
    dcs = nx.out_degree_centrality(g)
    return sorted(dcs, key=dcs.get, reverse=True)[:qty]

#Closeness-centrality
def ccSeeds(g, qty):
    ccs = nx.closeness_centrality(g, wf_improved=True)
    return sorted(ccs, key=ccs.get, reverse=True)[:qty]

#Information-centrality
#(a.k.a. current-flow closeness-centrality)
def infCSeeds(g, qty):
    ics = nx.information_centrality(nx.to_undirected(g))
    return sorted(ics, key=ics.get, reverse=True)[:qty]

#Betweenness-centrality
def btwnCSeeds(g, qty):
    btwns = nx.betweenness_centrality(g, k=50, normalized=False, seed=10)
    return sorted(btwns, key=btwns.get, reverse=True)[:qty]

#Approximate current-flow betweenness-centrality
def approxCfBtwnCSeeds(g, qty):
    acfBtwns = nx.approximate_current_flow_betweenness_centrality(nx.to_undi-
rected(g), normalized=False, kmax=200, seed=10)
    return sorted(acfBtwns, key=acfBtwns.get, reverse=True)[:qty]

#Load-centrality
def loadCSeeds(g, qty):
    lcs = nx.load_centrality(g, normalized=False, cutoff=2)
    return sorted(lcs, key=lcs.get, reverse=True)[:qty]

#Eigenvector-centrality
def evCSeeds(g, qty):
    evcs = nx.eigenvector_centrality(g, tol=0.005)
    return sorted(evcs, key=evcs.get, reverse=True)[:qty]

#Katz-centrality
def kCSeeds(g, qty):
    kcs = nx.katz_centrality_numpy(g, alpha=0.05, normalized=False)
    return sorted(kcs, key=kcs.get, reverse=True)[:qty]

#Subgraph-centrality
def subgCSeeds(g, qty):
    sgcs = nx.subgraph_centrality(nx.to_undirected(g))
    return sorted(sgcs, key=sgcs.get, reverse=True)[:qty]

#Harmonic-centrality
def harmCSeeds(g, qty):
    hcs = nx.harmonic_centrality(g, distance='distance')
    return sorted(hcs, key=hcs.get, reverse=True)[:qty]
```

```python
#VoteRank
def voteRankSeeds(g, qty):
    return set(nx.voterank(nx.to_undirected(g), qty))

#PageRank
def pageRankSeeds(g, qty):
    prs = nx.pagerank(g, alpha=0.95)
    return sorted(prs, key=prs.get, reverse=True)[:qty]

#HITS Hubs
def hitsHubSeeds(g, qty):
    hhs, has = nx.hits(g)
    return sorted(hhs, key=hhs.get, reverse=True)[:qty]

#HITS Authorities
def hitsAuthSeeds(g, qty):
    hhs, has = nx.hits(g)
    return sorted(has, key=has.get, reverse=True)[:qty]

#NetworkX seeds compiled into list (w/ random)
netSeeds = [(degCSeeds, 'degC'), (inDegCSeeds, 'inDeg'),
            (outDegCSeeds, 'outDeg'), (ccSeeds, 'closeC'),
            (infCSeeds, 'info'), (btwnCSeeds, 'btwnC'),
            (approxCfBtwnCSeeds, 'approxCfBtwnC'), (loadCSeeds, 'loadC'),
            (subgCSeeds, 'subG'), (harmCSeeds, 'harmC'),
            (voteRankSeeds, 'voteRank'), (pageRankSeeds, 'pageRank'),
            (hitsHubSeeds, 'Hubs'), (hitsAuthSeeds, 'Auth'),
            (randomSeeds, 'random')]

#Mixed greedy seed selection models
def mixedGreedy11(g, qty, its):
    S = set()
    for i in range(its):
        np.random.seed(i)
        remove = [(u, v) for (u, v, t) in g.edges.data('trust')
                  if np.random.uniform(0, 1) > (pp*t)]
        newG = deepcopy(g)
        newG.remove_edges_from(remove)
        rSnewG = reachable(newG, S)
        infs = [(node, len(reachable(newG, {node}))) for node in newG
                if node not in rSnewG]
    infs = sorted([(node, val/its) for (node, val) in infs],
                  key=itemgetter(1), reverse=True)
    S.add(infs[0][0])
    reach, infs = infs[0][1], infs[1:]
    for _ in range(qty-1):
        rSnewG = reachable(newG, S)
        sameTop = False
        while not sameTop:
            check = infs[0][0]
            if check in rSnewG:
                infs = infs[1:]
                continue
            infs[0] = (check, len(reachable(newG, {check})) - reach)
            infs = sorted(infs, key=itemgetter(1), reverse=True)
            sameTop = (infs[0][0] == check)
        S.add(infs[0][0])
        reach += infs[0][1]
        infs = infs[1:]
    return S
```

```
def mixedGreedy12(g, qty, its):
    S = set()
    for i in range(its):
        np.random.seed(i)
        remove = [(u, v) for (u, v, t) in g.edges.data('trust')
                  if np.random.uniform(0, 1) > (pp*t)]
        newG = deepcopy(g)
        newG.remove_edges_from(remove)
        rSnewG = reachable(newG, S)
        infs = [(node, len(reachable(newG, {node}))) for node in newG
                if node not in rSnewG]
    infs = sorted([(node, val/its) for (node, val) in infs],
                  key=itemgetter(1), reverse=True)
    S.add(infs[0][0])
    reach, infs = infs[0][1], infs[1:]
    firstrun = 1
    for _ in range(qty-1):
        if firstrun:
            rSnewG = reachable(newG, S)
            firstrun = 0
        else:
            rSnewG = reachable(newG, {Snew})
        newG.remove_nodes_from(rSnewG)
        sameTop = False
        while not sameTop:
            check = infs[0][0]
            if check in rSnewG or check not in newG:
                infs = infs[1:]
                continue
            infs[0] = (check, len(reachable(newG, {check})) - reach)
            infs = sorted(infs, key=itemgetter(1), reverse=True)
            sameTop = (infs[0][0] == check)
        Snew = infs[0][0]
        S.add(Snew)
        reach += infs[0][1]
        infs = infs[1:]
    return S

def mixedGreedy21(g, qty, its):
    S = set()
    edges, edgeCount = [], []
    for i in range(its):
        np.random.seed(i)
        newG = deepcopy(g)
        newG.remove_edges_from([(u,v) for (u,v,t) in g.edges.data('trust')
                                if np.random.uniform(0,1) > (pp*t)])
        newEdges = [e for e in newG.edges]
        edgeCount.append(len(newEdges))
        edges += newEdges
    counts = Counter(edges)
    finalEdges = (sorted(counts, key=counts.get, re-
verse=True))[:int(np.mean(edgeCount))]
    newGfinal = nx.DiGraph(finalEdges)
    rSnewG = reachable(newG, S)
    infs = sorted([(node, len(reachable(newG, {node}))) for node in newGfinal
                   if node not in rSnewG], key=itemgetter(1), reverse=True)
    S.add(infs[0][0])
    reach, infs = infs[0][1], infs[1:]
    for _ in range(qty-1):
        rSnewG = reachable(newGfinal, S)
```

```
        sameTop = False
        while not sameTop:
            check = infs[0][0]
            if check in rSnewG:
                infs = infs[1:]
                continue
            infs[0] = (check, len(reachable(newGfinal, {check})) - reach)
            infs = sorted(infs, key=itemgetter(1), reverse=True)
            sameTop = (infs[0][0] == check)
        S.add(infs[0][0])
        reach += infs[0][1]
        infs = infs[1:]
    return S

def mixedGreedy22(g, qty, its):
    S = set()
    edges, edgeCount = [], []
    for i in range(its):
        np.random.seed(i)
        newG = deepcopy(g)
        newG.remove_edges_from([(u,v) for (u,v,t) in g.edges.data('trust')
                                if np.random.uniform(0,1) > (pp*t)])
        newEdges = [e for e in newG.edges]
        edgeCount.append(len(newEdges))
        edges += newEdges
    counts = Counter(edges)
    finalEdges = (sorted(counts, key=counts.get, re-
verse=True))[:int(np.mean(edgeCount))]
    newGfinal = nx.DiGraph(finalEdges)
    rSnewG = reachable(newG, S)
    infs = sorted([(node, len(reachable(newG, {node}))) for node in newGfinal
                    if node not in rSnewG], key=itemgetter(1), reverse=True)
    S.add(infs[0][0])
    reach, infs = infs[0][1], infs[1:]
    firstrun = 1
    for _ in range(qty-1):
        if firstrun:
            rSnewG = reachable(newGfinal, S)
            firstrun = 0
        else:
            rSnewG = reachable(newGfinal, {Snew})
        newGfinal.remove_nodes_from(rSnewG)
        sameTop = False
        while not sameTop:
            check = infs[0][0]
            if check in rSnewG or check not in newGfinal:
                infs = infs[1:]
                continue
            infs[0] = (check, len(reachable(newGfinal, {check})) - reach)
            infs = sorted(infs, key=itemgetter(1), reverse=True)
            sameTop = (infs[0][0] == check)
        Snew = infs[0][0]
        S.add(Snew)
        reach += infs[0][1]
        infs = infs[1:]
    return S

#Custom Heuristic
def calculateRank(g, node, seeds):
    seedProb, nonSeedProb = 1, 1
    for neighbour in g.predecessors(node):
```

```
        if neighbour in seeds:
            seedProb *= (1 - (basicProb()*(g[neighbour][node]['trust'])))
    for neighbour in g.successors(node):
        if neighbour not in seeds:
            nonSeedProb += (basicProb()*(g[node][neighbour]['trust']))
    return seedProb * nonSeedProb

def customHeuristicSeeds(g, qty, *other):
    S = set()
    ranks = sorted({(node, calculateRank(g, node, S)) for node in g
                    if node not in S}, key=lambda x:x[1], reverse=True)
    for _ in range(qty):
        topNode = ranks[0][0]
        S.add(topNode)
        ranks = ranks[1:]
        for neighbour in g.successors(topNode):
            for pos,(node,rank) in enumerate(ranks):
                if node == neighbour:
                    ranks[pos] = (node, calculateRank(g, node, S))
                    #changed.append(ranks[pos])
                    break
        ranks = sorted(ranks, key=lambda x:x[1], reverse=True)
    return S

#Disconnect Greedy
#Could not get it to work, so omitted from results
def disconnectSeeds(g, qty, its=500):
    S, infs = set(), {}
    for node in g:
        reached = cascade(g, {node}, its, ret=True)
        infs[node] = {'nodes': reached, 'inf': len(reached)}
        maxSeed = max(infs, key=lambda x:infs[x]['inf'])
        S.add(maxSeed)
    Gx = deepcopy(g)
    Gx.remove_nodes_from(infs[maxSeed]['nodes'])
    del infs[maxSeed]
    for _ in (range(qty - 1)):
        for node in Gx:
            newReach = cascade(Gx, {node}, its, ret=True)
            infs[node] = {'nodes':newReach, 'inf':len(newReach)}
        maxSeed = max(infs, key=lambda x:infs[x]['inf'])
        S.add(maxSeed)
        Gx.remove_nodes_from(infs[maxSeed]['nodes'])
        del infs[maxSeed]
    return S

ogSeeds = [(mixedGreedy11, 'mixedGreedy11'),
           (mixedGreedy12, 'mixedGreedy12'),
           (mixedGreedy21, 'mixedGreedy21'),
           (mixedGreedy22, 'mixedGreedy22'),
           (customHeuristicSeeds, 'customHeuristic'),
           (randomSeeds, 'random')]

allSeeds1 = [(ogGreedySeeds, 'ogGreedy'),
             (celfSeeds, 'celf'),
             (impGreedySeeds, 'impGreedy'),
             (degDiscSeeds,'degDisc'),
             (inDegCSeeds, 'inDeg'),
             (outDegCSeeds, 'outDeg'),
             (ccSeeds, 'closeC'),
             (infCSeeds, 'info'),
```

```
              (btwnCSeeds, 'btwnC'),
              (approxCfBtwnCSeeds, 'approxCfBtwnC'),
              (loadCSeeds, 'loadC'),
              (subgCSeeds, 'subG'),
              (harmCSeeds, 'harmC'),
              (voteRankSeeds, 'voteRank'),
              (pageRankSeeds, 'pageRank'),
              (hitsHubSeeds, 'Hubs'),
              (hitsAuthSeeds, 'Auth'),
              (mixedGreedy11, 'mixedGreedy11'),
              (mixedGreedy12, 'mixedGreedy12'),
              (mixedGreedy21, 'mixedGreedy21'),
              (mixedGreedy22, 'mixedGreedy22'),
              (customHeuristicSeeds, 'customHeuristic'),
              (randomSeeds, 'random')]
allSeeds2 = [(degDiscSeeds, 'degreeDiscount'),
              (inDegCSeeds, 'inDegree'),
              (outDegCSeeds, 'outDegree'),
              (mixedGreedy22, 'mixedGreedy22'),
              (customHeuristicSeeds, 'customHeuristic'),
              (randomSeeds, 'randomSeeds')]

#  GRAPHING FUNCTIONS ::

def horzBar(lis, vals, msg, topGap):
    #return nothing if lists aren't same size
    if len(lis[1]) != len(vals[1]) != len(vals[2]):
        print("Error, not the same size")
        return
    lis[1], vals[1], vals[2] = lis[1][::-1], vals[1][::-1], vals[2][::-1]
    #subplot set up, gridlines drawn, max value calculated and y-limits set
    fig, ax = plt.subplots(2, 1, figsize=(12,2*len(vals[1])))
    for g in range(2):
        ax[g].grid(zorder=0)
        height, pos = 0.4, np.arange(len(vals[1]))
        #bar chart plotted
        ax[g].barh(lis[1], vals[g+1], height=height,
                facecolor='lightsteelblue', edgecolor='black',
                linewidth=2.5, zorder=3)
        #Subtitle, x-labels & y-labels are set for each axis
        ax[g].set_ylabel(lis[0], fontsize=20)
        ax[g].tick_params(axis='both', labelsize=15)
        ax[g].set_xlabel(vals[0][g], fontsize=20)
    #Titles are set and the layout (incl. padding/gaps) is set and adjusted
    fig.tight_layout(pad=5)
    fig.suptitle(msg + " Comparison:", fontsize=24, fontweight='bold')
    fig.subplots_adjust(top=topGap)

#Prepares values for comparing seed selection models, and plots bar chart
def prepareBar(seedMods, gs, qty=4, its=100, its2=100, topGap=0.92, gqty=0,
model='IC', timeFactor=0.01):
    if not gqty:
        gqty=len(gs)
    #Lists are intialized
    x = ['Models', seedMods]
    #Special cases where additional variables are required in
    #  seed selection are noted
    fourParam = ['ogGreedy', 'celf']
    threeParam = ['impGreedy', 'mixedGreedy11', 'mixedGreedy12',
                  'mixedGreedy21', 'mixedGreedy22']
    #For every graph in the given list,
```

```
    for gc, g in enumerate(gs):
        if gc+1 != gqty:
            continue
        y, seedTimes = [['Spread', 'Spread by Time'], [], []], []
        for c, seedSel in enumerate(x[1]):
            #Every seed selection model is run, seeds and the time
            #  elapsed are added to a list
            if seedSel[1] in threeParam:
                t = time()
                seeds = seedSel[0](gs[g], qty, its2)
                t = time() - t
                if t < 0.001:
                    t = 0.001
                seedTimes.append(t)
            elif seedSel[1] in fourParam:
                t = time()
                seeds = seedSel[0](gs[g], qty, its2, model)
                t = time() - t
                if t < 0.001:
                    t = 0.001
                seedTimes.append(t)
            else:
                t = time()
                seeds = seedSel[0](gs[g], qty)
                t = time() - t
                if t < 0.001:
                    t = 0.001
                seedTimes.append(t)
            casc = cascade(gs[g], seeds, its)
            y[1].append(casc)
            if seedTimes[c] > 1:
                y[2].append(casc/(seedTimes[c]))
            else:
                y[2].append(casc)
        #Seed selection labels are compiled and bar chart is plotted
        xLabels = ['Models', [seedMod[1] for seedMod in x[1]]]
        horzBar(xLabels, y, (g + " Seed Select Models:"), topGap)

#Past models - printing
print("Past models time testing (5 seeds)::\n")
for rndmGraph in rndmGraphs:
    print(rndmGraph +  ":\n")
    for seedSel in priorSeeds:
        measureTime2(seedSel[1], seedSel[0], rndmGraphs[rndmGraph], 5)

#Past models random graphs - bar charts for spread & spread/time
#"""
measureTime1NoPrint(prepareBar, priorSeeds, rndmGraphs, 5)
#"""

#NetworkX models - printing
print("Past models time testing (5 seeds)::\n")
for rndmGraph in rndmGraphs:
    print(rndmGraph +  ":\n")
    for seedSel in netSeeds:
        measureTime2(seedSel[1], seedSel[0], rndmGraphs[rndmGraph], 5)

#NetworkX models random graphs - bar charts for spread & spread/time
#"""
measureTime1NoPrint(prepareBar, netSeeds, rndmGraphs, 5, 100, 100, 0.96)
#"""
```

```
#NetworkX models real graphs - bar charts for spread & spread/time
#"""
measureTime1NoPrint(prepareBar, netSeeds, graphs, 5, 1, 1, 0.96, 1)
#"""
print("")

#New models - printing
print("New models time testing (5 seeds)::\n")
for rndmGraph in rndmGraphs:
    print(rndmGraph +  ":\n")
    for seedSel in ogSeeds:
        measureTime2(seedSel[1], seedSel[0], rndmGraphs[rndmGraph], 5, 500)

#New models real graphs - printing
print("New models time testing (5 seeds)::\n")
for graph in graphs:
    print(graph +  ":\n")
    for seedSel in ogSeeds:
        measureTime2(seedSel[1], seedSel[0], graphs[graph], 5, 5)
    print("")
    break

#New models random graphs - bar charts for spread & spread/time
#"""
measureTime1NoPrint(prepareBar, ogSeeds, rndmGraphs, 5, 100)
#"""
print("")

#New models real graphs - bar charts for spread & spread/time
#"""
measureTime1NoPrint(prepareBar, ogSeeds, graphs, 5, 5, 5, 0.92)
#"""
print("")

#All models random graphs - bar charts for spread & spread/time
#"""
measureTime1NoPrint(prepareBar, allSeeds1, rndmGraphs, 5, 100, 100, 0.96)
#"""
print("")

#All models random graphs - bar charts for spread & spread/time
#"""
measureTime1NoPrint(prepareBar, allSeeds2, graphs, 5, 50, 50, 0.92, 1)
#"""
print("")

#All models random graphs - bar charts for spread & spread/time
#"""
measureTime1NoPrint(prepareBar, allSeeds2, graphs, 5, 50, 50, 0.92, 2)
#"""
print("")

g, qty, its = rndmGraphs['mock3-random1'], 4, 250
testSeeds = [(randomSeeds, 'random'),
             (degDiscSeeds, 'degreeDiscount'),
             (degCSeeds, 'degreeCentrality'),
             (customHeuristicSeeds, 'customHeuristics')]
def testRun(g, qty, seedMod, its):
    print("Seed selection model: " + seedMod[1])
    s, t = measureTimeRet(seedMod[0], g, qty)
```

```
    print("Seeds: " + str(s) + "\nTime taken: " + str(round(t, 3)))
    inf, t = measureTimeRet(cascade, g, s, its)
    print("Spread: " + str(inf) + "\nTime taken: " + str(round(t, 3)) + "\n")

for testSeed in testSeeds:
    testRun(g, qty, testSeed, its)
```

upgrades.py (prototype and testing area)

```
#  ALL NECESSARY IMPORTS ::

#product for generating all permutations for seed selection parameter fine-tuning
from itertools import product
#itemgetter for sorting lists of tuples / dicts by their second element / value
from operator import itemgetter
#math.log for calculating logs of probabilities in WC1 and WC2
from math import log
#copy.deepcopy for deepcopying graphs in seed selection models
from copy import deepcopy
#numpy to manipulate lists, calculate means, etc.
import numpy as np
#
import pandas as pd
#time.time to calculate and compare running time and efficiency
from time import time
#Counter to count frequencies in lists, for averaging edges in seed selection
models
from collections import Counter
#networkx to generate and handle networks from data
import networkx as nx
#csv to extract network data from .csv files
import csv
#winsound to alert me when propagation is complete for long proccessing periods
import winsound
#matplotlib.pyplot for plotting graphs and charts
import matplotlib.pyplot as plt
#matplotlib.offsetbox.AnchoredText for anchored textboxes on plotted figures
from matplotlib.offsetbox import AnchoredText

#Time measuring functions

#Measure the time taken to perform a given function
def measureTime1(func, *pars):
    startT = time()
    print(func(*pars))
    print(str(round((time() - startT), 3)) + " secs\n")

#Measures time, and returns the time unrounded
def measureTimeRet1(func, *pars):
    startT = time()
    return (time() - startT)

#Measures time, and returns the result and the time
def measureTimeRet2(func, *pars):
    startT = time()
    return func(*pars), round((time() - startT), 3)

#Same as measureTime1, but prints a given message initially
def measureTime2(msg, func, *pars):
    print(msg + ":")
    startT = time()
    print(func(*pars))
    print(str(time() - startT) + " secs\n")

#Given a seed selection model, and can also take parameters for that, selects a
seed set and
#  measures the time taken to do so. Checks this seed set hasn't already been
propagated to,
```

```python
#  and if it hasn't performs a given propagation model on it and measures the
time it took.
#Also returns the seed set, so that it can be added to the set of propagated-to
seed sets.
def measureTime3(seedSel, propMod, oldSeeds, g, qty, its, *params):
    print(seedSel[1] + " Seed Selection:")
    startT = time()
    S = set(seedSel[0](g, qty, *params))
    print(str(S) + "\n" + str(time() - startT) + "\n")
    found = False
    for oldSeedSet in oldSeeds:
        if S in oldSeedSet:
            found = True
            print(seedSel[1] + "has the same seed set as " + oldSeedSet[1] +
                  ". No need for propagation, check previous results.\n")
    if found:
        return S
    print(propMod[1] + ": (" + str(its) + " iterations)")
    startT = time()
    print(str(cascade(g, S, its, propMod[0])))
    print(str(time() - startT) + "\n\n")
    return S

#Same as measureTime3 without the old seed checking
def measureTime4(seedSel, propMod, oldSeeds, g, qty, its, *params):
    print(seedSel[1] + " Seed Selection:")
    startT = time()
    S = set(seedSel[0](g, qty, *params))
    print(str(S) + "\n" + str(time() - startT) + "\n")
    print(propMod[1] + ": (" + str(its) + " iterations)")
    startT = time()
    print(str(cascade(g, S, its, propMod[0])))
    print(str(time() - startT) + "\n\n")

#INFLUENCE MODEL #1

def IndependentCascade1(g, s, its, pp):
    #Graph, Seed set, Iterations, Propagation probability
    #Time measured and overall spread list initalized
    startTime = time()
    spread = []
    #every iteration...
    for it in range(its):
        #randomness seeded for consistency
        np.random.seed(it)
        influenced, tried = [], []
        #new nodes set to seed nodes
        newlyInf = s
        #while nodes were influenced this turn...
        #(stops when propagation cannot continue)
        while newlyInf:
            #targets are compiled into a list from newly influenced nodes
            targets = []
            for node in newlyInf:
                for neighbour in g.neighbors(node):
                    if neighbour not in influenced:
                        targets.append(neighbour)
            lastTurn = newlyInf
            newlyInf = []
            #targets are influenced depending on pp
            for target in targets:
```

```
                tried.append(target)
                if np.random.random() < pp:
                    newlyInf.append(target)
            #newly influenced nodes added to overall list
            influenced.append(newlyInf)
        #total number of influenced nodes added to list
        spread.append(len(influenced))
    #mean of all iterations returned, with time taken
    return np.mean(spread), (str(round((time()-startTime), 4)) + " secs")


#propagation probability testing
print(IndependentCascade1(G3, [1,2], 100, 0.1))
print(IndependentCascade1(G3, [1,2], 100, 0.8))

#INFLUENCE MODEL #2

#Success functions #1
def successVars2(sign, qf):
    q = qf
    #Modify quality factor for negative influence
    if not sign:
        q = (1-q)
    return q

def successIC(sign, g, target, targeting, pp, qf):
    return np.random.uniform(0,1) < (pp*successVars2(sign, qf)*g[targeting][tar-
get]['trust'])

def successWC1(sign, g, target, targeting, pp, qf):
    recip = 1 / g.in_degree(target)
    return np.random.uniform(0,1) < (recip*successVars2(sign, qf)*g[target-
ing][target]['trust'])

def successWC2(sign, g, target, targeting, pp, qf):
    snd = 0
    for neighbour in g.predecessors(target):
        snd += 1
    reldeg = g.out_degree(targeting) / snd
    return np.random.uniform(0,1) < (reldeg*successVars2(sign, qf)*g[target-
ing][target]['trust'])

def propagation2(g, posNew, negNew, tried, successMod, pp, qf):
    posCurrent, negCurrent = set(), set()
    for node in negNew:
        for neighbour in g.neighbors(node):
            if (node, neighbour) not in tried:
                #Negative influence to neighbours of negative nodes
                if successMod(False, g, neighbour, node, pp, qf):
                    negCurrent.add(neighbour)
                tried.add((node, neighbour))
    for node in posNew:
        for neighbour in g.neighbors(node):
            if (node, neighbour) not in tried:
                #Positive influence to neighbours of positive nodes
                if neighbour not in negCurrent and successMod(True, g, neighbour,
node, pp, qf):
                    posCurrent.add(neighbour)
                #Negative influence to neighbours of positive nodes
                elif neighbour not in negCurrent and neighbour not in posCurrent
and successMod(False, g, neighbour, node, pp, qf):
```

```
                    negCurrent.add(neighbour)
                tried.add((node, neighbour))
    return(posCurrent, negCurrent, tried)

#Cascade Model #2
def iteration2(g, s, its, pp=0.2, qf=1, model='IC'):
    #Graph, Seed set, Iterations, Propagation probability
    if model == 'WC1':
        successFunc = successWC1
    elif model == 'WC2':
        successFunc = successWC2
    else:
        successFunc = successIC
    #Time measured and overall spread list initalized
    startTime = time()
    spread = []
    #every iteration...
    for it in range(its):
        #randomness seeded for consistency
        np.random.seed(it)
        #Sets initialised for influenced/newly influenced nodes and tried edges
        positive, posNew, negative, negNew, tried = set(), set(s), set(), set(),
set()
        #while nodes were influenced this turn...
        #(stops when propagation cannot continue)
        while posNew or negNew:
            #placeholder variables for new nodes
            posLastTurn, negLastTurn = posNew, negNew
            #propagation function is called
            posNew, negNew, tried = propagation2(g, posNew, negNew, tried, suc-
cessFunc, pp, qf)
            #newly influenced nodes added to overall lists
            positive = (positive.union(posNew, posLastTurn) - negNew)
            negative = (negative.union(negNew, negLastTurn) - posNew)
        #total number of influenced nodes added to list
        spread.append(len(positive))
    #mean of all iterations returned, with time taken
    return np.mean(spread), (str(round((time()-startTime), 4)) + " secs")


#Optimization testing for influence model method 1 -> 2
for infFunc in [IndependentCascade1, iteration2]:
    measureTime1(infFunc, G3, [1,2], 500, 0.5)

#quality factor testing for every model, for every real graph
for model in ['IC', 'WC1', 'WC2']:
    for gc, g in enumerate([G1, G2]):
        for pp in [0.2]:
            for qf in [0.2, 0.8]:
                print("G" + str(gc+1) + ":\n" + model + " Vars Testing\nPP = "
                      + str(pp) + ". QF = " + str(qf) + "\n" +
                      str(iteration2(g, [1], 50, pp, qf, model)) + "\n")


#FINAL INFLUENCE MODEL

#Determine propagation success for the various models
#(includes quality factor to differentiate positive/negative influence)
#(includes a switch penalty for nodes switching sign)

#Apply quality factor and switch factor variables
```

```
def successVars(sign, switch, qf, sf):
    if not switch:
        sf = 0
    if not sign:
        qf = (1-qf)
    return qf*(1-sf)

#Calculate whether propagation is successful (model-specific)
def success(successModel, sign, switch, timeDelay, g, target, targeting, pp, qf,
sf, a):
    if successModel == 'ICu':
        succ = (pp*successVars(sign, switch, qf, sf)*timeDelay)
    elif successModel == 'IC':
        succ = (pp*successVars(sign, switch, qf, sf)*g[targeting][tar-
get]['trust']*timeDelay)
    elif successModel == 'WC1':
        if a:
            recip = g.nodes[target]['degRecip']
        else:
            recip = (1 / g.in_degree(target))
        succ = (recip*successVars(sign, switch, qf, sf)*timeDelay*g[target-
ing][target]['trust'])
    elif successModel == 'WC2':
        if a:
            relDeg = g[targeting][target]['relDeg']
        else:
            snd = sum([(g.out_degree(neighbour)) for neighbour in g.predeces-
sors(target)])
            relDeg = (g.out_degree(targeting) / snd)
            #relDeg = mmNormalizeSingle(log(g.out_degree(targeting)/snd))
        succ = (relDeg*successVars(sign, switch, qf, sf)*timeDelay*g[target-
ing][target]['trust'])
    return np.random.uniform(0,1) < succ

#Returns probability with only the variables
#(no trust values, degree reciprocals or relational degrees)
def basicProb(weighted=False, *nodes):
    return pp * successVars(True, False)

#One complete turn of propagation from a given set of the newly
#  activated (positive & negative) nodes from the last turn.
#(1. new negative nodes attempt to negatively influence their neighbours)
#(2. new positive nodes attempt to positively influence their neighbours)
#(3. new positive nodes attempt to negatively influence their neighbours)
def propagateTurn(g, pn, pos, nn, neg, trv, td, successMod, pp, qf, sf, a):
    posCurrent, negCurrent = set(), set()
    for node in nn:
        for neighbour in g.neighbors(node):
            if (node, neighbour) not in trv:
                #Negative influence to neighbours of negative nodes
                if success(successMod, False, (neighbour in pos), td, g, neigh-
bour, node, pp, qf, sf, a):
                    negCurrent.add(neighbour)
                trv.add((node, neighbour))
    for node in pn:
        for neighbour in g.neighbors(node):
            if (node, neighbour) not in trv:
                #Positive influence to neighbours of positive nodes
                if neighbour not in negCurrent and success(successMod, True,
(neighbour in neg), td, g, neighbour, node, pp, qf, sf, a):
                    posCurrent.add(neighbour)
```

```
                #Negative influence to neighbours of positive nodes
                elif neighbour not in negCurrent and neighbour not in posCurrent
and success(successMod, False, (neighbour in pos), td, g, neighbour, node, pp,
qf, sf, a):
                    negCurrent.add(neighbour)
                trv.add((node, neighbour))
    return(posCurrent, negCurrent, trv)

#Calculate average positive spread over a given number of iterations
def iterate(g, s, its, successFunc, pp, qf, sf, tf, retNodes, a):
    #If no number of iterations is given, one is calculated based on the
    #  ratio of nodes to edges within the graph, capped at 2000.
    if not its:
        neRatio = (len(g)/(g.size()))
        if neRatio > 0.555:
            its = 2000
        else:
            its = ((neRatio/0.165)**(1/1.75))*1000
    influence = []
    for i in range(its):
        #Randomness seeded per iteration for repeatability & robustness
        np.random.seed(i)
        positive, posNew, negative, negNew, traversed, timeFactor = set(),
set(s), set(), set(), set(), 1
        #while there are newly influenced nodes from last turn...
        while posNew or negNew:
            #new nodes assigned to placeholder variables
            posLastTurn, negLastTurn = posNew, negNew
            #propagation turn is performed, returning positive&negative nodes and
traversed edges
            posNew, negNew, traversed = propagateTurn(g, posNew, positive, neg-
New, negative, traversed, timeFactor, successFunc, pp, qf, sf, a)
            #Positive and negative nodes are recalculated
            positive, negative = (positive.union(posNew, posLastTurn) - negNew),
(negative.union(negNew, negLastTurn) - posNew)
            #Time delay is taken away from the time factor
            if timeFactor < 0:
                timeFactor = 0
            else:
                timeFactor -= tf
        if retNodes:
            #Positive nodes added to list
            for p in positive:
                influence.append(p)
            #Number of nodes added to list
            infCount.append(len(positive))
        else:
            #Number of positive nodes added to list
            influence.append(len(positive))
    #If nodes are being returned
    if retNodes:
        #Average list of positive nodes are returned
        counts = Counter(influence)
        result = (sorted(counts, key=counts.get, re-
verse=True))[:int(np.mean(infCount))]
    #If nodes aren't being returned
    else:
        #Mean is returned
        result = np.mean(influence)
    return result
```

```
#Determine the cascade model and run the iteration function
#  with the appropriate success function
def cascade(g, s, its=0,
            model='IC', assign=1, ret=False,
            pp=0.2, qf=0.6, sf=0.7, tf=0.04):
    #g = graph, s = set of seed nodes, its = num of iterations
    #model = cascade model, #assign model, #return nodes?
    #pp = propagation probability, qf = quality factor
    #sf = switch factor, tf = time factor
    #Model is determined and appropriate success function is assigned
    #print(f'model = {model},  assign = {assign}  its = {its}\npp = {pp}, qf =
{qf}, sf = {sf}, tf = {tf} \n')
    if model != 'IC' and model != 'ICu' and assign:
        assignSelect(g, model, assign)
    success = model
    return iterate(g, s, its, success, pp, qf, sf, tf, ret, assign)

#Propagation models and their names are compiled into a list
propMods = [('IC', "Independent Cascade"),
            ('WC1', "Weighted Cascade 1"),
            ('WC2', "Weighted Cascade 2")]

#Methods that assign probabilities for WC1 & WC2 to nodes or edges

#Calculate manipulated degree-reciprocals for all nodes in a graph, and
#  assign them as node attributes for the Weighted Cascade 1 model

#Log-scaling method - default if not specified
def assignRecips1(g):
    drs = {}
    for target in g:
        if not g.in_degree(target):
            continue
        drs[target] = log(1 / g.in_degree(target))
    elMax = drs[max(drs, key=drs.get)]
    elMin = drs[min(drs, key=drs.get)]
    drs = mmNormalizeDict(drs, elMax, elMin)
    nx.set_node_attributes(g, drs, "degRecip")

#Square-rooting method
def assignRecips2(g):
    drs = {}
    for target in g:
        if not g.in_degree(target):
            continue
        drs[target] = ((1 / g.in_degree(target)) ** (1/2))
    nx.set_node_attributes(g, drs, "degRecip")

#Cube-rooting method
def assignRecips3(g):
    drs = {}
    for target in g:
        if not g.in_degree(target):
            continue
        drs[target] = ((1 / g.in_degree(target)) ** (1/3))
    nx.set_node_attributes(g, drs, "degRecip")

#Calculate manipulated relational-degrees for all edges in a graph, and
#  assign them as edge attributes for the Weighted Cascade 2 model

#Log-scaling method
```

```python
def assignRelDegs1(g):
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = 0
        for targeting in g.predecessors(target):
            snd += g.out_degree(targeting)
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = log(g.out_degree(targeting) / snd)
    #elMax = rds[max(rds, key=rds.get)]
    #elMin = rds[min(rds, key=rds.get)]
    rds = mmNormalizeDict(rds, max(rds.values()), min(rds.values()))
    nx.set_edge_attributes(g, rds, "relDeg")

#Square-rooting method
def assignRelDegs2(g):
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = 0
        for targeting in g.predecessors(target):
            snd += g.out_degree(targeting)
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = (((g.out_degree(targeting)) / snd) **
(1/2))
    nx.set_edge_attributes(g, rds, "relDeg")

#Cube-rooting method
def assignRelDegs3(g):
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = sum([(g.out_degree(neighbour))
                   for neighbour in g.predecessors(target)])
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = (((g.out_degree(targeting)) / snd) **
(1/3))
    nx.set_edge_attributes(g, rds, "relDeg")

#Assign method dictionary for selection depending on parameters
assignMods = {'WC1': {1: assignRecips1, 2: assignRecips2, 3: assignRecips3},
              'WC2': {1: assignRelDegs1, 2: assignRelDegs2, 3:assignRelDegs3}}

#Selects and runs appropriate assigning method
def assignSelect(g, propMod, assignMod):
    if assignMod:
        assignMods[propMod][assignMod](g)

#Normalize (Min-Max) every value in a given dictionary (method 2 & 3)
def mmNormalizeDict(dic, elMax, elMin):
    #for key, value in dic.items():
    #    dic[key] = ((value - elMin) / (elMax - elMin))
    #printResults("Assigned", dic.values())
    #print("Assigned normalization:\nMax = " + str(elMax) + "\nMin = "
    #      + str(elMin) + "\nMean = "
    #      + str(np.mean(list(dic.values()))))
    #return dic
    return {key: ((val - elMin)/(elMax - elMin)) for key,val in dic.items()}
```

```python
#switch factor testing for every model in BitcoinOTC graph
g = graphs['BitcoinOTC']
for model in propMods:
    print(model[1] + ":\n")
    for test in [0, 0.3, 0.6, 0.9]:
        measureTime2("Switch factor: " + str(test), cascade, g,
                     [1], 50, model[0], 1, False, 0.2, 0.6, test)
    print("")

#switch factor testing for every model in BitcoinOTC graph
g = graphs['BitcoinOTC']
for model in propMods:
    print(model[1] + ":\n")
    for test in [0, 0.05, 0.1, 0.5]:
        measureTime2("Time factor: " + str(test), cascade, g,
                     [1], 50, model[0], 1, False, 0.2, 0.6, 0.5, test)
    print("")

#Printing Methods

#Method 1 - seperate print calls
#"""
def printResults1(msg, lis):
    print(msg)
    print("Mean = " + str(round(np.mean(lis), 5)))
    print("Median = " + str(round(np.median(lis), 5)))
    print("Max = " + str(round(max(lis), 5)))
    print("Min = " + str(round(min(lis), 5)))
    print("Range = " + str(round((max(lis)-min(lis)), 5)))
#"""
#Method 2 - single print call
#"""
def printResults2(msg, lis):
    print(msg)
    print("\nMean = " + str(round(np.mean(lis), 5)) +
          "\nMedian = " + str(round(np.median(lis), 5)) +
          "\nMax = " + str(round(max(lis), 5)) +
          "\nMin = " + str(round(min(lis), 5)) +
          "\nRange = " + str(round((max(lis)-min(lis)), 5)))
#"""
#Method 3 - .join()
#"""
def printResults3(msg, lis):
    print(msg)
    strs = [("Mean = " + str(round(np.mean(lis), 5))),
             ("Median = " + str(round(np.median(lis), 5))),
             ("Max = " + str(round(max(lis), 5))),
             ("Min = " + str(round(min(lis), 5))),
             ("Range = " + str(round((max(lis)-min(lis)), 5)) + '\n')]
    sep = '\n'
    print(sep.join(strs))
#"""

#Functionality Testing
"""
ab = [np.random.randint(0,200) for _ in range(200)]
for c, p in enumerate([printResults1, printResults2, printResults3]):
    p(("Method " + str(c+1)), ab)
#"""
#Results:
```

```
#Means, Medians, Maxs, Mins, Ranges --> all identical

#Time Testing
"""
its = 250
for c, p in enumerate([printResults1, printResults2, printResults3]):
    startT = time()
    for it in range(its):
        #ab = [np.random.randint(0,200) for _ in range(200)]
        p("", [0])
    print("Method " + str(c+1) + ": " + str(time()-startT))
#"""
print("")
#Results: (250 iterations)
#printResults1----0.129
#printResults2----0.067
#printResults3----0.092

#printResults2 is the fastest
#One single print call with '\n'

#Dataset dictionary, needed for graph methods 3 & 4
#"""
#Title : directed, weighted, offset, filepath to .csv file
datasets = {
    #BitcoinOTC dataset (5881 nodes, 35592 edges)
    #(directed, weighted, signed)
    "BitcoinOTC": (True, True,
         r"D:\Sully\Documents\Computer Science BSc\Year 3\Term 2\Individual Pro-
ject\datasets\soc-sign-bitcoinotc.csv"),
    #Facebook dataset (4039 nodes, 88234 edges)
    #(undirected, unweighted, unsigned)
    "Facebook": (False, False,
         r"D:\Sully\Documents\Computer Science BSc\Year 3\Term 2\Individual Pro-
ject\datasets\facebook.csv")
}
#"""

#Used in graph generation
#Removes any unconnected components of a given graph
def removeUnconnected(g):
    components = sorted(list(nx.weakly_connected_components(g)), key=len)
    while len(components)>1:
        component = components[0]
        for node in component:
            g.remove_node(node)
        components = components[1:]

#Network generation methods

#First method - manual
"""
#Generate network from soc-BitcoinOTC dataset
#(5881 nodes, 35592 edges)
#(directed, weighted, signed)
#Initliaise directed graph
G11 = nx.DiGraph(Graph = "BitcoinOTC")
#Open files from path
with open(r"D:\Sully\Documents\Computer Science BSc\Year 3\Term 2\Individual Pro-
ject\datasets\soc-sign-bitcoinotc.csv") as csvfile1:
    #read file and seperate items by comma
```

```
    #  (file is in format: X, Y, W
    #  indicating an edge from node X to node Y with weight W)
    readFile = csv.reader(csvfile1, delimiter=',')
    #for every row in the file...
    for row in readFile:
        #add the edge listed to the graph
        #the edges are reversed to indicate influence
        G11.add_edge((int(row[1])-1), (int(row[0])-1), trust=(int(row[2])+10)/20)
removeUnconnected(G11)

#Generate network from ego-Facebook dataset
#(4039 nodes, 88234 edges)
#(undirected, unweighted, unsigned, no parallel edges)
#Initliaise standard graph
G21 = nx.DiGraph(Graph = "Facebook")
#Open file from path
with open(r"D:\Sully\Documents\Computer Science BSc\Year 3\Term 2\Individual Pro-
ject\datasets\facebook.csv") as csvfile:
    #read file and seperate items by comma
    #  (file is in format: X, Y
    #  indicating an edge from node X to node Y)
    readFile = csv.reader(csvfile, delimiter=',')
    #for every row in the file...
    for row in readFile:
        #add the edge listed to the graph
        #  (edges are reversed to indicate influence)
        G21.add_edge(int(row[1]), int(row[0]), trust=1)
        G21.add_edge(int(row[0]), int(row[1]), trust=1)

#Small, custom directed, unweighted graph
G31 = nx.DiGraph()
testedges = [(1,2), (2,4), (2,5), (2,6), (3,5), (4,5), (5,9), (5,10), (6,8),
             (7,8), (8,9)]
G31.add_edges_from(testedges)
nx.set_edge_attributes(G3, 1, 'trust')
#"""
#Second method - modularized with functions
"""
#Generates NetworkX graph from given file path:
def generateNetwork(name, weighted, directed, offset, path):
    newG = nx.DiGraph(Graph = name)
    with open(path) as csvfile:
        #read file and seperate items by comma
        #  (file is in format: X, Y, W - but may not contain W, indicating an
edge from node X to node Y with weight W)
        readFile = csv.reader(csvfile, delimiter=',')
        for row in readFile:
            tr, dis = 1, 1
            #add the edge listed to the graph (the edges are reversed to indicate
influence, & nodes are added automatically)
            #allow for custom weights in the csv file, distance = weight's recip-
rocal
            if weighted:
                tr = (int(row[2])+10)/20
                dis = 1-tr
            newG.add_edge(int(row[1])-offset, int(row[0])-offset, trust=tr, dis-
tance=dis)
            if not directed:
                newG.add_edge(int(row[0])-offset, int(row[1])-offset, trust=tr,
distance=dis)
        if directed:
```

```
            removeUnconnected(newG)
    return newG

#Generate graphs from real datasets:

#  Generate network graph from soc-BitcoinOTC dataset (5881 nodes, 35592 edges)
#  (directed, weighted, signed)
G12 = generateNetwork("BitcoinOTC Network", True, True, 1, r"D:\Sully\Docu-
ments\Computer Science BSc\Year 3\Term 2\Individual Project\datasets\soc-sign-
bitcoinotc.csv")

#  Generate network from ego-Facebook dataset (4039 nodes, 88234 edges)
#  (undirected, unweighted, unsigned, no parallel edges)
G22 = generateNetwork("Facebook Network", False, False, 0, r"D:\Sully\Docu-
ments\Computer Science BSc\Year 3\Term 2\Individual Project\datasets\face-
book.csv")

#Generate mock graphs for testing and debugging:

#Small, custom directed, unweighted graph
G32 = nx.DiGraph()
testedges = [(1,2), (2,4), (2,5), (2,6), (3,5), (4,5), (5,9), (5,10), (6,8),
             (7,8), (8,9)]
G32.add_edges_from(testedges)
nx.set_edge_attributes(G32, 1, 'trust')

#Medium-sized path graph
#(each node only has edges to the node before and/or after it)
G4 = nx.path_graph(100)
nx.set_edge_attributes(G4, 1, 'trust')

#Medium-sized, randomly generated directed, unweighted graph
G5 = nx.DiGraph(Graph = "G5: random, trust=1")
for i in range(50):
    for j in range(10):
        targ = np.random.randint(-40,50)
        if targ > -1:
            G5.add_edge(i, targ, trust=1)

#Medium-sized, randomly generated directed, randomly-weighted graph
G6 = nx.DiGraph(Graph = "G6: random, randomized trust vals")
for i in range(50):
    for j in range(10):
        targ = np.random.randint(-40,50)
        if targ > -1:
            tru = np.random.uniform(0,1)
            G6.add_edge(i, targ, trust=tru)
#"""

#Third method - modularized, using the itertuples iteration method and
#  dictionaries to allow for additional graphs to be added simply.
#Also, offest no longer needed as it is calculated in function.
#"""
def generateNetwork(name, weighted, directed, path):
    #graph is initialized and named, dataframe is initialized
    newG = nx.DiGraph(Graph = name)
    data = pd.DataFrame()
    #pandas dataframe is read from .csv file,
    #  with weight if weighted, without if not
    if weighted:
        data = pd.read_csv(path, header=None, usecols=[0,1,2],
```

```
                            names=['Node 1', 'Node 2', 'Weight'])
    else:
        data = pd.read_csv(path, header=None, usecols=[0,1],
                            names=['Node 1', 'Node 2'])
        data['Weight'] = 1
    #offset is calculated from minimum nodes
    offset = min(data[['Node 1', 'Node 2']].min())
    #each row of dataframe is added to graph as an edge
    for row in data.itertuples(False, None):
        #trust=weight, & distance=(1-trust)
        trustval = row[2]
        newG.add_edge(row[1]-offset, row[0]-offset,
                      trust=trustval, distance=(1-trustval))
        #if graph is undirected, edges are added again in reverse
        if not directed:
            newG.add_edge(row[0]-offset, row[1]-offset,
                          trust=trustval, distance=(1-trustval))
    #unconnected components are removed
    if directed:
        removeUnconnected(newG)
    return newG
#"""
print("")

#Functionality testing for method 1
#"""
for test in [(1,5), (4,5), (14,0)]:
    present = (test in G1.edges)
    print(str(test) + ": " + str(present))
#"""

test = [(G11, G21, G31), (G12, G22, G32)]
for gc in range(3):
    for graphlist in test:
        print(graphlist[gc].size())
        print(str(len(graphlist[gc])) + "\n")

#Graph compilation methods

#method 2
"""
#All real graphs
graphs = [G1, G2]
#All real graphs with their names attached
namedGraphs = [(G1, 'G1'), (G2, 'G2')]
#All real graphs with their optimal number of iterations
graphits = [(G1, 1000), (G2, 500)]
#All mock graphs
mockGraphs = [G3, G4, G5, G6]
#All mock graphs with their names attached
namedMockGraphs = [(G3, 'G3'), (G4, 'G4'), (G5, 'G5'), (G6, 'G6')]
#Randomly generated mock graphs
rndmGraphs = [G5, G6]
#All directed graphs
diGraphs = [G1, G2, G3, G5, G6]
#All directed graphs with their names attached
namedDiGraphs = [(G1, 'G1'), (G2, 'G2'), (G3, 'G3'), (G5, 'G5'), (G6, 'G6')]
#All graphs - real & mock
allGraphs = [G1, G2, G3, G4, G5, G6]
#"""
#method 3
```

```python
#"""
#Generate graphs and compile into dictionaries:

#Dictionaries for groups of graphs are intialized
graphs, mockGraphs, rndmGraphs, diGraphs, allGraphs = {}, {}, {}, {}, {}

#Generate graphs from real datasets using the datasets dictionary
for g in datasets:
    realGraph = generateNetwork((g + " Network"),
                                datasets[g][0], datasets[g][1],
                                datasets[g][2])
    graphs[g], diGraphs[g], allGraphs[g] = realGraph, realGraph, realGraph


#Generate various mock graphs for testing and debugging:

#Custom, small directed, unweighted graph
mockG, name = nx.DiGraph(), "mock1: Custom, small"
testedges = [(1,2), (2,4), (2,5), (2,6), (3,5), (4,5), (5,9), (5,10), (6,8),
             (7,8), (8,9)]
mockG.add_edges_from(testedges)
nx.set_edge_attributes(mockG, 1, 'trust')
mockGraphs[name], diGraphs[name] = mockG, mockG

#Medium-sized path graph
#(each node only has edges to the node before and/or after it)
mockG, name = nx.path_graph(100), "mock2: Path graph, 100 nodes"
nx.set_edge_attributes(mockG, 1, 'trust')
mockGraphs[name] = mockG

#Medium-sized, randomly generated directed, unweighted graph
mockG, name = nx.DiGraph(), "mock3: Random, trustvals=1"
for i in range(50):
    for j in range(10):
        targ = np.random.randint(-40,50)
        if targ > -1:
            mockG.add_edge(i, targ, trust=1)
mockGraphs[name], rndmGraphs[name], diGraphs[name] = mockG, mockG, mockG

#Medium-sized, randomly generated directed, randomly-weighted graph
mockG, name = nx.DiGraph(), "mock4: Random, trustvals=random"
for i in range(50):
    for j in range(10):
        targ = np.random.randint(-40,50)
        if targ > -1:
            tru = np.random.uniform(0,1)
            mockG.add_edge(i, targ, trust=tru)
mockGraphs[name], rndmGraphs[name], diGraphs[name] = mockG, mockG, mockG
#"""
print("")

#Functional testing for graph method 3
#Print numbers of nodes & edges
#"""
for graphlist in [namedGraphs, namedMockGraphs]:
    for g in graphlist:
        print(g[1] + ": " + str(g[0].size()))
        print(g[1] + ": " + str(len(g[0])) + "\n")

for graphlist in [realGraphs, mockGraphs]:
    for g in graphlist:
```

```
        print(g + ": " + str(graphlist[g].size()))
        print(g + ": " + str(len(graphlist[g])) + "\n")
#"""
print("")

#Functions needed for graph methods 2 & 3, for time testing

def removeUnconnected2(g):
    for component in list(nx.weakly_connected_components(g)):
        if len(component) < 3:
            for node in component:
                g.remove_node(node)

def removeUnconnected(g):
    components = sorted(list(nx.weakly_connected_components(g)), key=len)
    while len(components)>1:
        component = components[0]
        for node in component:
            g.remove_node(node)
        components = components[1:]

#Generates NetworkX graph from given file path:
def generateNetwork(name, weighted, directed, offset, path):
    NG = nx.DiGraph(Graph = name)
    with open(path) as csvfile:
        #read file and seperate items by comma
        #  (file is in format: X, Y, W - but may not contain W, indicating an
edge from node X to node Y with weight W)
        readFile = csv.reader(csvfile, delimiter=',')
        for row in readFile:
            tr, dis = 1, 1
            #add the edge listed to the graph (the edges are reversed to indicate
influence, & nodes are added automatically)
            #allow for custom weights in the csv file, distance = weight's recip-
rocal
            if weighted:
                tr = (int(row[2])+10)/20
                dis = 1-tr
            NG.add_edge(int(row[1])-offset, int(row[0])-offset, trust=tr, dis-
tance=(dis))
            if not directed:
                NG.add_edge(int(row[0])-offset, int(row[1])-offset, trust=tr,
distance=(dis))
        if directed:
            removeUnconnected(NG)
    return NG


#Graphing methods 1 vs 2 time testing
#  Manual --> Modular, slight time improvement
#"""
def compareGraphMethods(its):
    a, b = 0, 0
    for it in range(its):

        startT1 = time()
        #Generate network from soc-BitcoinOTC dataset
        #(5881 nodes, 35592 edges)
        #(directed, weighted, signed)
        #Initliaise directed graph
        G1 = nx.DiGraph(Graph = "BitcoinOTC")
```

```
        #Open files from path
        with open(r"D:\Sully\Documents\Computer Science BSc\Year 3\Term 2\Indi-
vidual Project\datasets\soc-sign-bitcoinotc-EDITED.csv") as csvfile1:
            #read file and seperate items by comma
            #  (file is in format: X, Y, W
            #  indicating an edge from node X to node Y with weight W)
            readFile = csv.reader(csvfile1, delimiter=',')
            #for every row in the file...
            for row in readFile:
                #if the first node is not in the graph already,
                if (int(row[0])-1) not in G1:
                    #add it
                    G1.add_node((int(row[0]))-1)
                #if the second node is not in the graph already,
                if (int(row[1])-1) not in G1:
                    #add it
                    G1.add_node((int(row[1]))-1)
                #add the edge listed to the graph
                #(this happens every row without fail)
                #the edges are reversed to indicate influence
                G1.add_edge((int(row[1])-1), (int(row[0])-1),
trust=(int(row[2])+10)/20)
        removeUnconnected(G1)

        #Generate network from ego-Facebook dataset
        #(4039 nodes, 88234 edges)
        #(undirected, unweighted, unsigned, no parallel edges)
        #Initliaise standard graph
        G2 = nx.DiGraph(Graph = "Facebook")
        #Open file from path
        with open(r"D:\Sully\Documents\Computer Science BSc\Year 3\Term 2\Indi-
vidual Project\datasets\facebook_combined.csv") as csvfile:
            #read file and seperate items by comma
            #  (file is in format: X, Y
            #  indicating an edge from node X to node Y)
            readFile = csv.reader(csvfile, delimiter=',')
            #for every row in the file...
            for row in readFile:
                #if the first node is not in the graph already,
                if int(row[0]) not in G2:
                    #add it
                    G2.add_node(int(row[0]))
                #if the second node is not in the graph already,
                if int(row[1]) not in G2:
                    #add it
                    G2.add_node(int(row[1]))
                #add the edge listed to the graph
                #  (this happens every row without fail, and
                #  the edges are reversed to indicate influence)
                G2.add_edge(int(row[1]), int(row[0]), trust=1)
                G2.add_edge(int(row[0]), int(row[1]), trust=1)
        a += (time()-startT1)

        startT2 = time()
        #  Generate network graph from soc-BitcoinOTC dataset (5881 nodes, 35592
edges)
        #  (directed, weighted, signed)
        G1 = generateNetwork("BitcoinOTC Network", True, True, 1, r"D:\Sully\Doc-
uments\Computer Science BSc\Year 3\Term 2\Individual Project\datasets\soc-sign-
bitcoinotc-EDITED.csv")
```

```
        #  Generate network from ego-Facebook dataset (4039 nodes, 88234 edges)
        #  (undirected, unweighted, unsigned, no parallel edges)
        G2 = generateNetwork("Facebook Network", False, False, 0, r"D:\Sully\Doc-
uments\Computer Science BSc\Year 3\Term 2\Individual Project\datasets\face-
book_combined.csv")
        b += (time() - startT2)

    return (("First manual method: ", a), ("Second modular method: ", b))

#Function to run them a given number of times, and return running times to com-
pare
#1. 64.046   2. 58.004
#Second modular approach is faster, as expected, but not by much.

#print(compareGraphMethods(100))

#Graph method 3 testing different iteration methods
methods = {'index','loc','iloc','iterrows','itertuples'}

#Generates NetworkX graph from given file path with a given method:
def generateNetwork2(name, weighted, directed, offset, path, itermethod):
    #graph is initialized and named, dataframe is initialized
    newG = nx.DiGraph(Graph = name)
    data = pd.DataFrame()

    #pandas dataframe is created from .csv file,
    #  with weight if weighted, without if not
    if weighted:
        data = pd.read_csv(path, header=None, usecols=[0,1,2],
                           names=['Node 1', 'Node 2', 'Weight'])
    else:
        data = pd.read_csv(path, header=None, usecols=[0,1],
                           names=['Node 1', 'Node 2'])
        data['Weight'] = 1

    #offset is calculated from minimum nodes
    offset = min(data[['Node 1', 'Node 2']].min())

    #if graph is undirected, edges are added twice in parallel
    #different iteration methods below:
    #index itermethod
    if itermethod == 'index':
        for i in data.index:
            trustval = data['Weight'][i]
            newG.add_edge(data['Node 2'][i]-offset,
                          data['Node 1'][i]-offset,
                          trust=trustval, distance=1-trustval)
            if not directed:
                newG.add_edge(data['Node 1'][i]-offset,
                              data['Node 2'][i]-offset,
                              trust=trustval,
                              distance=1-trustval)
    #loc ietrmethod
    elif itermethod == 'loc':
        for i in range(len(data)):
            trustval = data.loc[i, 'Weight']
            newG.add_edge(data.loc[i, 'Node 2']-offset,
                          data.loc[i, 'Node 1']-offset,
                          trust=trustval, distance=1-trustval)
            if not directed:
                newG.add_edge(data.loc[i, 'Node 1']-offset,
```

```python
                              data.loc[i, 'Node 2']-offset,
                              trust=trustval, distance=1-trustval)
    #iloc itermethod
    elif itermethod == 'iloc':
        for i in range(len(data)):
            trustval = data.iloc[i, 2]
            newG.add_edge(data.iloc[i, 1]-offset,
                          data.iloc[i, 0]-offset,
                          trust=trustval, distance=1-trustval)
            if not directed:
                newG.add_edge(data.iloc[i, 0]-offset, data.iloc[i, 1]-offset,
                              trust=trustval, distance=1-trustval)

    #iterrows itermethod
    elif itermethod == 'iterrows':
        for i, row in data.iterrows():
            trustval = row['Weight']
            newG.add_edge(row['Node 2']-offset, row['Node 1']-offset,
                          trust=trustval, distance=1-trustval)
            if not directed:
                newG.add_edge(row['Node 1']-offset,
                              row['Node 2']-offset,
                              trust=trustval, distance=1-trustval)

    #itertuples itermethod
    elif itermethod == 'itertuples':
        for row in data.itertuples(False, None):
            trustval = row[2]
            newG.add_edge(row[1]-offset, row[0]-offset,
                          trust=trustval, distance=(1-trustval))
            if not directed:
                newG.add_edge(row[0]-offset, row[1]-offset,
                              trust=trustval, distance=(1-trustval))

    #unconnected components are removed
    if directed:
        removeUnconnected(newG)
    return newG


#Functionality testing function
def itermethodEqual(method1, methodlist, cleared):
    for count, g in enumerate(datasets):
        if not len(cleared[count]):
            cleared[count].append(g + ": ")
        networks = [(generateNetwork2((g + " Network"),
                                      datasets[g][0], datasets[g][1],
                                      datasets[g][2], datasets[g][3],
                                      method1), method1)]
        for method in methodlist:
            if method not in cleared[count]:
                networks.append((generateNetwork2((g + " Network"),
                                                  datasets[g][0],
                                                  datasets[g][1],
                                                  datasets[g][2],
                                                  datasets[g][3],
                                                  method), method))
        missingnodes = [(g + " " + method1 + " missing nodes:")]
        clear = True
        for c, network in enumerate(networks[1:]):
            if set(network[0].nodes) == set(networks[0][0].nodes):
```

```
                continue
            unequal = False
            for node in network[0]:
                if node not in networks[0][0]:
                    if not unequal:
                        missingnodes[c].append(method1 + " - " + network[1])
                        unequal = True
                    missingnodes[c].append(node)
                    clear = False
        if clear:
            cleared[count].append(method1)
            print("Cleared methods so far:\n" + str(cleared[0])
                  + "\n" + cleared[1] + "\n")
        else:
            for l in missingnodes:
                print(l)
            print("")
    return cleared

#Functionality testing area
"""
clearmethods = [[], []]
for method in methods:
    clearmethods = itermethodEqual(method, (methods - set(method)), clearmethods)
#"""
#Results:
#none had equal sets of nodes -> led me to typo in offset
#iloc was unequal to all others -> led me to typo in iloc
#when typos were fixed -> all were identical


#Time testing function to generate all real graphs for a given method
#  and return the time taken to do so + the time taken so far.
def itermethodTime(method, timeSoFar):
    startTime = time()
    testGraphs = {}
    for g in datasets:
        testGraphs[g] = generateNetwork2((g + " Network"),
                                         datasets[g][0], datasets[g][1],
                                         datasets[g][2], datasets[g][3],
                                         method)
    return timeSoFar + (time()-startTime)

#Time testing area - Repeatedly generates graphs for a number of iterations,
#  for each method, and prints the time elapsed for each
"""
for method in methods:
    t = 0
    for i in range(10):
        t = itermethodTime(method, t)
    print(method + " = " + str(t))
#"""
#Results: (10 iterations)
#index--------38.352
#iterrows-----95.219
#loc----------45.991
#itertuples---5.541
#iloc---------130.186
#Itertuples is the fastest by far, so was implemented

print("")
```

```
#Failed section: method 3 for graph iteration method functionality testing
"""
        checknodes = {}
        checkedges = {}
        m1 = methodlist[0]
        test = networks[m1]
        testnodes = {node for node in test}
        testedges = {edge for edge in test}
        for m2 in networks:
            if m2 == networks[m1]:
                continue
            check = networks[m2]
            if m2 == m1:
                print("Same graph\n")
                continue
            for node in check:
                if node not in testnodes:
                    if node not in checknodes[m2 + " " + m1] and node not in
checknodes[m1 + " " + m2]:
                        checknodes[m1 + " " + m2].append(node)
            for edge in check:
                if edge not in testedges:
                    if edge not in checkedges[m2 + " " + m1] and edge not in
checknodes[m1 + " " + m2]:
                        checkedges[m1 + " " + m2].append(edge)

        print(g + ":\n" + "Node dict, key/values pair by pair:\n")
        for nodepair in checknodes:
            print(nodepair + "\n" + str(checknodes[nodepair]) + "\n")
        print("\n" + g + ":\n" +"Edge dict, key/values pair by pair:\n")
        for edgepair in checkedges:
            print(edgepair + "\n" + str(checkedges[edgepair]) + "\n")
"""
"""
            for g2 in networks:
                check = networks[g2]
                checknodes = {node for node in check}
                checkedges = {edge for edge in check}
                if g2 == g1:
                    "Same graph\n"
                #    continue
                for node in networks[g2]:
                    if node not in testnodes:
                        checknodes[(g1 + " " + g2)].append(node)
                if not (testnodes == checknodes):
                    print("Nodes different in " + g + "!\n"
                          + g1 + " - " + g2 + "\n")
                if not (testedges == checkedges):
                    print("Edges different in " + g + "!\n"
                          + g1 + " - " + g2 + "\n")
"""

#Accessing degree recips & rel degs for probabilities

#After deciding on log-scaling -> requires minMaxNormalizing,
#  but that requires knowing the min and max of all logged probs.
#So I implemented a manual method of calculating the normalized prob
#  during the cascade process.
```

```
#Then I improved upon this with a method of assigning them to dictionaries,
#  made other improvements/optimizations and ran multiple tests

#Method 1 - manually while cascading, everytime when needed
#  (requires entire list to be calculated first for mmNormalize)
"""
def mmNormalizeLis(lis):
    elMax, elMin = max(lis), min(lis)
    return list(map(lambda x : ((x - elMin)/(elMax - elMin)), lis))

def allRelDegs(g):
    #allRds = []
    allRds, allRdsDict = [], {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = 0
        for neighbour in g.predecessors(target):
            snd += 1
        for targeting in g.predecessors(target):
            rdval = log(g[targeting][target]['trust']*(g.out_degree(targeting) /
snd))
            allRds.append(rdval)
            allRdsDict[(targeting, target)] = log((g.out_degree(targeting) /
snd))
    return allRds, allRdsDict

relDegsTest1 = allRelDegs(graphs['Facebook'])
#relDegsTest1, relDegsTestDict = allRelDegs(graphs['Facebook'])
elMax, elMin = max(relDegsTest1), min(relDegsTest1)
relDegsTest2 = mmNormalizeLis(relDegsTest1)
#relDegsTest3 = mmNormalizeDict(relDegsTestDict,
#                              max(relDegsTestDict.values()),
#                              min(relDegsTestDict.values()))

def mmNormalizeSingle(val):
    #elMax, elMin = max(relDegsTest1), min(relDegsTest1)
    #normLis = list(map(lambda x: ((x-elMin)/(elMax-elMin)), relDegsTest1))
    #print("Single test normalization:\nMaximum = " + str(elMax) +
    #      "\nMinimum = " + str(elMin) + "Mean = " +
    #      str(np.mean(relDegsTest1)) + "\n")
    return ((val - elMin)/(elMax - elMin))

#printResults("Test list: ", relDegsTest1)
#printResults("Test normalized list: ", relDegsTest2)
#printResults("Test normalized dict: ", list(relDegsTestDict.values()))

#startT = time()
#print("Test normalized dict: " + str(np.mean(list(relDegsTest3.values())))
#             + "\n" + str(time()-startT) + " secs\n")


#Functionality & quality testing of assignment functions
#"""
for assignTest in [0,1]:
    print('assign method: ' + str(assignTest))
    measureTime1(cascade, graphs['Facebook'], [1], 15, 'WC2', assignTest,
                 0.5, 0.7, 0.7, 0.08)
    print("")
#"""
print("")
```

```
#Results: (S=[1], 75its, pp=0.5, qf=0.7, sf=0.7, tf=0.04)
#Manual log-scaling----------1.427 spread, 0.368secs
#Pre-assigned log-scaling----888.773 spread, 77.491secs
#Initially not equal --> typo in allRelDegs (return line indented so no loop)

#Lowered iterations due to it taking so long to process the manual method
#Results: (S=[1])



#"""
#Method 2 - assign to dictionary, at the start of WC1 or WC2
#  3 different functions: logscale, squareroot, cuberoot
"""
#Log-scaling method - default if not specified
def assignRecips1(g):
    drs = {}
    for target in g:
        if not g.in_degree(target):
            continue
        drs[target] = log(1 / g.in_degree(target))
    elMax = drs[max(drs, key=drs.get)]
    elMin = drs[min(drs, key=drs.get)]
    drs = mmNormalizeDict(drs, elMax, elMin)
    nx.set_node_attributes(g, drs, "degRecip")

#Square-rooting method
def assignRecips2(g):
    print("bloop")
    drs = {}
    for target in g:
        if not g.in_degree(target):
            continue
        drs[target] = ((1 / g.in_degree(target)) ** (1/2))
    nx.set_node_attributes(g, drs, "degRecip")

#Cube-rooting method
def assignRecips3(g):
    drs = {}
    for target in g:
        if not g.in_degree(target):
            continue
        drs[target] = ((1 / g.in_degree(target)) ** (1/3))
    nx.set_node_attributes(g, drs, "degRecip")

#Calculate manipulated relational-degrees for all edges in a graph, and
#  assign them as edge attributes for the Weighted Cascade 2 model

#Log-scaling method
def assignRelDegs1(g):
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = 0
        for targeting in g.predecessors(target):
            snd += g.out_degree(targeting)
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = log(g[targeting][tar-
get]['trust']*(g.out_degree(targeting) / snd))
```

```
    #elMax = rds[max(rds, key=rds.get)]
    #elMin = rds[min(rds, key=rds.get)]
    rds = mmNormalizeDict(rds, max(rds.values()), min(rds.values()))
    nx.set_edge_attributes(g, rds, "relDeg")

#Square-rooting method
def assignRelDegs2(g):
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = sum([(g[neighbour][target]['trust']*g.out_degree(neighbour))
                   for neighbour in g.predecessors(target)])
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = (((g[targeting][target]['trust']*g.out_de-
gree(targeting)) / snd) ** (1/2))
    nx.set_edge_attributes(g, rds, "relDeg")

#Cube-rooting method
def assignRelDegs3(g):
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = sum([(g[neighbour][target]['trust']*g.out_degree(neighbour))
                   for neighbour in g.predecessors(target)])
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = (((g[targeting][target]['trust']*g.out_de-
gree(targeting)) / snd) ** (1/3))
    nx.set_edge_attributes(g, rds, "relDeg")

#if,elif,else statement added to cascade
    if model == 'WC1':
        assignRecips1(g)
        success = model
    elif model == 'WC2':
        assignRelDegs1(g)
        success = model
    else:
        success = model
#"""
#Method 3 - method 2 with AssignSelect func, to select which
#  assign function from variable in cascade (logscale=default)
#Same code as Method 2, with the following addition & change to cascade
"""
#Assign method dictionary for selection depending on parameters
assignMods = {'WC1': {1: assignRecips1, 2: assignRecips2, 3: assignRecips3},
              'WC2': {1: assignRelDegs1, 2: assignRelDegs2, 3:assignRelDegs3}}
#Selects and runs appropriate assigning method
def assignSelect(g, propMod, assignMod):
    if assignMod:
        assignMods[propMod][assignMod](g)

#assign paramater added to cascade, along with those 3 lines
def cascade(g, s, it=0, model='IC', assign=1, pp=0.2, qf=1, sf=1, tf=0):
    if model != 'IC' and assign:
        assignSelect(g, model, assign)
    success = model
#"""

#Access/Assign Methods 1 & 3 Functionality&Time Testing
```

```
"""

"""


#Access/Assign Methods 1 & 3 Time Testing

"""

"""
#Results: (S=[1], 75its, pp=0.5, qf=0.7, sf=0.7, tf=0.04)
#Manual log-scaling----------1.427 spread, 0.368secs
#Pre-assigned log-scaling----888.773 spread, 77.491secs
#Initially not equal --> typo in allRelDegs (return line indented so no loop)
#Due to the typo these results are erroneous

#Lowered iterations due to it taking so long to process the manual method
#Results: (S=[1], 75its, pp=0.5, qf=0.7, sf=0.7, tf=0.04)
#Manual log-scaling---------1188.533 spread, 328.085secs
#Pre-assigned log-scaling---1188.533 spread, 13.405secs

#Assign Method 2/3 Optimization

#Optimization for calculating, normalizing & assigning probabilities
#  (log-scaled RelDegs - WC2 here, but applicable to all methods)
#Specifically optimizing the way in which the sum of all a target's
#  neighbours' degrees or maximums/minimums of a dictionary are obtained.

#Sum neighbour degree
#method 1 - Integer & For-loop Method
"""
def assignRelDegs11(g):
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = sum([(g[neighbour][target]['trust']*g.out_degree(neighbour))
                  for neighbour in g.predecessors(target)])
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = log((g[targeting][tar-
get]['trust']*g.out_degree(targeting)) / snd)
    rds = mmNormalizeDict(rds, max(rds.values()), min(rds.values()))
    nx.set_edge_attributes(g, rds, "relDeg")
"""
#method 2 - List Comprehension method
"""
def assignRelDegs12(g):
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = 0
        for neighbour in g.predecessors(target):
            snd += (g[neighbour][target]['trust']*g.out_degree(neighbour))
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = log((g[targeting][tar-
get]['trust']*g.out_degree(targeting)) / snd)
    rds = mmNormalizeDict(rds, max(rds.values()), min(rds.values()))
    nx.set_edge_attributes(g, rds, "relDeg")
"""
```

```
##Dictionary Maximum & Minimum
#method 1 - .values()
"""
def assignRelDegs21(g):
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = 0
        for targeting in g.predecessors(target):
            snd += (g[targeting][target]['trust']*g.out_degree(targeting))
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = log((g[targeting][tar-
get]['trust']*g.out_degree(targeting)) / snd)
    rds = mmNormalizeDict(rds, max(rds.values()), min(rds.values()))
    nx.set_edge_attributes(g, rds, "relDeg")
"""
#method 2 - index and key.get Method
"""
def assignRelDegs22(g):
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = 0
        for targeting in g.predecessors(target):
            snd += (g[targeting][target]['trust']*g.out_degree(targeting))
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = log((g[targeting][tar-
get]['trust']*g.out_degree(targeting)) / snd)
    elMax = rds[max(rds, key=rds.get)]
    elMin = rds[min(rds, key=rds.get)]
    rds = mmNormalizeDict(rds, elMax, elMin)
    nx.set_edge_attributes(g, rds, "relDeg")
"""
#method 3 - itemgetter(1)
"""
def assignRelDegs23(g):
    rds = {}
    for target in g:
        if not g.in_degree(target):
            continue
        snd = 0
        for targeting in g.predecessors(target):
            snd += (g[targeting][target]['trust']*g.out_degree(targeting))
        for targeting in g.predecessors(target):
            rds[(targeting, target)] = log((g[targeting][tar-
get]['trust']*g.out_degree(targeting)) / snd)
    elMax = max(rds.items(), key=itemgetter(1))[1]
    elMin = min(rds.items(), key=itemgetter(1))[1]
    rds = mmNormalizeDict(rds, elMax, elMin)
    nx.set_edge_attributes(g, rds, "relDeg")
"""

#SumNeighbourDegree Time Testing
"""
methods, its = [("SumListComp", assignRelDegs11),
                ("IntegerForLoop", assignRelDegs12)], 20
for method in methods:
    startT = time()
```

```
    for _ in range(its):
        method[1](gr)
    print(method[0] + ": " + str(time()-startT) + " secs")
#"""
#Results: (20 iterations)
#SumListComp-------29.476
#IntegerForLoop----28.361

#IntegerForLoop was marginally quicker, probably due to the lack of
#  creating a new list each time.

#MaxMinDictionary Time Testing
"""
methods, its = [("values()", assignRelDegs21),
                (".get() & index", assignRelDegs22),
                ("itemgetter(1)", assignRelDegs23)], 20
for method in methods:
    startT = time()
    for _ in range(its):
        method[1](gr)
    print(method[0] + ": " + str(time()-startT) + " secs")
"""
print("")
#Results: (20 iterations)
#.values()------------36.638
#.get()&index---------38.029
#itemgetter(1)&[1]----38.418

#.values() was marginally faster than the others

#Printing





#Comparing histograms of normalized probabilites
#Original method
"""
a = calcRelDegs(G1, False)
figs, axs = plt.subplots(1, 2, figsize=(8, 5), sharey=True)
axs[0].hist(a)
axs[0].set(xlabel="Probabilities", ylabel="Base relational degrees")
axs[1].hist(varsList(a))
axs[1].set(xlabel="Probabilities", ylabel="Base relational degrees w/ probability
variables")

b = rootList(a, (1/2))
figs, axs = plt.subplots(1, 2, figsize=(8, 5), sharey=True)
axs[0].hist(b)
axs[0].set(xlabel="Probabilities", ylabel="Square rooted")
axs[1].hist(varsList(b))
axs[1].set(xlabel="Probabilities", ylabel="Square rooted w/ probability varia-
bles")

b = rootList(a, (1/3))
figs, axs = plt.subplots(1, 2, figsize=(8, 5), sharey=True)
axs[0].hist(b)
axs[0].set(xlabel="Probabilities", ylabel="Cube rooted")
axs[1].hist(varsList(b))
axs[1].set(xlabel="Probabilities", ylabel="Cube rooted w/ probability variables")
```

```
b = mmNormalize(a)
figs, axs = plt.subplots(1, 2, figsize=(8, 5), sharey=True)
axs[0].hist(b)
axs[0].set(xlabel="Probabilities", ylabel="min-max normalized")
axs[1].hist(varsList(b))
axs[1].set(xlabel="Probabilities", ylabel="min-max normalized w/ probability var-
iables")

b = zNormalize(a)
figs, axs = plt.subplots(1, 2, figsize=(8, 5), sharey=True)
axs[0].hist(b)
axs[0].set(xlabel="Probabilities", ylabel="z-score normalized")
axs[1].hist(varsList(b))
axs[1].set(xlabel="Probabilities", ylabel="z-score normalized w/ probability var-
iables")

b = robustNormalize(a)
figs, axs = plt.subplots(1, 2, figsize=(8, 5), sharey=True)
axs[0].hist(b)
axs[0].set(xlabel="Probabilities", ylabel="robust normalized")
axs[1].hist(varsList(b))
axs[1].set(xlabel="Probabilities", ylabel="robust normalized w/ probability vari-
ables")

b = logList(a)
figs, axs = plt.subplots(1, 2, figsize=(8, 5), sharey=True)
axs[0].hist(b)
axs[0].set(xlabel="Probabilities", ylabel="logList")
axs[1].hist(varsList(b))
axs[1].set(xlabel="Probabilities", ylabel="logList w/ probability variables")
"""

#Scaled values between 0 and 1
"""
a = calcRelDegs(G1, False)
if max(b) > 1 or min(b) < -1:
    b = mmNormalize(b)
figs, axs = plt.subplots(1, 2, figsize=(8, 5), sharey=True)
axs[0].hist(a)
axs[0].set(xlabel="Probabilities", ylabel="Base relational degrees")
axs[1].hist(varsList(a))
axs[1].set(xlabel="Probabilities", ylabel="Base relational degrees w/ probability
variables")

b = rootList(a, (1/2))
if max(b) > 1 or min(b) < -1:
    b = mmNormalize(b)
figs, axs = plt.subplots(1, 2, figsize=(8, 5), sharey=True)
axs[0].hist(b)
axs[0].set(xlabel="Probabilities", ylabel="Square rooted")
axs[1].hist(varsList(b))
axs[1].set(xlabel="Probabilities", ylabel="Square rooted w/ probability varia-
bles")

b = rootList(a, (1/3))
if max(b) > 1 or min(b) < -1:
    b = mmNormalize(b)
figs, axs = plt.subplots(1, 2, figsize=(8, 5), sharey=True)
axs[0].hist(b)
axs[0].set(xlabel="Probabilities", ylabel="Cube rooted")
```

```
axs[1].hist(varsList(b))
axs[1].set(xlabel="Probabilities", ylabel="Cube rooted w/ probability variables")

b = mmNormalize(a)
if max(b) > 1 or min(b) < -1:
    b = mmNormalize(b)
figs, axs = plt.subplots(1, 2, figsize=(8, 5), sharey=True)
axs[0].hist(b)
axs[0].set(xlabel="Probabilities", ylabel="min-max normalized")
axs[1].hist(varsList(b))
axs[1].set(xlabel="Probabilities", ylabel="min-max normalized w/ probability var-
iables")

b = zNormalize(a)
if max(b) > 1 or min(b) < -1:
    b = mmNormalize(b)
figs, axs = plt.subplots(1, 2, figsize=(8, 5), sharey=True)
axs[0].hist(b)
axs[0].set(xlabel="Probabilities", ylabel="z-score normalized")
axs[1].hist(varsList(b))
axs[1].set(xlabel="Probabilities", ylabel="z-score normalized w/ probability var-
iables")

b = robustNormalize(a)
if max(b) > 1 or min(b) < -1:
    b = mmNormalize(b)
figs, axs = plt.subplots(1, 2, figsize=(8, 5), sharey=True)
axs[0].hist(b)
axs[0].set(xlabel="Probabilities", ylabel="robust normalized")
axs[1].hist(varsList(b))
axs[1].set(xlabel="Probabilities", ylabel="robust normalized w/ probability vari-
ables")

b = logList(a)
if max(b) > 1 or min(b) < -1:
    b = mmNormalize(b)
figs, axs = plt.subplots(1, 2, figsize=(8, 5), sharey=True)
axs[0].hist(b)
axs[0].set(xlabel="Probabilities", ylabel="logList")
axs[1].hist(varsList(b))
axs[1].set(xlabel="Probabilities", ylabel="logList w/ probability variables")
"""

#Modular approach
"""

"""

#Previous methdos for variable comparison / graph plotting:
#Manual approach
"""
qty, its = 5, 50

for g in graphs:
    S = randomSeeds(graphs[g], qty)
    print(g + "\nseed set: " + str(S) + "\n")
    for propMod in propMods:
        res, t = measureTimeRet(cascade, graphs[g], S, its, propMod[0])
        print(propMod[1] + " " + str(its) + " iterations")
        print(str(res) + " " + str(t) + " secs")
    print("\n")
```

```
qty, its = 8, 250

S = randomSeeds(G1, qty)
print(str(G1.graph) + "\nseed set: " + str(S) + "\n")
g1values = [cascade(G1, S, its, 'IC', qf=q*0.1) for q in range(0, 11, 1)]
print("G1 results: " + str(g1values) + "\n")

S = randomSeeds(G2, qty)
print(str(G2.graph) + "\nseed set: " + str(S) + "\n")
g2values = [cascade(G2, S, its, 'IC', qf=q*0.1) for q in range(0, 11, 1)]
print("G2 results: " + str(g2values) + "\n")

#g1values = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100]
#g2values = [50, 100, 150, 200, 250, 300, 460, 600, 720, 900, 1050]
xValues = [x*0.1 for x in range(0, 11, 1)]

fig, axs = plt.subplots(figsize=(10,6))
axs.set_xlabel("Quality factor")
axs.set_ylabel("Spread")
axs.set_title("Effect of quality factor on spread within a network")
axs.plot(xValues, g1values, label="G1: Bitcoin")
axs.plot(xValues, g2values, label="G2: Facebook")
axs.legend()

plt.show()
#"""
#Modular approach
"""#
#Compare positive influence spreads for a given list of graphs with
#  a range of different quality factors, and plot a line graph to show.
def comparePP(gs, qty, its, seedFunc, model, pps):
    values = []
    for i,g in enumerate(gs):
        startTime = time()
        S = seedFunc(gs[g], qty)
        print(g + "\nseed set: " + str(S) + "\n" +
              str(round((time() - startTime), 5)) + " secs\n")
        startTime = time()
        values.append([cascade(gs[g], S, its, model, pp=p) for p in pps])
        print(g + ":\n" + str(values[i]) + "\n" +
              str(round((time() - startTime), 5)) + " secs\n")
    figs, axs = plt.subplots(figsize=(10,6))
    axs.set_xlabel("Propagation probability")
    axs.set_ylabel("Spread")
    axs.set_title("Effect of propagation probability on spread within a net-
work\n" +
                  "Cascade model: " + model)
    for i,g in enumerate(gs):
        axs.plot(pps, values[i], label=g)
    axs.legend()
    plt.show()

comparePP(graphs, 4, 1000, randomSeeds, 'IC', [pp*0.05 for pp in range(1,20)])

#Compare positive influence spreads for a given list of graphs with
#  a range of different quality factors, and plot a line graph to show.
def compareQF(gs, qty, its, seedFunc, model, qfs, sw):
    values = []
    for i,g in enumerate(gs):
        startTime = time()
```

```
        S = seedFunc(gs[g], qty)
        print(g + "\nseed set: " + str(S) + "\n" +
              str(round((time() - startTime), 5)) + " secs\n")
        startTime = time()
        values.append([cascade(gs[g], S, its, model, qf=q, sf=sw) for q in qfs])
        print(g + ":\n" + str(values[i]) + "\n" +
              str(round((time() - startTime), 5)) + " secs\n")
    figs, axs = plt.subplots(figsize=(10,6))
    axs.set_xlabel("Quality factor")
    axs.set_ylabel("Spread")
    axs.set_title("Effect of Quality & Switch Factors\n" +
                  "Switch factor = " + str(sw))
    for i,g in enumerate(gs):
        axs.plot(qfs, values[i], label=g)
    axs.legend()
    plt.show()

for sw in [b*0.1 for b in range(1)]:
    compareQF(graphs, 5, 2, randomSeeds, 'IC', [q*0.1 for q in range(0,11,1)],
sw)
#compareQF(graphs, 8, 50, randomSeeds, 'WC1', [q*0.1 for q in range(0,11,1)])
#compareQF(graphs, 8, 50, randomSeeds, 'WC2', [q*0.1 for q in range(0,11,1)])

#Compare positive influence spreads for a given list of graphs with
#  a range of different switch factors, and plot a line graph to show.
def compareSF(gs, qty, its, seedFunc, model, sfs, qual):
    values = []
    for i,g in enumerate(gs):
        startTime = time.time()
        S = seedFunc(g, qty)
        print(str(g.graph) + "\nseed set: " + str(S) + "\n" +
              str(round((time.time() - startTime), 5)) + " secs\n")
        startTime = time.time()
        values.append([cascade(g, S, its, model, qf=qual, sf=sw) for sw in sfs])
        print(str(g.graph) + ":\n" + str(values[i]) + "\n" +
              str(round((time.time() - startTime), 5)) + " secs\n")
    figs, axs = plt.subplots()
    axs.set_xlabel("Switch factor")
    axs.set_ylabel("Spread")
    axs.set_title("Effect of switch factor on spread within a network\n" +
                  "Cascade model: " + model + ". Quality factor: " + str(qual))
    for i,g in enumerate(gs):
        axs.plot(sfs, values[i], label=str(g.graph))
    axs.legend()
    plt.show()

#Compare positive influence spreads for a given list of graphs with
#  a range of different switch factors, and plot a line graph to show.
def compareSF2(g, s, its, sfs, qfs, col):
    values, labels = [], [str(q) for q in qfs]
    #values, labels = [[] for _ in range(len(qfs))], [str(q) for q in qfs]
    for q in (range(len(qfs))):
        #print("Quality factor: " + str(qfs[q]))
        for sw in range(len(sfs)):
            startTime = time()
            values.append(cascade(graphs['BitcoinOTC'], S, its, pp=0.6,
qf=qfs[q], sf=sfs[sw]))
            #print("Switch factor: " + str(sfs[sw])
            #      + ":\n" + str(values[q]) + "\n" +
            #      str(round((time() - startTime), 5)) + " secs\n")
    figs, axs = plt.subplots()
```

```
    for q in range(len(qfs)):
        axs.plot(sfs, values, label=g, color=col)
    axs.set_xlabel("Switch factor")
    axs.set_ylabel("Spread")
    axs.set_title(g + "\nQuality factor: " + str(qual))
    axs.legend()

compareSF(graphs, 8, 25, randomSeeds, 'IC', [sf*0.1 for sf in range(0,11,1)],
0.8)
compareSF(graphs, 8, 50, randomSeeds, 'WC1', [sf*0.1 for sf in range(0,11,1)],
0.8)
compareSF(graphs, 8, 50, randomSeeds, 'WC2', [sf*0.1 for sf in range(0,11,1)],
0.8)
compareSF(graphs, 8, 50, randomSeeds, 'IC', [sf*0.1 for sf in range(0,11,1)],
0.5)
compareSF(graphs, 8, 50, randomSeeds, 'WC1', [sf*0.1 for sf in range(0,11,1)],
0.5)
compareSF(graphs, 8, 50, randomSeeds, 'WC2', [sf*0.1 for sf in range(0,11,1)],
0.5)
compareSF(graphs, 8, 50, randomSeeds, 'IC', [sf*0.1 for sf in range(0,11,1)],
0.3)
compareSF(graphs, 8, 50, randomSeeds, 'WC1', [sf*0.1 for sf in range(0,11,1)],
0.3)
compareSF(graphs, 8, 50, randomSeeds, 'WC2', [sf*0.1 for sf in range(0,11,1)],
0.3)

#Compare positive influence spreads for a given list of graphs with
#  a range of different time factors, and plot a line graph to show.
def compareTF(gs, qty, its, seedFunc, model, tfs):
    values = []
    for i,g in enumerate(gs):
        startTime = time.time()
        S = seedFunc(g, qty)
        print(str(g.graph) + "\nseed set: " + str(S) + "\n" +
              str(round((time.time() - startTime), 5)) + " secs\n")
        startTime = time.time()
        values.append([cascade(g, S, its, model, tf=t) for t in tfs])
        print(str(g.graph) + ":\n" + str(values[i]) + "\n" +
              str(round((time.time() - startTime), 5)) + " secs\n")
    figs, axs = plt.subplots()
    axs.set_xlabel("Time factor")
    axs.set_ylabel("Spread")
    axs.set_title("Effect of time factor on spread within a network\n" +
                  "Cascade model: " + model)
    for i,g in enumerate(gs):
        axs.plot(tfs, values[i], label=str(g.graph))
    axs.legend()
    plt.show()

compareTF(graphs, 8, 50, randomSeeds, 'IC', [tf*0.01 for tf in range(0,11,1)])
compareTF(graphs, 8, 50, randomSeeds, 'WC1', [tf*0.01 for tf in range(0,11,1)])
compareTF(graphs, 8, 50, randomSeeds, 'WC2', [tf*0.01 for tf in range(0,11,1)])
"""
#Unfinished comparison function
"""
def compareVariables(g, s, its, compareVars, compare):
    startTT, values = time(), []
    for c1, var1 in enumerate(compareVars[0]):
        for c2, var2 in enumerate(compareVars[1]):
            for c3, var3 in enumerate(compareVars[2]):
                startT = time()
```

```
                casc = cascade(gs[g], s, its, pp=var1[1], qf=var2[1], sf=var3[1])
                endT = round((time()-startT), 5)
                print(str(c1+c2+c3+1) + " cascades completed so far!\n" +
                      str(round((time()-startT), 5)) + " secs\n" +
                      str(round((time()-startTT), 5)) + " secs total\n")
                values.append(())

            measureTimeRet2(cascade, g, S, its)

def plotComparison(vals, cols):
    labels = [str(u*0.1) for u in range(11)]
    figs, axs = plt.subplots(figsize=(12,6))
    for v in range(len(vals)):
        axs.plot(vals[v], label="QF=" + labels[v], color=cols[v])
    axs.legend()
#"""
print("")

#seed selection model comparison unfinished/erroneous methods

#old method of comparing seed selection models' printed results
"""
#Selects seeds with a given model, uses those seeds with each cascade model
#  and prints the resultant spreads.
def compareSeedSelMods(qty, seedMods):
    for g in graphits:
        doneSeeds = set()
        for seedMod in seedMods:
            S, t = measureTimeRet(seedMod[0], g[0], qty)
            print(str(g[0].graph) + " " + seedMod[1] +
                  "\n" + str(S) + " " + str(t) + " secs\n")
            found = False
            for check in doneSeeds:
                if S in check:
                    print(seedMod[1] + " has the same seed set as " + check[1])
                    found = True
            doneSeeds.add((frozenset(S), seedMod[1]))
            if not found:
                for propMod in propMods:
                    try:
                        measureTime2(propMod[1], propMod[0], g[0], S, g[1])
                    except Exception as e:
                        print(e)

qty = 8
seedMods = [(degDiscSeeds1, "degreeDiscount1"), (degDiscSeeds2, "degreeDis-
count2"),
            (degCSeeds, "degreeCentrality"), (inDegCSeeds, "inDegreeCentrality"),
            (outDegCSeeds, "outDegreeCentrality"), (ccSeeds, "ClosenessCentral-
ity"),
            (infCSeeds, "infoCentrality"), (btwnCSeeds, "BetweennessCentrality"),
            (approxCfBtwnCSeeds, "approxCF-BetweennessCentrality"), (loadCSeeds,
"loadCentrality"),
            (evCSeeds, "eigenvector"), (kCSeeds, "katz"),
            (subgCSeeds, "subgraph"), (harmCSeeds, "harmonic"),
            (voteRankSeeds, "voteRank"), (pageRankSeeds, "pageRank"),
            (hitsHubSeeds, "HITS Hubs"), (hitsAuthSeeds, "HITS Auths")]
compareSeedSelMods(qty, seedMods)

#Special case for mixed greedy models
#25 iterations for MixedGreedy 1.1, 1.2, 2.1 & 2.2 took a very long time
```

```
#  in G2, so it was run with both 10 and 25 iterations for comparison.
qty, its1, its2 = 8, 25, [10, 25]
graphits1, graphits2 = [(G1, 1000)], [(G2, 500)]
seedMods = [(mixedGreedy11, "Mixed1.1"), (mixedGreedy12, "Mixed1.2"),
            (mixedGreedy21, "Mixed2.1"), (mixedGreedy22, "Mixed2.2")]

#Seed selection and propagation with those seeds for MixedGreedy models in G1
for g in graphits1:
    for seedMod in seedMods:
        S, t = measureTimeRet(seedMod[0], g[0], qty, its)
        print(str(g[0].graph) + " " + seedMod[1] +
              "\n" + str(S) + " " + str(t) + " secs\n")
        for propMod in propMods:
            measureTime2(propMod[1], propMod[0], g[0], S, g[1])

#Seed selection and propagation with those seeds for MixedGreedy models in G2
its = [10, 25]
for g in graphits2:
    for it in its:
        for seedMod in seedMods:
            S, t = measureTimeRet(seedMod[0], g[0], qty, it)
            print(str(g[0].graph) + " " + seedMod[1] + " " + str(it) +
                  " iterations\n" + str(S) + " " + str(t) + " secs\n")
            for propMod in propMods:
                measureTime2(propMod[1], propMod[0], g[0], S, g[1])
#"""
#unfinished comparison of all models on one graph
"""
def compareAllBar(lis, vals):
    labels = [label[1] for label in lis[1]]
    fig, ax = plt.subplots()
    y = np.arange(len(lis[1]))
    height = 0.4
    ax.grid(zorder=0)
    spreads = ax.barh(y - height/2, seedModels[0], height=height,
                      label='Spread', zorder=3)
    spreadsT = ax.barh(y + height/2, seedModels[1], height=height,
                       label='Spread / (Time*0.1)', zorder=3)

    ax.set_xlabel('Spread')
    ax.set_yticks(y)
    ax.set_yticklabels(lis[1])
    ax.set_title('Spreads of various models')
    ax.legend(loc=0)

    fig.tight_layout()

allSeeds = priorSeeds + netSeeds + ogSeeds
prepareBar(allSeeds, rndmGraphs)
allSeeds = (degDiscSeeds, 'degDisc') + netSeeds
#"""
#old comparison methods that were improved upon
"""
x, y = ['Graphs', ['ogGreedy','celf','impGreedy','degDisc']], ['Spread', []]
gtest, seedsels = graphs['BitcoinOTC'], [ogGreedySeeds(mockGraphs["mock4-ran-
dom2"],  4, 100, 'IC'),
                                          celfSeeds(mockGraphs["mock4-random2"],
4, 100, 'IC'),
                                          impGreedySeeds(mockGraphs["mock4-ran-
dom2"], 4, 100),
```

```
                                          degDiscSeeds(mockGraphs["mock4-ran-
dom2"], 4)]
for s in range(len(seedsels)):
    st = time()
    y[1].append(cascade(mockGraphs["mock4-random2"], seedsels[s], 1000))
    print(x[1][s] + " = " + str(round((time()-st), 4)))
vertBar(x, y, "Seed Select Models")

x = ['Models', ]
qty, its, model, its2, timeFactor = 4, 1000, 'IC', 1000, 1

for g in mockGraphs:
    y, seedTimes = ['Spread', []], []
    for c, seedSel in enumerate(x[1]):
        if c > 2:
            t = time()
            seeds = seedSel[0](mockGraphs[g], qty)
            t = time() - t
            if t < 0.001:
                t = 0.001
            seedTimes.append(t)
        elif c > 1:
            t = time()
            seeds = seedSel[0](mockGraphs[g], qty, its2)
            t = time() - t
            if t < 0.001:
                t = 0.001
            seedTimes.append(t)
        else:
            t = time()
            seeds = seedSel[0](mockGraphs[g], qty, its, model)
            t = time() - t
            if t < 0.001:
                t = 0.001
            seedTimes.append(t)
        y[1].append(cascade(mockGraphs[g], seeds, its))

    xLabels = ['Models', [seedMod[1] for seedMod in x[1]]]
    print(y)
    vertBar(xLabels, y, (g + " Seed Select Models: Spread"))

    print(seedTimes)
    for seedSpread in range(len(y[1])):
        y[1][seedSpread] = y[1][seedSpread] / (timeFactor * seedTimes[seed-
Spread])
    print(y)
    vertBar(xLabels, y, (g + " Seed Select Models: Spread / Time"))


#"""

#Plotting bar charts for seed selection models

#Vertical bar chart for each seed select model on one graph
"""
def vertBar(lis, vals, msg):
    #return nothing if lists aren't same size
    if len(lis[1]) != len(vals[1]):
        print("Error, not the same size")
        return
    #subplot set up, gridlines drawn, max value calculated and y-limits set
```

```
    fig, ax = plt.subplots(1, 1, figsize=(16,len(vals[1])))
    ax.grid(zorder=0)
    topVal = max(vals[1])
    ax.set_ylim([0, topVal*1.25])
    #bar chart plotted
    bars = ax.bar(lis[1], vals[1], width=0.4, facecolor='lightsteelblue',
                  edgecolor='black', linewidth=2.5, zorder=3)
    #ax.bar_label(bars, fmt='%.3f')
    #Subtitle, x-labels & y-labels are set for each axis
    ax.set_xlabel(lis[0], fontsize=20)
    ax.set_ylabel(vals[0], fontsize=20)
    ax.tick_params(axis='both', labelsize=15)
    #Titles are set and the layout (incl. padding/gaps) is set and adjusted
    fig.tight_layout(pad=5)
    fig.suptitle(msg + " Comparison:", fontsize=24, fontweight='bold')
    fig.subplots_adjust(top=0.88)
#"""

#Time testing simple inequality functions, as practice for
#  timing other functions

"""
#Time function
def timeFunc(func, its, a, b, count):
    startTime = time()
    for _ in range(its):
        count = func(a, b, count)
    return count, (time() - startTime)

#Method1
def notEq1(a, b, count):
    if a != b:
        count += 1
    return count

#Method2
def notEq2(a, b, count):
    if not a == b:
        count += 1
    return count

#Method3
def notEq3(a, b, count):
    if a is not b:
        count += 1
    return count

#Method4
def notEq4(a, b, count):
    if a == b:
        cdefg=None
    else:
        count+=1
    return count

#Dictionary for results
eqMethod = {}
for i in range(4):
    eqMethod[i+1] = [['True', 0],
                     ['False', 0],
                     ['Uniterable', 0],
```

```
                         ['Iterable', 0]]
#Testing loops
for num1, (aa,bb) in enumerate([(1200, 7),
                                    ("Blah", "Yoyoyoyoyoyoyoyo"),
                                    (None, True),
                                    ([1,2,3],[7,8,9]),
                                    ({1,2,3,4,5}, {4,5,6,7}),
                                    (('abc', 123), ('abc', 125))]):
    for x in range(2):
        #print("Params #" + str(num1+1) + " " + str(aa==bb) + ":\n"
        #      + str(aa) + " " + str(bb) + "\n")
        for num2, f in enumerate([notEq1, notEq2, notEq3, notEq4]):
            qty = 0
            qty, t = timeFunc(f, 10000000, aa, bb, qty)
            index = 0
            if x:
                index += 1
            if num1 > 1:
                index += 2
            eqMethod[num2+1][index][1] += t
        bb = aa

#Print results
for a in eqMethod:
    print("Method " + str(a))
    print(eqMethod[a][0])
    print(eqMethod[a][1])
    print(eqMethod[a][2])
    print(eqMethod[a][3])
    print('\n')
for b in eqMethod:
    print("Types of tests:")
    print(eqMethod[0][b])
    print(eqMethod[1][b])
    print(eqMethod[2][b])
    print(eqMethod[3][b])
"""
print("")
```