Influence: Deconstructed

Influence Maximization in Large Networks

with Interchangeable Positive and Negative Influence

Table of Contents

Chapter 1: Introduction

1.1 Project Overview

This project explored the problem of influence maximization (IM), analysing previous solutions to gain a better understanding and aid in formulating original solutions. The problem is essentially where within a social network should original ideas or advertisements be placed to achieve the maximum resultant spread of positive influence, and what actions, models or strategies would best help to find these optimal locations?

1.1.1 Problem Description

Social networks play an increasingly important role in society, through which opinions, ideas and information spread. IM is an algorithmic extension of social network analysis, considered an optimization problem by computer scientists, that also has wide-reaching applications and implications.

The problem is as follows: how can the spread of influence through a network be best maximised, and what factors are directly correlated. Furthermore, how can simulations represent social networks, and what conclusions could possibly be drawn, if any. More specifically: how can a set of seed nodes of size *k* be found, to maximise propagation through a network with those nodes.

It was formulated as an approximation problem (Domingos and Richardson, 2001), and has been increasingly common in research since, with opportunities of practical application growing.

It was also proven to be NP-hard (Kempe, Kleinberg and Tardos, 2003), meaning it is suspected to be impossible to solve in polynomial time. This dynamic contributes to it being so interesting and well-studied.

The IM function was also proven to be submodular and monotone (Leskovec *et al.*, 2007), and these properties were used to further optimize approximations. Submodular functions have diminishing returns of additional input values – in larger seed sets, extra seed nodes provide less marginal spread. Monotone functions preserve the given order of input values – in this case, nodes.

1.2 Aim and Objectives

The aim of this project is to develop an optimized solution for IM, and in doing so gain a deeper understanding of how influence propagates through a network, what factors affect it most and what insights that reveals into human society and interpersonal interaction.

1.2.1 Main Objectives

1. Implement, analyse and experiment with influence-maximizing seed selection models, and develop an original model that outperforms existing models.
2. Analyse and evaluate parameters involved with influence propagation, their correlations and the insights they provide, and implement an original parameter to further develop those insights.

1.2.2 Subobjectives

1. Implement a general method for generating network graphs from datasets and different models to simulate influence propagation through them.
2. Ensure network generation is generalised for reproducibility, and easy extension to other network graphs using other dataset files.
3. Analyse propagation parameters and implement some original ones, to better simulate social networks.
4. Analyse and implement different seed selection models.
5. Perform statistical data analysis on networks, comparing their properties to influence spreads.
6. Implement a method of comparing propagation spreads and times with different networks, parameters and seed sets.
7. Plot different types of graphs, to analyse and present different propagation models, propagation parameters and seed selection models, in regard to spread and time.
8. Using established literature, evaluate and interpret the results, providing insight into social interaction, and discuss accompanying implications.
9. Gain a deeper understanding of Python's research capabilities, and software regarding network graphs, using NetworkX.

1.2.3 Research Questions

1. What parameters affect the spread of influence most in simulations and do these translate to real-life?
2. What are the best approximation strategies for IM?
3. How could propagation models better reflect real social networks, and what limitations restrict them?
4. How accurately can simulations reflect real-life networks, and how could irrationality be introduced to a model?

1.2.4 Research Hypotheses

1. Networks with more nodes will take longer to perform propagation on, accounting for the spread.
2. In a model where positive and negative influences occur and can 'steal' influence from the other, the more likely nodes are to switch influence, the influence with the higher general probability will increase.
3. New seed selection models based solely on network analysis metrics will outperform existing research models.

A substantial change in how the project was conducted compared to the PDD, was the absence of network visualization. While planning on utilising it throughout, it was decided that it was not the main focus of the research and should not be prioritised, although given more time it would have provided a complementary visual aid.

1.3. Anticipated Beneficiaries

There are a great many different groups of people who are currently interested in IM, with mathematicians and computer scientists studying the problem analytically, advertisers seeking to maximise profits or outreach by way of refined viral marketing campaigns (Richardson and Domingos, 2002), and recently even doctors exploring how research could be applied to pandemics (Blower and Go, 2011), and even maximise vaccine rollouts to the general public.

The scope of people who could potentially benefit from IM research or application in the future is incredibly far-reaching, and only growing with time. Those who could be directly positively affected by IM include:

- Businessowners / Advertisers looking to reach as many potential customers as possible with limited capital to maximise success. Since IM maximises outreach while minimising costs, it makes advertising cheaper for everyone – effectively levelling the playing field. This especially applies to entrepreneurs and smaller business owners.
- Brand experts and social media influencers would directly benefit from gaining understanding about the spread through a social network, allowing them to increase or maximise their own gains.
- Individuals looking to research further into the problem of IM for academic purposes – mathematical, computing, sociological or otherwise.
- Individuals interested in learning more about how ideas or influences spread.

1.4 Assumptions & Limitations

1.4.1 Assumptions

Multiple assumptions were made during research and should be considered when viewing results. Firstly, it is assumed the anonymised datasets acquired from SNAP (Section 4.3.1) are accurate, as all results and conclusions hinge on their authenticity.

Furthermore, assuming their accuracy, a further assumption is made regarding whether connections in these social networks genuinely constitute potential for influence. This does not account for a user only being influenced by a handful of their online friends/following/etc. This is especially true on undirected and unweighted graphs, where every node and edge are treated as equally influential, further abstracting the problem away from real human online interactions and relationships.

Finally, there is an assumption that all users/nodes in the network are active and does not account for inactive users being documented. This will be particularly impactful if the inactive node in question is relatively influential (in a well-connected position or has strong edge weights).

1.4.2 Limitations

This project, considering the computationally expensive nature of an NP-hard approximation problem (thousands of iterations are desirable), was severely limited by the single laptop machine used, as many research papers in the field had access to supercomputers and other machines with higher levels of performance.

Chapter 2: Output Summary

Output 2.1: Cascade Program
Program capable of generating network graphs from .csv files, performing various propagation models on them, and analysing propagation parameters with plotted graphs.
Fully functional Python script: 533 lines of code (excluding comments)
Intended for IM researchers or people interested in IM. Facilitates reproduction or further development of my implementation. Produces further results (output 2.8).
Results: Sections **5.1**, **5.2** and **5.5**, Appendix **C.1**

Output 2.2: Network Analysis Program
Programs capable of generating network graphs, performing statistical analysis on them, normalizing network properties in various ways and comparing results through graph plotting.
Fully functional Python script: 531 lines of code (excluding comments)
Intended for IM researchers, or anyone interested in social network analysis. Facilitates reproduction or further development of my analysis. Produces further results (outputs 2.4 & 2.5).
Results: Section **5.3**, Appendix **C.2**

Output 2.3: Seed Selection Program
Program capable of generating network graphs, fine-tuning seed selection model parameters, selecting seeds through various models and comparing influence spreads of different models through graph plotting.
Fully functional Python scripts: 1136 lines of code (excluding comments).
Intended for IM researchers or people interested in IM. Facilitates reproduction or further development of my implementation. Produces further results (outputs 2.6 & 2.7).
Results: Sections **5.4** and **5.6**, Appendix **C.3**

Output 2.4: Normalization Technique Comparison Graphs
Histograms and pie charts displaying distributions of degree reciprocals and relational degrees when various normalization techniques are applied.
Plotted pie charts, histograms and line graphs
Intended for myself to analyse the distribution of probabilities and determine which normalization technique best suited the datasets and propagation models and should be applied.
Results: Section **5.3.1**

Output 2.5: Network-wide Properties Comparison Graphs
Bar charts comparing various network-wide properties of different networks
Plotted bar charts
Intended for developers of project to aid analysis and comparison of networks.
Results: Section **5.3.2**, Appendix **D.3**

Output 2.6: Fine-tuned Seed Selection Parameters
List of values collated in a spreadsheet, denoting different propagation spreads from different sets of values of parameters in NetworkX seed selection models.
Conditionally formatted (colour) Excel spreadsheet (list of values)
Intended for developers of project to determine optimal values for the NetworkX seed selection models, before finalising their implementation.
Results: Section **5.4.3**, Appendix **D.4**

Output 2.7: Seed Selection Model Comparison Graphs
Horizontal bar charts comparing the spreads and (spread/time)s of different seed selection models, on random and real graphs separately.
Plotted horizontal bar charts
Intended for IM researchers, people interested in maximising their own influence or people studying graphing possibilities in Python. Compares results of various seed selection models.
Results: Section **5.6**

Output 2.8: Propagation Parameter Comparison Graphs
Line graphs comparing influence spreads, varying the model and the propagation parameters used.
Plotted line graphs
Intended for myself or anyone who in future will work on this project. Used to modify NetworkX seed selection models with optimal parameters, improving effectiveness of models.
Results: Section **5.7**

Chapter 3: Literature Review

3.1 Influence Maximization

Past solutions and variations of the IM problem were studied to develop an understanding of the problem, including seed selection techniques, datasets used and possible variations of the problem through additional parameters to improve insightful conclusions.

Kempe, Kleinberg and Tardos

The first formal solution to the approximation problem of IM that is studied today was formulated by (Kempe, Kleinberg and Tardos, 2003). Building on Domingos and Richardson's work (2001, 2002), they developed the greedy solution, which is used as a benchmark for all future solutions. It achieved a high level of approximation but was inefficient and failed to scale well to large networks, (described in Section 4.6.1), due to the nature of its calculations. They also proved the problem to be NP-hard, setting the limits of all future solutions and increasing mathematical and computer-science related interest, as proving NP-hard problems can be solved in polynomial time is of great interest. This paper employed a collaboration graph, arguing that 'co-authorship networks capture many of the key features of social networks more generally' – *High-energy physics theory Section of the e-print arXiv (10,748 nodes, 53,000 edges including parallel edges).*

Leskovec *et al*

One of the earliest significant improvements to Kempe's greedy approach was the CELF (cost-effective lazy-forward) solution (Leskovec *et al.*, 2007), which proved the submodularity of the problem and used that property to improve efficiency of approximation with no lack of quality guarantee whatsoever.

The lead scientist involved in that paper – Leskovec – also designed SNAP (Stanford Network Analysis Program), a publicly accessible library of large networks for network analysis, from which the datasets in this project are sourced. Specifically, the BitcoinOTC dataset used in this project was created by Leskovec himself through surveys in another paper he proposed, through surveys. Leskovec is a long-time contributor in the field of IM, and his influence on this project cannot be understated.

Chen, Wang and Yang / Chen *et al*

A further improvement to the greedy and CELF solutions were the improved greedy and heuristic solutions provided in (Chen, Wang and Yang, 2009). The heuristic solution being so significant that it is the last major improvement despite being over a decade ago, improving efficiency by over a factor of 200. The paper used two 'academic collaboration networks from … two different Sections of arXiv … the same source used in the study by Kempe et al (2003)' – *High Energy Physics – Theory (15,233 nodes, 58,891 edges) and the full paper list of the Physics Section (37,154 nodes, 231,584 edges)*

Chen was also primarily involved in another paper (Chen *et al.*, 2011), where a quality factor was introduced to enable both positive and negative influence, expanding the boundaries of potential conclusive insights. This was directly implemented into this project and was used as inspiration for additional propagation parameters to be added to the model, improving realism and opportunities for practical application. This paper used 'three real-world networks of increasing sizes' – *The high-energy physics theory Section of the e-print arXiv (15,000 nodes, 31,000 edges including parallel edges), The WikiVote Wikipedia voting history network (7,000 nodes, 101,000 edges), and the Epinions product review network.* For the latter two, the edges were reversed to represent influence as they interpreted '*v* voting/trusting *u* as *u* having an influence on *v*', exemplifying excellent dataset preparation to better represent social networks of influence.

3.2 Social Network Analysis

Before the formal definition of IM, social network analysis has long been studied by sociologists, whose social understanding of networks can provide useful insights when discussing results or proposing applications. Analysis of organizational structure (Tichy, Tushman and Fombrun, 1979) or contemporary social science research (Borgatti *et al.*, 2009) are just two examples of the significant overlaps in these fields. Additionally, sociologists have long since used properties such as density and centrality to compare and understand networks, which are still useful metrics of analysis today, and provide an extra level of comparison when researching IM.

Chapter 4: Methods

This chapter outlines the methodologies followed, the necessary requirements and components and every stage of development (including revisited components for optimization/improvements).

4.1 Specifications

This Section describes the functional components necessary to achieve the objectives described in Section 1.2.

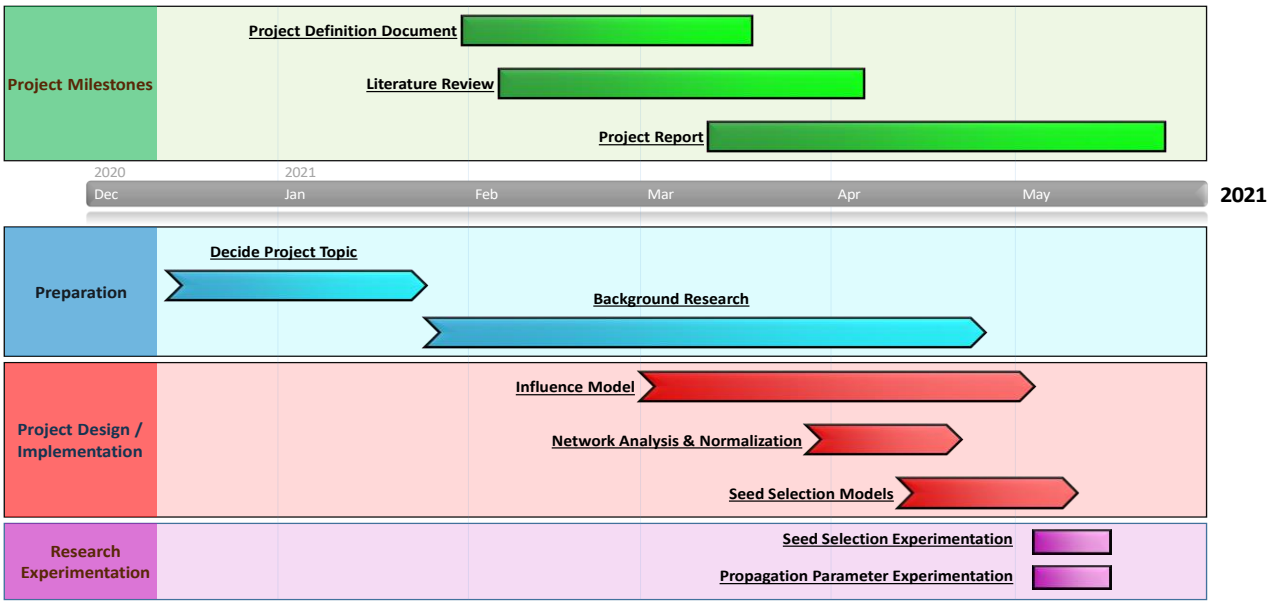
4.1.1 Necessary Components

1. Network graph generation from datasets.
2. Propagation models and comparison.
3. Additional propagation parameters.
4. Network data analysis.
5. Seed selection models and comparisons.
6. Graph plotting / presenting of results.
7. Evaluation, conclusion & discussion.

4.2 Methodologies

This Section outlines the development timeline and the methodologies followed to develop the software and research.

4.2.1 Development Timeline



4.2.2 Software Development Lifecycle

A sequential development lifecycle was planned to build each component successively. However, due to experiencing unforeseen obstacles (mentioned in Section 5.3), as well as the optimization-heavy nature of the task, an iterative approach was followed with multiple components experiencing multiple builds.

4.2.3 Research Methodology

The methodology used to design the study, collect the data and govern the analysis was experimental in design, concerned with the collecting of data in controlled environments to identify cause-and-effect relationships between variables – like influence spread and propagation parameters.

Most of methods used are quantitative, collecting and analysing the measures, proportions and correlations of data rather than the traits or characteristics. Networks are analysed and seed selection models are grouped according to their source, but this is fairly trivial.

4.3 Software Used

This Section describes the software used in implementation, their suitability to the task and justifications made.

4.3.1 Programming Languages & Libraries

Python

This project was coded in Python 3.6, a multipurpose language favouring simplistic solutions, which is advantageous when optimization is key. It also has extensive official documentation, many useful external libraries and an active community.

NetworkX

NetworkX was used to generate and analyse network graphs due to scaling well, extensive documentation and built-in network analysis functions.

Available alternatives:

- the Pajek program – developed for IM ((Mrvar and Batagelj, 2016)
- statnet – a suite of packages for C and the open-source language of R
- graphViz or gephi – graph visualization tools (which lack network analysis functionality)

Numpy

Numpy is a library of mathematical functions – ideal for manipulating large network graphs, generating random numbers, seeding randomness and calculating averages. Written in C, Numpy is interpreted by CPython, not compiled, meaning it executes faster than Python, improving efficiency.

Matplotlib

Matplotlib is a versatile library for plotting graphs (essential for research), with in-depth documentation and an active community. It was used to plot graphs during network analysis and experimentation.

Other Packages

- Time.time to optimize code.
- Math.log for log-scaling normalization
- Copy.deepcopy for making deep copies (copies of values, not references – a distinct object) of network graphs.
- Operator.itemgetter for sorting iterable by their second elements.
- Itertools.product for producing every permutation of an iterable of iterables, used in seed selection parameter fine-tuning.

LaTeX

LaTeX is an open-source document preparation system for coding scientific documents. Coding in LaTeX on the website Overleaf enables production of high-quality algorithm visualisations, with clear mathematical symbols.

4.3.2 Applications

This subSection notes the programming languages and applications utilised in design, implementation and testing.

Jupyter Notebook Web Application

Python scripts were written using the Jupyter Notebook Web Application – highly suitable for research due to its language independency and clear visual display.

Microsoft Excel

Excel was used to collate seed selection parameter fine-tuning results with analytical conditional colour formatting.

4.4 Influence Model (Build #1)

Simulating influence propagation through generated networks was the primary stage of development. This chapter outlines the steps taken to build the foundational model – including analysis of datasets and implementation of network generation and influence propagation, as well as preliminary testing for functionality.

4.4.1 Datasets

Analysis

Datasets differ in important ways, such as quantities and arrangements of nodes or edges and whether direction or weight is included. Certain properties were used as criteria to select a range of graphs to provide insightful comparisons:

Preferred Criteria	Larger datasets facilitate comparison with similarly sized datasets from past papers and real large-scale social networks – however, being more computationally expensive, available resources were considered.
	Different types of datasets (weightedness, directedness) broaden analysis, resulting in broader conclusions or specific differences in findings.
Necessary Requirements	Public availability and access for reproduction, verification or any further development.
	Completely anonymised to protect subjects' privacy and comply with GDPR.

Design

Datasets sourced from Stanford Network Analysis Platform (SNAP) library (Leskovec and Krevl, 2014):

Name:	*Size:*	*Properties:*	*Description:*
BitcoinOTC:	5,881 nodes 35,592 edges	Directed Weighted Signed	Network of weighted edges (between -10 and 10) indicating trust between users of BitcoinOTC (a platform for trading the cryptocurrency Bitcoin)
Facebook:	4,039 nodes 88,234 edges	Undirected Unweighted Unsigned	Anonymised network of friends lists from Facebook collected from survey participants, first developed by McAuley and Leskovec

Implementation

The datasets needed to be uniformly formatted before being read from files into graphs, with weights indicating influence between nodes. Common format ensures generalised functionality.

4.4.2 Network Generation

Analysis

Generation of network graphs to simulate social networks was implemented first. They can be directed or undirected, weighted or unweighted, and these properties affect their behaviour and subsequent results.

Design

Graphs are implemented as data structures from the NetworkX library read from CSV files. Additionally, smaller, user-defined graphs were created for comparison and debugging purposes.

Implementation

Directed graphs were generated from datasets, for directed or undirected, and weighted or unweighted networks. Additionally, a smaller graph was generated with customised architecture. Its purpose: incremental analysis and debugging, where its small size and simplicity make errors easier to locate. It is here-on referred to as 'G3' or the small graph:

Testing

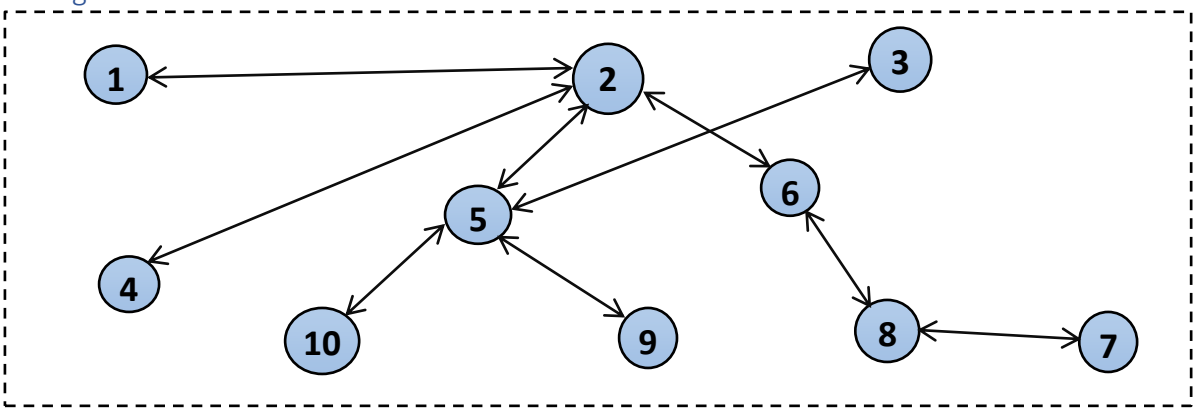


Figure 1. User-defined graph G3

Dataset-generated graphs were tested by observing rows from the files and verifying those edges are present.

4.4.3 Propagation Model (First Build)

Analysis

Propagation models share a general underlying framework but determine propagation success differently. Implementing a variety broadens research.

Design

The independent cascade model – propagation determined with a single network-wide variable uniformly applied – was implemented first due to simplicity. It could be viewed as reductive (other models incorporate relationships between probability and network characteristics), or as robust and universal – relationships outlined in other models will not always hold true, as Blower stated: 'increasing model complexity did not always increase accuracy' in regards to understanding and predicting epidemics (Blower and Go, 2011).

Simpler structure means simpler debugging, introducing complex features after functionality is established. It is also widely used in research, aiding comparison.

Propagation Model 1: Independent Cascade (G, S, R, P)

Input: *Graph G, Seeds S, Iterations R, Probability P*
Output: *Mean spread of influence*

```
1  totalSpread ← 0
2  for iter = 1 to R do
3      tried ← ∅
4      newNodes ← S
5      while newNodes ≠ ∅ do
6          currentNodes ← newNodes
7          newNodes ← ∅
8          for each node x : x ∈ newNodes do
9              for each node y : y ∈ N_x and y ∉ S do
                   /* (N_x = set of node x's neighbour nodes)
10                 if Random(0, 1) < P then
11                     newNodes += node y
12                 tried += (node x, node y)
                   /* (line 13 alternative:  tried += node y)
13         S = S ∪ currentNodes ∪ newNodes
14     totalSpread += Len(S)
15 return (totalSpread / R)
```

Figure 2. Independent Cascade Model Algorithm coded in LaTeX

Implementation and Testing

The model was implemented according to the algorithm above and tested using G3. Testing and results in Section 5.2.3.

4.5 Influence Model (Build #2)

This Section describes improvements made to the influence model, including refactoring of code for efficiency, modularization for generalization and the introduction of additional propagation parameters.

4.5.1 Refactoring/Modularization

Analysis

Improvements were required to improve efficiency and generalization. Some useful community posts discussed relevant topics:

- Best and/or fastest way to create lists in Python (ecampver, 2013) {https://stackoverflow.com/questions/20816600/best-and-or-fastest-way-to-create-lists-in-python}
- Python Sets vs Lists (Mantas Vidutis, 2010) {https://stackoverflow.com/questions/2831212/python-sets-vs-lists}

Design

Improvements made were modularization and specialising data structures.

Design and Implementation

Network generation was modularised to generalise the process so all each dataset needed defined was its file-path and some variables (directed, weighted, offset).

Testing

Testing verified the code's functionality, and measured improvements in efficiency. Tests and results can be found in Section 5.3.1.

4.5.2 Negative Influence (Quality Factor)

Analysis

Propagation parameters are introduced to influence models to improve realism, aiding simulation and application. These usually imitate and incorporate a specific property of social networks into simulated models.

Design

A quality factor enabling negative influence, introduced first by Chen et al (2011), was implemented.

Implementation

As in Chen et al (2011): the quality factor *qf* is implemented negatively for negative influence (1 – *qf*), and negativity bias is incorporated – people incline to negative opinions and influences. In keeping with this, the order of influence propagation implemented:

1. Negative nodes negatively influence (1 – qf)
2. Positive nodes positively influence (qf)
3. Positive nodes negatively influence (1 – qf)

4.5.3 Weighted Cascade 1

Analysis

Additional propagation models were included to broaden research and complexity. Some models introduce insightful, sociological relationships to improve realism.

Design

The weighted cascade model implies that nodes with more incoming edges (many influences) will be harder to influence.

It calculates propagation success using a target node's degree reciprocal – (1 – its in-degree).

Propagation Model 2: Weighted Cascade 1 (G, S, R)

```
1  totalSpread ← 0
2  for iter = 1 to R do
3      tried ← ∅
4      newNodes ← S
5      while newNodes ≠ ∅ do
6          currentNodes ← newNodes
7          newNodes ← ∅
8          for each node x : x ∈ newNodes do
9              for each node y : y ∈ N_x and y ∉ S do
10                 if Random(0, 1) < (1 / in-degree(y)) then
11                     newNodes += node y
12                 tried += (node x, node y)
                   // (line 12 alternative:  tried += node y)
13         S = S ∪ currentNodes ∪ newNodes
14     totalSpread += Len(S)
15 return (totalSpread / R)
```

Figure 3. Weighted Cascade 1 Model Algorithm coded in LaTeX

A disadvantage to this model is that it treats all edges equally in terms of calculating probability when some connections will be more significant than others.

4.5.4 Weighted Cascade 2

Analysis

A variation on the weighted cascade model was formulated (Hong and Liu, 2019) which considers popular nodes persuasiveness as well as popular nodes being difficult to influence.

Design

Propagation Model 3: Weighted Cascade 2 (G, S, R)

```
totalSpread ← 0
for iter = 1 to R do
    tried ← ∅
    newNodes ← S
    while newNodes ≠ ∅ do
        currentNodes ← newNodes
        newNodes ← ∅
        for each node x : x ∈ newNodes do
            for each node y : y ∈ N_x and y ∉ S do
                SND ← 0
                for each node z : z ∈ N_y do
                    SND += out-degree(z)
                if Random(0, 1) < (out-degree(x) / SND) then
                    newNodes += node y
                tried += (node x, node y)
                // (line 12 alternative:  tried += node y)
        S = S ∪ currentNodes ∪ newNodes
    totalSpread += Len(S)
return (totalSpread / R)
```

Figure 4. Weighted Cascade 2 Model Algorithm coded in LaTeX

This model determines propagation success using relational degrees: the proportion of the target node's in-degrees the targeting node's out-degree comprises.

Testing

New influence models were tested on the BitcoinOTC and Facebook graphs, and influence spreads were compared. For results, see Section 5.3.4.

These models will be from here on referred to as WC1 and WC2.

4.6 Network Analysis

This Section describes network analysis – involving statistical analysis of degree reciprocals and relational degrees to identify why weighted cascade probabilities were so low and how to rectify them – and comparison of network-wide properties undertaken.

4.6.1 Probability Analysis and Normalization

Analysis

Degree reciprocals and relational degrees were calculated and various normalization and scaling techniques were researched, utilised and compared. Useful online resources:

- {https://machinelearningmastery.com/robust-scaler-transforms-for-machine-learning/#:~:text=One%20approach%20to%20standardizing%20input,standardization%20or%20robust%20data%20scaling.} - (Brownlee, 2020)
- {https://developers.google.com/machine-learning/data-prep/transform/normalization.} - (*Data Preparation and Feature Engineering for Machine Learning: Normalization*, 2020)
- {https://www.codecademy.com/articles/normalization#:~:text=Min%2Dmax%20normalization%20is%20one,decimal%20between%200%20and%201.} - (*Normalization*, 2020)

Design

The probabilities needed to be between 0 and 1, evenly spread and averaging above 0.2.

Normalization/scaling techniques applied:

Technique	*Description:*
Rooting	All values rooted (square, cube, etc). Values increase, more for lower values.
Min-Max Normalization	For every value: subtract the minimum, divide by the range. Scales values between 0 and 1. Doesn't handle outliers well. Useful for other techniques that require scaling.
Z-score Normalization	For every value: minus the mean, divide by the standard deviation. Handles outliers well. Requires scaling.
Interquartile Normalization	Normalises using interquartile range. Handles outliers well.
Log-scaling	Every value is logged. Requires scaling.

Implementation and Testing

Normalised probabilities are calculated and compared, and the most ideal technique is incorporated into the propagation models – for results see Section 5.4.2.

4.6.2 Network-wide Analysis

Analysis

Network-wide properties were compared, for supplementary comparison in results.

Design

Metrics compared:

Metric	*Description:*
Weak components / *Strong components*	Weakly connected components: every node can reach or be reached by every other node. Strongly connected components: every node can reach every other node. Number of weak components divided by number of strong components is calculated (undirected graphs = 100%).
Mutuality	Percentage of edges (x -> y) where (y -> x) is also present.
Density	Percentage of possible edges that are present.
Nodes with no incoming edges	Percentage of nodes with no incoming edges.
Nodes with no outgoing edges	Percentage of nodes with no outgoing edges.

Implementation and Testing

Metrics are calculated and compared – see Section 5.4.3 for results.

4.7 Seed Selection Models

The next stage was implementing seed selection models; some from previous papers, building some using NetworkX metrics and some new models.

4.7.1 Models from Past Papers

Models explored and implemented from past papers:

Metric:	*Description:*
Original Greedy (Kempe, Kleinberg and Tardos, 2003)	Calculates spread of seed set + node for every node in graph, selects highest ranking as a seed. Repeats until seed set full. High level of success, but scales inefficiently due to repeated propagations.
*CELF (Cost-effective Lazy-forward)*** (Leskovec *et al.*, 2007)	Optimizes using submodularity of IM (as seed set grows, new seeds improve spread less). Ranks nodes by influence spread. Top node is selected, removed from the list. New top node's spread recalculated (minus spread from the seed set), list is resorted, and if it remains on top, is selected as a seed and the process repeats. If not, process repeats for the new top node. No seed node will ever outperform their performance of the previous turn.
Improved Greedy (Chen, Wang and Yang, 2009)	Generates new graph, removing edges with probability (1 – pp). Nodes are selected on their reach within this new graph. Removed the need for repeated propagation, but still inefficient due to generating new graphs every iteration for every seed.
Degree Discount (Chen, Wang and Yang, 2009)	Heuristic approximation method. Ranks nodes based on degree, and discounts rankings of neighbour nodes of selected seeds, indicating neighbours of seeds are less likely to be good nodes themselves. No iterations required – most efficient approximation method.

**An online implementation (King, 2018), which can be found at this link (https://hautahi.com/im_greedycelf), was reused/reappropriated into my implementation.

4.7.2 Network-Analysis-based Models

Analysis and Design

Models were developed using NetworkX metrics, investigating which metrics indicate good seed nodes. Metrics used:

Metric:	*Node Description:*
Degree Centrality	Fraction of nodes it is connected to.
In-degree Centrality	Fraction of nodes its incoming edges are connected to.
Out-degree Centrality	Fraction of nodes its outgoing edges are connected to.
Closeness Centrality	Reciprocal of the average shortest path distance over all reachable nodes.
Information Centrality	Variant of closeness centrality – incorporates effective resistance between nodes.
Betweenness Centrality	Sum of the fraction of all-pairs shortest paths that pass through it.
Approximate Current-flow Centrality	Approximates betweenness centrality (uses electrical current model) within absolute error of epsilon.
Load Centrality	Fraction of all shortest paths that pass through it.
Eigenvector Centrality	Centrality based on the centrality of its neighbours.
Katz Centrality	Generalization of eigenvector centrality.
Subgraph Centrality	Sum of all weighted walks starting and ending at this node. Weights decrease with path length.
Harmonic Centrality	Sum of the reciprocal of shortest path distances from all other nodes.
VoteRank	Based on a voting scheme. Nodes vote for in-neighbours and node with highest nodes is elected iteratively. Voting ability of out-neighbours decreases subsequently.
PageRank	Based on structure of incoming links. Designed to rank web pages.
HITS Hubs	Value based on outgoing links.
HITS Authorities	Value based on incoming links.

4.7.3 Parameter Fine-tuning

Analysis

Some NetworkX-based models can take parameters – to enhance effectiveness, the parameters were investigated to determine ideal values.

Design

For each parameter, a range of values was compiled for comparison.

Implementation

Every permutation of parameters is calculated, and the seed set obtained with each has its influence spread measured.

Testing

Influence spreads and selection times are compared. Results were stored in a conditionally formatted spreadsheet for staggered entry due to long processing times, enabling easy analysis and comparison. For results see Section 5.4.3.

Evaluation

Once collection was complete, the optimal parameters were set as defaults in their respective seed selection functions.

4.7.4 New Seed Selection Models

Analysis

To accomplish the main objective (1), new seed selection models were developed to rival past models.

Design and Implementation

Some were developed through combining existing strategies, and others extended or generalised characteristics of existing models.

4.8 Influence Model (Build #3)

Final improvements and additional propagation parameters.

4.8.1 Switch Factor

Analysis and Design

Signed influence presents opportunities for nodes to switch sign of influence, representing people changing their mind about something. To incorporate this insight a 'switch factor' was implemented.

Implementation

Enables nodes to switch sign of influence with a defined penalty in probability (1 – *sf*). The higher the switch factor (*sf*), the less likely the switch in influence.

Testing

Effects of switch factor were assessed, holding other propagation parameters constant – for results see Section 5.5.

4.8.2 Time Factor

Analysis

How recent something is affecting its influence is an insight included in multiple papers (Liu *et al.*, 2014), referencing how as time goes on (as products/opinions age), influence probabilities decrease.

Design and Implementation

Propagation success is multiplied by a variable decreasing by the time factor each turn of propagation.

Testing

Effects of time factor were assessed, holding other propagation parameters constant – for results see Section 5.5.

4.8.3 Network Generation (Method #3)

Analysis

Generalising network generation to enable seamless extension to further datasets is important in ensuring reproducibility and aiding extension of the model.

Design

The requirement is that anyone can add their own network dataset and use my propagation models or my network analysis tools on it, simply.

Implementation

This consisted of storing information about datasets (name, weighted, directed, file-path) in a dictionary, and completely refashioning the network generation function.

Testing

Functionality testing is performed – efficiency testing against old method unnecessary, as the extra functionality is prioritised. However, iteration methods within the function are optimized through testing.

4.8.4 Probability Assignment

Analysis

Degree reciprocals and their normalizations do not change during the program, so to improve efficiency a method was implemented to

Design

Implementation

Probabilities for the weighted cascade models (1 and 2) are calculated, the collection of them normalised and added to a dictionary for easy access during propagation.

Testing

Iteration methods tested for efficiency, and entire function tested for functionality against previous method.

4.9 Testing and Experimentation Plans

To conclude the project and finalise the main objectives, two phases of experiments are performed – comparison of seed selection models' performances in time and IM, and analysis of propagation parameters and their effects on influence spread.

4.9.1 Functionality Testing and Optimization

Throughout development, components were improved and optimized. Functionality testing verified that improved components still function the same, and efficiency tests evaluate if they optimize code.

4.9.2 Experiments

Analysis

Seed selection models are compared on time taken to select seeds and their resulting spreads, and propagation parameters are compared on their effects on influence spread.

Design

Different graphs are plotted for different experiments – line graphs for propagation parameters and horizontal bar charts for seed selection models.

Implementation

For seed selection models, every model is executed, with their returned seed sets and time elapsed measured. Bar charts are plotted for the real graphs and the random graphs for the slower models to be compared.

For propagation parameters, propagation is performed varying one parameter (*pp*¸ *qf*, *sf*, *tf*) and line graphs are plotted to analyse its effect on influence spread.

Chapter 5: Results

This chapter outlines important parts of underlying software and all results from Methods, including functionality and efficiency testing and experimentation.

5.1 Influence Model (Build #1)

5.1.1 Datasets

Implementation

Datasets were formatted into 2 or 3 columns – node, node, weight (if present) – each row representing an edge of the graph (new nodes added automatically). Weights on edges represent the likelihood of propagation between them, scaled between 0 and 1.

The BitcoinOTC dataset was in .csv (comma-separated-values) format, while the Facebook file was in .txt (elements separated by spaces). Preliminary formatting involved replacing every space in the Facebook file with a comma (Ctrl-f -> Advanced -> Replace all) and saving the file in .csv format. An optional step is to delete any values from column 4 onwards, as they won't be used and may decrease efficiency in very large dataset files.

5.1.2 Network Generation

Design

Directed graphs were used for all networks, keeping the data structure uniform aiding universality – some NetworkX functions are exclusive to directed graphs.

For different types of graphs, Boolean variables indicate directedness and weightedness. Unweighted graphs will be assigned a weight of 1 to all edges; for undirected graphs, every edge will be added in parallel. Also, edges are reversed when added, as discussed in (Chen *et al.*, 2011) to represent influence – an edge indicating node x *trusts* node y shows node y has *influence* over node x.

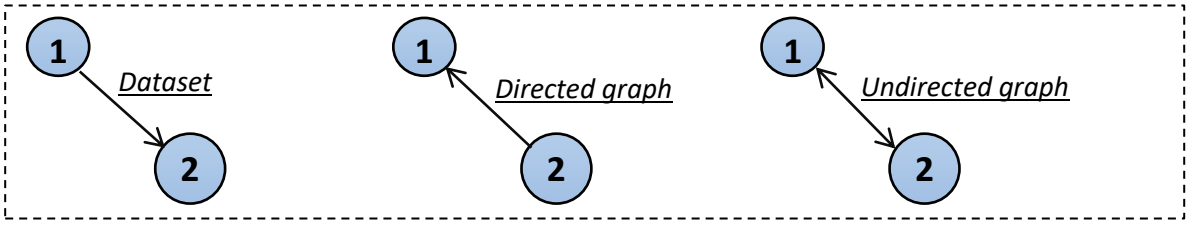


Figure 5. Edge formatting diagram

Implementation

Network generation was manual and done individually for each graph. This way, their parallel edges (if undirected) and/or weight values (if weighted) could be adjusted accordingly. The BitcoinOTC dataset needed its weight values scaled from between -10 and 10 to between 0 and 1. Also, nodes in BitcoinOTC began at 1, and in Facebook at 0. So, BitcoinOTC node keys (numbers) had 1 subtracted for uniformity.

BitcoinOTC example (Facebook graph follows similar format):

```
#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\Individual Project\datasets\soc-sign-bitcoinotc.csv") as csvfile1:
    #read file and separate 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
        #edges are reversed to indicate influence
        G1.add_edge((int(row[1])-1), (int(row[0])-1),
trust=(int(row[2])+10)/20)
```

G3 example:

```
#Small, custom directed, unweighted graph
G3 = 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)]
G3.add_edges_from(testedges)
nx.set_edge_attributes(G3, 1, 'trust')
```

Testing

Manual testing if edges are present:

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

	A	B	C
1	6	2	4
2	6	5	2
3	1	15	1

Results

Edge:	*Present in BitcoinOTC Graph:*
1 -> 5 (6 -> 2 in dataset)	True
4 -> 5 (6 -> 4 in dataset)	True
14 -> 0 (1 -> 15 in dataset)	True

Evaluation

Testing was successful: multiple rows from the .csv files were present in the networks, proving functionality of network generation.

5.1.3 Initial Propagation Model

Implementation

The initial implementation was developed as a foundational model focused on functionality, using lists and nested for-loops extensively, avoiding complexity to simplify debugging. It would later be refactored, optimized and modularized.

(See Appendix D.1 for variations in propagation model)

Testing

Propagation was tested on G3, with a seed set of [1, 2] and one of two propagation probabilities over 100 iterations:

```
print(IndependentCascade(G3, [1,2], 100, 0.1))
print(IndependentCascade(G3, [1,2], 100, 0.8))
```

Results

Propagation Probability (pp)	*Influence Spread*	*Time elapsed*
0.1	1.42	0.001
0.8	4.56	0.003

Evaluation

Testing was successful, indicating the model functioned as intended and could be expanded to include additional features. However, propagation on either the BitcoinOTC or Facebook graph was too inefficient to run, necessitating optimization.

5.2 Influence Model (Build #2)

5.2.1 Refactoring/Modularization

Implementation

Improvements were made, using relevant community posts (Section 4.4.1) as a broad guide.

The influence model was modularised into three functions – iteration, propagation and success.

Refactoring consisted of removing unnecessary loops and specializing data structures. Sets are quicker than lists when searching for an element. Sets cannot contain duplicates, however, so the model was restructured without 'targets' lists.

Network generation was also modularised and semi-generalised:

(see Appendix D.1 for variations in network generation method)

Network generation from datasets:

```
#  Generate network graph from soc-BitcoinOTC dataset (5881 nodes, 35592
edges)
#  (directed, weighted, signed)
G1 = generateNetwork("BitcoinOTC Network", True, True, 1,
r"D:\Sully\Documents\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)
G2 = generateNetwork("Facebook Network", False, False, 0,
r"D:\Sully\Documents\Computer Science BSc\Year 3\Term 2\Individual
Project\datasets\facebook.csv")
```

Testing

Modularised network generation functionality testing:

```
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")
```

Results

Network generation method 2 results:

Network Generation Method:	*Graph 1 Size (edges), Length (nodes)*	*Graph 2 Size (edges), Length (nodes)*	*Graph 3 Size (edges), Length (nodes)*
Method 1 (manual)	35587, 5875	176468, 4039	11, 10
Method 2 (semi-modularised)	35587, 5875	176468, 4039	11, 10

Evaluation

Testing was successful, indicating modularized network generation was functionally identical to the previous method.

5.2.2 Quality Factor

Implementation

The quality factor was implemented in the success function, with *qf* being applied to probability for positive influence and (1 – *qf*) for negative influence:

```
    q = qf
    #Modify quality factor for negative influence
    if not sign:
        q = (1-q)
    return q
```

Data structures used required restructuring to allow for negative nodes and newly negative nodes, avoiding interference with positive nodes.

Quality factor was tested along with new propagation models.

5.2.4 Additional Propagation Models (WC1 & WC2)

Implementation

New propagation models were introduced:

```
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[targeting][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[targeting][target]['trust'])
```

Testing

```
#Testing the change in quality factor for every model, for every 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")
```

Results

Graph:	*Model (Build #2):*	*Propagation Probability (pp):*	*Quality Factor (qf):*	*Spread:*	*Time elapsed (50 iterations):*
BitcoinOTC	Independent Cascade	0.2	0.2	7.32	5.458 secs
			0.8	759.1	7.438 secs
	Weighted Cascade 1	0.2	0.2	5.68	0.524
			0.8	27.8	0.2777 secs
	Weighted Cascade 2	0.2	0.2	353.44	19.153 secs
			0.8	3593.74	20.76 secs
Facebook	Independent Cascade	0.2	0.2	0.88	45.028 secs
			0.8	1560.36	37.475 secs
	Weighted Cascade 1	0.2	0.2	1.28	0.039 secs
			0.8	2.68	0.032
	Weighted Cascade 2	0.2	0.2	0.0	0.368 secs
			0.8	1696.64	98.8408

Evaluation

Results verified that the quality factor increased influence when high and decreased influence when low – further analysis in Section 5.8.

However, WC1 and WC2 models had noticeably low spreads with a low quality factor (orange), and WC1 even with a high quality factor (red). This indicated probabilities based on degree reciprocals or relational degrees were too low. These models comprised much of the planned experimentation, and the spreads needed to be sufficiently high to enable comparison with other models. This led to in-depth statistical analysis of probabilities and normalization techniques.

5.3 Network Analysis

5.3.1 Probability Analysis and Normalization

Analysis

As previously mentioned, probabilities based on degree reciprocals (WC1) or relational degrees (WC2) were too low, especially in the Facebook graph, necessitating investigation into normalization techniques.

Implementation

Degree reciprocals and relational degrees are calculated and compiled into lists, various normalization techniques are applied and histograms of results are plotted. Normalization is intended to achieve probabilities with a mean between 0.3 and 0.7 and an even spread of values. Example functions are included below, different variations are included in appendices (REFERENCE).

Relational degree calculation function:

```
#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][target]['trust'])
            else:
                relDegs.append(g.out_degree(targeting)/snd)
    return relDegs
```

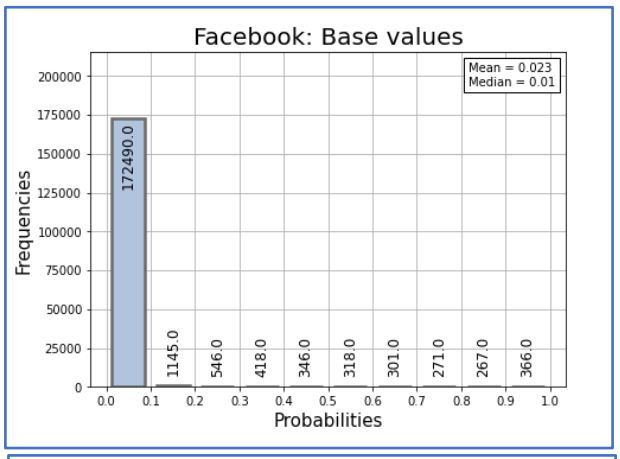
Example normalization function (square rooting):

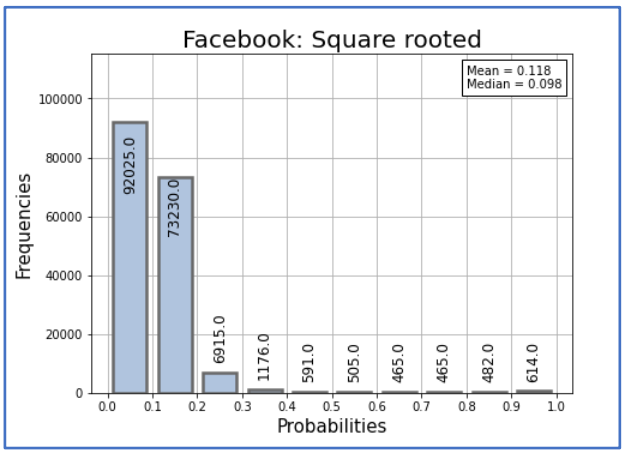
```
#Convert all elements in a list to a given root
def rootList(lis, root):
    return list(map(lambda x : x**root, lis))

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

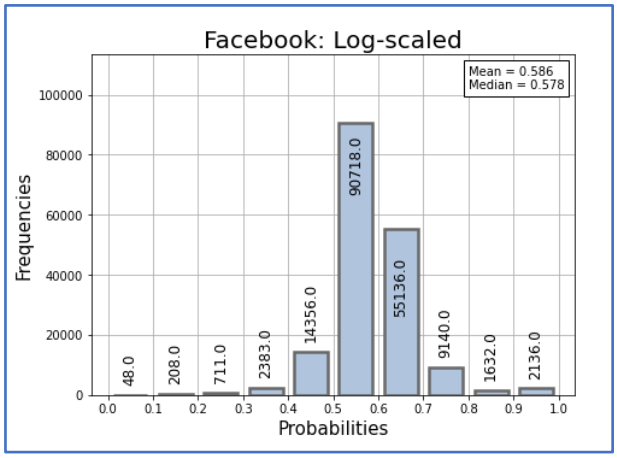
See Appendix C.2 for plot histogram functions

Results

Example histograms for Facebook graph relational degrees:







The complete collection of histograms can be found in Appendix D.2.

Evaluation

Reviewing the histograms, log-scaling was chosen as it results in sufficient probabilities for both graphs and has an even spread. Despite increasing probabilities, relationships still hold due to all values being normalized.

5.3.2 Network-wide Analysis

Implementation

Network-wide analysis was also performed for network comparison. Different metrics are calculated for each network, compiled into lists and plotted in bar charts.

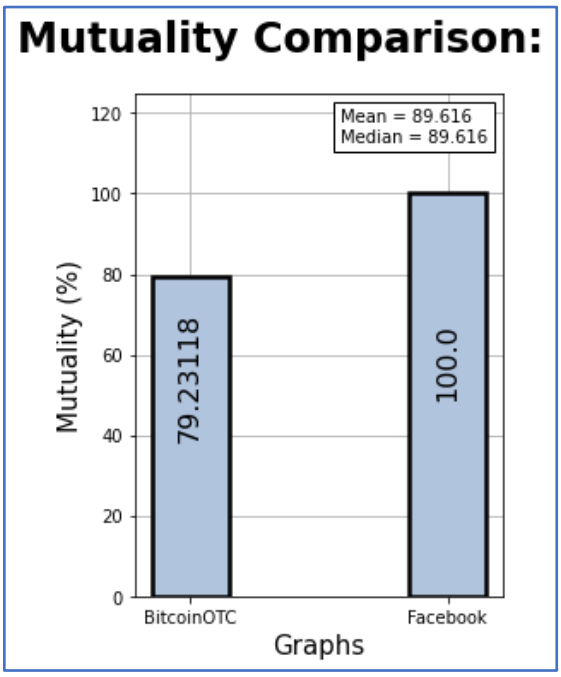
Example metric calculation function (mutuality):

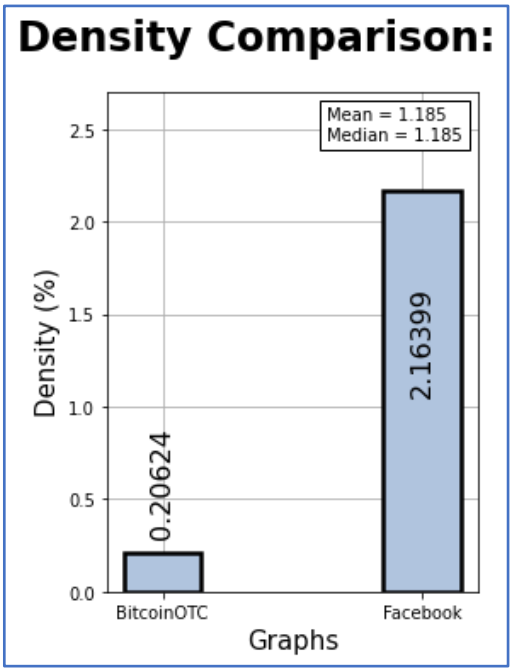
```
#Returns the mutuality-percentage of a given graph
#(how many of the edges are parallel/bi-directional)
def mutuality(g):
    edgeSet = set(g.edges)
    count = 0
    for (u,v) in edgeSet:
        if (v,u) in edgeSet:
            count += 1
    return round((count/g.size())*100, 5)
```

(see Appendix C.2 for network metric bar chart plotting functions and D.3 for all graphs)

Results

BitcoinOTC and Facebook – mutuality and density:





Evaluation

The density bar chart shows the Facebook graph to be significantly higher than the BitcoinOTC graph, with a density of ~2.2% compared to ~0.2%. This explains why the Facebook graph is slower to propagate through – as more edges per nodes means more edges must be attempted before propagation is completed. The other charts showed no significant correlation to improve analysis.

5.4 Seed Selection Models

5.4.1 Past Models

Analysis and Design

Recreating past models involved studying algorithms from relevant papers and implementing them in Python.

Implementation

Example of past models: (original greedy)

```
#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
```

(All models can be found in Appendix C.3.)

5.4.2 Network-Analysis-based Models

Design

All models follow a similar framework: sort all nodes by the given metric using the appropriate built-in NetworkX function, and extract the *k* top nodes, where *k* is the seed set size.

Implementation

Two examples of NetworkX-based models:

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


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

(All models can be found in C.3.)

5.4.3 Parameter Fine-tuning

Design

Range of values used in fine-tuning:

Closeness Centrality	Wasserman-Faust improvement						
	True		False				
Betweenness Centrality	Node samples for estimation				Randomness seed		
	25	50	100	200	10	None	
Approx. Current-flow Betweenness Centrality	Maximum number of sample node pairs						
	10000	500	*50	*200			
Load Centrality	Cut-off length						
	1	2	3	4			
Eigenvector Centrality	Max iterations				Error tolerance		
	100	500	1000		0.001	0.0025	0.005
Katz Centrality	Beta (neighbourhood weight)				Alpha (attenuation factor)			
	0.75	1	1.25	1.5	0.05	0.1	0.15	0.2
Harmonic Centrality	Distance edge attribute						
	None		Trust				
PageRank	Alpha (damping parameter)						
	0.65	0.75	0.85	0.95			

*the parameter will be the graph's size divided by this value.

Implementation

Each seed selection model was run with every possible permutation of values with functions defined with variable parameters:

Example variable parameter function:

```
#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]
```

Functions obtained seeds using every permutation of parameters for every seed selection model, ran propagation with those seeds and printed the results. Important function 'paramFineTune' can be found in Appendix C.3.

It also tracked which seeds had been propagated to prevent propagation using any previously tried seed sets as it would return the same result and waste time.

Testing

```
#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)
```

Results

Results were compiled into a formatted excel spread sheet, an excerpt of which is shown here:

Seed Selection Parameter Tuning:				Graph 1: BitcoinOTC (weighted, directed)											
Seed Selection Model:	*Parameters* Param1	Param2	Param3	*Seed node set* 1	2	3	4	5	6	7	8	*Spread:*	*Time:* ~120 secs (pp=0.2, its=1000)	*Spread:*	*Time:*
				Optimal configuration in blue								(Colour ranked within model)		(Colour ranked overall)	
Betweenness Centrality	Optimal Parameters & their Results -->			UNWEIGHTED EDGES, k=50, Seeds=10								578.043	2.007		
	UNWEIGHTED	Seeds=10	k=25	0	34	6	904	2027	4558	2641	1809	573.985	1.033	573.985	1.033
			k=50	0	34	6	904	4171	2027	2641	1809	578.043	2.007	578.043	2.007
			k=100	0	34	6	904	4171	2027	2641	1809	578.043	2.989	578.043	2.989
			k=200	0	34	6	904	4171	2027	2641	1809	578.043	5.868	578.043	5.868
		No Seed	k=25	34	1351	904	3017	2641	1809	2387	1017	558.328	0.84	558.328	0.84
			k=50	1952	0	34	904	2124	2896	2641	1809	564.188	1.76	564.188	1.76
			k=100	34	6	904	4171	2124	2027	2641	1809	573.974	3.252	573.974	3.252
			k=200	0	34	904	4171	2124	2027	2641	1809	573.062	6.54	573.062	6.54
	WEIGHTED	Seeds=10	k=25	1952	134	1351	4171	2027	3758	3759	1809	520.318	1.665	520.318	1.665
			k=50	1952	4171	2027	3755	3758	3759	1809	3743	505.926	3.392	505.926	3.392
			k=100	1952	4171	2027	3755	3758	3759	1809	3743	505.926	6.062	505.926	6.062
			k=200	1952	2027	3755	3758	3759	1809	2295	3743	498.688	12.801	498.688	12.801
		No Seed	k=25	1952	4171	2027	3755	3758	1809	2295	3743	510.46	1.67	510.46	1.67
			k=50	1952	2027	4171	3755	3758	3759	1809	2066	514.47	3.689	514.47	3.689
			k=100	1952	4171	3755	2027	3758	3759	1809	3743	505.926	6.646	505.926	6.646
			k=200	1952	3755	3756	3758	3759	2066	2295	3743	472.022	13.231	472.022	13.231
Load Centrality	Optimal Parameters & their Results -->			cutoff=2, NO Weight								575.847	42.277		
	UNWEIGHTED		cutoff=1	0	1	2	3	4	5	14	15	482.49	0.114	482.49	0.114
			cutoff=2	34	6	904	2027	2124	4171	2641	1809	573.974	3.68	573.974	3.68
			cutoff=3	0	34	6	904	2027	2124	2641	1809	575.847	42.277	575.847	42.277
			cutoff=4	34	2641	1809	904	2124	0	2027	4171	573.062	120.109	573.062	120.109
	WEIGHTED		cutoff=1	1542	904	2027	3743	1809	2641	3313	831	524.195	49.377	524.195	49.377
			cutoff=2	34	1351	904	2027	3743	2641	1809	831	557.207	237.793	557.207	237.793
			cutoff=3									557.207	312.618	557.207	312.618
			cutoff=4									557.207	313.358	557.207	313.358

(The complete table can be found in Appendix D.4 and the file can be found in additional appendices) (values may have changed before the final version due to further tweaks in the cascade implementation, but the rankings will be consistent.)

Conditional formatting coloured cells depending on their rank (by spread or time), once per model and once overall. There were also formulas to calculate averages and display the maximum spread and minimum time overall.

This was repeated for both the Bitcoin and Facebook graphs, and the optimal parameters were hard-coded into the seed selection models.

5.4.4 New Models

The first new model implemented, 'MixedGreedy', was theorised but not detailed in Efficient Influence Maximization in Social Networks (Chen, Wang and Yang, 2009). It uses the improved greedy technique to generate a modified graph, removing edges by probability, and uses the CELF technique to continuously re-rank and choose the best seed nodes (both in Section 4.6.1). Four variations of this model were implemented.

The next model implemented was a heuristic strategy closely based on the degree discount model and another heuristic solution described broadly by (Hong and Liu, 2019) which ranks nodes by degree and discounts rankings of seed node's neighbours.

The final model implemented, 'Disconnect' model, removes nodes reached by seed nodes from the graph iteratively, however this model was not successfully implemented and was never developed to working order.

5.5 Influence Model (Build #3)

5.5.1 Switch Factor

Implementation

If an already influenced node is being influenced with the opposite sign of influence (positive/negative) a factor of (1 – *sf*) where *sf* is the switch factor is applied to the probability. If the node is uninfluenced, no switch factor is applied.

```
success(successMod, False, (neighbour in pos)#...
#...other code...#
def success(successModel, sign, switch,#...
    if successModel == 'IC':
        succ = (pp*successVars(sign, switch, qf, sf)
#...other code...#
def successVars(sign, switch, qf, sf):
    if not sign:
        qf = (1-qf)
    if not switch:
        sf = 0
    return qf*(1-sf)
```

Testing

```
#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("")
```

Results

Switch factor testing on BitcoinOTC graph:

Propagation Model	*Switch Factor (sf):*	*Influence Spread*	*Time elapsed (50 iterations):*
Independent Cascade	0	847	8.51 secs
	0.3	845.36	8.12 secs
	0.6	902.18	8.13 secs
	0.9	975.56	8.35 secs
Weighted Cascade 1	0	1561.02	9.58 secs
	0.3	1628.18	9.67 secs
	0.6	1695.58	10.06 secs
	0.9	1867.92	11.1 secs
Weighted Cascade 2	0	1937.3	10.09 secs
	0.3	1978.8	9.74 secs
	0.6	2097.32	9.97 secs
	0.9	2295.3	10.91 secs

Evaluation

The switch factor results are opposite to what was initially expected, as a higher switch factor was expected to reduce influence. The only possible explanation is that due to the incorporation of negativity bias (Section 4.5.2.), it reduces negative influence more than it reduces positive influence, resulting in an overall increase in positive influence. An unexpected, but insightful result.

Further Testing

Due to the switch factor producing unexpected results, it was thought it may be due to the positive quality factor – referring to research hypothesis (2) – and if quality factor < 0.5, switch factor would have an inverse effect.

```
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("")
```

Results

Quality Factor (qf):	*Switch Factor (sf):*	*Positive influence*	*Negative influence*
0.2	0	14.6	2245.52
	0.2	14.84	2249.68
	0.4	18.32	2243.64
	0.6	20.0	2239.04
	0.8	24.6	2229.84
	1	37.24	2218.96
0.8	0	1854.56	416.16
	0.2	1865.32	404.52
	0.4	1868.28	389.92
	0.6	1874.72	381.92
	0.8	1892.92	375.24
	1	1903.64	359.6

Evaluation

It seems regardless of what value the quality factor is, a higher switch factor always benefits positive influence over negative influence. The logical explanation is that it counteracts the negativity bias implicit in the model.

5.5.2 Time Factor

Implementation

A time variable is initialised at 1. Every turn of propagation, the time factor is subtracted from the time variable, and is multiplied during propagation probability.

E.g., time factor = 0.02:

First turn: prob = 1 x prob. Second turn: prob = 0.98 x prob Third turn prob = 0.96 x prob

```
        timeFactor = 1
        #while there are newly influenced nodes from last turn...
        while posNew or negNew:
            #...other code...#
            #Time delay is taken away from the time factor
            if timeFactor < 0:
                timeFactor = 0
            else:
                timeFactor -= tf
```

Testing

```
#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("")
```

Results

Time factor testing on BitcoinOTC graph:

Propagation Model	*Time Factor (tf):*	*Influence Spread*	*Time elapsed*
Independent Cascade	0	929.36	8.69 secs
	0.05	869.84	8.88 secs
	0.1	770.34	8.44 secs
	0.5	111.76	2.71 secs
Weighted Cascade 1	0	1679.78	9.9 secs
	0.05	1648.88	10.59 secs
	0.1	1579.5	10.44 secs
	0.5	297.6	5.08 secs
Weighted Cascade 2	0	2105.56	10.697 secs
	0.05	2028.0	10.25 secs
	0.1	1926.14	10.67 secs
	0.5	427.88	7.56 secs

Evaluation

The results show that time factor works exactly as expected: as it increases, influence spread decreases. This is due to less propagation turns being possible each iteration, as the delay made to the time delay variable is greater.

5.5.3 Network Generation

Implementation

Completely generalized network generation method is implemented using a dictionary to store information about datasets and the network graphs themselves. This enables new datasets to be added to the program requiring only two lines of code – add file-path to datasets dictionary and list whether it is weighted and/or directed.

Adding new datasets is explained in full detail in Appendix B.

(unconnected function can be found in Appendix C.1, C.2, C.3, D.1 and final network generation function can be found in Appendix C.1, or D.1)

Dataset dictionary and network generation:

```
#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
Project\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
Project\datasets\facebook.csv")
}

#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
```

Testing and Results

There were various iteration methods available in Pandas to iterate through the rows of the .csv files, so they were tested for efficiency:

(code for iteration method efficiency testing can be found in Appendix D.1)

Pandas Row Iteration Model:	*Time elapsed: (10 iterations)*
index	38.35 secs
loc	45.99 secs
iloc	130.19 secs
itertuples	5.54 secs
iterrows	95.22 secs

Secondly, the new method is functionally tested, against the old method, to ensure they work identically.

Network Generation Method:	*Graph:*	*Size (# of edges):*	*Length (# of nodes):*
Method 2 (Semi-generalised function with manual dataset parameter entry)	G1: BitcoinOTC	35587	5875
	G2: Facebook	176468	4039
	G3: Custom	11	10
Method 3 (Fully generalised function with dataset parameter dictionary)	G1: BitcoinOTC	35587	5875
	G2: Facebook	176468	4039
	G3: Custom	11	10

Evaluation

The itertuples iteration method was the most efficient and was implemented in network generation method 3. Method 3 and method 2 are seen to be functionally identical, so the new network generation method was a success.

5.5.4 Probability Assignment

Implementation

Assign functions were implemented to calculate all degree reciprocals (WC1) or relational degrees (WC2), normalise them and assign them as node (WC1) or edge (WC2) attributes. The downside to this approach was that the normalization method had to be manually predetermined.

This was then improved with a more generalised method, where the cascade function had an additional parameter *assign* which would specify which normalization method should be used in the case of WC1 or WC2. Furthermore, if *assign* was set to 0, no assigning would occur and any calculations for degree reciprocal or relational degree were done in the success function, as necessary. This final feature meant that comparative testing could take place.

Testing and Results

Optimization and functionality tests were run to confirm it still functioned identically and compare performance. The tests were run on the BitcoinOTC graph with the parameters: Seed set = [1], 75 iterations, propagation probability = 0.5, quality factor = 0.7, switch factor = 0.7, time factor = 0.04:

(testing can be found in Appendix D.1)

The initial results can be seen below:

Assign Model:	*Description:*	*Spread:*	*Time elapsed:*
Assign = 0 (Manual log-scaling)	Calculate log-scale of probability every time it's needed, during the success function.	1.427	0.368 secs
Assign = 3 (Log-scaling and assigning method)	Calculate and normalise all Relational Degrees initally and assign them as edge attributes	888.773	77.491 secs

As they are not equal, there was a problem in implementation. Upon inspection it is clear there was a mistyped line in the *AllRelDegs* function (used to calculate all the relational degrees, to obtain maximums and minimums, which are used for normalization when manually calculating for *Assign* = 0), and further running of the tests were carried out:

Assign Model:	*Description:*	*Spread:*	*Time elapsed:*
Assign = 0 (Manual log-scaling)	Calculate log-scale of probability every time it's needed, during the success function.	1188.533	328.085
Assign = 3 (Log-scaling and assigning method)	Calculate and normalise all Relational Degrees initially and assign them as edge attributes	1188.533	13.405

It is clear from these results, that the two methods are functionally identical and the newer assigning method improves efficiency by a factor of over 20.

5.6 Seed Selection Model Comparison

Design

Bar charts were used to compare the seed selection models, as each model is its own distinct entity, and as such they form discrete variables. The random user-defined graph was used initially, due to the past models being too inefficient to feasibly run on the BitcoinOTC or Facebook graph (even running the original greedy with 1 iteration on the Facebook graph would take over 15 minutes).

Implementation

After collecting and compiling all the fine-tuning results, bar charts were plotted to better compare the different models, using different preparation and plotting functions:

The function ‘prepareBar’ used to prepare the data, given seed selection models and their labels can be found in Appendix C.3.

Function to plot horizontal bar charts with the prepared data, comparing different seed selection models:

```
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)
```

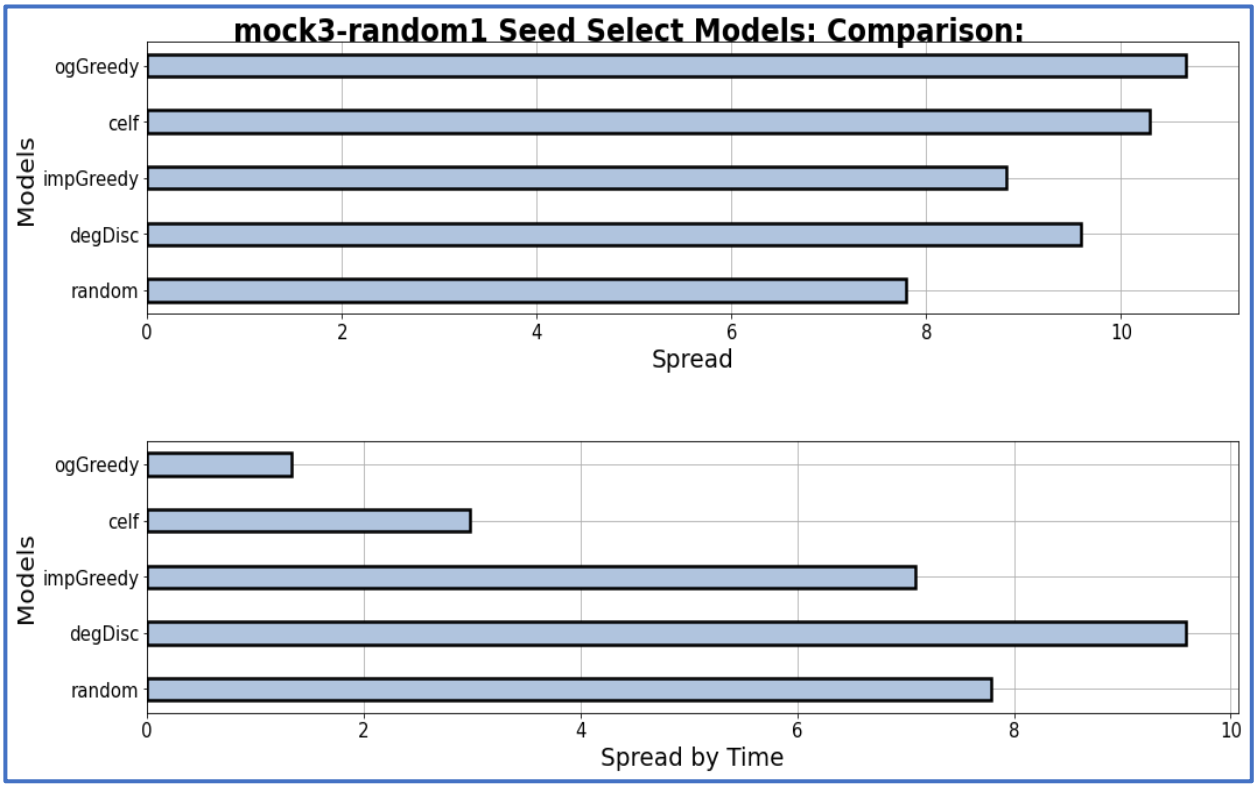
Results

Each model has bar charts plotted for spreads and for spreads divided by a tenth of the seed selection time. Past models are compared using the random graphs, as they take too long to run on the BitcoinOTC and Facebook graphs, and the others are compared on random and real graphs. Finally, all models are compared on the random graphs – once comparing spread, and once comparing spread divided by a tenth of the seed selection time (to compare effectiveness in regards to efficiency).

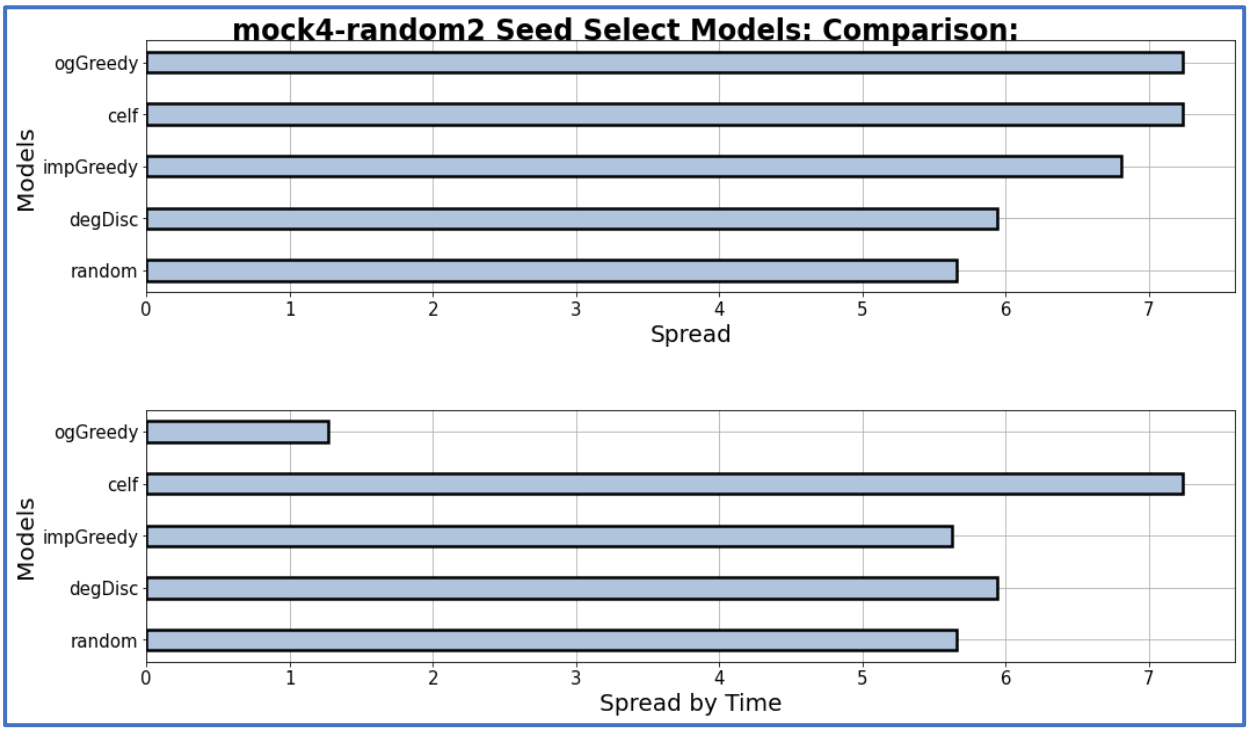
Past Models

Results

Past models – random graph with uniform trust values:



Past models – random graph with randomised trust values:



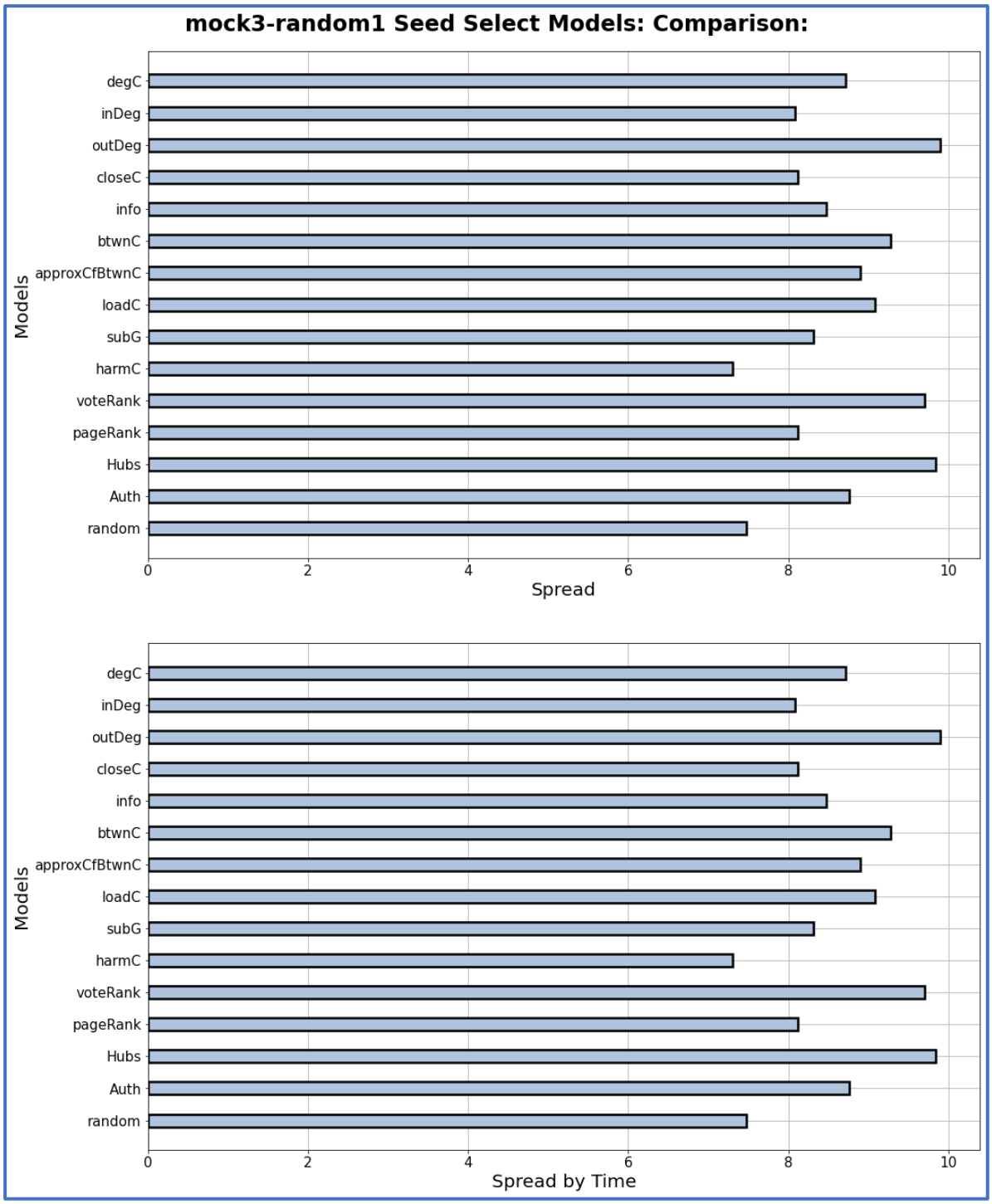
Evaluation

On the first random graph, original greedy, CELF and degree discount provide the best spread, but only degree discount retains its high ranking when time is considered. In the second graph, however, degree discount does not perform as well, with CELF retaining a very high spread.

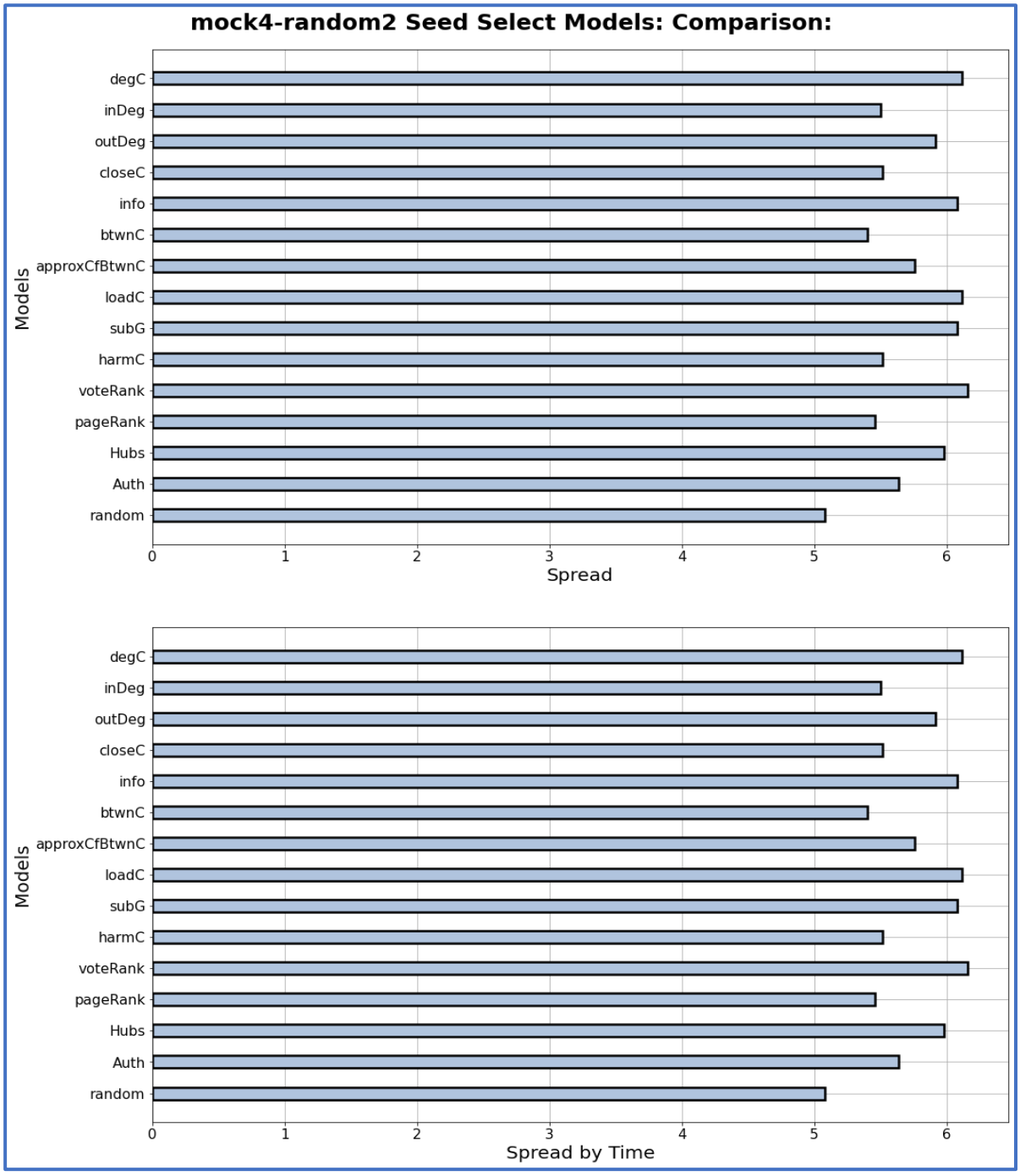
NetworkX Models

Results

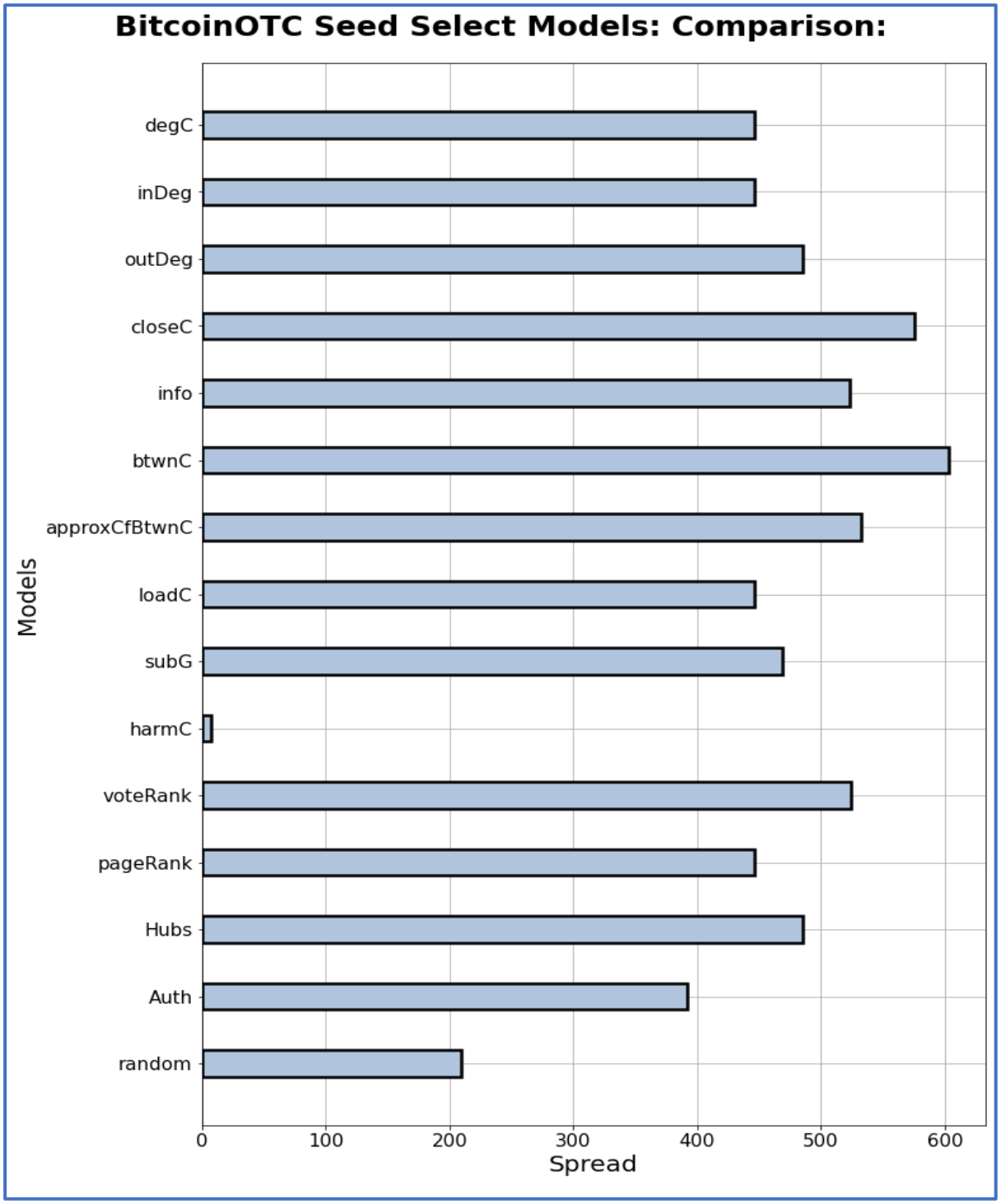
NetworkX models – random graph with uniform trust values

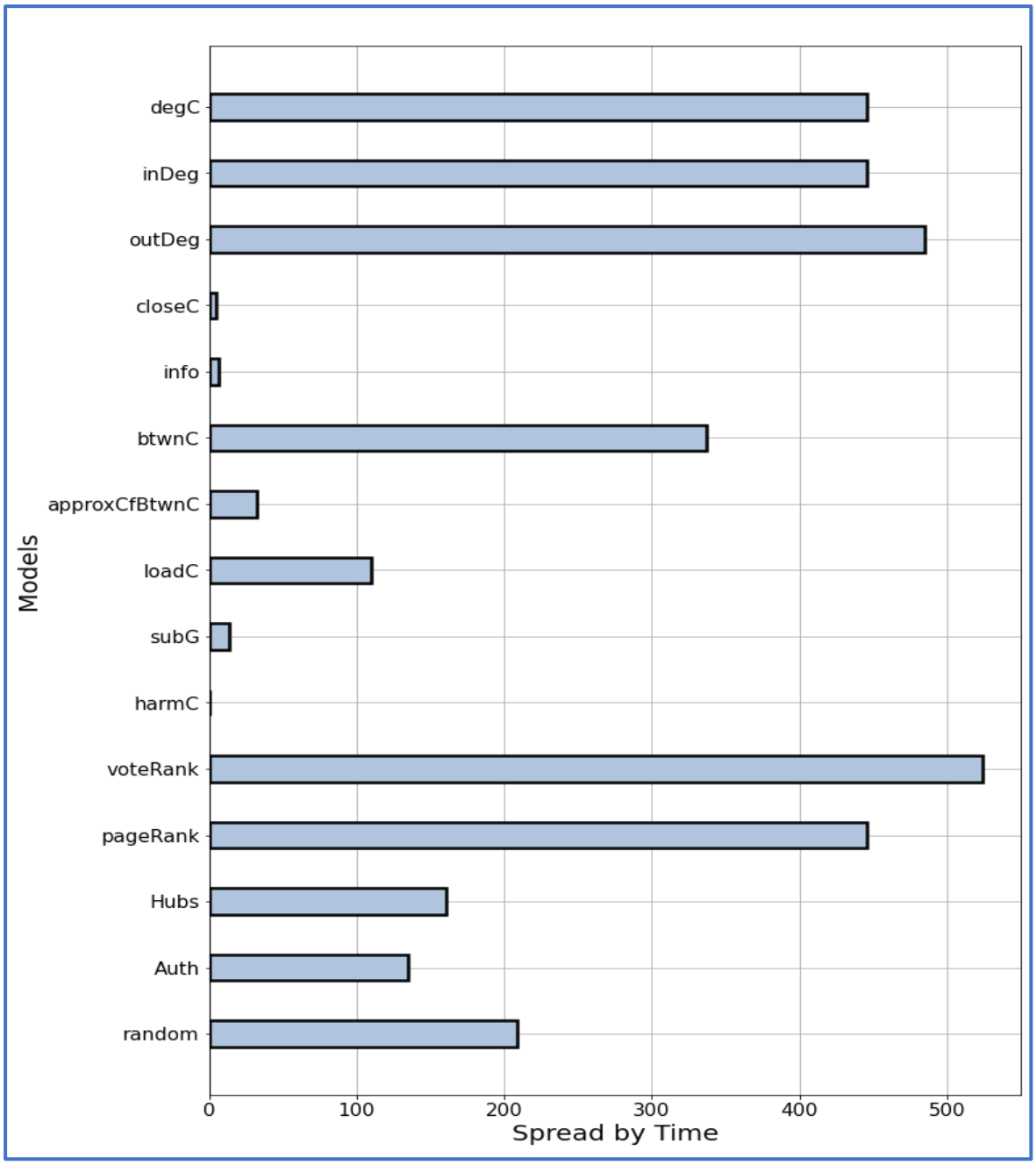


NetworkX models – random graph with randomised trust values:



NetworkX models – BitcoinOTC graph:





Evaluation

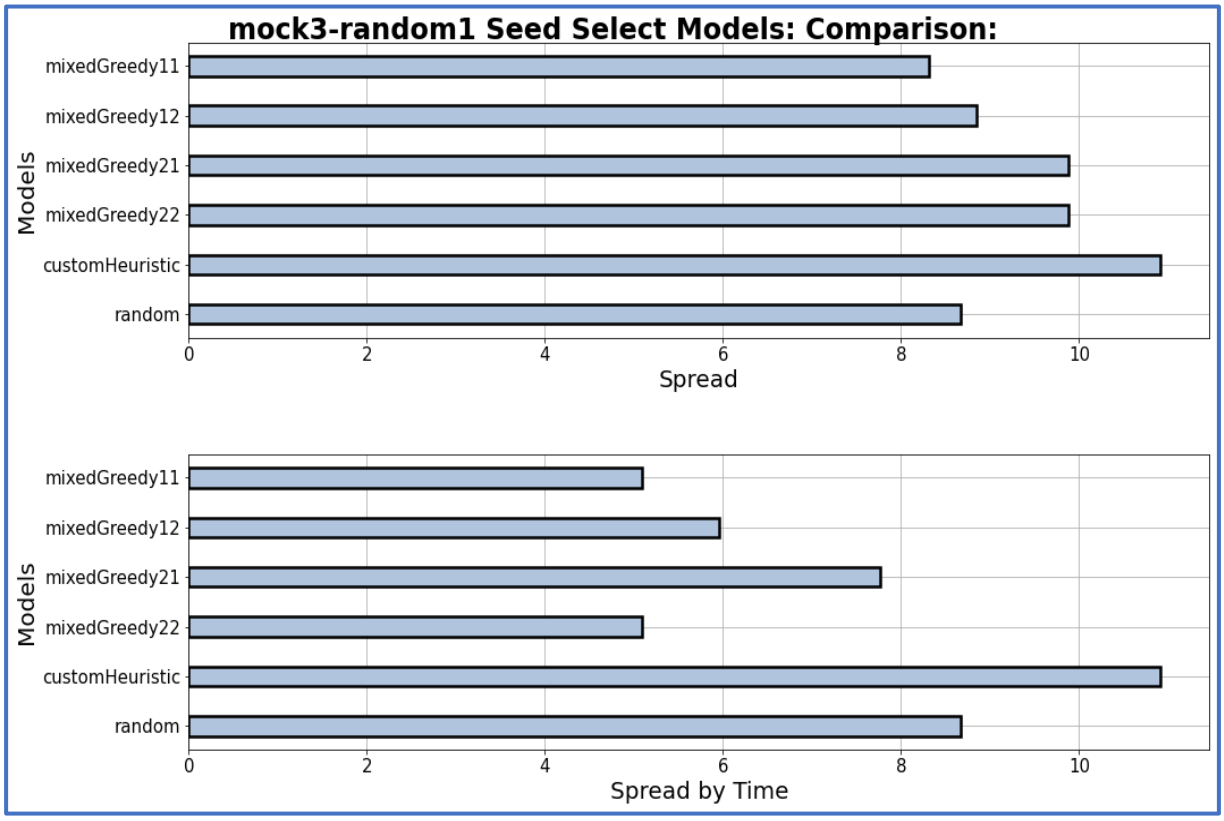
In both random graphs, considering time had no effect on spreads achieved, implying that all models selected seeds in a reasonable time. In the first graph, out degree centrality, vote rank and page rank were the best performing, and in the second graph, degree centrality, info centrality, load centrality, subgraph centrality and vote rank all perform the best.

However, considering time has a much more tangible effect on the Bitcoin graph. Closeness centrality and betweenness centrality performed the best on spread alone, but out-degree centrality and vote rank were the best by far when considering time also, with closeness centrality severely dropping down the rankings.

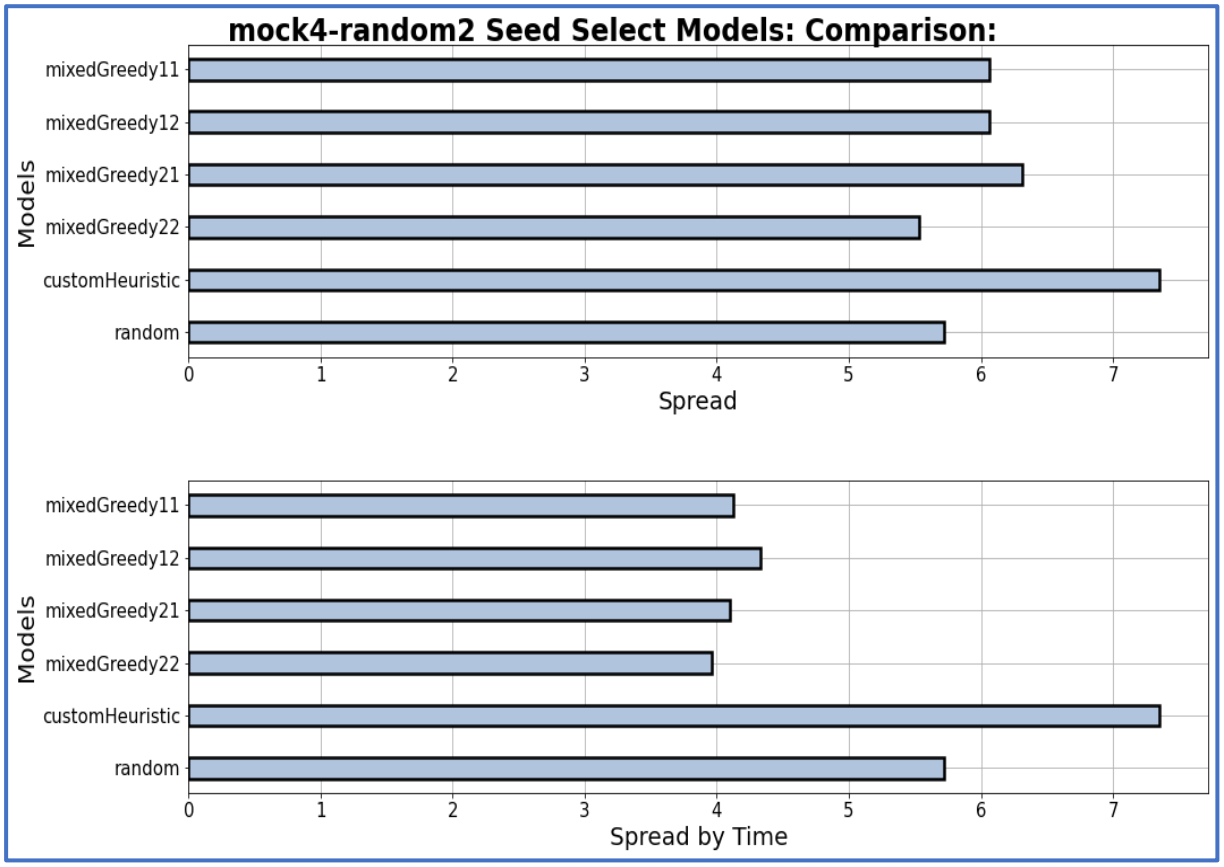
New Models

Results

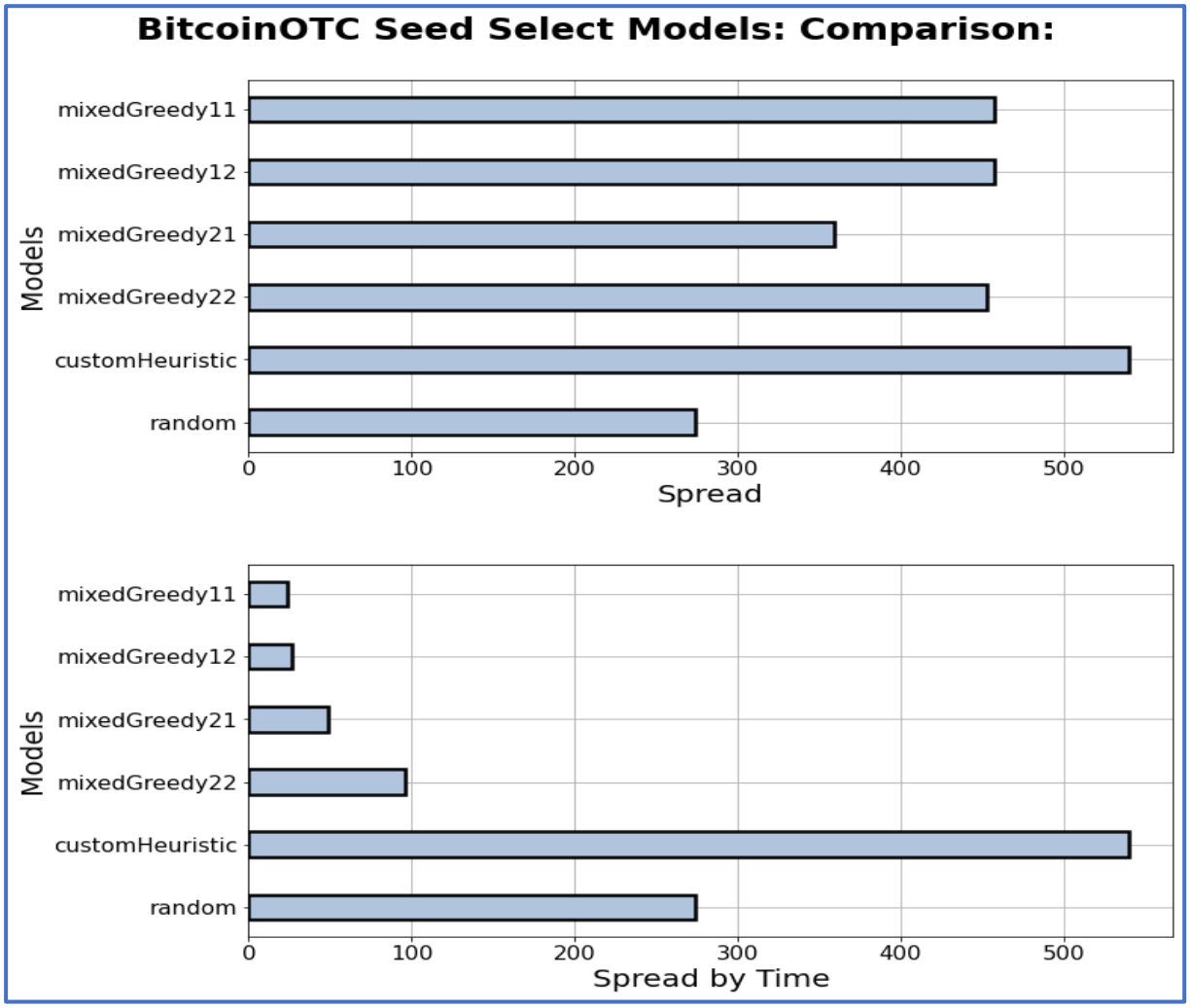
New models – random graph with uniform trust values:



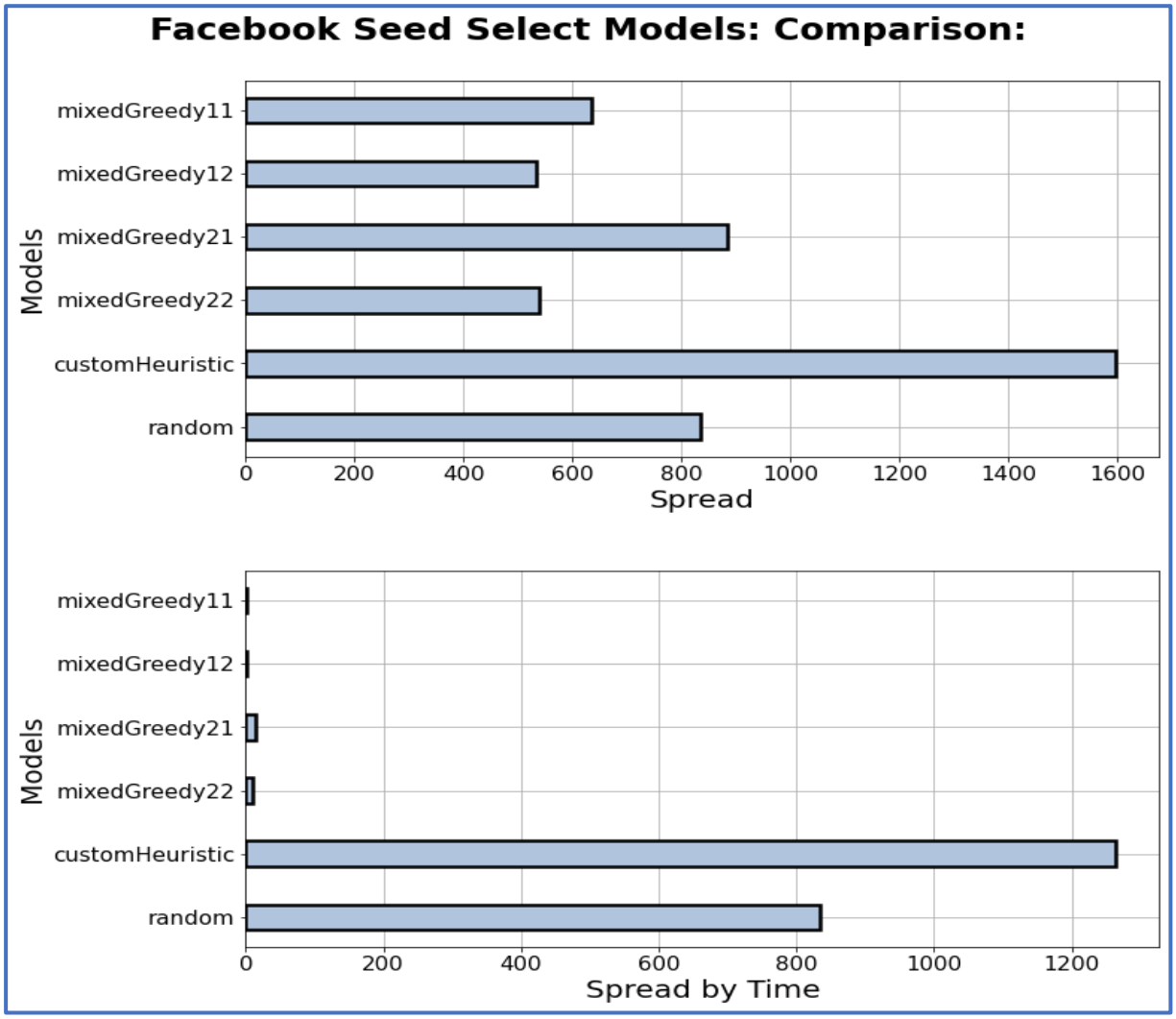
New models – random graph with randomized trust values:



New models – BitcoinOTC graph:



New models – Facebook graph:



Evaluation

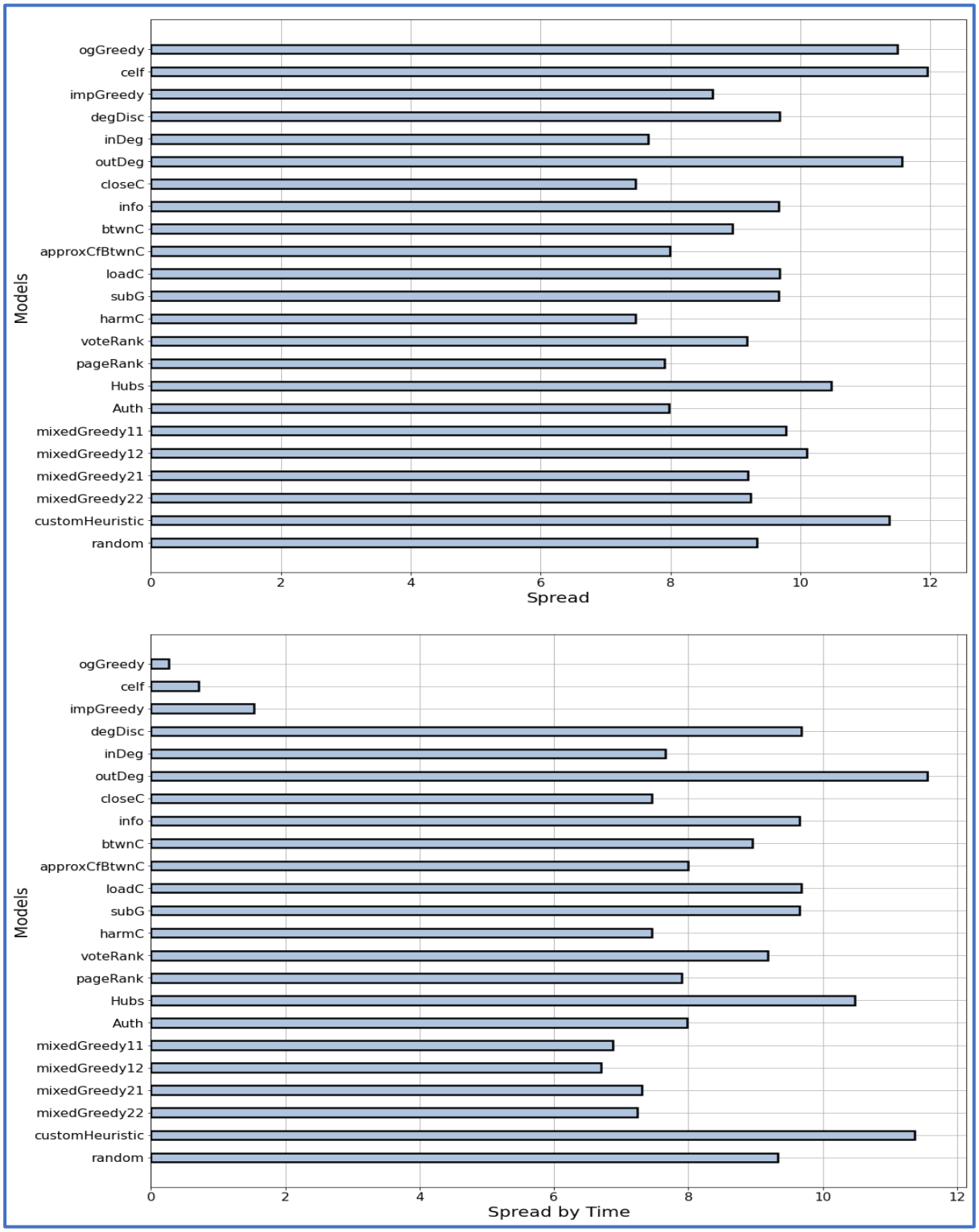
On both random graphs, custom heuristic performed the best, and the mixed greedy models had reasonable spreads until time was considered.

On the BitcoinOTC and Facebook graphs this was even more pronounced, with custom heuristic still being the best performing, but with the mixed greedy models dropping off even more steeply.

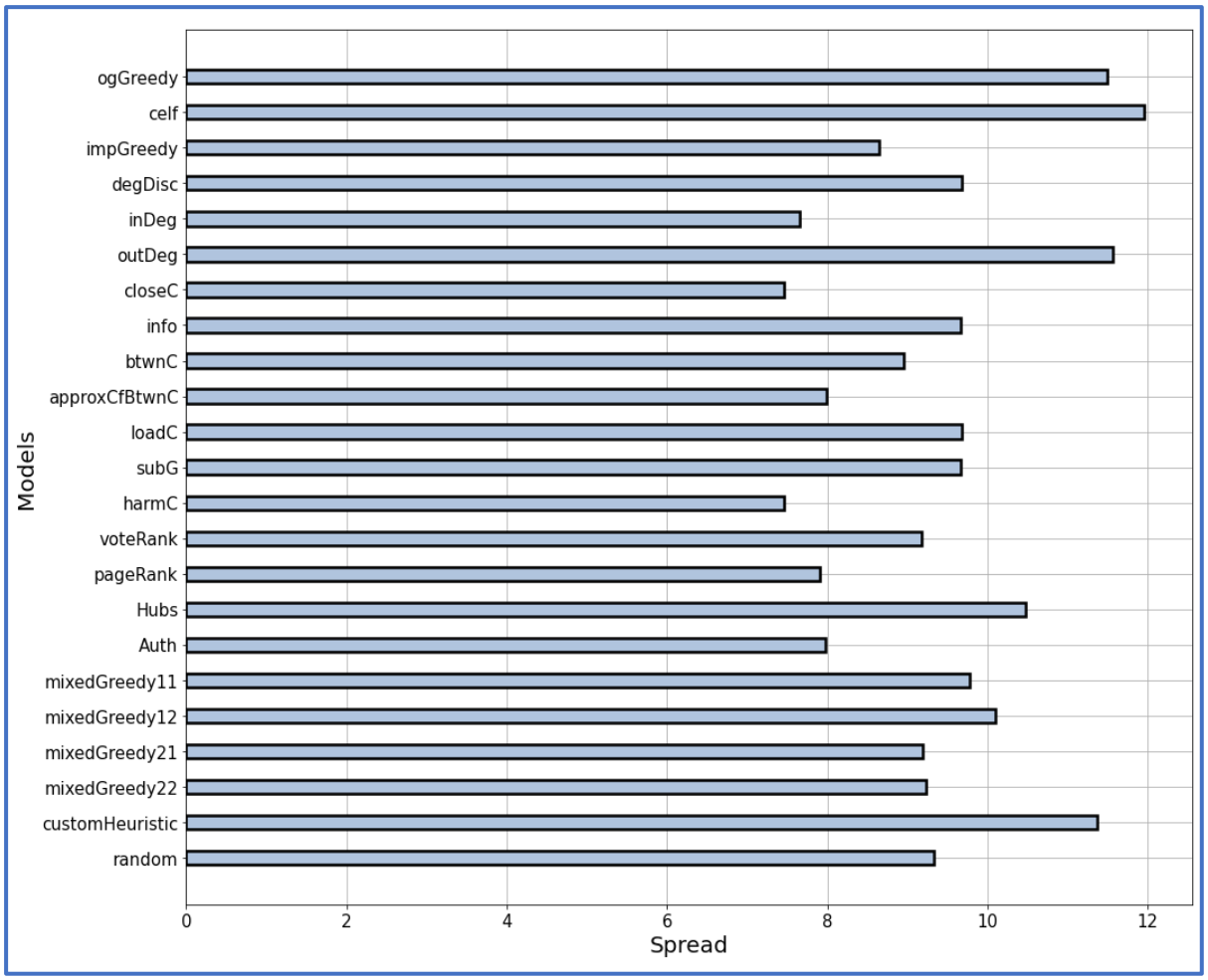
All Models

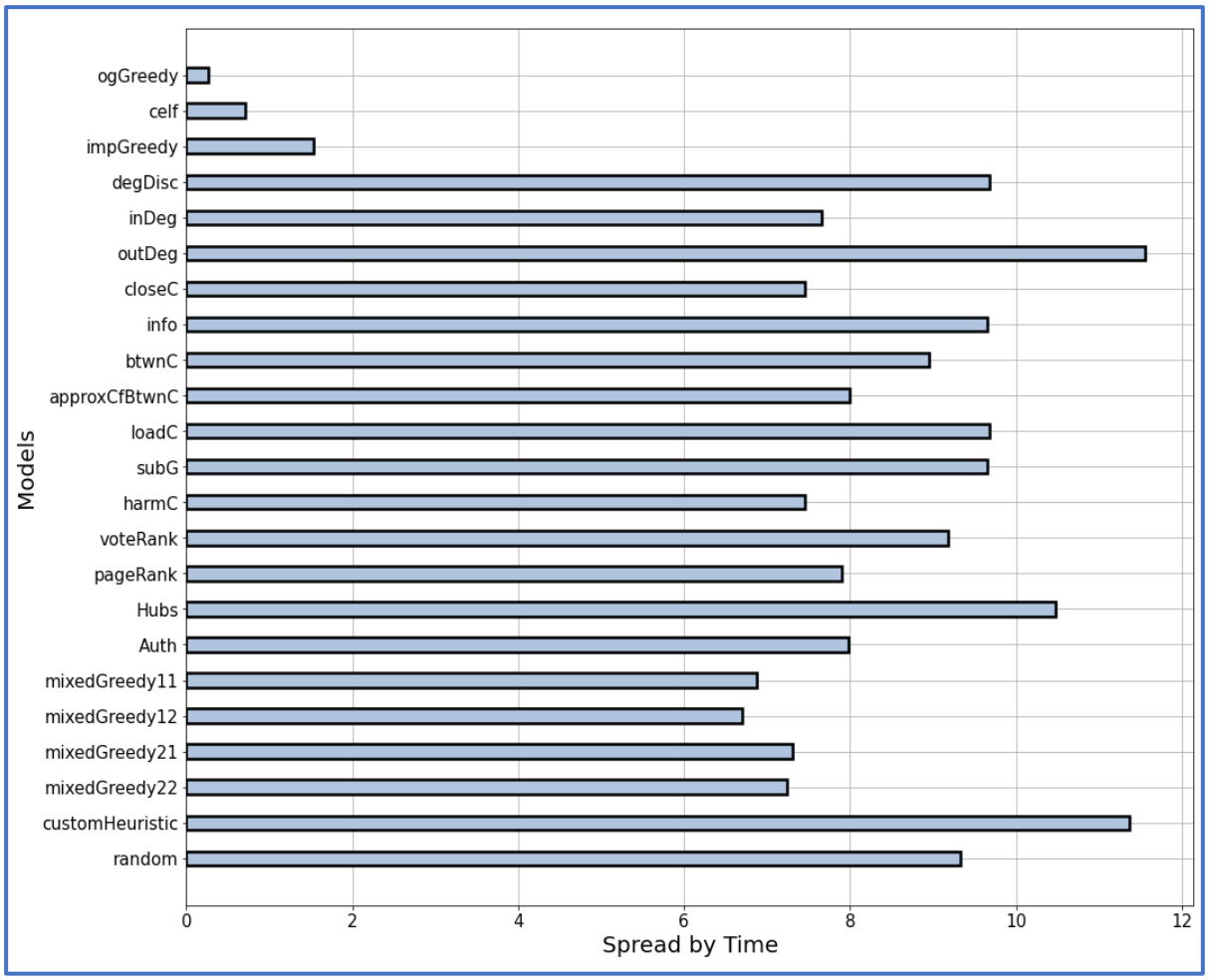
Results

All models – random graph with uniform trust values

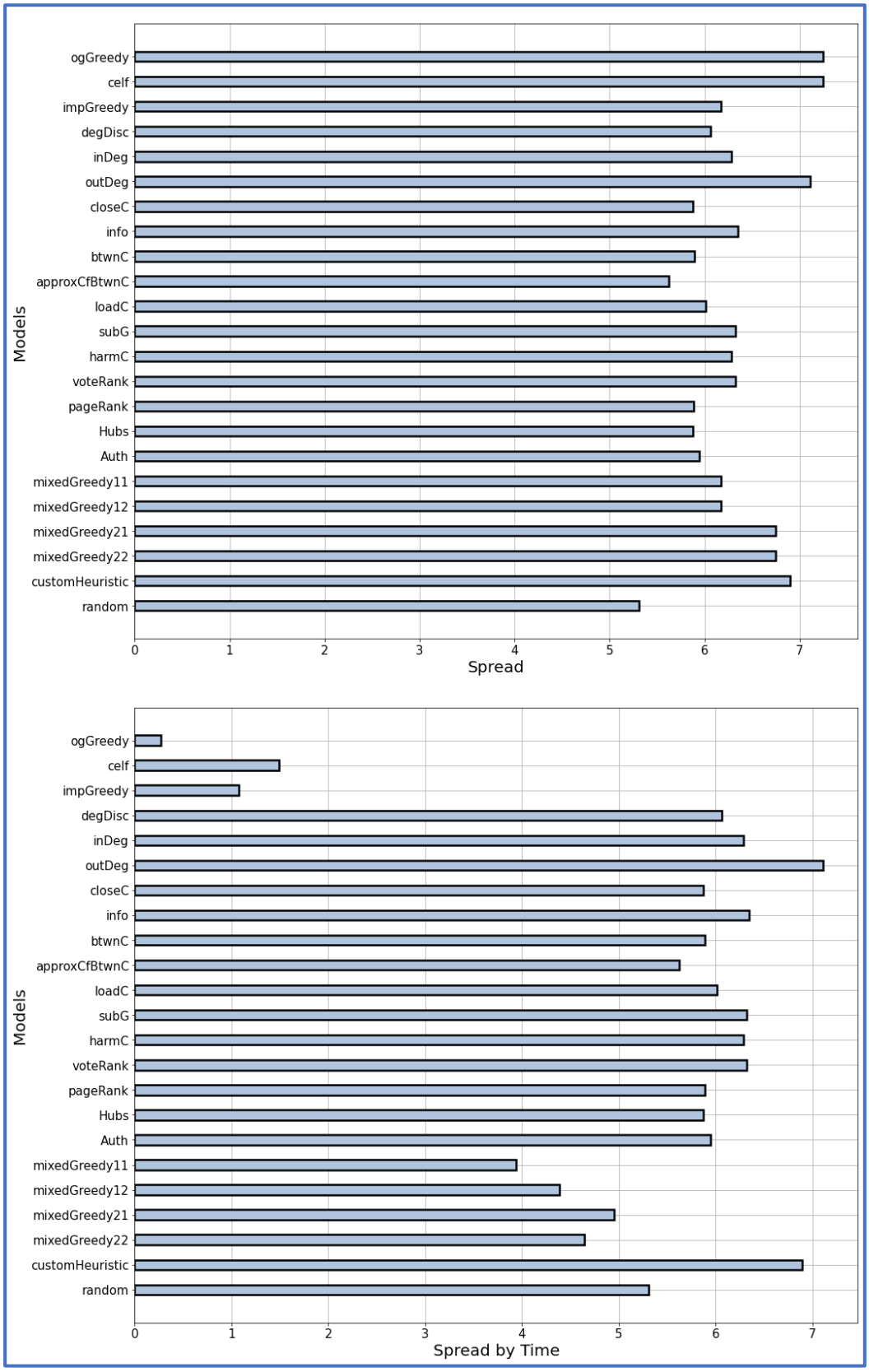


(zoomed out below)

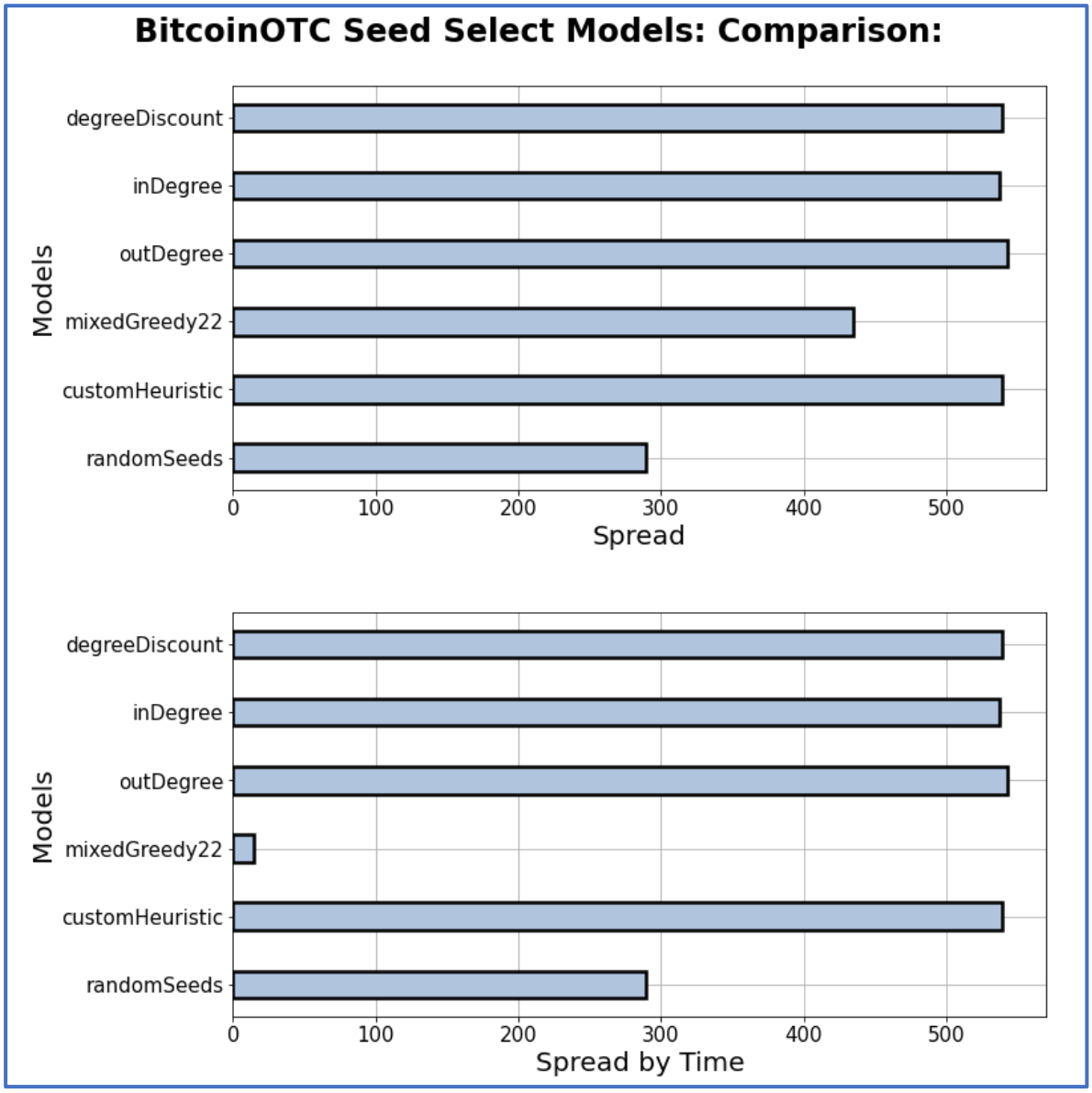

ogGreedy
celf
impGreedy
degDisc
inDeg
outDeg
closeC
info
btwnC
approxCfBtwnC
loadC
subG
harmC
voteRank
pageRank
Hubs
Auth
mixedGreedy11
mixedGreedy12
mixedGreedy21
mixedGreedy22
customHeuristic
random
Models
0
2
4
6
8
10
12
Spread


ogGreedy
celf
impGreedy
degDisc
inDeg
outDeg
closeC
info
btwnC
approxCfBtwnC
loadC
subG
harmC
voteRank
pageRank
Hubs
Auth
mixedGreedy11
mixedGreedy12
mixedGreedy21
mixedGreedy22
customHeuristic
random
Models
0
2
4
6
8
10
12
Spread by Time

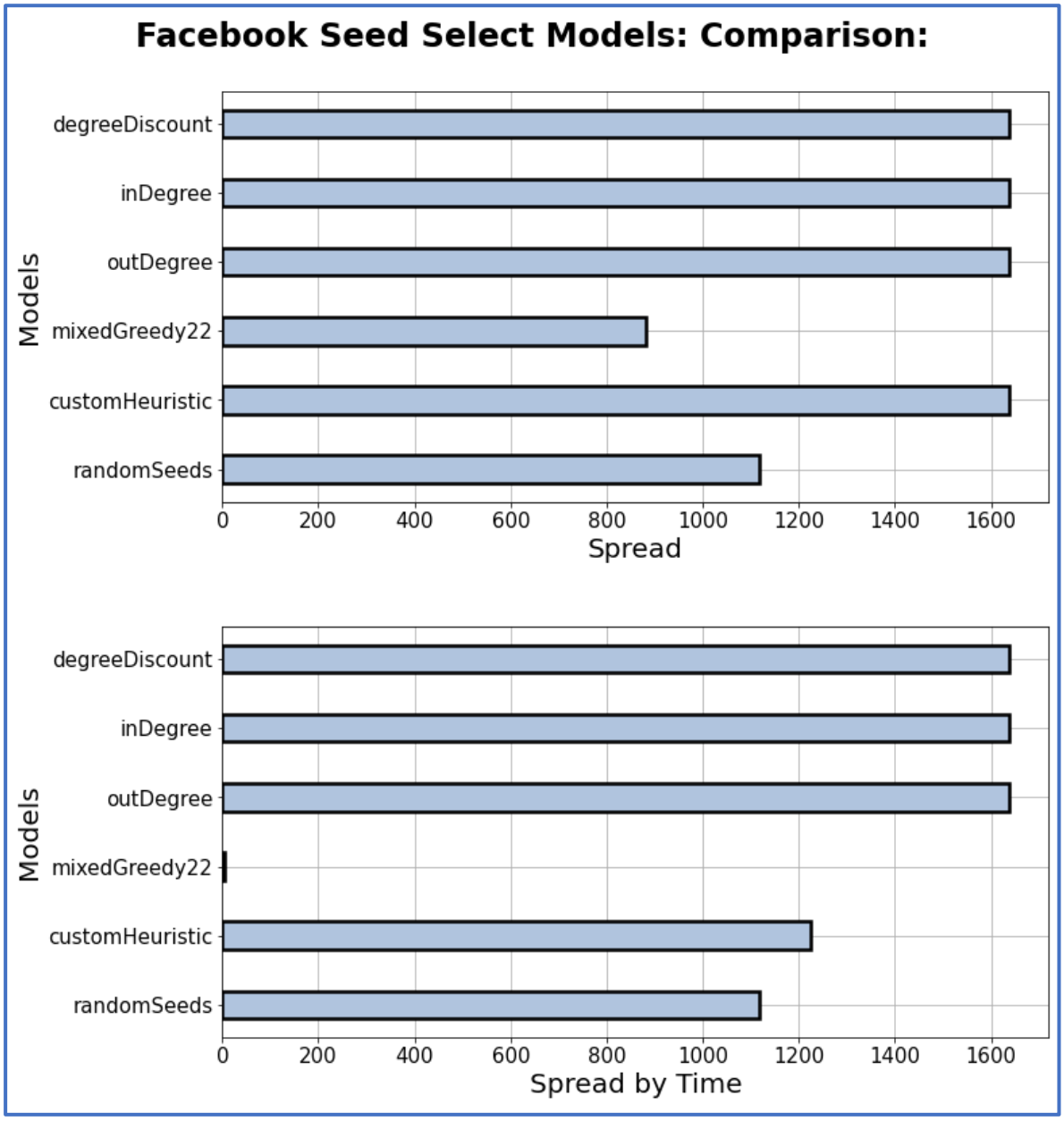
All models – random graph with randomized trust values



Some models (efficient enough to run) – BitcoinOTC graph



Some models (efficient enough to run) – Facebook graph



Evaluation

On the random graphs, CELF, original greedy, out degree centrality and custom heuristic performed the best by spread, and when time was considered CELF and original greedy went from some of the best to some of the worst.

On the real graphs, degree discount, in-degree centrality and out-degree centrality were consistently the best models, with custom heuristic performing much better on the BitcoinOTC graph than the Facebook graph.

Overall, the best models seem to be out-degree centrality, degree discount and custom heuristic.

5.7 Propagation Parameter Comparison

Design

Line graphs were plotted to indicate the effects of the parameters across a continuous range – 0 to 1 (0 to 0.1 for time factor).

A range of parameter values had to be used to illustrate the changes and correlations:

- Propagation probability = {0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1}
- Quality factor = {0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1}
- Switch factor = {0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1}
- Time factor = {0, 0.02, 0.04, 0.06, 0.08, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1}

Default values for parameters when not being altered and plotted:

- Propagation probability = 0.2
- Quality factor = 0.6
- Switch factor = 0.7
- Time factor = 0.04

Implementation

The function ‘compareVars’ for comparing and plotting multiple parameters at varying values using independent cascade can be found in Appendix C.1.

The function ‘compareVars2’ for comparing and plotting varying values of one parameter for all three propagation models can also be found in Appendix C.1.

Each propagation parameter (*pp*, *qf*, *sf*, *tf*) was tested on the independent cascade model (200 iterations) and across all three models (50 iterations), comparing spreads resulting from varying values. Finally, all parameters are compared in the independent cascade model (200 iterations).

Comparing *pp* in IC:

```
#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")
```

Comparing *pp* across all three propagation models:

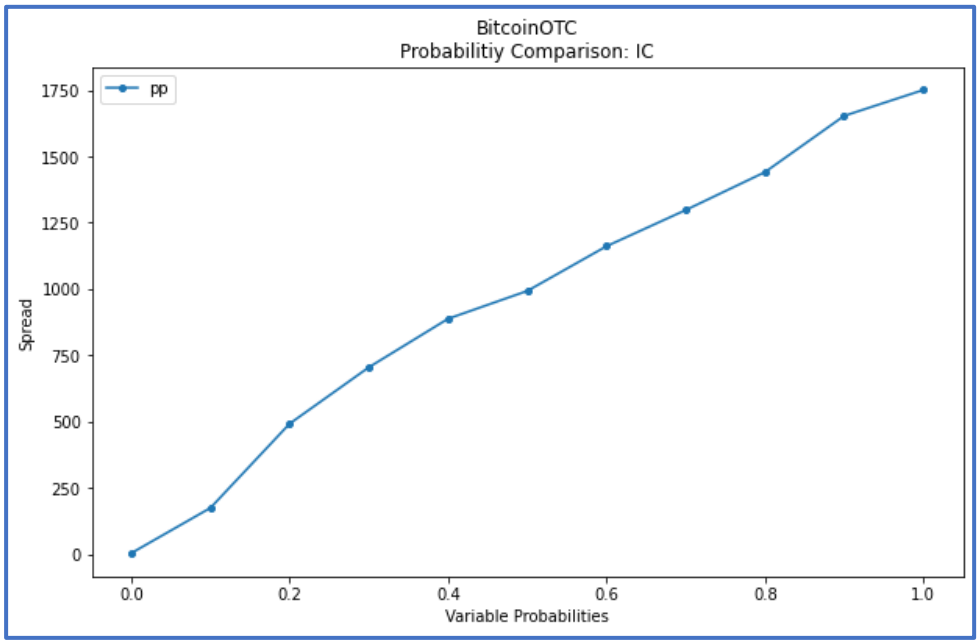
```
#pp experiment for all models
qty, its, models, gs = 5, 50, ['IC','WC1','WC2'], [graphs]
vss = [['pp', 0, [x*0.1 for x in range(11)]]]
startTime = time()
s = randomSeeds(graphs['BitcoinOTC'], qty)
print(g + "\nseed set: " + str(s) + "\n" +
      str(round((time() - startTime), 5)) + " secs\n")
compareVars2('BitcoinOTC',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

Comparing all parameters in IC:

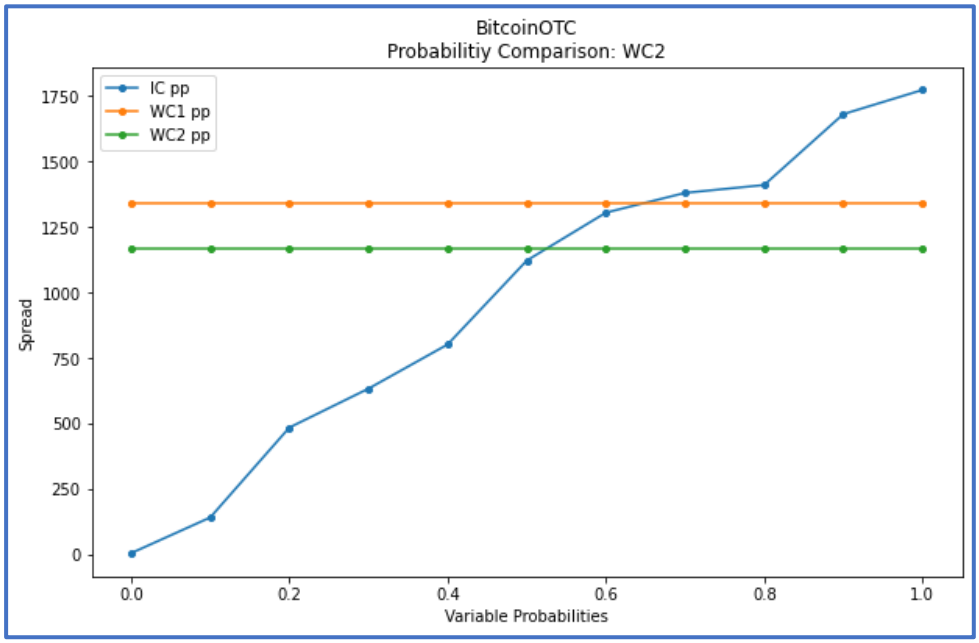
```
#all parameters experiment
qty, its, model = 5, 200, 'IC'
vss = [['qf', 0, [x*0.1 for x in range(11)]],
       ['sf', 0, [x*0.1 for x in range(11)]],
       ['pp', 0, [x*0.1 for x in range(11)]],
       ['tf', 0, [x*0.02 for x in range(5)] + [y*0.1 for y in
range(1,11)]]]
startTime = time()
s = randomSeeds(graphs['BitcoinOTC'], 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")
```

Results

Propagation probability – independent cascade:



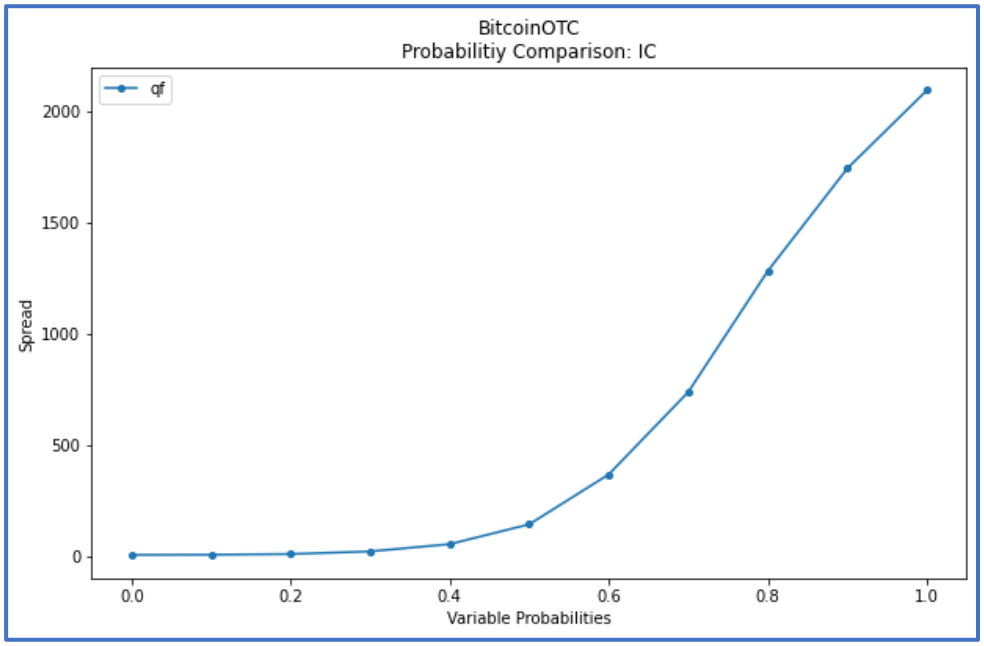
Propagation probability – all propagation models:



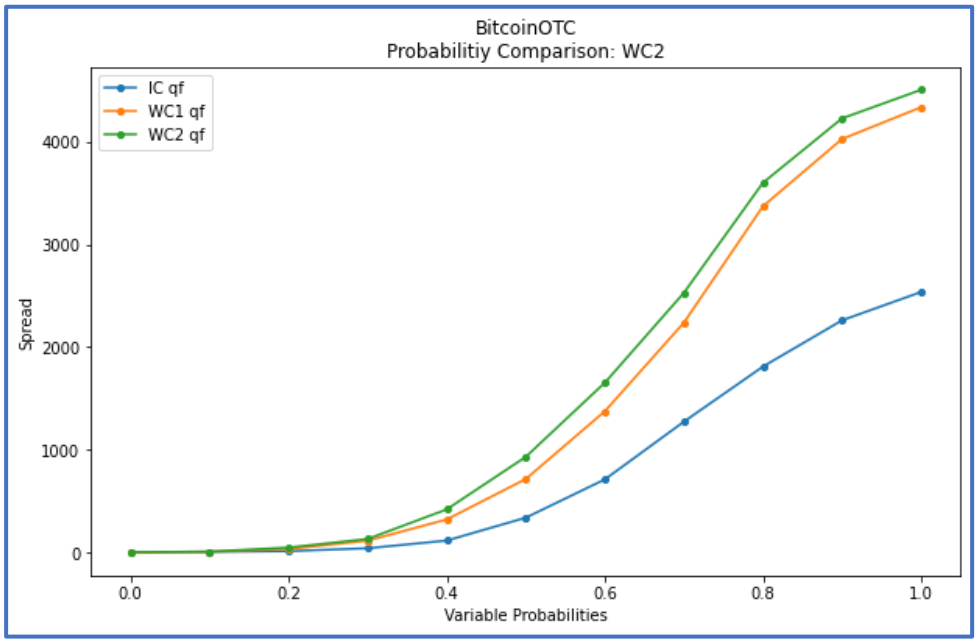
Evaluation

Propagation probability (*pp*) has a strong positive correlation with influence spread in the independent cascade model – increasing *pp* from 0 to 1 increased spread from 0 to ~1750. The weighted cascade models do not incorporate *pp* so remain constant throughout.

Quality factor – independent cascade:



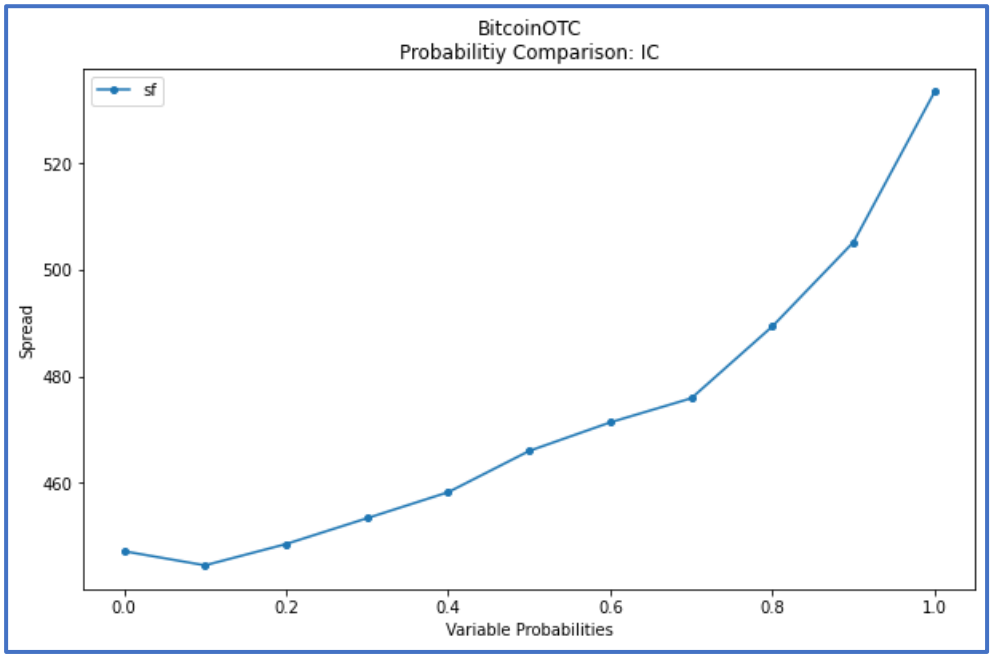
Quality factor – all propagation models:



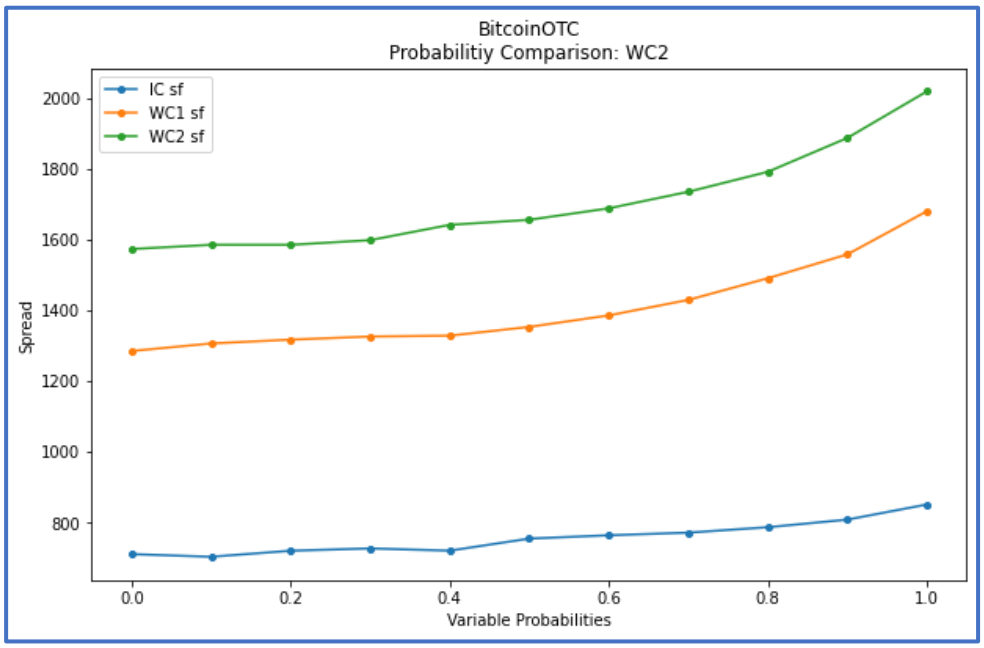
Evaluation

Quality factor (*qf*) has an increasingly positive correlation with influence spread – increasing *qf* from 0 to 0.5 increases spread from 0 to ~200, but increasing *qf* from 0.5 to 1 shows a rapid increase of spread from ~200 to ~2000 with the independent cascade model. The increasingly positive correlation is stronger with the weighted cascade models.

Switch factor – independent cascade:



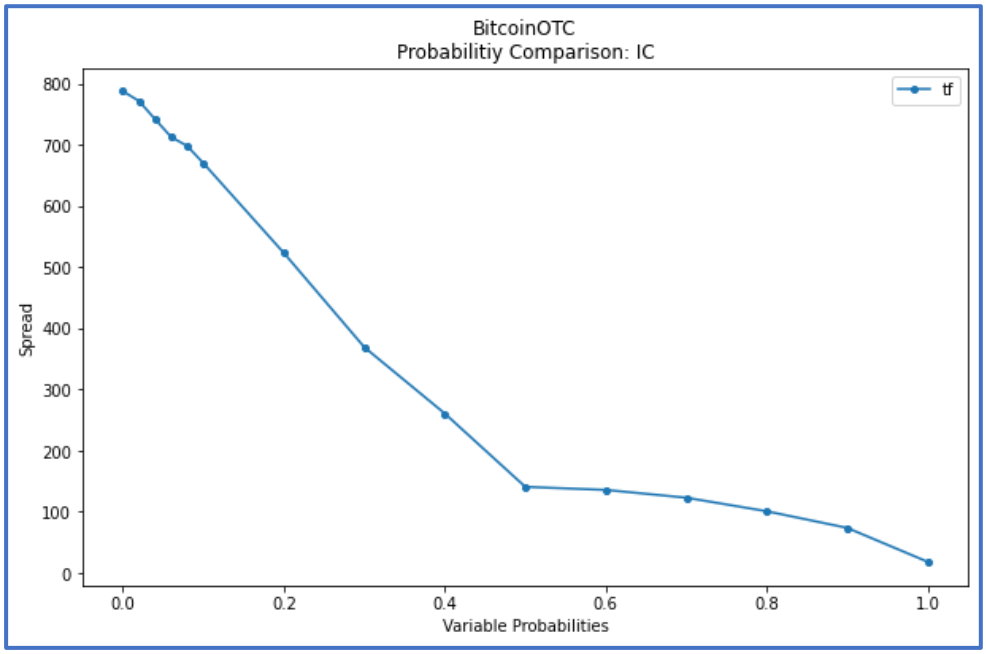
Switch factor – all propagation models:



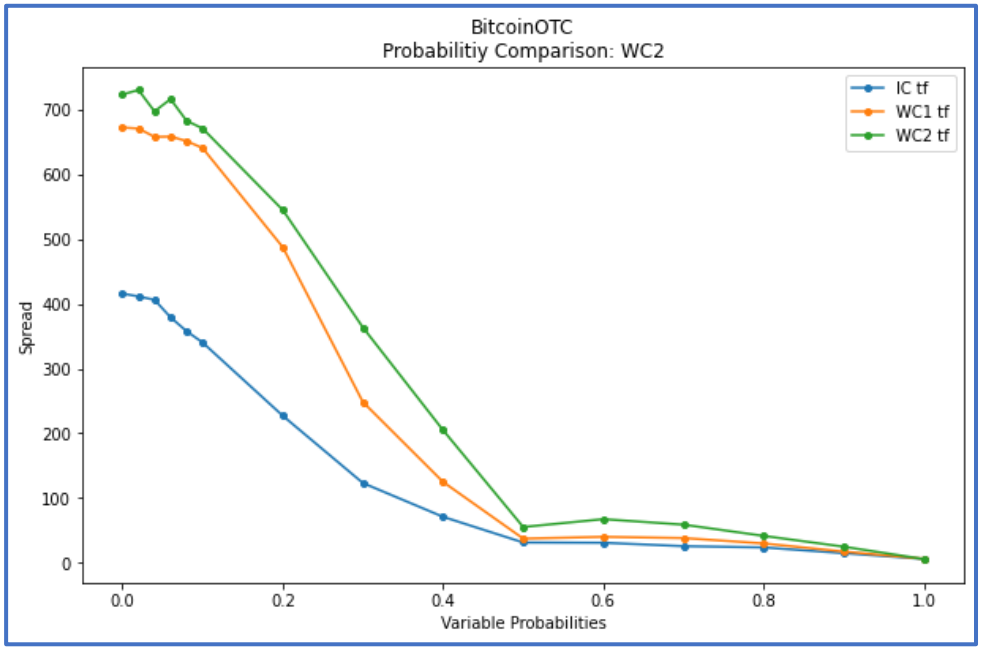
Evaluation

Switch factor (*sf*) has a very gradual positive correlation with influence spread, much lower than that of *pp* or *qf*, achieving a ~20% increase in spread over the full distribution of inputs with the independent cascade model. With the weighted cascade models, the correlation is more significant, but still quite gradual.

Time factor – independent cascade:



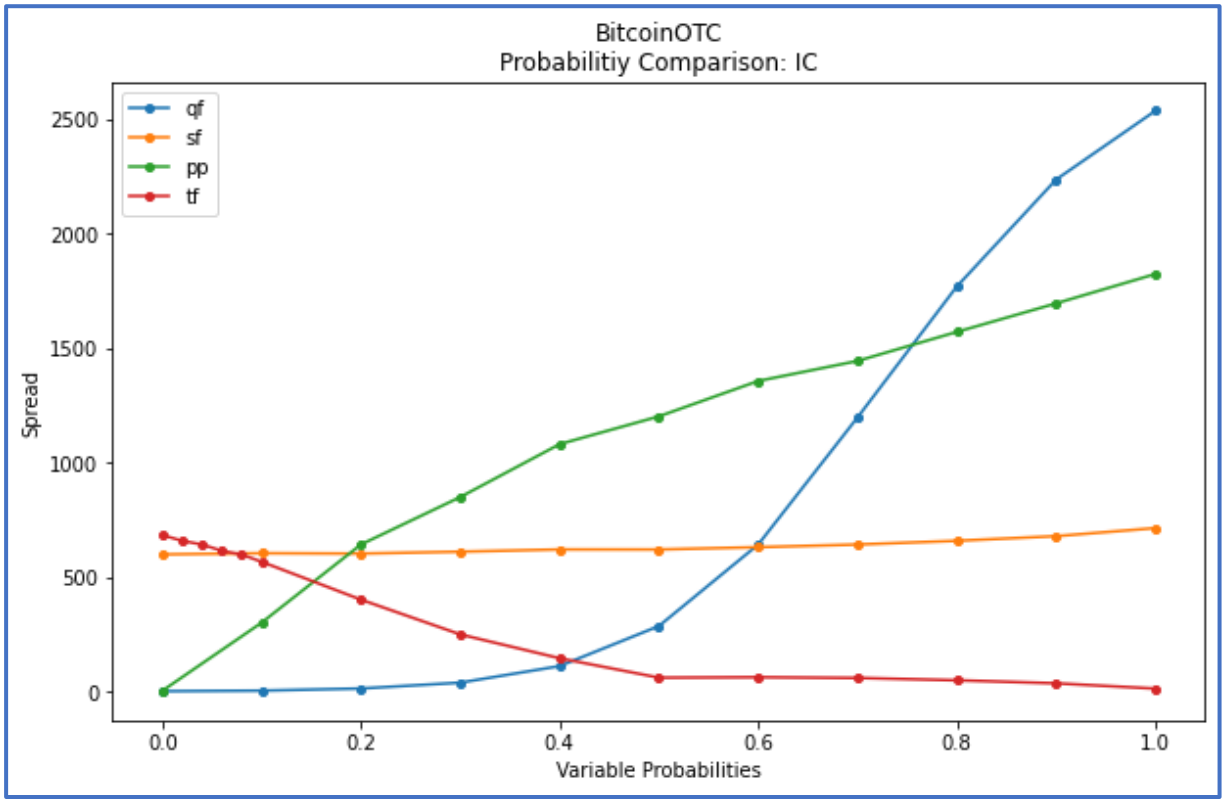
Time factor – all propagation models:



Evaluation

Time factor (*tf*) has a negative correlation with independent cascade influence spread – increasing *tf* from 0.01 to 0.1 results in a small decrease in spread from ~420 to ~370 , increasing from 0.1 to 0.5 shows a drastic drop off in influence spread from ~370 to ~50, and increasing from 0.5 to 1 steadily decreases the spread to 0. The correlation follows a similar trend with the weighted cascade models, beginning with higher spreads but dropping to the same lows by *tf* = 0.5.

All parameters – independent cascade



Evaluation

When all parameters are compared, propagation probability and quality factor are the most impactful on spread, with *pp* = 0.7 and *qf* = 0.7 returning very high spreads. Switch factor has a weak correlation in comparison, being more significant with the weighted cascade models. Time factor, when compared to the others, has a relatively small effect, affecting spread by less than 750 nodes. In contrast, quality factor can increase spread by +2000 nodes from *qf* = 0.5 to *qf* = 1.

Chapter 6: Conclusion

This chapter concludes the research undertaken, discusses the results, revisits the objectives and proposes future work that could be done in this field.

6.1 Objectives Revisited

6.1.1 Main Objectives

The first main objective of the project was to experiment with seed selection models and develop an original model that outperforms existing models. This objective was met as described in Sections 5.4 and 5.6, where multiple seed selection models were implemented and experimented with. The new models that were implemented all outperformed the original greedy, CELF and improved greedy models when time is taken into consideration, and the custom heuristic model was second only to the out-degree centrality and degree discount models. However, the new models created were not as original as initially intended, instead creating new models by augmenting and combining elements of existing models. The only completely original model, *Disconnect*, was never fully developed.

The second main objective of the project was to analyse and evaluate propagation parameters, their correlations and the insights they provide into real social networks, culminating in the implementation of an original parameter. This objective was definitely achieved, as laid out in Section 5.7 where experiments took place. Parameters from existing models – quality factor, time factor – were implemented and their effects on influence spread analysed, as well as a new parameter, the switch factor. This was an entirely original contribution, and the results from the switch factor provided unexpected insight into negativity bias and its effect on influence when influences can switch.

6.1.2 Sub Objectives

1. *Implement a general method for generating network graphs from datasets and different models to simulate influence propagation through them.*
   -Satisfied. See Sections 4.4, 4.5, 5.1 and 5.2 for successful network generation and propagation models. See Section 5.3.1 for probability analysis and normalization to maintain additional models and ensure functionality.

2. *Ensure network generation is generalised for reproducibility, and easy extension to other network graphs using other dataset files.*
   -Satisfied. See Section 5.5 and Appendix B.3 for completely generalized network generation that easily allows for new datasets to be additionally studied.

3. *Analyse propagation parameters and implement an original one, to better simulate social networks.*
   -Satisfied. See Sections 5.2, 5.5 and 5.7 for propagation parameters being implemented and analysed.

4. *Analyse and implement different seed selection models.*
   -Satisfied. See Section 5.6 for multiple implemented seed selection models.

5. *Perform statistical data analysis on networks, comparing their properties to influence spreads.*
   -Semi-satisfied. See Section 5.3 for extensive statistical network analysis, but these analyses were not applied to influence spreads for comparison.

6. *Implement a method of comparing propagation spreads and times with different networks, parameters and seed sets.*
   -Satisfied. See Section 5.7 for proof of comparing propagation spreads and times with different parameters and Section 5.6 for comparison of different seed sets and networks.

7. *Plot different types of graphs, to analyse and present different propagation models, propagation parameters and seed selection models, in regard to spread and time.*
   -Satisfied. See Sections 5.6 and 5.7 for various graphs used in analysis and experimentation.

8. *Using established literature, evaluate and interpret the results, providing insight into social interaction, and discuss accompanying implications.*
   -Discussed in upcoming Section 6.2. Chen's incorporation of negativity bias into quality factor was studied and used.

9. *Gain a deeper understanding of Python's research capabilities, and software regarding network graphs, using NetworkX*.
   -Satisfied. I have learnt a great deal about using NetworkX and more broadly about researching and plotting graphs using Python.

6.1.3 Research Questions

1. *What parameters affect the spread of influence most in simulations and do these translate to real-life?*
   -Quality factor, propagation probability, time factor and switch factor, in that order of significance (Section 5.7.). It remains totally unclear whether these parameters can actually be directly translated to real-life phenomena.

2. *What are the best approximation strategies for IM?*
   -Degree discount and out-degree centrality if efficiency is considered, CELF or original greedy if not (Section 5.6.).

3. *How could propagation models better reflect real social networks, and what limitations restrict them?*
   -Further propagation parameters in keeping with real human social phenomena. There are limitations regarding the datasets and how truly accurate they are (Section 1.5.).

4. *How accurately can simulations reflect real-life networks, and how could irrationality be introduced to a model?*
   -More real-world experiments would have to be undertaken to confirm any suspicions. Perhaps irrationality could be introduced through some ongoing random manipulation of probabilities throughout propagation, but again, human experiments would be necessary to substantiate any hypotheses.

6.1.4 Research Hypotheses

1. *Networks with more nodes will take longer to perform propagation on, accounting for the spread.*
   -Proven false. The Facebook graph consistently took longer to process and it had less nodes than the BitcoinOTC graph. It seems number of edges is much more significant than number of nodes.

2. *In a model where positive and negative influences occur and can 'steal' influence from the other, the more likely nodes are to switch influence, the influence with the higher general probability will increase.*
   -Proven false. Regardless of which influence was more likely, positive influence always benefited from a higher switch factor, probably due to it alleviating the effect of negativity bias within the model (Section 5.5.1.).

3. *New seed selection models based solely on network analysis metrics will outperform existing research models.*
   -Proven true. The out-degree centrality model performed better than many existing models. When time was considered, it outperformed every single seed selection model except sometimes degree discount, vote rank or custom heuristic (Section 5.6.).

6.2 Discussion

6.2.1 Seed selection

In terms of seed selection models, this project was a personal success. In Section 5.6 all models were experimented and compared, and it was seen that the customHeuristic model and out-degree centrality models outperformed past models like improved greedy and CELF. Only degree-discount consistently beat the original models which I consider to be a significant achievement. Linking back to Leskovec (Leskovec *et al.*, 2007) or (Kempe, Kleinberg and Tardos, 2003) who set the very foundations of this task, to outdo their model is satisfying and encouraging.

6.2.2 Propagation parameters

The analysis, experimentation and implementation of propagation parameters was an overall success. The results from Section 5.7 capture the exact effects of the parameters on the simulation – quality factor and propagation probability having huge effects, time factor and switch factor having more minor effect, and the insights gained from the switch factor regarding how it counteracts negativity bias is, I believe, a significant, original contribution. As discussed by Chen (Chen *et al.*, 2011) with the negativity bias and their quality factor, incorporating human nature into these simulations is beneficial in producing more meaningful results, and switch factor accomplished just that. Replicating a real human trait – changing one's mind – has been implemented and improved the realism of the model.

6.3 Future Work

There's a great deal of potential for future work in this field. Firstly, in the way of improving approximative solutions. As Chen stated (Chen, Wang and Yang, 2009), research should be focused on 'more effective heuristics' approaching the accuracy of greedy solutions, considering their extremely fast speeds. The heuristic models proposed so far have all used degree in quite simplistic ways, so it is likely better heuristic solutions could be found.

Another way in which future work could contribute to the field would be by way of expanding the realms of the problem to improve realism – formulating new propagation methods which more accurately represent human influence, implementing new propagation parameters which make existing propagation models more realistic or any other method of expanding the problem as is. There seems to be a distinct gap in how computer scientists and sociologists approach social network analysis and IM. Bridging that gap – by combining soft assumptions of theoretical sociology and algorithmic simulations of computer science – may produce opportunities for more practical applications.

Glossary

Network Graphs	Network-like graphs of nodes and edges (edges can be weighted), used to represent social networks. They can be directed or undirected, and weighted or unweighted.
Influence Propagation	The diffusion of influence through a network. Starting with a source of seed nodes, each node will attempt to influence its neighbours (connected nodes), and any influenced nodes will attempt to influence theirs, and so on.
Seed Selection	Selecting a specific number of optimal seed nodes from a graph, to maximise influence propagation.

References

Blower, S. and Go, M.-H. (2011) 'The importance of including dynamic social networks when modeling epidemics of airborne infections: does increasing complexity increase accuracy?', *BMC Medicine*, 9(1), p. 88. doi: 10.1186/1741-7015-9-88.

Borgatti, S. P. *et al.* (2009) 'Network Analysis in the Social Sciences', *Science*, 323(5916), pp. 892–895. doi: 10.1126/science.1165821.

Brownlee, J. (2020) *How to Scale Data With Outliers for Machine Learning*, *Machine Learning Mastery*. Available at: https://machinelearningmastery.com/robust-scaler-transforms-for-machine-learning/#:~:text=One%20approach%20to%20standardizing%20input,standardization%20or%20robust%20data%20scaling. (Accessed: 21 April 2021).

Chen, W. *et al.* (2011) 'Influence Maximization in Social Networks When Negative Opinions May Emerge and Propagate', in *Proceedings of the 2011 SIAM International Conference on Data Mining*. *The 2011 SIAM International Conference on Data Mining*, Mesa, AZ, USA: Society for Industrial and Applied Mathematics, pp. 379–390. doi: 10.1137/1.9781611972818.33.

Chen, W., Wang, Y. and Yang, S. (2009) 'Efficient influence maximization in social networks', in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '09*. *The 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, Paris, France: ACM Press, pp. 199–208. doi: 10.1145/1557019.1557047.

Data Preparation and Feature Engineering for Machine Learning: Normalization (2020) *Developers Google*. Available at: https://developers.google.com/machine-learning/data-prep/transform/normalization (Accessed: 21 April 2021).

Domingos, P. and Richardson, M. (2001) 'Mining the network value of customers', in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '01*. *The seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, San Francisco, CA, USA: ACM Press, pp. 57–66. doi: 10.1145/502512.502525.

ecampver (2013) *Best and/or fastest way to create lists in python*, *Stack Overflow*. Available at: https://stackoverflow.com/questions/20816600/best-and-or-fastest-way-to-create-lists-in-python (Accessed: 20 April 2021).

Hong, T. and Liu, Q. (2019) 'Seeds selection for spreading in a weighted cascade model', *Physica A: Statistical Mechanics and its Applications*, 526, p. 120943. doi: 10.1016/j.physa.2019.04.179.

Kempe, D., Kleinberg, J. and Tardos, E. (2003) 'Maximizing the Spread of Influence through a Social Network', in *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. *The ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, Washington, DC, USA: ACM, pp. 137–146. doi: 10.1145/956750.956769.

King, H. (2018) *Influence Maximization in Python - Greedy vs CELF*, *Hautahi*. Available at: https://hautahi.com/im_greedycelf (Accessed: 20 April 2021).

Leskovec, J. *et al.* (2007) 'Cost-effective outbreak detection in networks', in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '07*. *The 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, San Jose, CA, USA: ACM Press, pp. 420–429. doi: 10.1145/1281192.1281239.

Liu, B. *et al.* (2014) 'Influence Spreading Path and Its Application to the Time Constrained Social Influence Maximization Problem and Beyond', *IEEE Transactions on Knowledge and Data Engineering*, 26(8), pp. 1904–1917. doi: 10.1109/TKDE.2013.106.

Mantas Vidutis (2010) *Python Sets vs Lists*, *Stack Overflow*. Available at: https://stackoverflow.com/questions/2831212/python-sets-vs-lists (Accessed: 20 April 2021).

Mrvar, A. and Batagelj, V. (2016) 'Analysis and visualization of large networks with program package Pajek', *Complex Adaptive Systems Modeling*, 4(1), p. 6. doi: 10.1186/s40294-016-0017-8.

Normalization (2020) *CodeAcademy*. Available at: https://www.codecademy.com/articles/normalization#:~:text=Min%2Dmax%20normalization%20is%20one,decimal%20between%200%20and%201. (Accessed: 21 April 2021).

Richardson, M. and Domingos, P. (2002) 'Mining Knowledge-Sharing Sites for Viral Marketing', in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. *The eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, Edmonton, Alberta, Canada: ACM Press, pp. 61–70. doi: 10.1145/775047.

Tichy, N. M., Tushman, M. L. and Fombrun, C. (1979) 'Social Network Analysis for Organizations', *The Academy of Management Review*, 4(4), p. 507. doi: 10.2307/257851.

Appendices

Appendix A: Project Definition Document

Sean Sullivan

Course Title: IN3007

Consultant: Constantino Carlos Reyes-Aldasoro

Network Analysis and Competitive Influence Maximization within a Dynamic Directed Network, with Ethical Considerations

Proposal

Problems to be Solved:

How does information propagate throughout social networks? Social networks play an increasingly important role as a means to spread ideas, influence, and information among its users, and I plan to study in detail how we can understand and best utilise them to maximise this information propagation.

The problem I will be solving in my project is to find out and understand where within a social network original ads and ideas (seed nodes) would be best placed to maximise their spread and influence. The exact problem is as follows: given a number *k*, find within a network (nodes adjoined by edges) a *k* set of nodes which will result in the largest possible set of activated nodes, once a propagation method has been continuously applied to the network until no further nodes may be reached.

I will be doing this by generating and visualising a variety of network graphs from matrix-adjacency tables which I will generate based on publicly available datasets, performing network analysis, and implementing and experimenting with multiple different propagation methods (including one I will make myself) to maximise success, while also optimizing for efficiency.

This is a well-studied topic, given its rising relevance and importance in recent decades, and I will be building on the related work done over past years by many others whilst still ensuring I am making original and valuable contributions. One aspect that has not been studied thus far, is to explore the consequences of multiple competing influences within a dynamic (nodes appearing/disappearing) directed network graph, where both positive and negative influence may propagate. This will improve upon the so far mainly theoretical research, providing results which would be more applicable in the real world, where such aspects of the problem will always exist in tandem. Furthermore, I plan to invent and implement an entirely original propagation method (algorithm), to ensure I can contribute additional value to the topic.

Finally, due to the unavoidable prevalence of social networks and advertising in our lives I will use my findings to investigate the moral and ethical considerations of such technology and research, as well as how and why they may be used for malicious purposes. The ethics of advertising is not often considered by the public, despite the fact it pervades all of our lives constantly. I will explore ways in which we can safeguard ourselves and others from such malice, including technological safeguards and making the public more aware of the impact advertising and social networks – and potentially bad actors using them – could and will affect our lives.

Project Objectives:

Primary Objectives:

1. Implement a way to visualise large-scale (10,000+ nodes) social networks using network graphs.
2. Create a new propagation algorithm that improves upon its predecessors in success and/or efficiency and prove such with robust experimentation.
3. Explore the moral and ethical considerations of such technology.

Sub-Objectives:

1. Implement a way of representing network graphs visually from datasets.
2. Optimise this so it can be used for large-scale datasets and still provide insight.
3. Implement working propagation methods on small-scale custom datasets and test they work as expected.
4. Create and implement my own propagation methods, testing it works and improves upon others.
5. Experiment with all these methods on multiple datasets and collate my results.
6. Explore the morals and ethics of advertising and influence maximisation.

Project Beneficiaries:

Viral marketing is of great importance to advertisers in the modern age, and as such my results will be of great benefit to them, and many others who seek to spread information through a social network, including but not limited to: politicians running online campaign strategies, social media influencers, religious missionaries, etc. Of course, interest in my work will not be limited to those who would gain from it, but also include anyone who is interested in the algorithmic problem of Influence Maximization and the related field of network analysis.

Work Plan:

I understand I am extremely behind where I should be at this stage, but as you may or may not know I was extremely ill for almost the entirety of February, but I have applied and been granted a 2 week extension through EC. Nevertheless, I believe I will be able to finish this project to a high standard in that space of time, and have detailed my work plan from this day forward below:

1. Provide a literature review which studies previous related work and highlights how I can effectively and originally build on that and contribute further value. *Tues 23/03/21*
2. Write code to implement and compare multiple ways of representing large datasets with network graphs using Python. Also, compare the many large datasets applicable to this project and decide which few I use. *Fri 26/03/21*
3. Write code to implement various propagation methods (independent cascade, linear threshold, etc.) and verify they work successfully on very small custom-made network graphs. *Fri 2/04/21*
4. Optimise and improve until they are sufficient for the large-scale network graphs I will be using. *Tues 6/04/21*
5. Studying and comparing the various methods mentioned above, as well as different techniques used by many researchers over the past 18 years, create a completely original propagation method algorithm, and implement it in Python. *Sun 11/04/21*
6. Experiment with all of the implemented methods to collect and present results. *Sun 18/04/21*
7. Draft report and show to consultant. Fri *23/04/21*
8. Research the ethical and moral considerations of social networks and influence maximization and add to the report. *Weds 28/04/21*
9. Draft final report, including references and appendices. *Thu 13/05/21*
10. Submit final project and report. *Tue 18/5/21*

Project Risks:

1. Being delayed and missing deadlines. This would be disastrous for me, due to my already delayed situation.
   Solution: Avoid procrastination and break every task into subtasks with manageable deadlines.
2. I spend too much time on visualising large network graphs and don't leave enough time for experimenting.
   Solution: Prioritise implementation and results over visualisation, treat visualisation as an added extra if it takes up too much time.
3. My propagation method does not meet the standards I am attaining for.
   Solution: Leave a significant amount of time for implementing and testing propagation methods.
4. Living alone adds stress and subtracts time away from my situation.
   Solution: when I am able on March 29th, I will return to stay with my mother to keep on track.

Appendix B: Installation Guide

B.1 Files & Capabilities

This project is made up of 4 files:

1. cascade_final.py
    a. Generate networks from datasets
    b. Generate some user-defined graphs
    c. Perform propagation on a network with multiple optional parameters such as propagation probability, quality factor, propagation model, switch factor, time factor and more.
    d. Plot line graphs to compare the influence spreads of different models and different parameters.
2. network_analysis_final.py
    a. Generate networks from datasets
    b. Generate some user-defined graphs
    c. Calculate various network properties such as mutuality, density and more
    d. Create a list of degree reciprocals and relational degrees from a network
    e. Normalize those lists with various techniques
    f. Plot bar charts to compare network properties of networks
    g. Plot pie charts to represent probability distributions
    h. Plot histograms to represent probability distributions with various normalization techniques applied
3. seed_selection_final.py
    a. Generate networks from datasets
    b. Generate some user-defined graphs

c. Fine-tune NetworkX seed selection models (run NetworkX seed selection models multiple times with a range of different parameter values, printing the resulting spreads of each, to compare and find the optimal values)

d. Seed selection models – select seed nodes from a network with various different techniques such as original greedy, degree centrality, VoteRank and others.

e. Plot horizontal bar charts to compare different seed selection models in terms of spread or spread divided by a tenth of time taken to select seeds.

4. upgrades.py – This file is not necessary in order to run the code and test out the project, but instead is where old and new methods of code components are stored, as well as the functionality and efficiency tests that were ran to build the model.

B.2 Installation Guide

To run this program, you must have Python 3.6 installed; using Jupyter Notebook is recommended but not required. You should download the three Python files (cascade_final.py, network_analysis_final.py and seed_selection_final.py), the 2 dataset files (soc-sign-bitcoinotc.csv and facebook.csv) and any other dataset .csv files you wish to analyse. Then you should continue as described in the section below.

B.3 New Dataset(s)

To successfully run the program with the datasets used in this project, the file-paths will have to be redefined to wherever they are in your local files. In the first section of code:

```
#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 Project\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 Project\datasets\facebook.csv")
}
```

These lines must be modified.

New datasets can also be added to the program, and functionality is generalized so that you can apply the software to this new dataset with ease. Steps needed to implement new dataset:

1. Have the dataset in .csv file format, and make sure each row contains a target node (column1), a targeting node (column2) and optionally an edge weight (column3).

2. Enter in a new element in the dataset dictionary (seen above) on a new line in the format:
   "Name": (directed, weighted, file-path)
   where directed and weighted are Boolean values indicating whether the network you're implementing is directed and/or weighted (True or False), and where file-path is the path to the file in your local files written in an r-string (e.g.: r"file\path\goes\here.csv")

Appendix C: Source Code

C.1 cascade_final.py

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


#  ALL NECESSARY IMPORTS ::


# In[2]:


#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
Project\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
Project\datasets\facebook.csv")
}


# In[3]:


#  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


# In[4]:


#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][target]['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[targeting][target]['trust'])
    elif successModel == 'WC2':
        if a:
            relDeg = g[targeting][target]['relDeg']
        else:
            snd = sum([(g.out_degree(neighbour)) for neighbour in
g.predecessors(target)])
            relDeg = (g.out_degree(targeting) / snd)
            #relDeg = mmNormalizeSingle(log(g.out_degree(targeting)/snd))
        succ = (relDeg*successVars(sign, switch, qf,
sf)*timeDelay*g[targeting][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)


# In[5]:


#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,
neighbour, 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)


# In[6]:


#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,
negNew, 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,
reverse=True))[:int(np.mean(infCount))]
    #If nodes aren't being returned
    else:
        #Mean is returned
        result = np.mean(influence)
    return result


# In[7]:


#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")]


# In[8]:


#  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


# In[9]:


#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:]


# In[10]:


#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'])
    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


# In[11]:


#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("")


# In[12]:


#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))


# In[13]:


#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("")


# In[14]:


"""

#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("")


# In[15]:


#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 = 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)


# In[16]:


#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


# In[17]:


#  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


# In[18]:


#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 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(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")

#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("")


# In[19]:


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


# In[20]:


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


# In[21]:
```

```
#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()


# In[22]:


#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()


# In[22]:


#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


# In[34]:


#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


# In[23]:


#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


# In[23]:


#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


# In[24]:


#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


# In[24]:


#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


# In[22]:


#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


# In[25]:


#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


# In[23]:
```

```
#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")


# In[23]:


#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])
```

C.2 network_analysis_final.py

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


#  ALL NECESSARY IMPORTS ::


# In[ ]:


#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
Project\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
Project\datasets\facebook.csv")
}


# In[ ]:


#  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


# In[ ]:


#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:]


# In[ ]:


#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


# In[ ]:


#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("")


# In[ ]:


#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")


# In[ ]:


#  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.


# In[ ]:


#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("")


# In[ ]:


#  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


# In[ ]:


#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("")


# In[ ]:


#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("")


# In[ ]:


#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("")


# In[ ]:


#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_degree(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("")


# In[ ]:


#  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


# In[ ]:


#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][target]['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
Degrees")]

#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("")


# In[ ]:


#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


# In[ ]:


#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("")


# In[ ]:


#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
normalized", True)
    print("")
"""
print("")


# In[ ]:


#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("")


# In[ ]:


#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("")


# In[ ]:


#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
normalized", True)
    print("")
#"""
print("")


# In[ ]:


#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("")
```

```
# In[ ]:


#  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.


# In[ ]:


#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:",
fontsize=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("")


# In[ ]:


#Plot pie chart for probability distribution #2
def compareNetworksPie2(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(figsize=(10, 6))
    ax.grid(zorder=0)
    wedges = ax.pie(values, labels=fracnames,
                            wedgeprops=dict(width=0.5),
                            autopct='%1f%%', startangle=-30)
    #
    bbox_props = dict(boxstyle="square,pad=0.3",
                      fc="w", ec="k", lw=0.72)
    kw = dict(arrowprops=dict(arrowstyle="-"),
              bbox=bbox_props, zorder=0, va='center')

    ax.legend(wedges, ingredients)

    for i, p in enumerate(wedges):
        ang = (p.theta2 - p.theta1)/2. + p.theta1
        y = np.sin(np.deg2rad(ang))
        x = np.cos(np.deg2rad(ang))
        horizontalalignment = {-1: "right", 1: "left"}[int(np.sign(x))]
        connectionstyle = "angle,angleA=0,angleB={}".format(ang)
        kw["arrowprops"].update({"connectionstyle": connectionstyle})
        ax.annotate(recipe[i], xy=(x, y), xytext=(1.35*np.sign(x), 1.4*y),
                    horizontalalignment=horizontalalignment, **kw)

    fig.suptitle(gname + " - " + func[1] + " Probability Distribution:",
fontsize=16, fontweight='bold')
    fig.subplots_adjust(top=0.88)

#"""
for gs in [graphs]:
    for g in gs:
        for probFunc in probFuncs:
            compareNetworksPie2(gs[g], g, probFunc)
#"""
print("")


# In[ ]:


#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)


# In[ ]:


#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("")


# In[ ]:


#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 array 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("")


# In[ ]:


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


# In[ ]:


#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
probabilities
    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("")


# In[ ]:


#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
probabilities
    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("")
```

```
# In[ ]:


#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
probabilities
    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("")


# In[ ]:


#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("")


# In[ ]:


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("")


# In[ ]:


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("")
```

C.3 seed_selection_final.py

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


#  ALL NECESSARY IMPORTS ::


# In[2]:


#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
Project\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
Project\datasets\facebook.csv")
}


# In[3]:


#  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


# In[4]:


#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][target]['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][target]['trust']*timeDelay)
    elif successModel == 'WC2':
        if a:
            relDeg = g[targeting][target]['relDeg']
        else:
            snd = sum([(g.out_degree(neighbour)) for neighbour in
g.predecessors(target)])
            relDeg = (g.out_degree(targeting) / snd)
```

```
        succ = (relDeg*successVars(sign, switch, qf,
sf)*timeDelay*g[targeting][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)


# In[5]:


#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,
neighbour, 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)


# In[6]:


#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,
negNew, 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,
reverse=True))[:int(np.mean(infCount))]
    #If nodes aren't being returned
    else:
        #Mean is returned
        result = np.mean(influence)
    return result


# In[7]:


#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")]


# In[8]:


#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


# In[9]:


#  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


# In[10]:


#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:]
```

```
# In[11]:


#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


# In[12]:


#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("")


# In[13]:


#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()}


# In[14]:
```

```
#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)


# In[15]:


#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))
```

```
# In[16]:


#  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


# In[17]:


#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


# In[18]:


#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")


# In[19]:


#  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.


# In[20]:


#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_undirected(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]


# In[21]:


#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
propagations.
        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)


# In[22]:


#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("")


# In[23]:


#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("")


# In[24]:


#  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)


# In[ ]:


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


# In[ ]:


#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')]
```

```
# In[ ]:


#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_undirected(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]

#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')]


# In[ ]:


#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,
reverse=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,
reverse=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')]


# In[ ]:


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')]


# In[ ]:


#  GRAPHING FUNCTIONS ::


# In[25]:


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)


# In[26]:


#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)


# In[75]:


#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)


# In[89]:


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


# In[91]:


#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)


# In[95]:


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


# In[30]:


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


# In[109]:
```

```
#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)


# In[113]:


#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


# In[115]:


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


# In[35]:


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


# In[134]:


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


# In[139]:


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

```
# In[34]:


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


# In[34]:


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)


# In[ ]:
```

C.4 upgrades.py

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


#  ALL NECESSARY IMPORTS ::


# In[3]:


#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


# In[13]:


#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,
#  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(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")


# In[9]:


#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")


# In[22]:


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


# In[11]:


#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][target]['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[targeting][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[targeting][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,
successFunc, 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")


# In[16]:


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


# In[47]:


#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")


# In[42]:


#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][target]['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[targeting][target]['trust'])
    elif successModel == 'WC2':
        if a:
            relDeg = g[targeting][target]['relDeg']
        else:
            snd = sum([(g.out_degree(neighbour)) for neighbour in
g.predecessors(target)])
            relDeg = (g.out_degree(targeting) / snd)
            #relDeg = mmNormalizeSingle(log(g.out_degree(targeting)/snd))
        succ = (relDeg*successVars(sign, switch, qf,
sf)*timeDelay*g[targeting][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,
neighbour, 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,
negNew, 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,
reverse=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
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()}


# In[41]:


#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("")
```

```
# In[44]:


#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("")


# In[22]:


#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'


# In[26]:


#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
Project\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
Project\datasets\facebook.csv")
}
#"""


# In[27]:


#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:]


# In[28]:


#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 Project\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 Project\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
reciprocal
            if weighted:
                tr = (int(row[2])+10)/20
                dis = 1-tr
            newG.add_edge(int(row[1])-offset, int(row[0])-offset, trust=tr,
distance=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\Documents\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\Documents\Computer Science BSc\Year 3\Term 2\Individual
Project\datasets\facebook.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("")


# In[8]:


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


# In[22]:


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")


# In[29]:
```

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


# In[67]:


#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("")


# In[13]:


#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
reciprocal
            if weighted:
                tr = (int(row[2])+10)/20
                dis = 1-tr
            NG.add_edge(int(row[1])-offset, int(row[0])-offset, trust=tr,
distance=(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


# In[11]:


#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\Individual 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\Individual 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\Documents\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\Documents\Computer Science BSc\Year 3\Term 2\Individual
Project\datasets\facebook_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
compare
#1. 64.046   2. 58.004
#Second modular approach is faster, as expected, but not by much.

#print(compareGraphMethods(100))


# In[56]:


#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,
                              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("")


# In[73]:




# In[ ]:


#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")
"""


# In[ ]:


#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][target]['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_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[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_degree(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
#"""
```

```
# In[ ]:


#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


# In[ ]:


#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][target]['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][target]['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][target]['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][target]['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][target]['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


# In[ ]:


#Printing


# In[ ]:




# In[ ]:




# In[ ]:
```

```
#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
variables")

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 variables")

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 variables")

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
variables")

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
variables")

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 variables")

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 variables")

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
variables")
```

```
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
"""

"""


# In[ ]:


#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
network\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("")
```

```
# In[ ]:


#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,
"degreeDiscount2"),
            (degCSeeds, "degreeCentrality"), (inDegCSeeds,
"inDegreeCentrality"),
            (outDegCSeeds, "outDegreeCentrality"), (ccSeeds,
"ClosenessCentrality"),
            (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-
random2"],  4, 100, 'IC'),
                                         celfSeeds(mockGraphs["mock4-
random2"], 4, 100, 'IC'),
                                         impGreedySeeds(mockGraphs["mock4-
random2"], 4, 100),
                                         degDiscSeeds(mockGraphs["mock4-
random2"], 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[seedSpread])
    print(y)
    vertBar(xLabels, y, (g + " Seed Select Models: Spread / Time"))


#"""


# In[ ]:


#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)
#"""


# In[ ]:


#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("")
```

C.5 Latex Algorithm Code

```
\title{AlgorithmTemplate}
\author{Sean}

\documentclass[12pt]{article}
\usepackage{fullpage}
\usepackage[T1]{fontenc}
\usepackage[linesnumbered,ruled,vlined]{algorithm2e}
\include{pythonlisting}

\begin{document}

\begin{algorithm}
\SetAlgorithmName{Propagation Model}
\SetAlgoLined
\DontPrintSemicolon
%\KwIn{\textit{Graph G, Seeds S, Iterations R, Probability P}}
%\KwOut{\textit{Mean spread of influence}}
$\textsf{totalSpread} \gets \textsf{0}$ \\
\For{$\textsf{iter} = \textsf{1 to R }$}{
 $\textsf{tried} \gets \emptyset$ \\
 $\textsf{newNodes} \gets \textsf{S}$ \\
 \While{$\textsf{newNodes} \neq \emptyset  $}{
 $\textsf{currentNodes} \gets \textsf{newNodes} $\\
         $\textsf{newNodes} \gets \emptyset $ \\
        \For{$\textsf{each node x : x} \in \textsf{newNodes }$}{
             \For{$\textsf{each node y : y} \in \bf N_x \textsf{ and y}
\notin \textsf{S }$}{
                %\tcc{\small{($\bf{N_x}$ = set of node x's neighbour
nodes)}}
                 \If{$\textsf{Random(0, 1)} <  \textsf{P}$}{
                     $\textsf{newNodes += node y}$ \\
                   }
                 $\textsf{tried += (node x, node y)}$ \\
                 \tcp{\small{(line 12 alternative: tried += node y)}}
                }
                }
         $\textsf{S = S} \cup \textsf{currentNodes} \cup
\textsf{newNodes}$}
    $\textsf{totalSpread += Len(S)}$}
\Return $\textsf{(totalSpread / R)}$
\caption{{\bf Independent Cascade (G, S, R, P)} \label{Algorithm}}
\end{algorithm}

\begin{algorithm}
\SetAlgorithmName{Propagation Model}
\SetAlgoLined
\DontPrintSemicolon
%\KwIn{\textit{Graph G, Seeds S, Iterations R}}
%\KwOut{\textit{Mean spread of influence}}
$\textsf{totalSpread} \gets \textsf{0}$ \\
\For{$\textsf{iter} = \textsf{1 to R }$}{
 $\textsf{tried} \gets \emptyset$ \\
 $\textsf{newNodes} \gets \textsf{S}$ \\
 \While{$\textsf{newNodes} \neq \emptyset  $}{
 $\textsf{currentNodes} \gets \textsf{newNodes} $\\
         $\textsf{newNodes} \gets \emptyset $ \\
        \For{$\textsf{each node x : x} \in \textsf{newNodes }$}{
             \For{$\textsf{each node y : y} \in \bf N_x \textsf{ and y}
\notin \textsf{S }$}{
```

```
                %\tcp{\small{($\bf{N_x}$ = set of node x's neighbour
nodes)}}
                 \If{$\textsf{Random(0, 1)} <  \textsf{(1 / in-
degree(y))}$}{
                     $\textsf{newNodes += node y}$ \\
                    }
                 $\textsf{tried += (node x, node y)}$ \\
                 \tcp{\small{(line 12 alternative: tried += node y)}}
                }             }
         $\textsf{S = S} \cup \textsf{currentNodes} \cup
\textsf{newNodes}$}
    $\textsf{totalSpread += Len(S)}$}
\Return $\textsf{(totalSpread / R)}$
\caption{{\bf Weighted Cascade 1 (G, S, R)} \label{Algorithm}}
\end{algorithm}

\begin{algorithm}
\SetAlgorithmName{Propagation Model}
\SetAlgoLined
\DontPrintSemicolon
%\KwIn{\textit{Graph G, Seeds S, Iterations R}}
%\KwOut{\textit{Mean spread of influence}}
$\textsf{totalSpread} \gets \textsf{0}$ \\
\For{$\textsf{iter} = \textsf{1 to R }$}{
 $\textsf{tried} \gets \emptyset$ \\
 $\textsf{newNodes} \gets \textsf{S}$ \\
 \While{$\textsf{newNodes} \neq \emptyset  $}{
 $\textsf{currentNodes} \gets \textsf{newNodes} $\\
         $\textsf{newNodes} \gets \emptyset $ \\
        \For{$\textsf{each node x : x} \in \textsf{newNodes }$}{
            \For{$\textsf{each node y : y} \in \bf N_x \textsf{ and y}
\notin \textsf{S }$}{
                %\tcp{\small{($\bf{N_x}$ = set of node x's neighbour
nodes)}}
                $\textsf{SND} \gets 0 $ \\
                \For{$\textsf{each node z : z} \in \bf N_y$}{
                $\textsf{SND += out-degree(z)}$}
                 \If{$\textsf{Random(0, 1)} <  \textsf{(out-degree(x) /
SND)}$}{
                     $\textsf{newNodes += node y}$ \\
                    }
                 $\textsf{tried += (node x, node y)}$ \\
                 \tcp{\small{(line 12 alternative: tried += node y)}}
                }             }
         $\textsf{S = S} \cup \textsf{currentNodes} \cup
\textsf{newNodes}$}
    $\textsf{totalSpread += Len(S)}$}
\Return $\textsf{(totalSpread / R)}$
\caption{{\bf Weighted Cascade 2 (G, S, R)} \label{Algorithm}}
\end{algorithm}

\end{document}
```

Appendix D: Full Results

D.1 Influence Model Build #1 Testing

Network Generation Testing

Edge:	*Present in BitcoinOTC Graph:*
1 -> 5 (6 -> 2 in dataset)	True
4 -> 5 (6 -> 4 in dataset)	True
14 -> 0 (1 -> 15 in dataset)	True

Propagation Model Testing

Propagation Probability (pp)	*Influence Spread*	*Time elapsed*
0.1	1.42	0.001
0.8	4.56	0.003

D.2 Influence Model Build #2

Modularized Network Generation Testing

Network Generation Method:	*Graph 1 Size (edges), Length (nodes)*	*Graph 2 Size (edges), Length (nodes)*	*Graph 3 Size (edges), Length (nodes)*
Method 1 (manual)	35587, 5875	176468, 4039	11, 10
Method 2 (semi-modularised)	35587, 5875	176468, 4039	11, 10

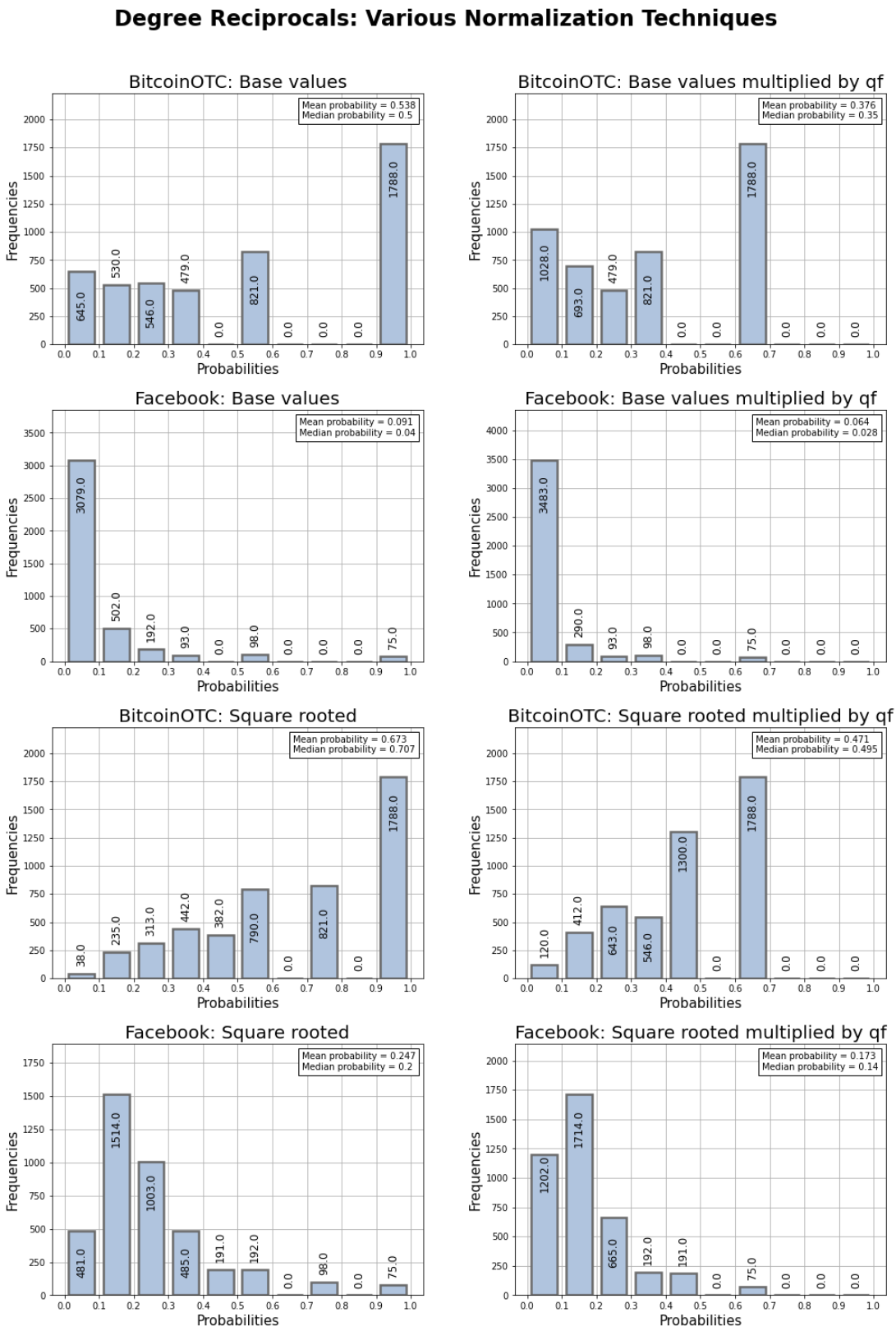
Quality Factor, Weighted Cascade 1 & Weighted Cascade 2 Models Testing

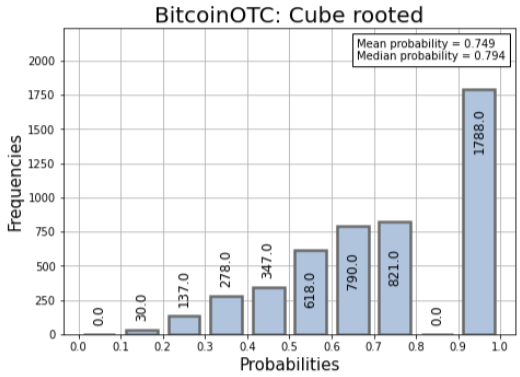
Graph:	*Model (Build #2):*	*Propagation Probability (pp):*	*Quality Factor (qf):*	*Spread:*	*Time elapsed (50 iterations):*
BitcoinOTC	Independent Cascade	0.2	0.2	7.32	5.458 secs
			0.8	759.1	7.438 secs

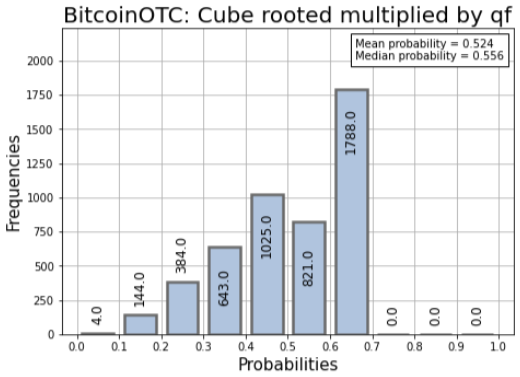
	Weighted Cascade 1	0.2	0.2	5.68	0.524
			0.8	27.8	0.2777 secs
	Weighted Cascade 2	0.2	0.2	353.44	19.153 secs
			0.8	3593.74	20.76 secs
Facebook	Independent Cascade	0.2	0.2	0.88	45.028 secs
			0.8	1560.36	37.475 secs
	Weighted Cascade 1	0.2	0.2	1.28	0.039 secs
			0.8	2.68	0.032
	Weighted Cascade 2	0.2	0.2	0.0	0.368 secs
			0.8	1696.64	98.8408

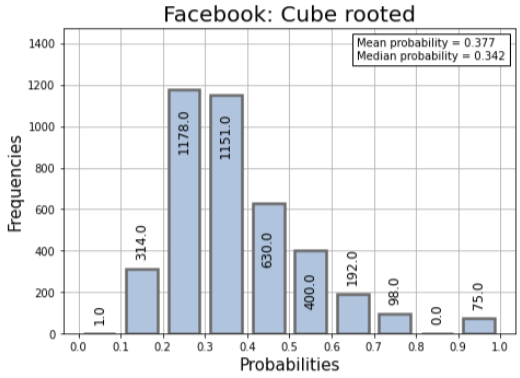
D.3 Probability Histograms

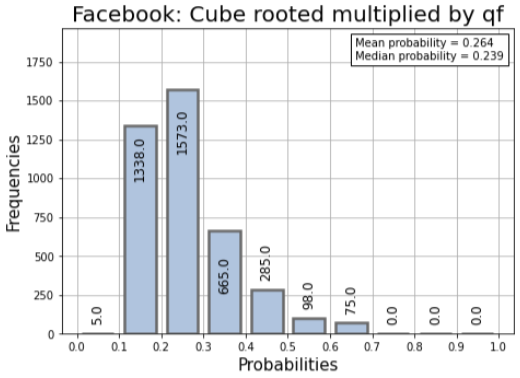
Degree Reciprocals (WC1) BitcoinOTC & Facebook:

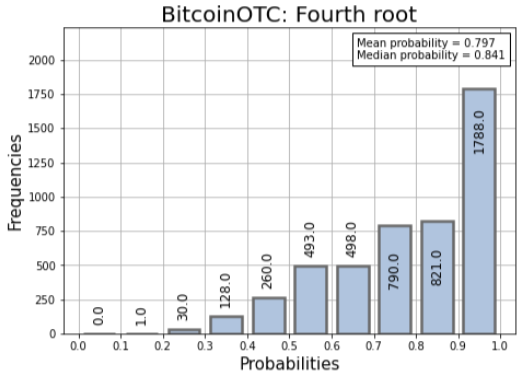


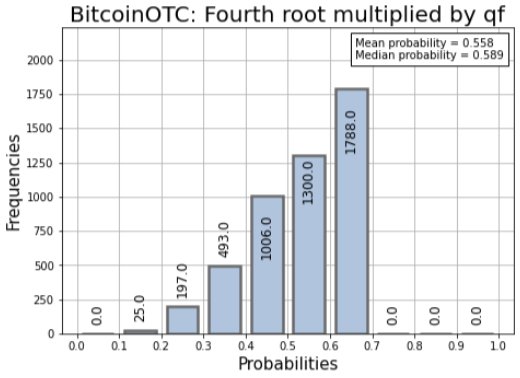

BitcoinOTC: Cube rooted
Mean probability = 0.749
Median probability = 0.794
Frequencies
Probabilities
0.0
30.0
137.0
278.0
347.0
618.0
790.0
821.0
0.0
1788.0

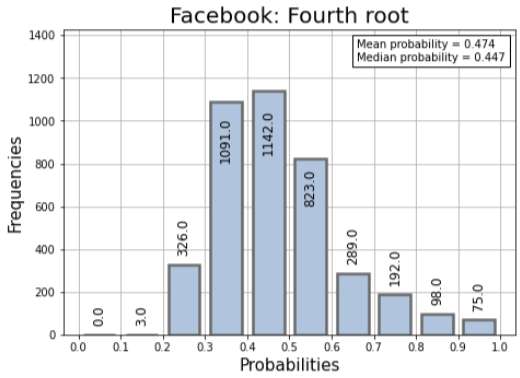

BitcoinOTC: Cube rooted multiplied by qf
Mean probability = 0.524
Median probability = 0.556
Frequencies
Probabilities
4.0
144.0
384.0
643.0
1025.0
821.0
1788.0
0.0
0.0
0.0

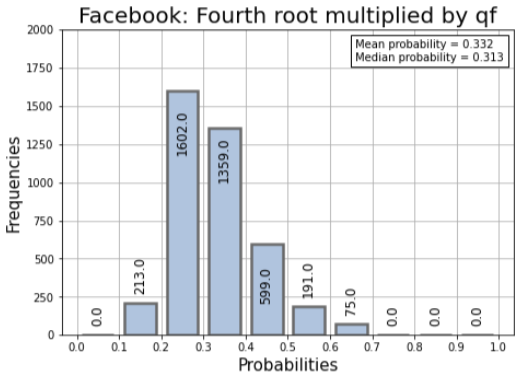

Facebook: Cube rooted
Mean probability = 0.377
Median probability = 0.342
Frequencies
Probabilities
1.0
314.0
1178.0
1151.0
630.0
400.0
192.0
98.0
0.0
75.0

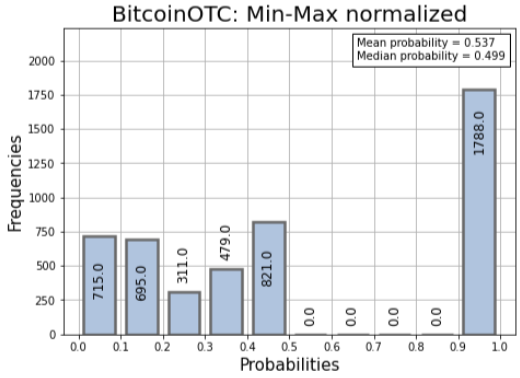

Facebook: Cube rooted multiplied by qf
Mean probability = 0.264
Median probability = 0.239
Frequencies
Probabilities
5.0
1338.0
1573.0
665.0
285.0
98.0
75.0
0.0
0.0
0.0

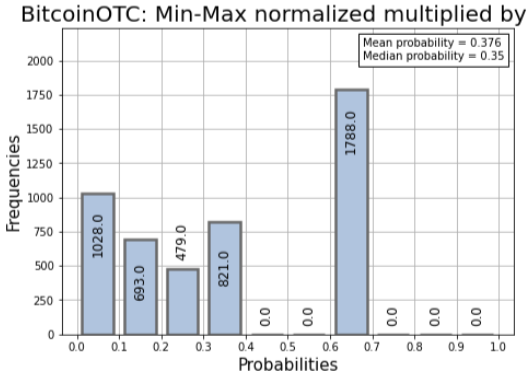

BitcoinOTC: Fourth root
Mean probability = 0.797
Median probability = 0.841
Frequencies
Probabilities
0.0
1.0
30.0
128.0
260.0
493.0
498.0
790.0
821.0
1788.0

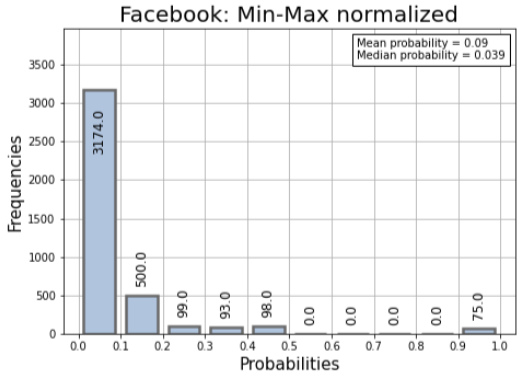

BitcoinOTC: Fourth root multiplied by qf
Mean probability = 0.558
Median probability = 0.589
Frequencies
Probabilities
0.0
25.0
197.0
493.0
1006.0
1300.0
1788.0
0.0
0.0
0.0

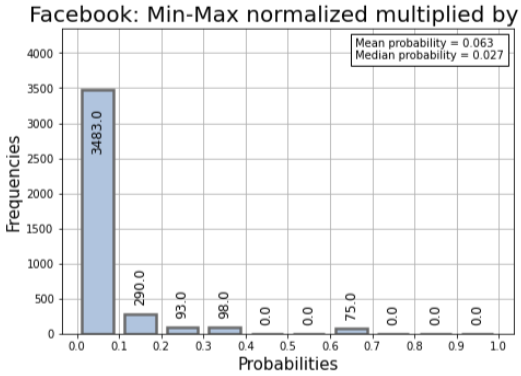

Facebook: Fourth root
Mean probability = 0.474
Median probability = 0.447
Frequencies
Probabilities
0.0
3.0
326.0
1091.0
1142.0
823.0
289.0
192.0
98.0
75.0

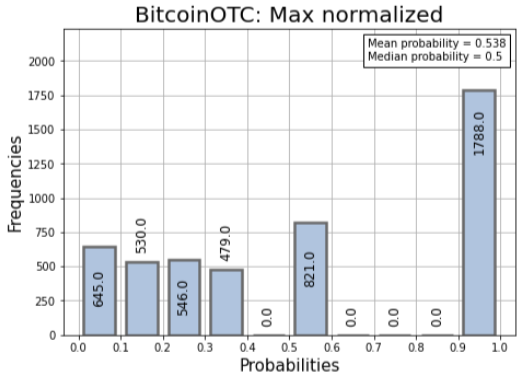

Facebook: Fourth root multiplied by qf
Mean probability = 0.332
Median probability = 0.313
Frequencies
Probabilities
0.0
213.0
1602.0
1359.0
599.0
191.0
75.0
0.0
0.0
0.0

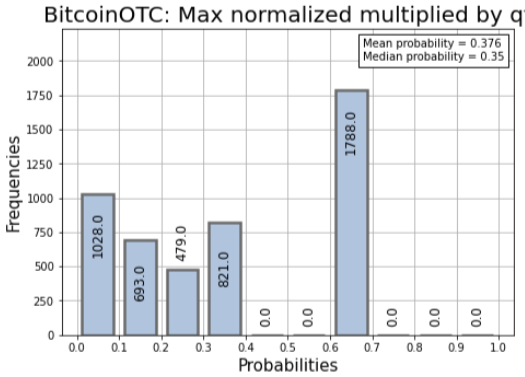

BitcoinOTC: Min-Max normalized
Mean probability = 0.537
Median probability = 0.499
Frequencies
Probabilities
715.0
695.0
311.0
479.0
821.0
0.0
0.0
0.0
0.0
1788.0

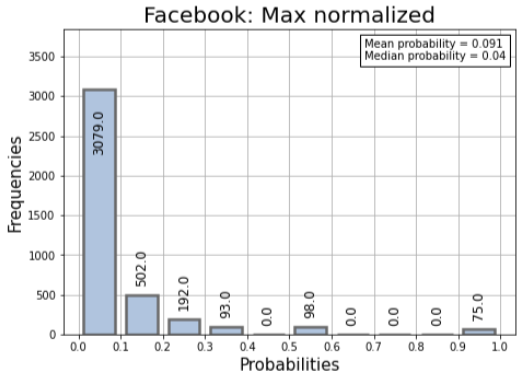

BitcoinOTC: Min-Max normalized multiplied by qf
Mean probability = 0.376
Median probability = 0.35
Frequencies
Probabilities
1028.0
693.0
479.0
821.0
0.0
0.0
1788.0
0.0
0.0
0.0

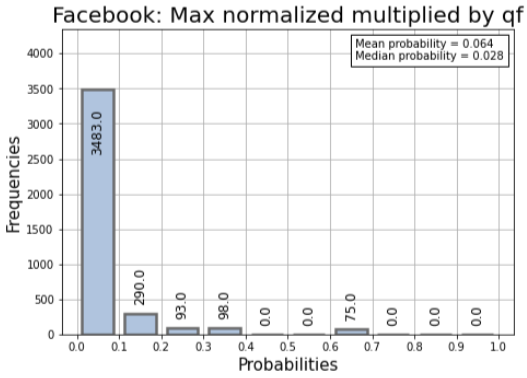

Facebook: Min-Max normalized
Mean probability = 0.09
Median probability = 0.039
Frequencies
Probabilities
3174.0
500.0
99.0
93.0
98.0
0.0
0.0
0.0
0.0
75.0

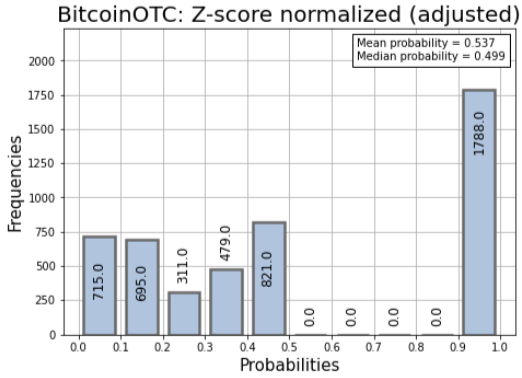

Facebook: Min-Max normalized multiplied by qf
Mean probability = 0.063
Median probability = 0.027
Frequencies
Probabilities
3483.0
290.0
93.0
98.0
0.0
0.0
75.0
0.0
0.0
0.0

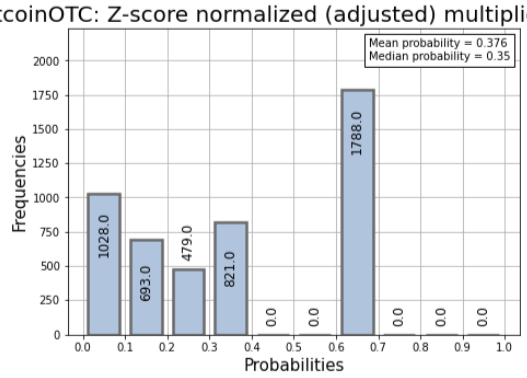

BitcoinOTC: Max normalized
Mean probability = 0.538
Median probability = 0.5
Frequencies
Probabilities
645.0
530.0
546.0
479.0
0.0
821.0
0.0
0.0
0.0
1788.0

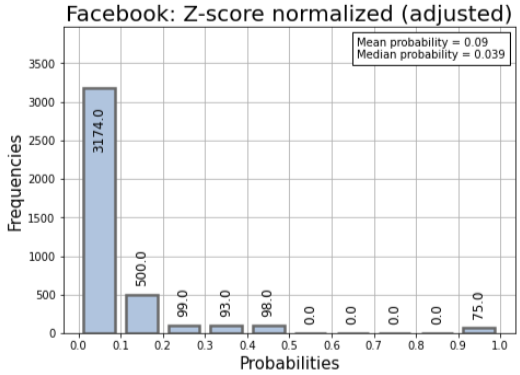

BitcoinOTC: Max normalized multiplied by qf
Mean probability = 0.376
Median probability = 0.35
Frequencies
Probabilities
1028.0
693.0
479.0
821.0
0.0
0.0
1788.0
0.0
0.0
0.0

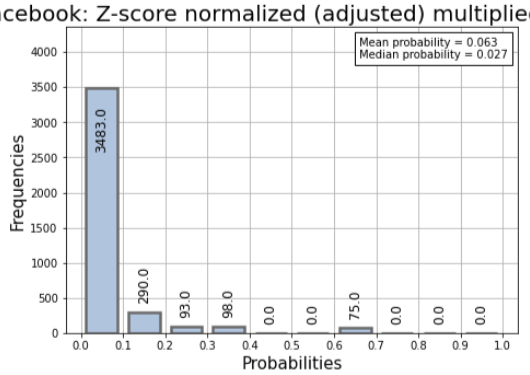

Facebook: Max normalized
Mean probability = 0.091
Median probability = 0.04
Frequencies
Probabilities
3079.0
502.0
192.0
93.0
0.0
98.0
0.0
0.0
0.0
75.0

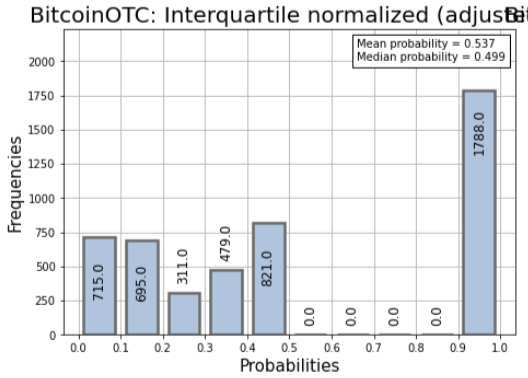

Facebook: Max normalized multiplied by qf
Mean probability = 0.064
Median probability = 0.028
Frequencies
Probabilities
3483.0
290.0
93.0
98.0
0.0
0.0
75.0
0.0
0.0
0.0

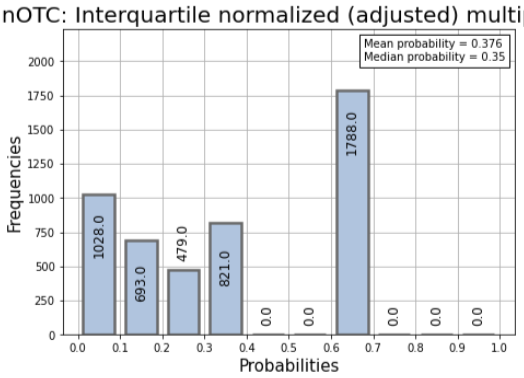

BitcoinOTC: Z-score normalized (adjusted)
Mean probability = 0.537
Median probability = 0.499
Frequencies
Probabilities
715.0
695.0
311.0
479.0
821.0
0.0
0.0
0.0
0.0
1788.0

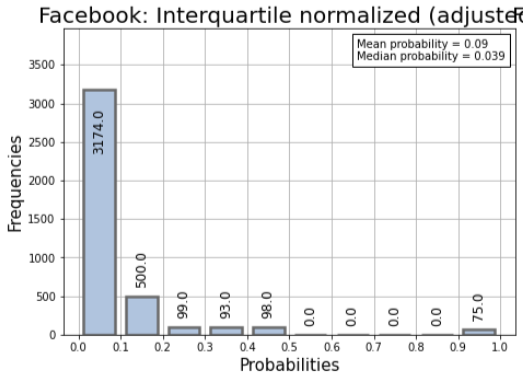

BitcoinOTC: Z-score normalized (adjusted) multiplied by qf
Mean probability = 0.376
Median probability = 0.35
Frequencies
Probabilities
1028.0
693.0
479.0
821.0
0.0
0.0
1788.0
0.0
0.0
0.0

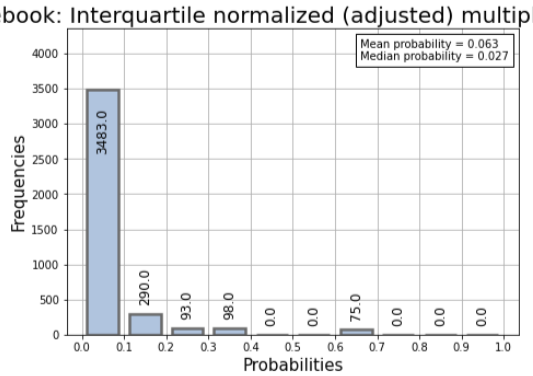

Facebook: Z-score normalized (adjusted)
Mean probability = 0.09
Median probability = 0.039
Frequencies
Probabilities
3174.0
500.0
99.0
93.0
98.0
0.0
0.0
0.0
0.0
75.0

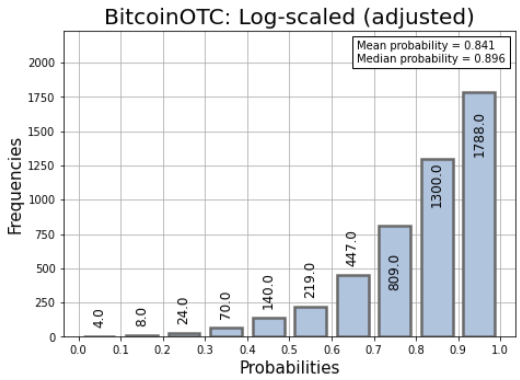

Facebook: Z-score normalized (adjusted) multiplied by qf
Mean probability = 0.063
Median probability = 0.027
Frequencies
Probabilities
3483.0
290.0
93.0
98.0
0.0
0.0
75.0
0.0
0.0
0.0

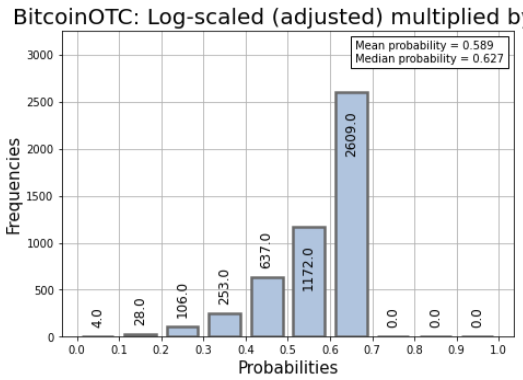

BitcoinOTC: Interquartile normalized (adjusted)
Mean probability = 0.537
Median probability = 0.499
Frequencies
Probabilities
715.0
695.0
311.0
479.0
821.0
0.0
0.0
0.0
0.0
1788.0

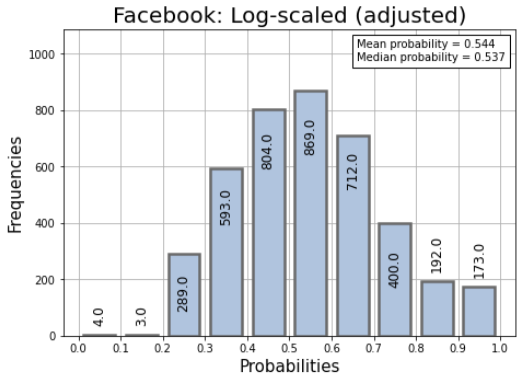

BitcoinOTC: Interquartile normalized (adjusted) multiplied by qf
Mean probability = 0.376
Median probability = 0.35
Frequencies
Probabilities
1028.0
693.0
479.0
821.0
0.0
0.0
1788.0
0.0
0.0
0.0

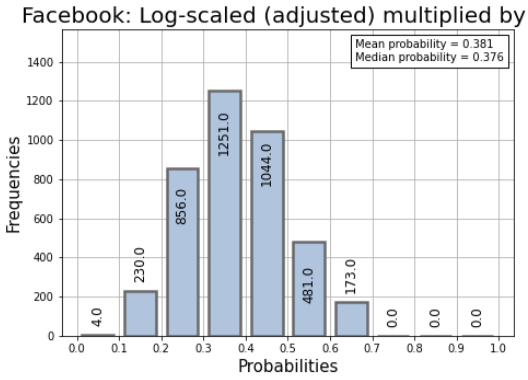

Facebook: Interquartile normalized (adjusted)
Mean probability = 0.09
Median probability = 0.039
Frequencies
Probabilities
3174.0
500.0
99.0
93.0
98.0
0.0
0.0
0.0
0.0
75.0


Facebook: Interquartile normalized (adjusted) multiplied by qf
Mean probability = 0.063
Median probability = 0.027
Frequencies
Probabilities
3483.0
290.0
93.0
98.0
0.0
0.0
75.0
0.0
0.0
0.0

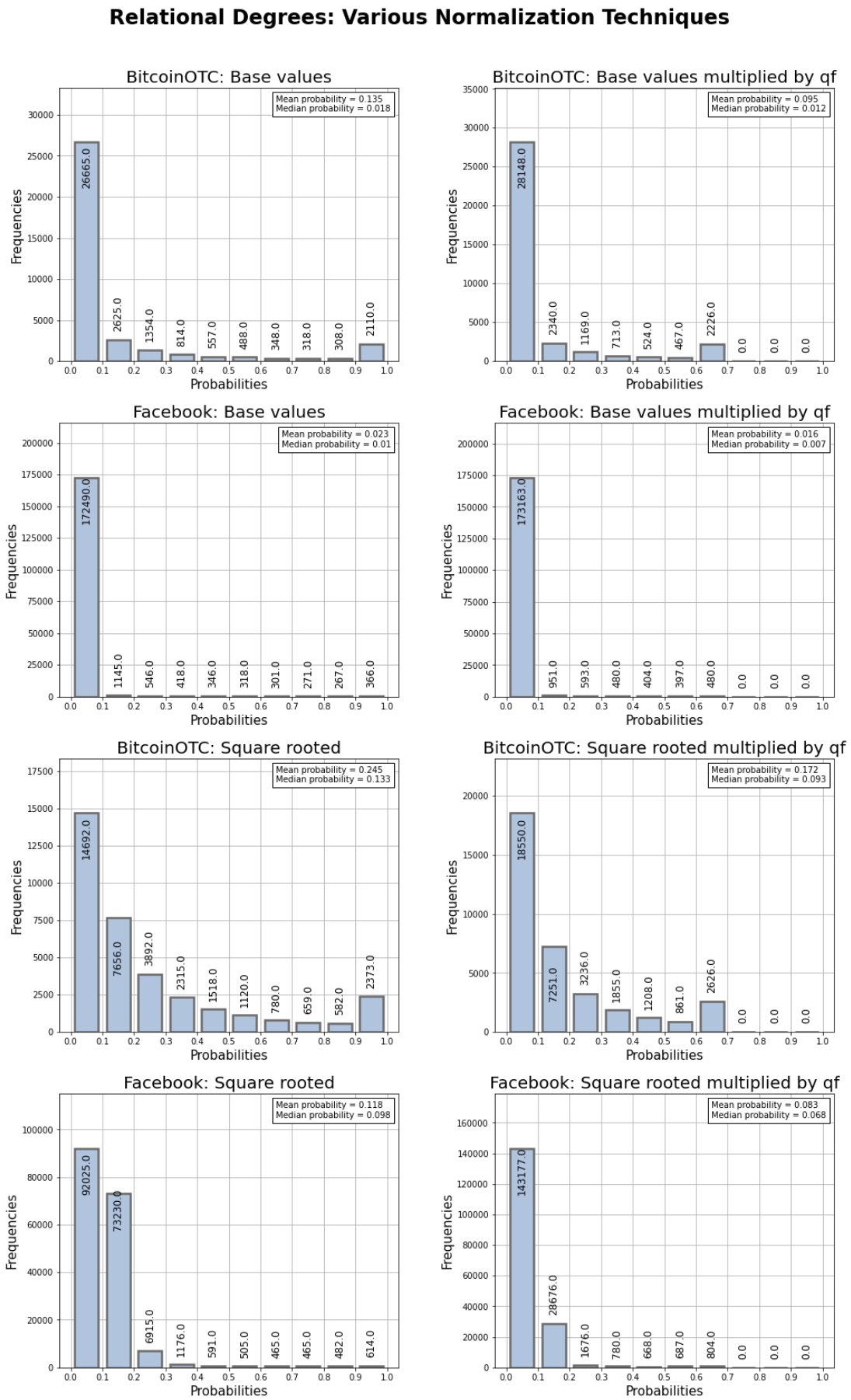

BitcoinOTC: Log-scaled (adjusted)
Mean probability = 0.841
Median probability = 0.896
Frequencies
Probabilities
4.0
8.0
24.0
70.0
140.0
219.0
447.0
809.0
1300.0
1788.0

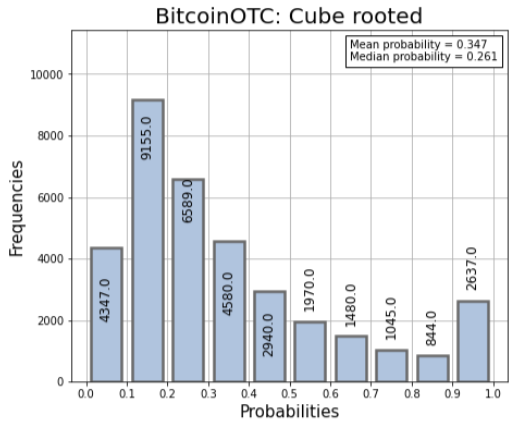

BitcoinOTC: Log-scaled (adjusted) multiplied by qf
Mean probability = 0.589
Median probability = 0.627
Frequencies
Probabilities
4.0
28.0
106.0
253.0
637.0
1172.0
2609.0
0.0
0.0
0.0

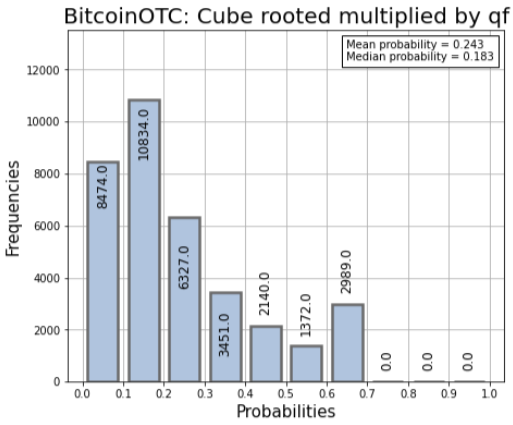

Facebook: Log-scaled (adjusted)
Mean probability = 0.544
Median probability = 0.537
Frequencies
Probabilities
4.0
3.0
289.0
593.0
804.0
869.0
712.0
400.0
192.0
173.0

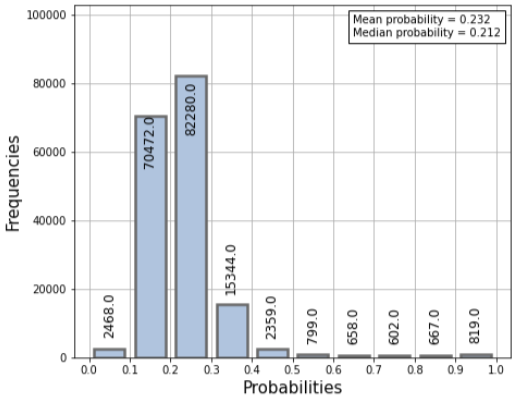

Facebook: Log-scaled (adjusted) multiplied by qf
Mean probability = 0.381
Median probability = 0.376
Frequencies
Probabilities
4.0
230.0
856.0
1251.0
1044.0
481.0
173.0
0.0
0.0
0.0

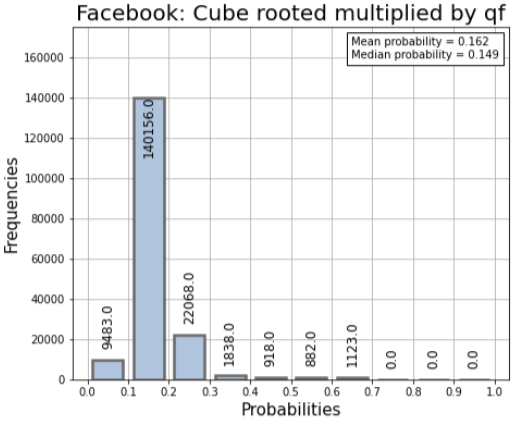
Relational Degrees (WC2) BitcoinOTC & Facebook:

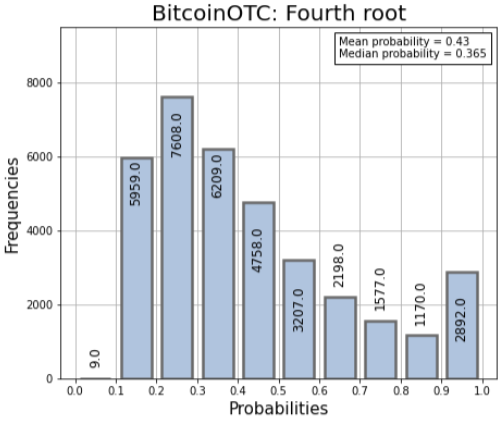


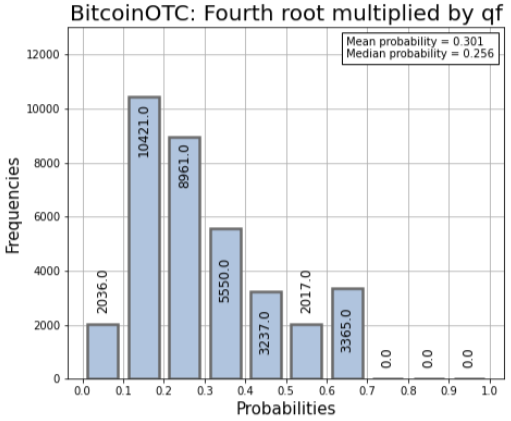

BitcoinOTC: Cube rooted
Mean probability = 0.347
Median probability = 0.261
Frequencies
Probabilities
4347.0
9155.0
6589.0
4580.0
2940.0
1970.0
1480.0
1045.0
844.0
2637.0

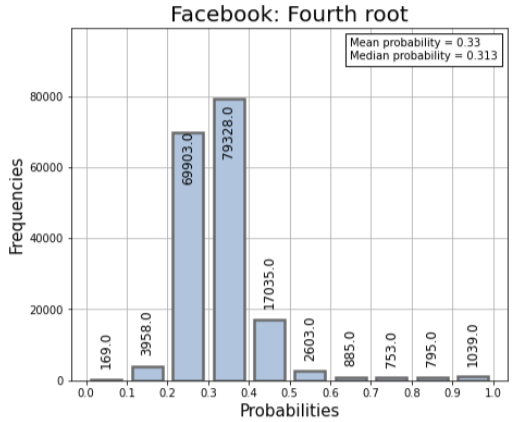

BitcoinOTC: Cube rooted multiplied by qf
Mean probability = 0.243
Median probability = 0.183
Frequencies
Probabilities
8474.0
10834.0
6327.0
3451.0
2140.0
1372.0
2989.0
0.0
0.0
0.0

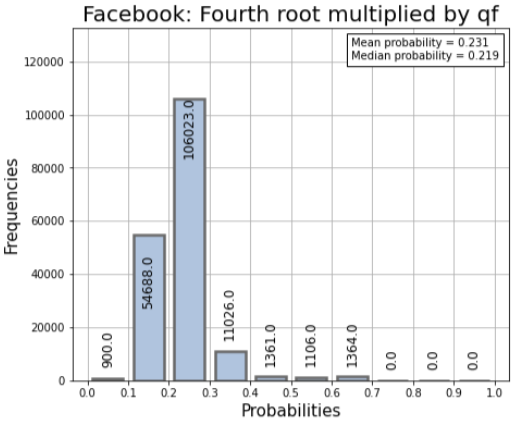

Facebook: Cube rooted
Mean probability = 0.232
Median probability = 0.212
Frequencies
Probabilities
2468.0
70472.0
82280.0
15344.0
2359.0
799.0
658.0
602.0
667.0
819.0

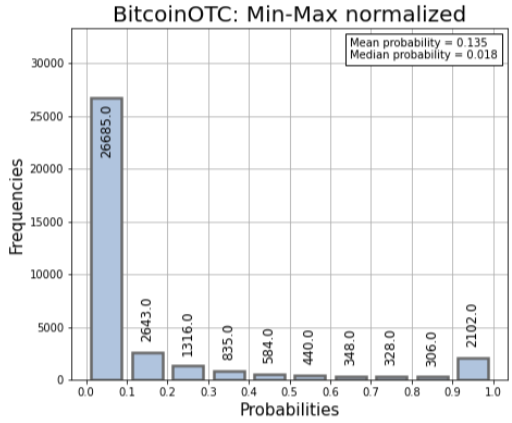

Facebook: Cube rooted multiplied by qf
Mean probability = 0.162
Median probability = 0.149
Frequencies
Probabilities
9483.0
140156.0
22068.0
1838.0
918.0
882.0
1123.0
0.0
0.0
0.0

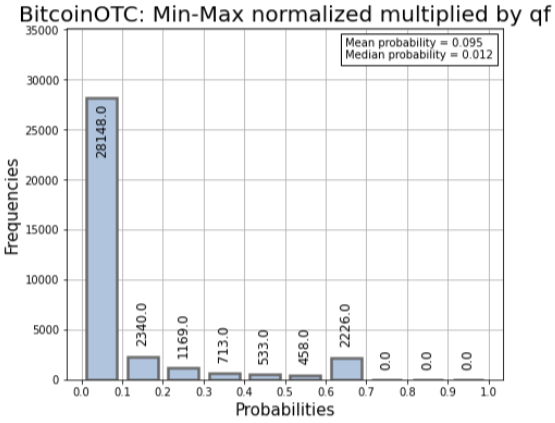

BitcoinOTC: Fourth root
Mean probability = 0.43
Median probability = 0.365
Frequencies
Probabilities
9.0
5959.0
7608.0
6209.0
4758.0
3207.0
2198.0
1577.0
1170.0
2892.0

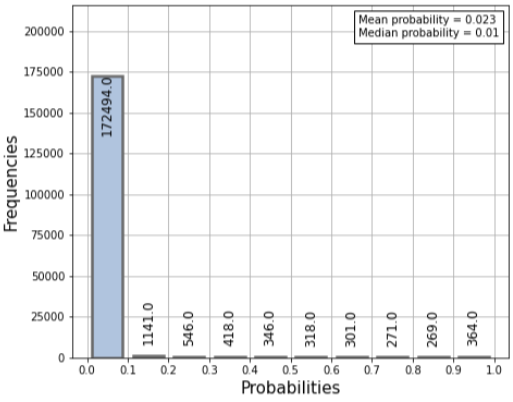

BitcoinOTC: Fourth root multiplied by qf
Mean probability = 0.301
Median probability = 0.256
Frequencies
Probabilities
2036.0
10421.0
8961.0
5550.0
3237.0
2017.0
3365.0
0.0
0.0
0.0

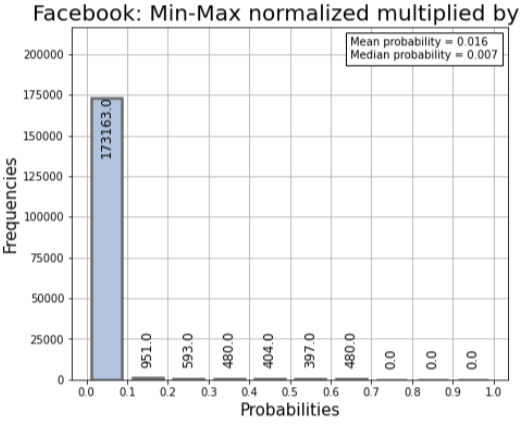

Facebook: Fourth root
Mean probability = 0.33
Median probability = 0.313
Frequencies
Probabilities
169.0
3958.0
69903.0
79328.0
17035.0
2603.0
885.0
753.0
795.0
1039.0

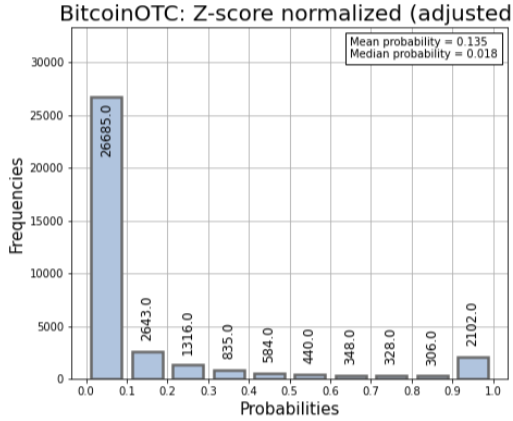

Facebook: Fourth root multiplied by qf
Mean probability = 0.231
Median probability = 0.219
Frequencies
Probabilities
900.0
54688.0
106023.0
11026.0
1361.0
1106.0
1364.0
0.0
0.0
0.0

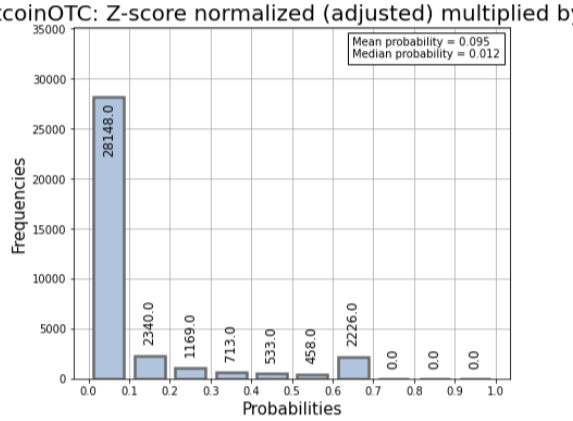

BitcoinOTC: Min-Max normalized
Mean probability = 0.135
Median probability = 0.018
Frequencies
Probabilities
26685.0
2643.0
1316.0
835.0
584.0
440.0
348.0
328.0
306.0
2102.0

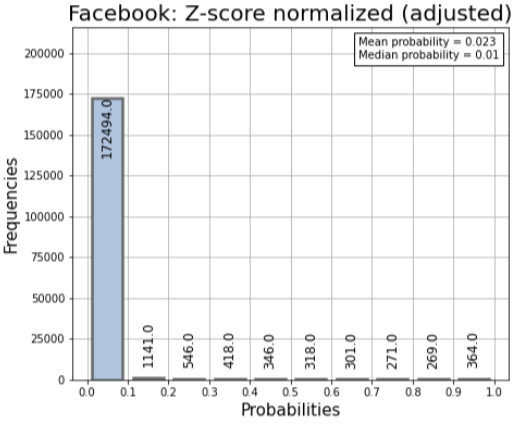

BitcoinOTC: Min-Max normalized multiplied by qf
Mean probability = 0.095
Median probability = 0.012
Frequencies
Probabilities
28148.0
2340.0
1169.0
713.0
533.0
458.0
2226.0
0.0
0.0
0.0

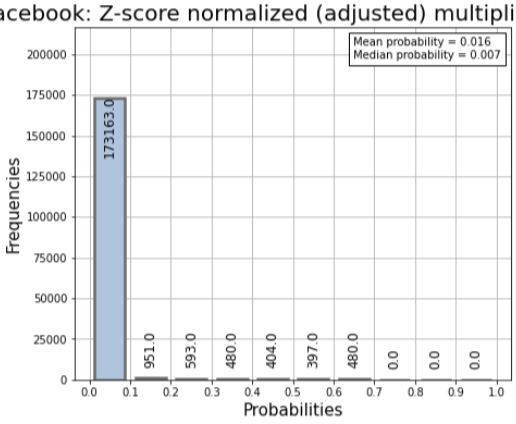

Facebook: Min-Max normalized
Mean probability = 0.023
Median probability = 0.01
Frequencies
Probabilities
172494.0
1141.0
546.0
418.0
346.0
318.0
301.0
271.0
269.0
364.0

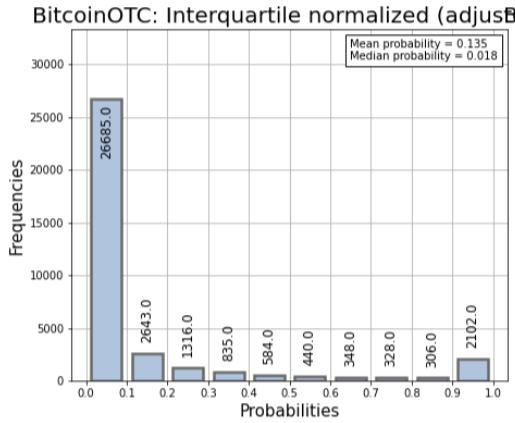

Facebook: Min-Max normalized multiplied by qf
Mean probability = 0.016
Median probability = 0.007
Frequencies
Probabilities
173163.0
951.0
593.0
480.0
404.0
397.0
480.0
0.0
0.0
0.0

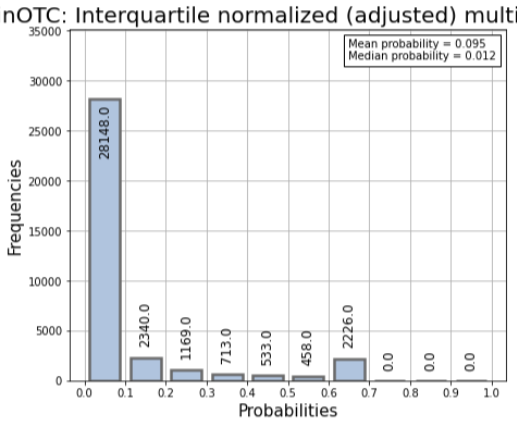

BitcoinOTC: Z-score normalized (adjusted)
Mean probability = 0.135
Median probability = 0.018
Frequencies
Probabilities
26685.0
2643.0
1316.0
835.0
584.0
440.0
348.0
328.0
306.0
2102.0

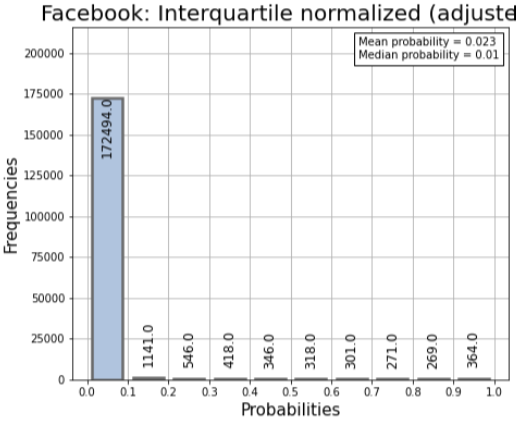

BitcoinOTC: Z-score normalized (adjusted) multiplied by qf
Mean probability = 0.095
Median probability = 0.012
Frequencies
Probabilities
28148.0
2340.0
1169.0
713.0
533.0
458.0
2226.0
0.0
0.0
0.0

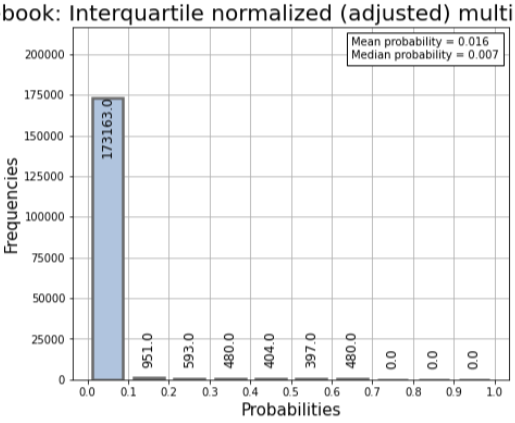

Facebook: Z-score normalized (adjusted)
Mean probability = 0.023
Median probability = 0.01
Frequencies
Probabilities
172494.0
1141.0
546.0
418.0
346.0
318.0
301.0
271.0
269.0
364.0

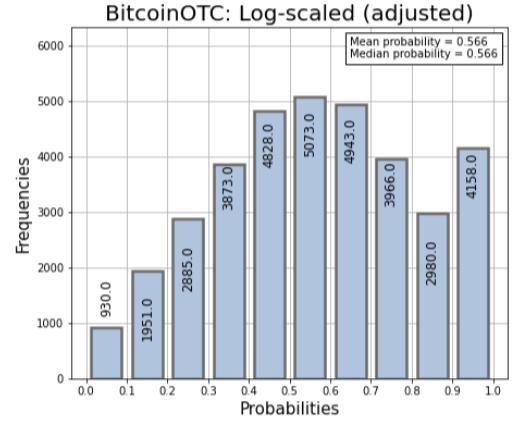

Facebook: Z-score normalized (adjusted) multiplied by qf
Mean probability = 0.016
Median probability = 0.007
Frequencies
Probabilities
173163.0
951.0
593.0
480.0
404.0
397.0
480.0
0.0
0.0
0.0

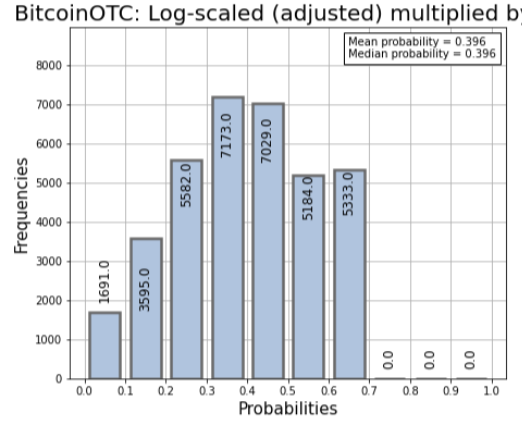

BitcoinOTC: Interquartile normalized (adjusted)
Mean probability = 0.135
Median probability = 0.018
Frequencies
Probabilities
26685.0
2643.0
1316.0
835.0
584.0
440.0
348.0
328.0
306.0
2102.0

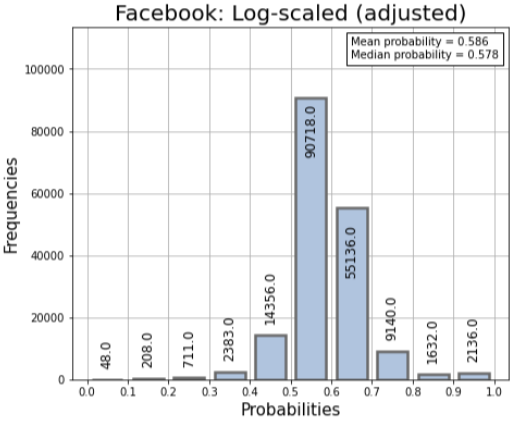

BitcoinOTC: Interquartile normalized (adjusted) multiplied by qf
Mean probability = 0.095
Median probability = 0.012
Frequencies
Probabilities
28148.0
2340.0
1169.0
713.0
533.0
458.0
2226.0
0.0
0.0
0.0

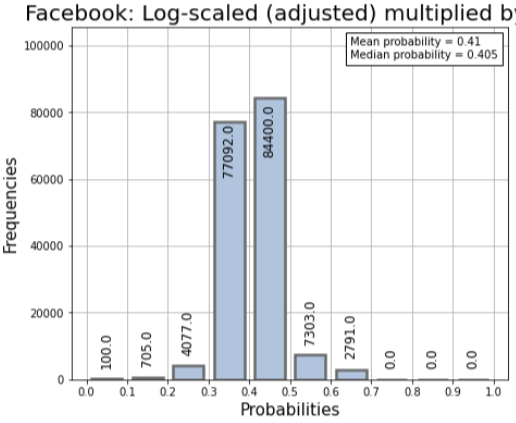

Facebook: Interquartile normalized (adjusted)
Mean probability = 0.023
Median probability = 0.01
Frequencies
Probabilities
172494.0
1141.0
546.0
418.0
346.0
318.0
301.0
271.0
269.0
364.0


Facebook: Interquartile normalized (adjusted) multiplied by qf
Mean probability = 0.016
Median probability = 0.007
Frequencies
Probabilities
173163.0
951.0
593.0
480.0
404.0
397.0
480.0
0.0
0.0
0.0


BitcoinOTC: Log-scaled (adjusted)
Mean probability = 0.566
Median probability = 0.566
Frequencies
Probabilities
930.0
1951.0
2885.0
3873.0
4828.0
5073.0
4943.0
3966.0
2980.0
4158.0

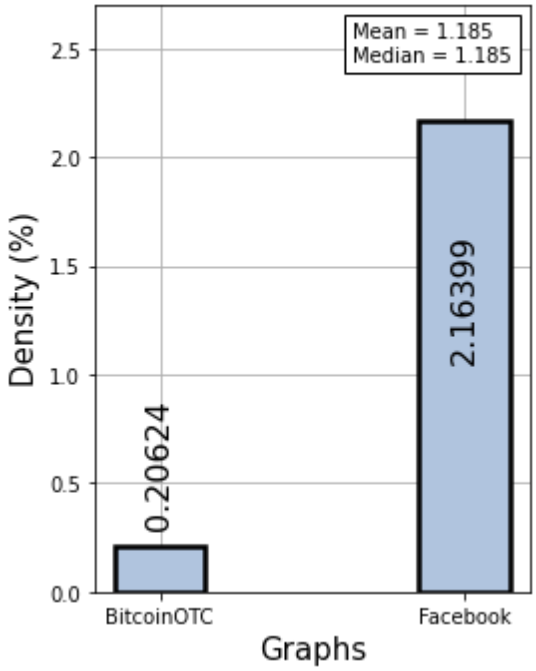

BitcoinOTC: Log-scaled (adjusted) multiplied by qf
Mean probability = 0.396
Median probability = 0.396
Frequencies
Probabilities
1691.0
3595.0
5582.0
7173.0
7029.0
5184.0
5333.0
0.0
0.0
0.0


Facebook: Log-scaled (adjusted)
Mean probability = 0.586
Median probability = 0.578
Frequencies
Probabilities
48.0
208.0
711.0
2383.0
14356.0
90718.0
55136.0
9140.0
1632.0
2136.0

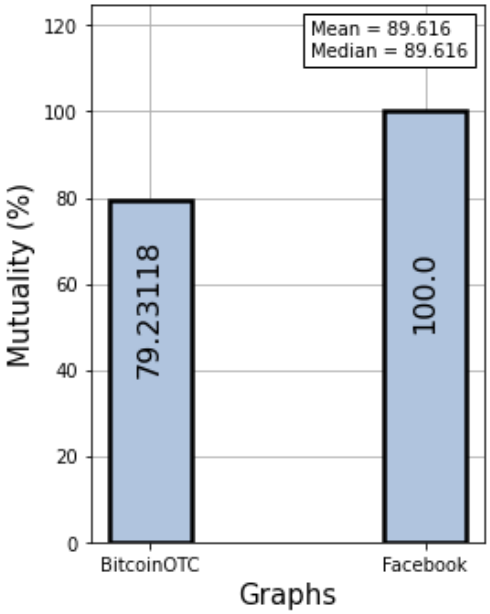

Facebook: Log-scaled (adjusted) multiplied by qf
Mean probability = 0.41
Median probability = 0.405
Frequencies
Probabilities
100.0
705.0
4077.0
77092.0
84400.0
7303.0
2791.0
0.0
0.0
0.0

D.4 Network-wide Metric Bar Charts

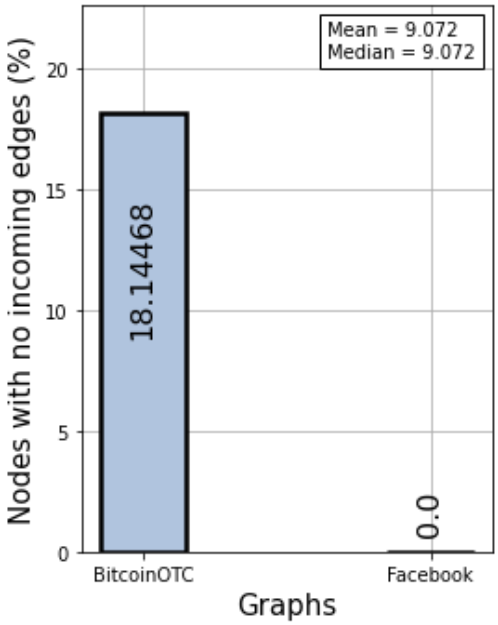
Density Comparison:



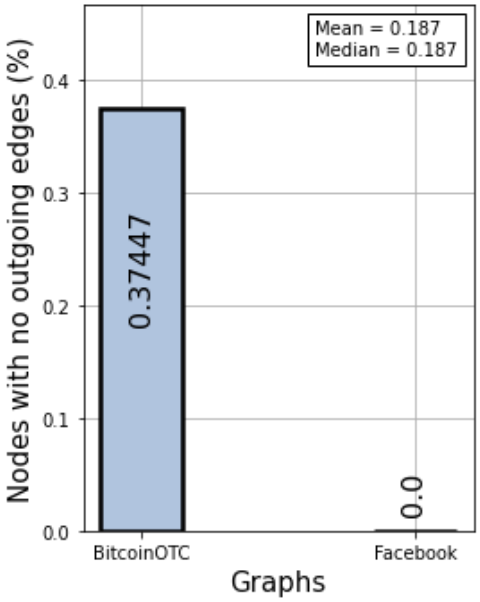
Mutuality Comparison:



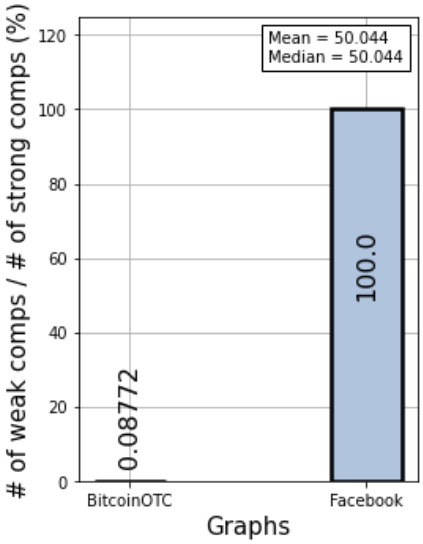
Nodes with no incoming edges Comparison:



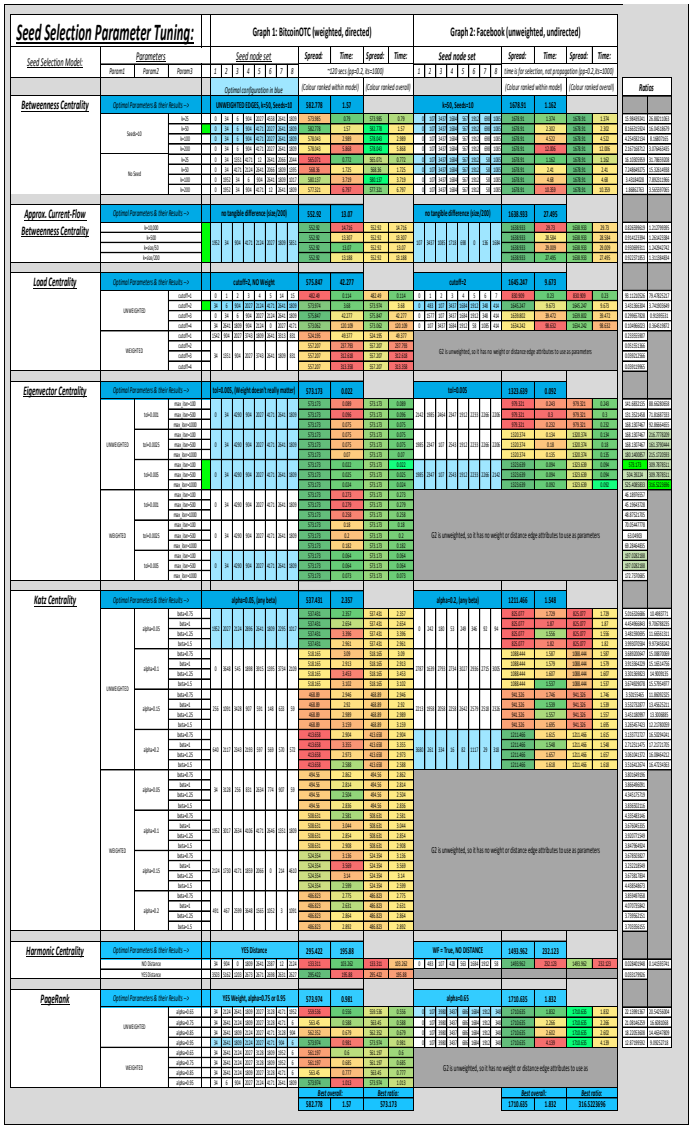
Nodes with no outgoing edges Comparison:



of weak comps / # of strong comps Comparison:



D.5 Seed Selection Parameter Fine-tuning Excel Spreadsheet



(file included in additional appendices)

D.6 Influence Build #3

Switch Factor Testing

Propagation Model	*Switch Factor (sf):*	*Influence Spread*	*Time elapsed (50 iterations):*
Independent Cascade	0	847	8.51 secs
	0.3	845.36	8.12 secs
	0.6	902.18	8.13 secs
	0.9	975.56	8.35 secs
Weighted Cascade 1	0	1561.02	9.58 secs
	0.3	1628.18	9.67 secs
	0.6	1695.58	10.06 secs
	0.9	1867.92	11.1 secs
Weighted Cascade 2	0	1937.3	10.09 secs
	0.3	1978.8	9.74 secs
	0.6	2097.32	9.97 secs
	0.9	2295.3	10.91 secs

Switch Factor / Quality Factor Testing

Quality Factor (qf):	*Switch Factor (sf):*	*Positive influence*	*Negative influence*
0.2	0	14.6	2245.52
	0.2	14.84	2249.68
	0.4	18.32	2243.64
	0.6	20.0	2239.04
	0.8	24.6	2229.84
	1	37.24	2218.96
0.8	0	1854.56	416.16
	0.2	1865.32	404.52
	0.4	1868.28	389.92
	0.6	1874.72	381.92
	0.8	1892.92	375.24

	1	1903.64	359.6

Time Factor Testing

Propagation Model	*Time Factor (tf):*	*Influence Spread*	*Time elapsed*
Independent Cascade	0	929.36	8.69 secs
	0.05	869.84	8.88 secs
	0.1	770.34	8.44 secs
	0.5	111.76	2.71 secs
Weighted Cascade 1	0	1679.78	9.9 secs
	0.05	1648.88	10.59 secs
	0.1	1579.5	10.44 secs
	0.5	297.6	5.08 secs
Weighted Cascade 2	0	2105.56	10.697 secs
	0.05	2028.0	10.25 secs
	0.1	1926.14	10.67 secs
	0.5	427.88	7.56 secs

Network Generation Testing

Pandas Row Iteration Model:	*Time elapsed: (10 iterations)*
index	38.35 secs
loc	45.99 secs
iloc	130.19 secs
itertuples	5.54 secs
iterrows	95.22 secs

Network Generation Method:	*Graph:*	*Size (# of edges):*	*Length (# of nodes):*
Method 2 (Semi-generalised function with manual dataset parameter entry)	G1: BitcoinOTC	35587	5875
	G2: Facebook	176468	4039
	G3: Custom	11	10

Method 3 (Fully generalised function with dataset parameter dictionary)	G1: BitcoinOTC	35587	5875
	G2: Facebook	176468	4039
	G3: Custom	11	10

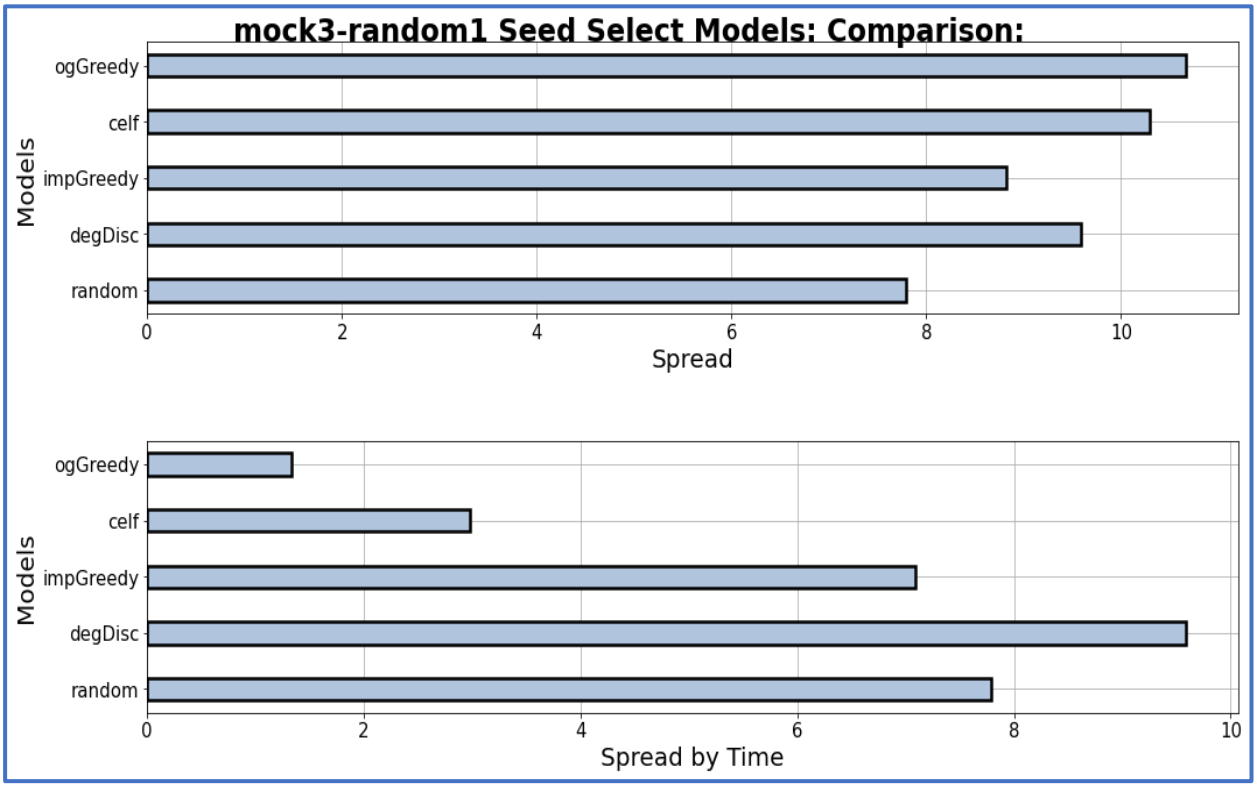
Probability Assignment Testing

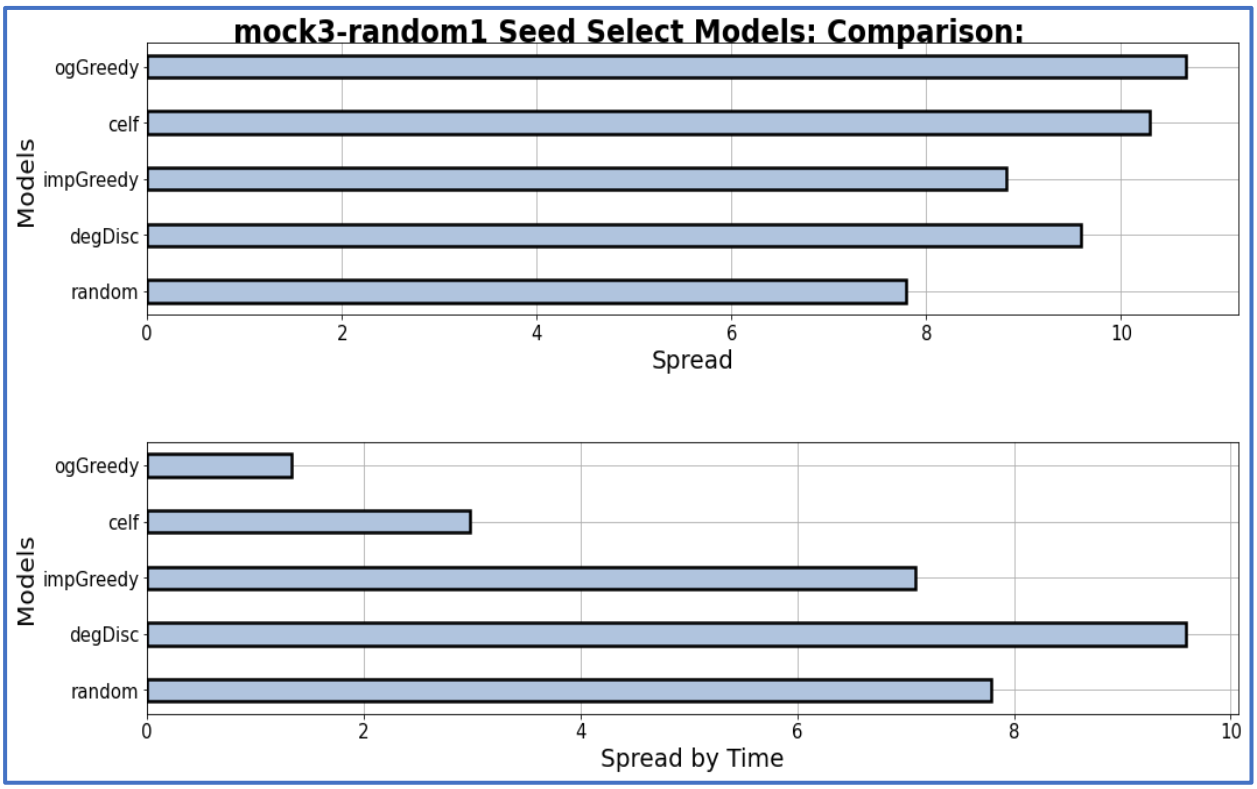
Assign Model:	*Description:*	*Spread:*	*Time elapsed:*
Assign = 0 (Manual log-scaling)	Calculate log-scale of probability every time it's needed, during the success function.	1.427	0.368 secs
Assign = 3 (Log-scaling and assigning method)	Calculate and normalise all Relational Degrees initally and assign them as edge attributes	888.773	77.491 secs

Assign Model:	*Description:*	*Spread:*	*Time elapsed:*
Assign = 0 (Manual log-scaling)	Calculate log-scale of probability every time it's needed, during the success function.	1188.533	328.085
Assign = 3 (Log-scaling and assigning method)	Calculate and normalise all Relational Degrees initially and assign them as edge attributes	1188.533	13.405

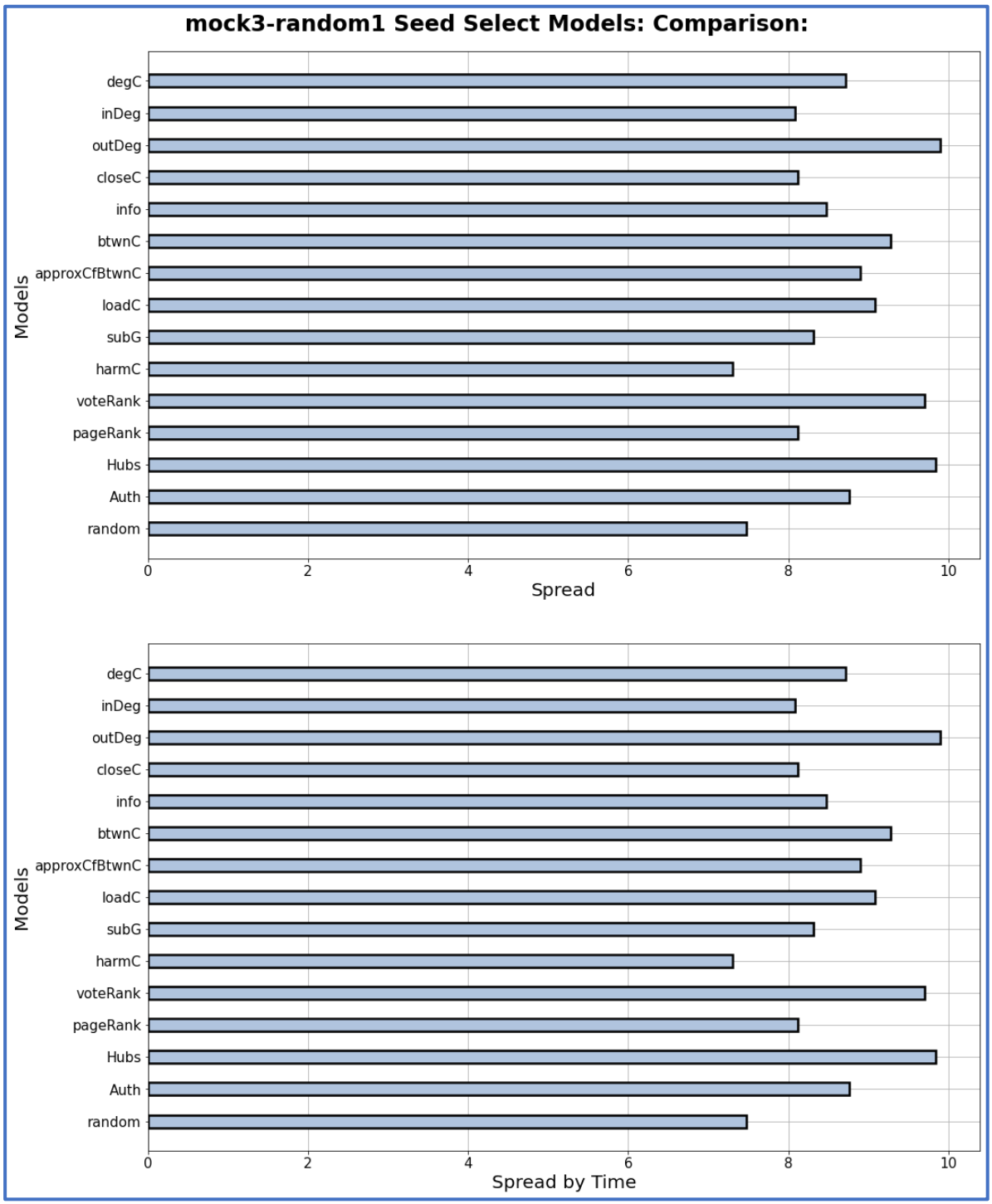
D.7 Seed Selection Model Comparison

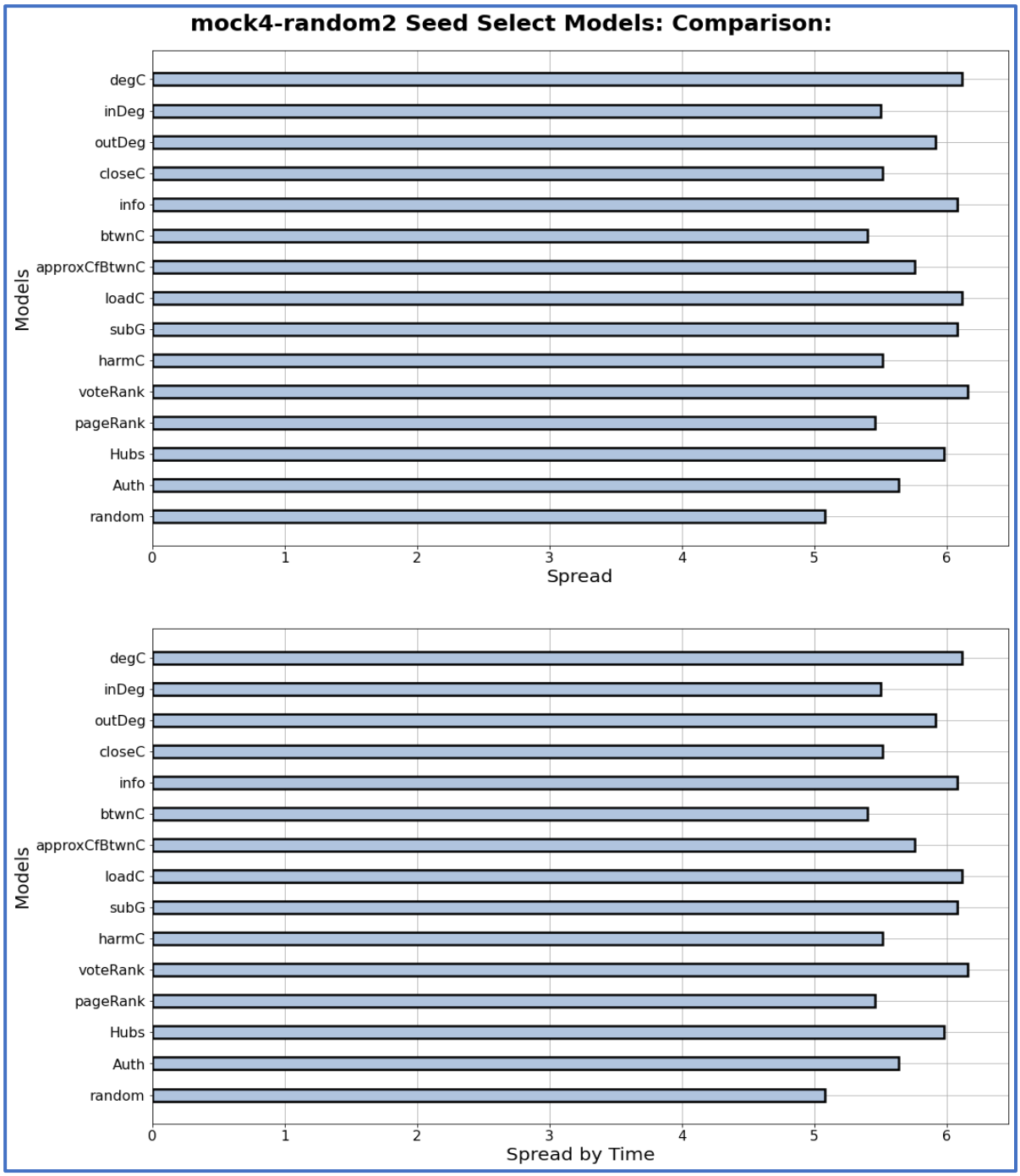
Past Models

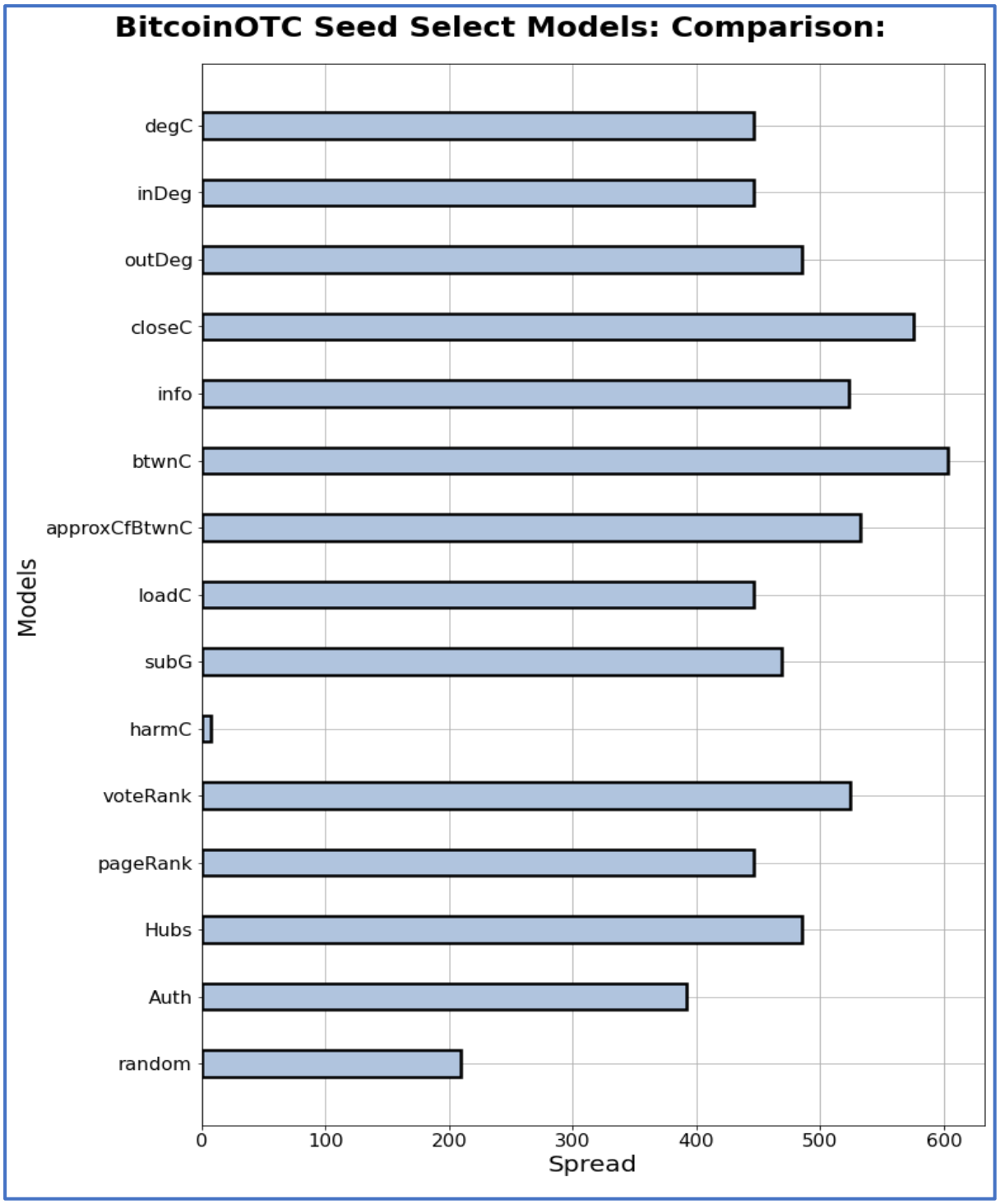


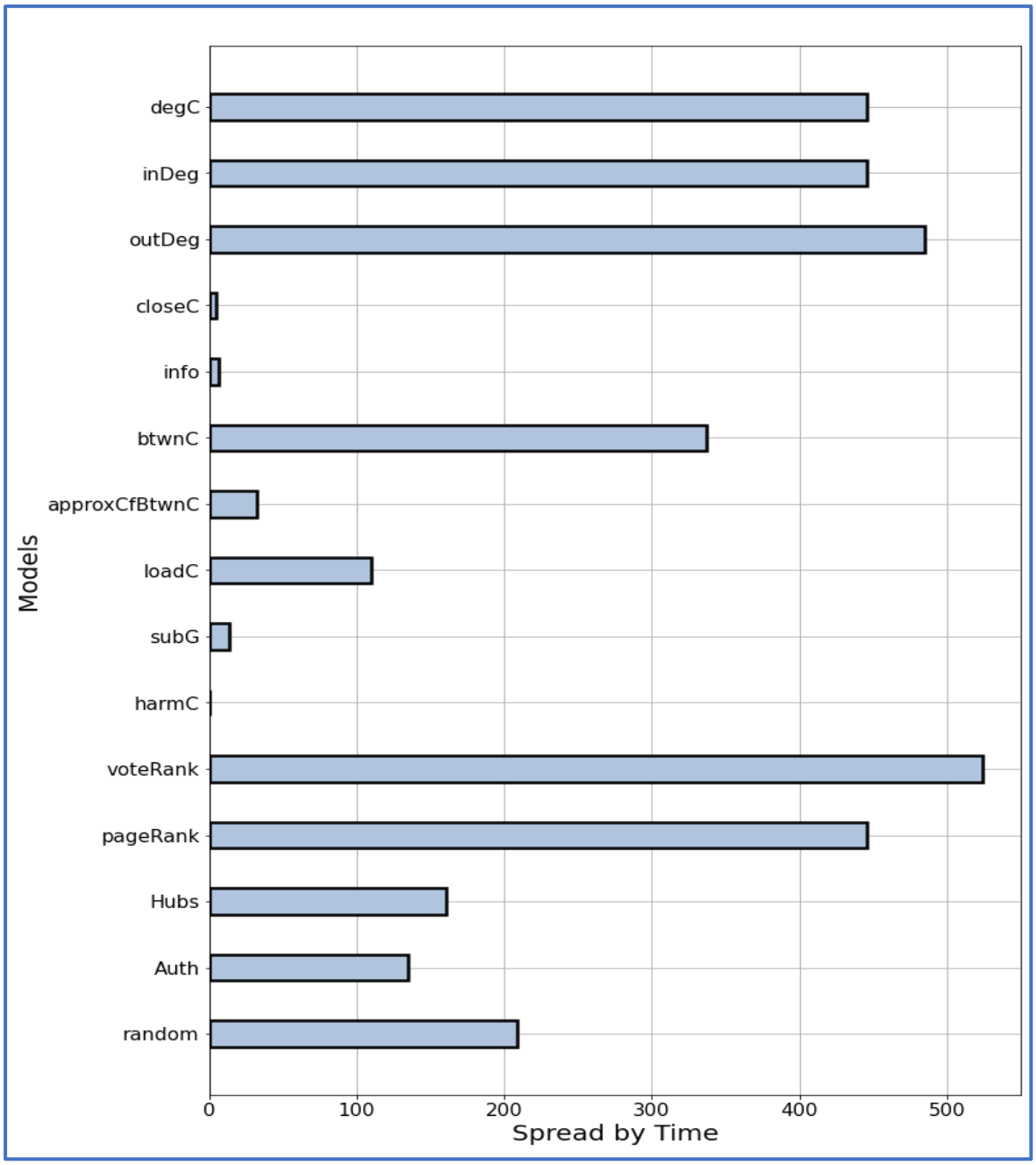


NetworkX Models

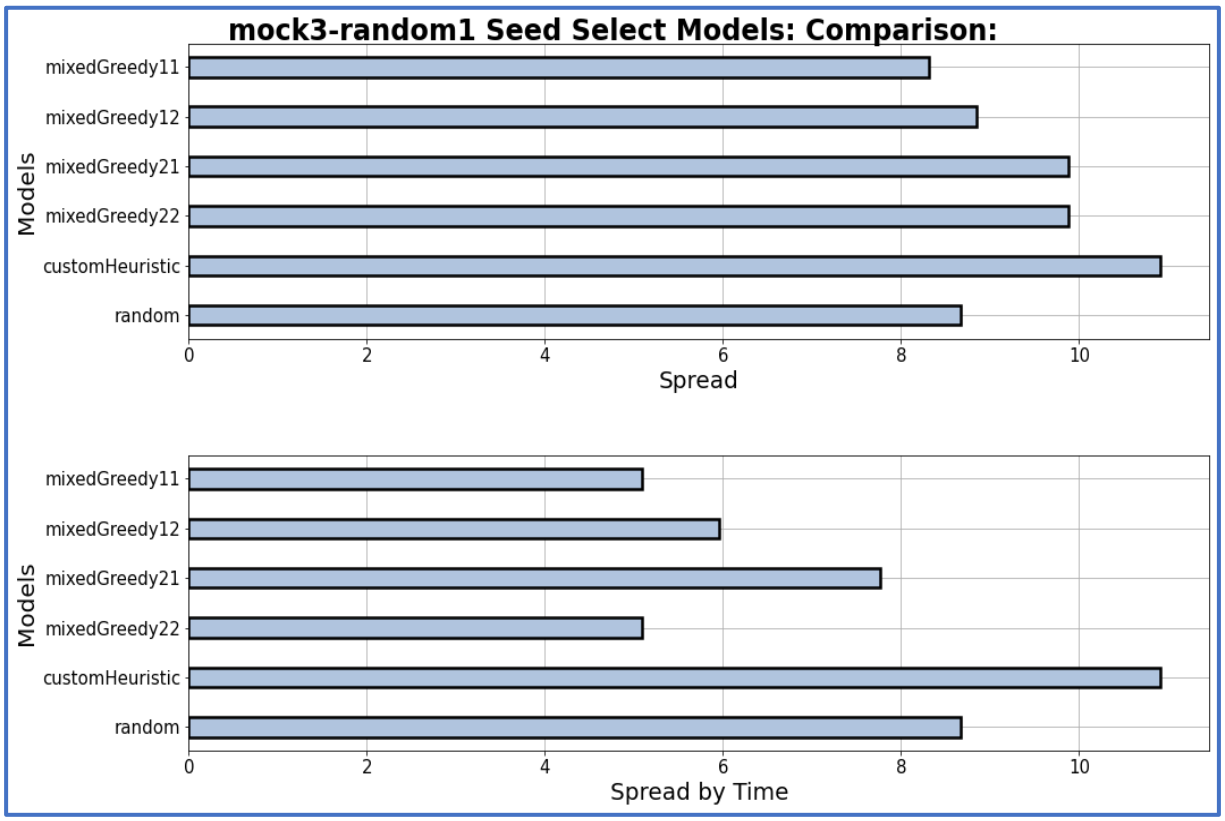


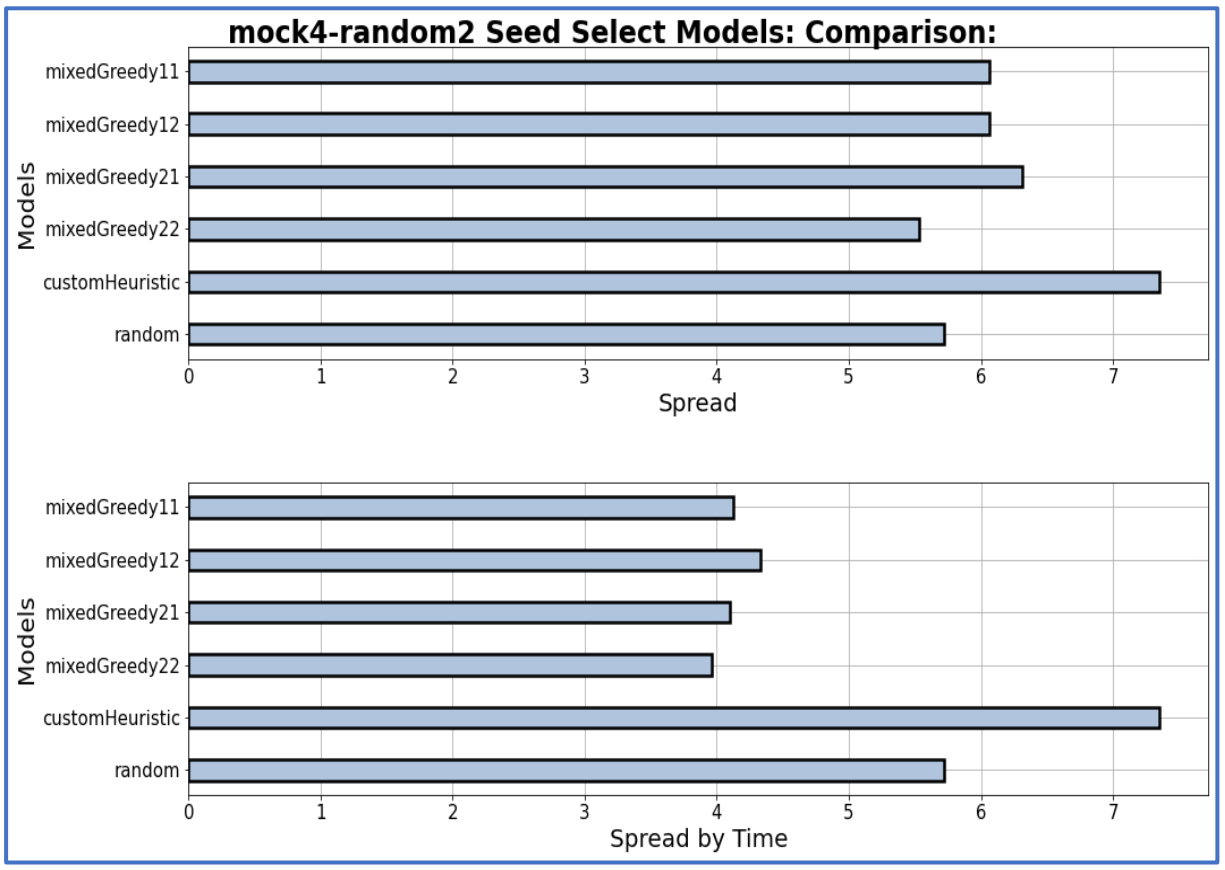

mock4-random2 Seed Select Models: Comparison:
Models
degC
inDeg
outDeg
closeC
info
btwnC
approxCfBtwnC
loadC
subG
harmC
voteRank
pageRank
Hubs
Auth
random
0
1
2
3
4
5
6
Spread
Models
degC
inDeg
outDeg
closeC
info
btwnC
approxCfBtwnC
loadC
subG
harmC
voteRank
pageRank
Hubs
Auth
random
0
1
2
3
4
5
6
Spread by Time

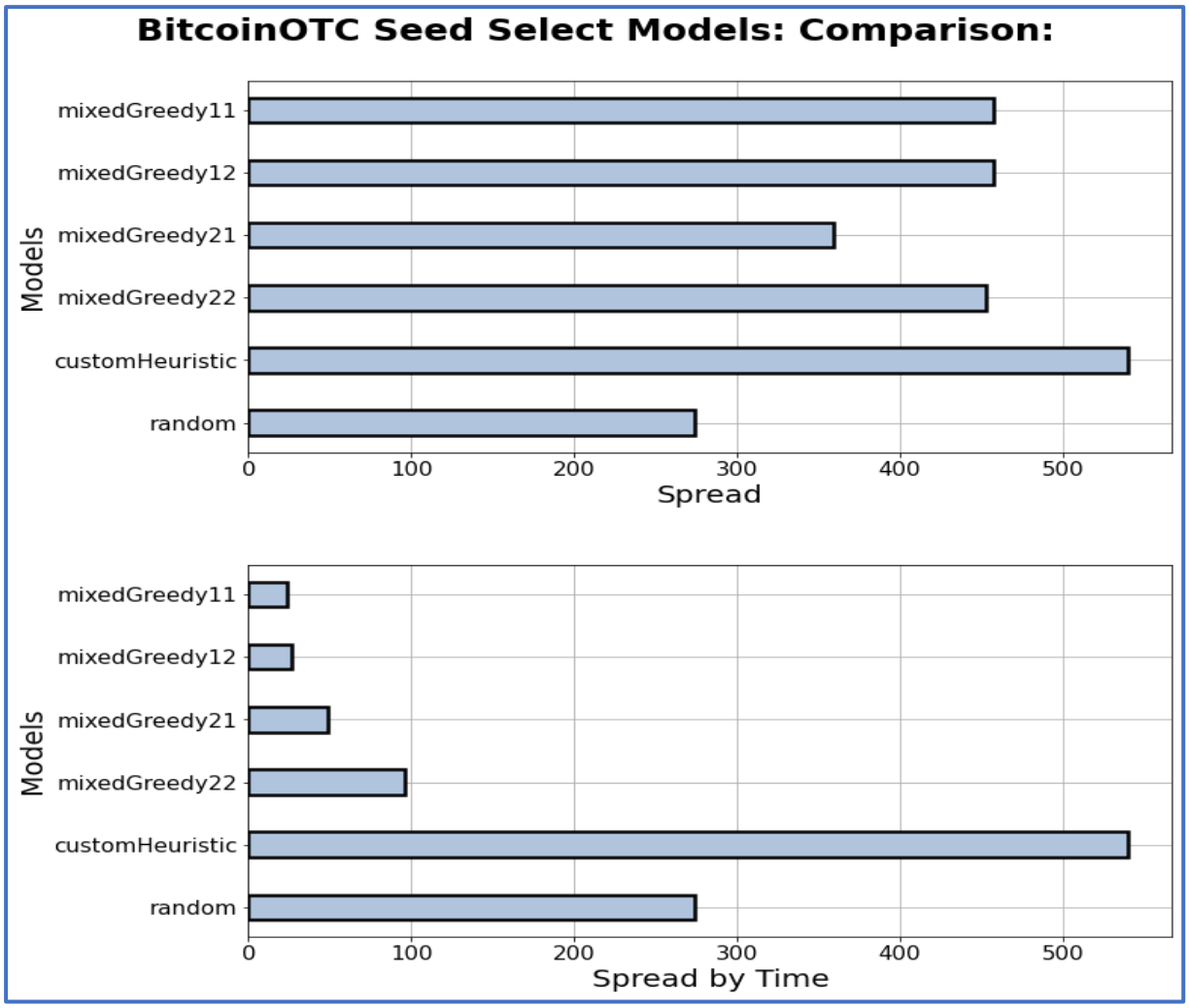

BitcoinOTC Seed Select Models: Comparison:
Models
degC
inDeg
outDeg
closeC
info
btwnC
approxCfBtwnC
loadC
subG
harmC
voteRank
pageRank
Hubs
Auth
random
0
100
200
300
400
500
600
Spread

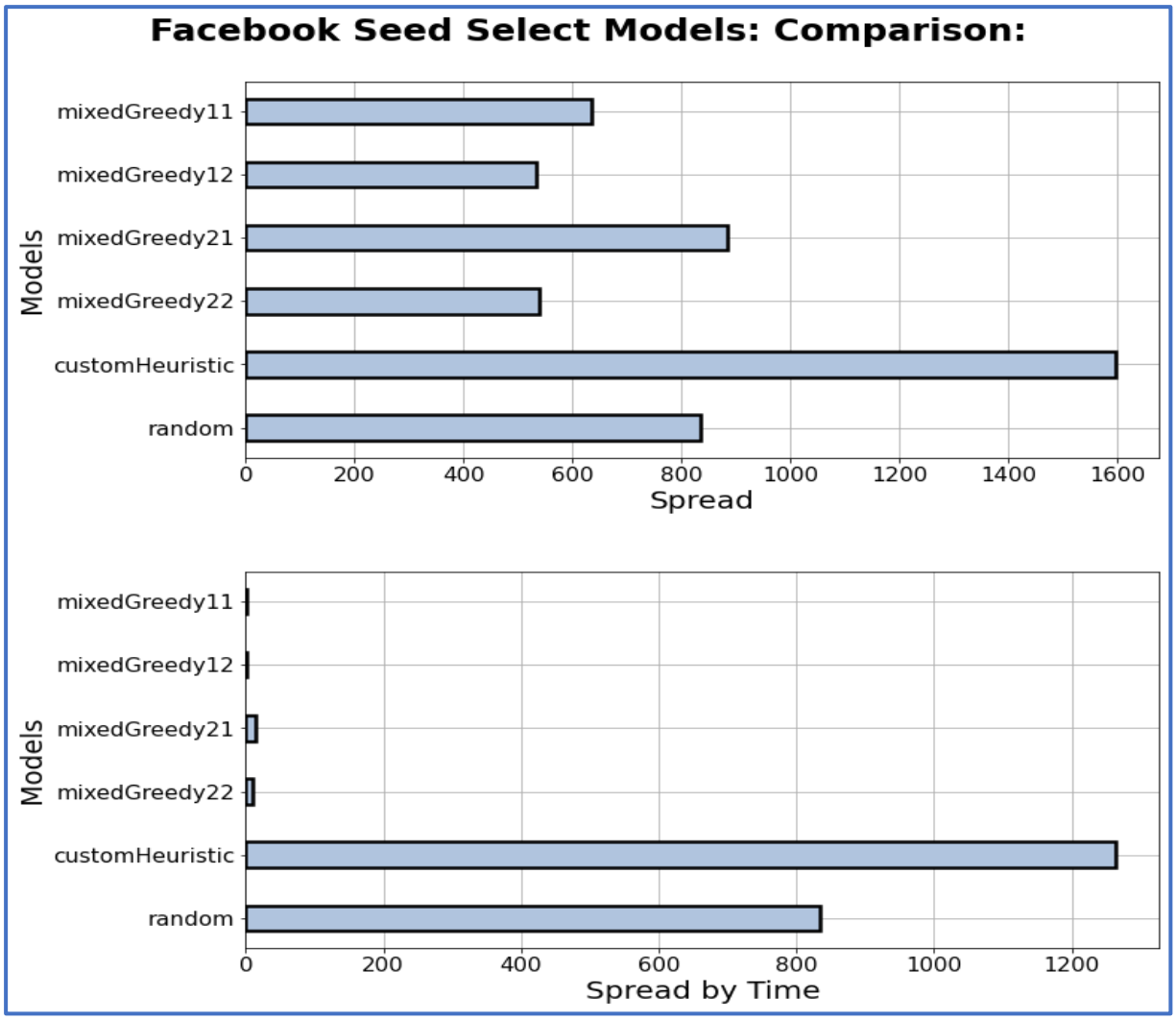

degC
inDeg
outDeg
closeC
info
btwnC
approxCfBtwnC
loadC
subG
harmC
voteRank
pageRank
Hubs
Auth
random
Models
0
100
200
300
400
500
Spread by Time

New/Original Models

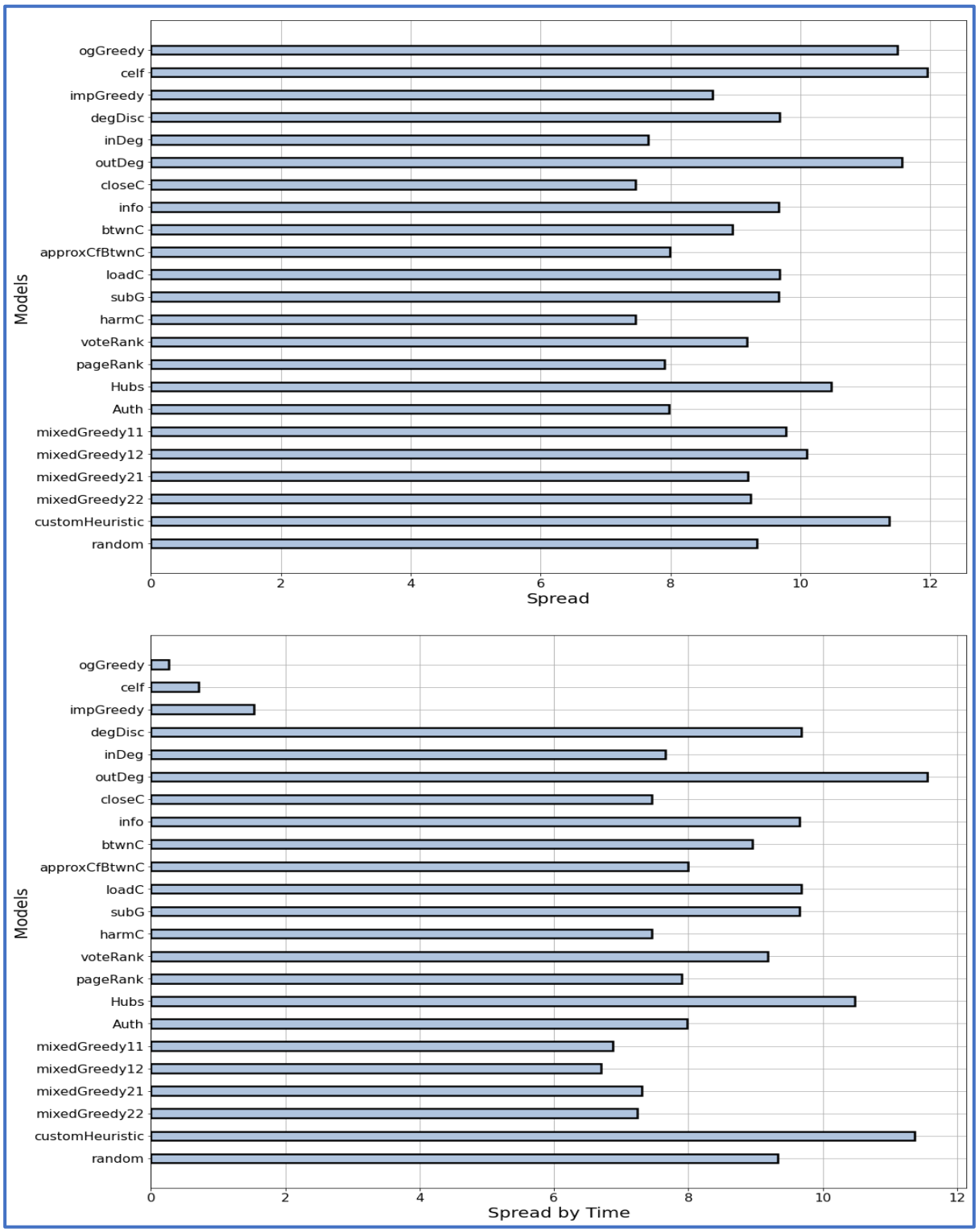




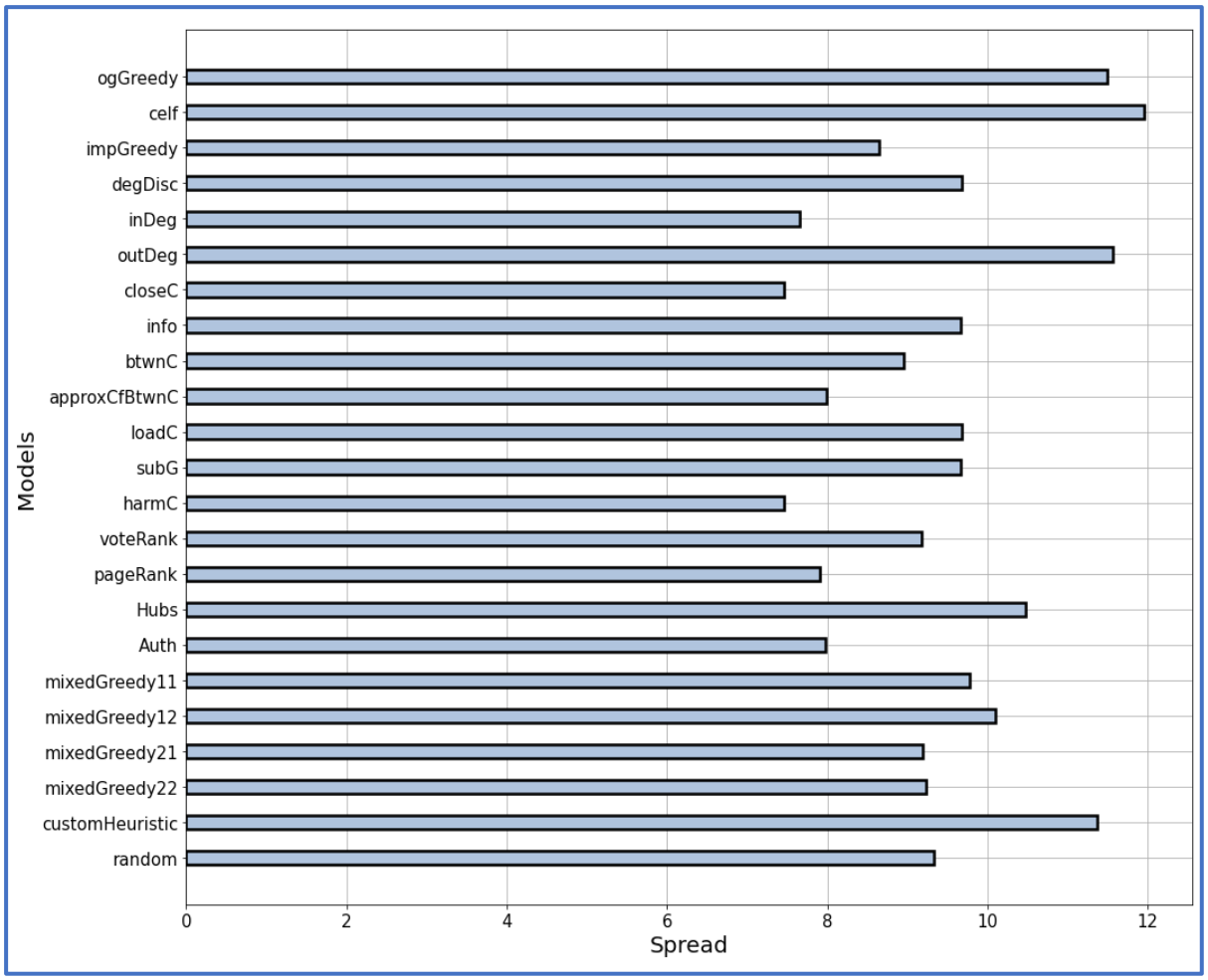

BitcoinOTC Seed Select Models: Comparison:
Models
mixedGreedy11
mixedGreedy12
mixedGreedy21
mixedGreedy22
customHeuristic
random
0
100
200
300
400
500
Spread
Models
mixedGreedy11
mixedGreedy12
mixedGreedy21
mixedGreedy22
customHeuristic
random
0
100
200
300
400
500
Spread by Time

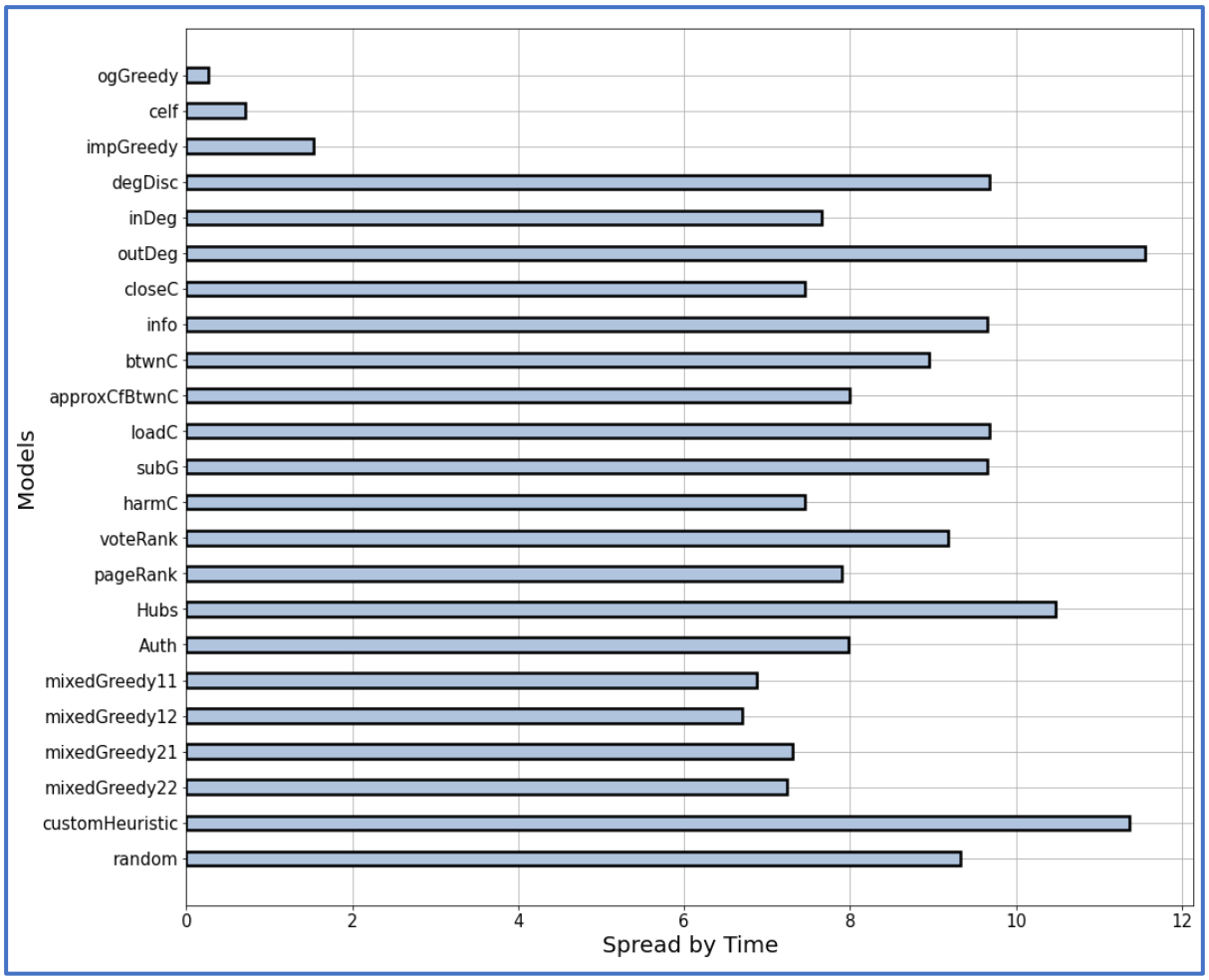

Facebook Seed Select Models: Comparison:
Models
mixedGreedy11
mixedGreedy12
mixedGreedy21
mixedGreedy22
customHeuristic
random
0
200
400
600
800
1000
1200
1400
1600
Spread
Models
mixedGreedy11
mixedGreedy12
mixedGreedy21
mixedGreedy22
customHeuristic
random
0
200
400
600
800
1000
1200
Spread by Time

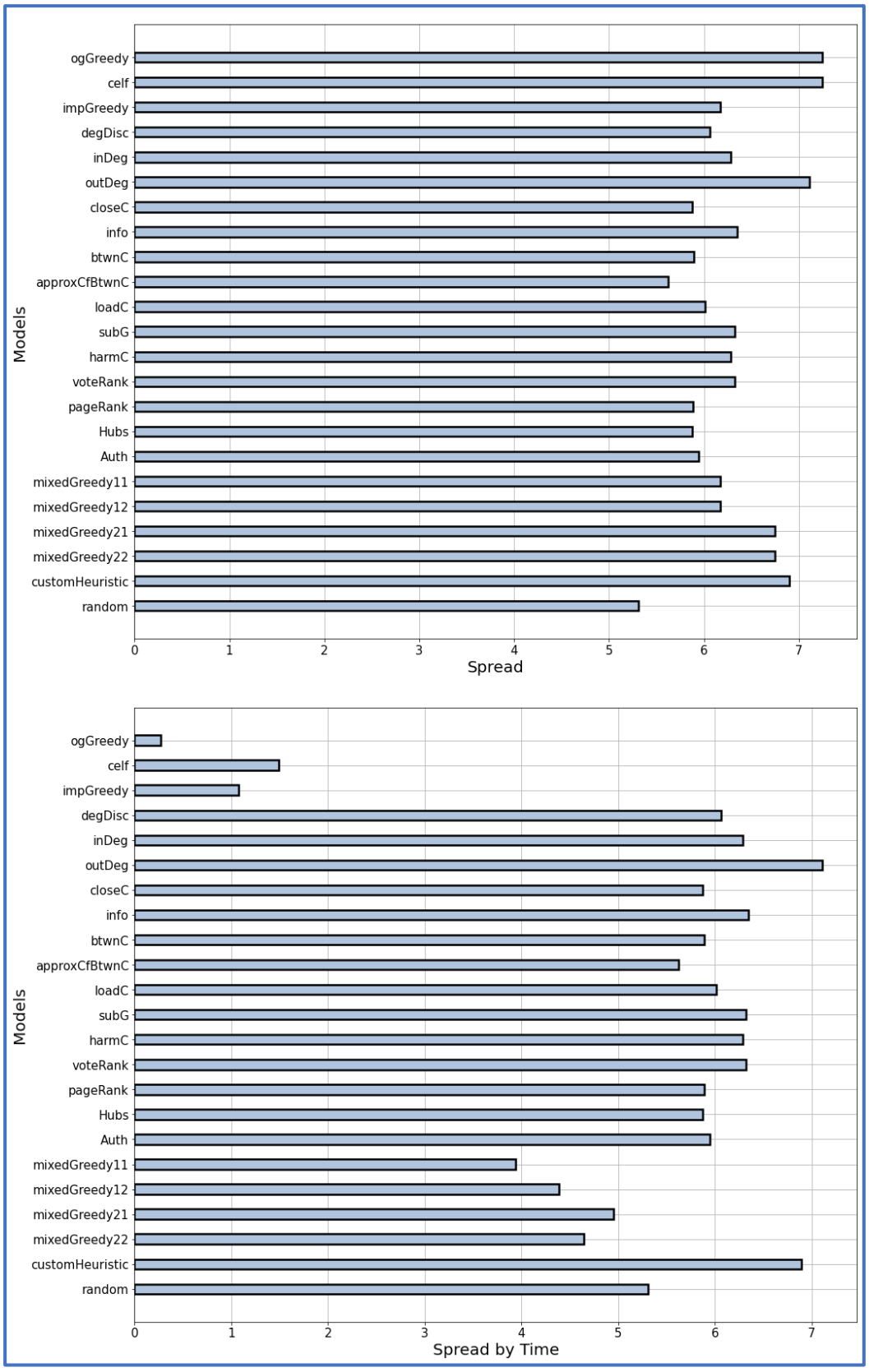
All Models

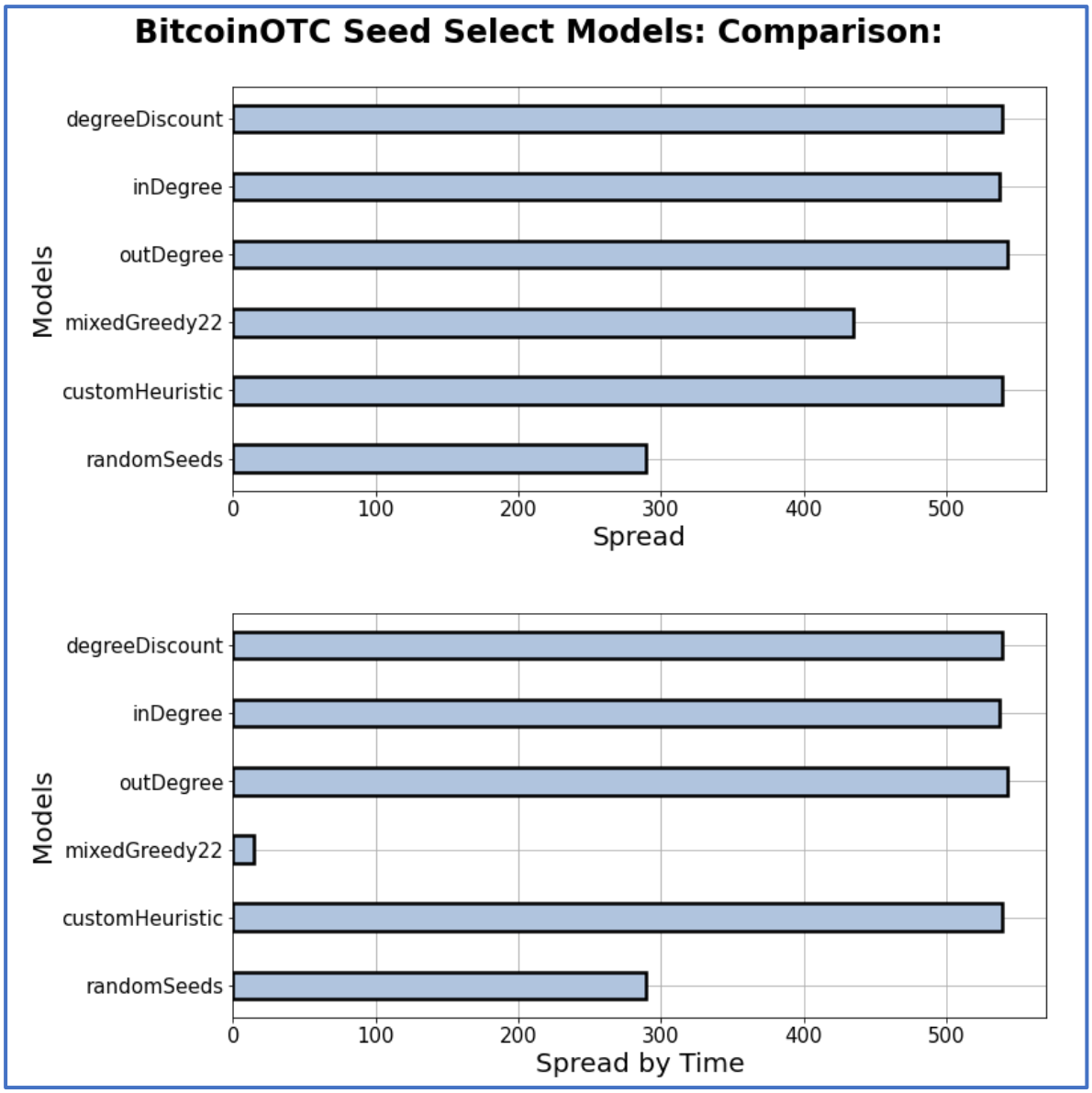


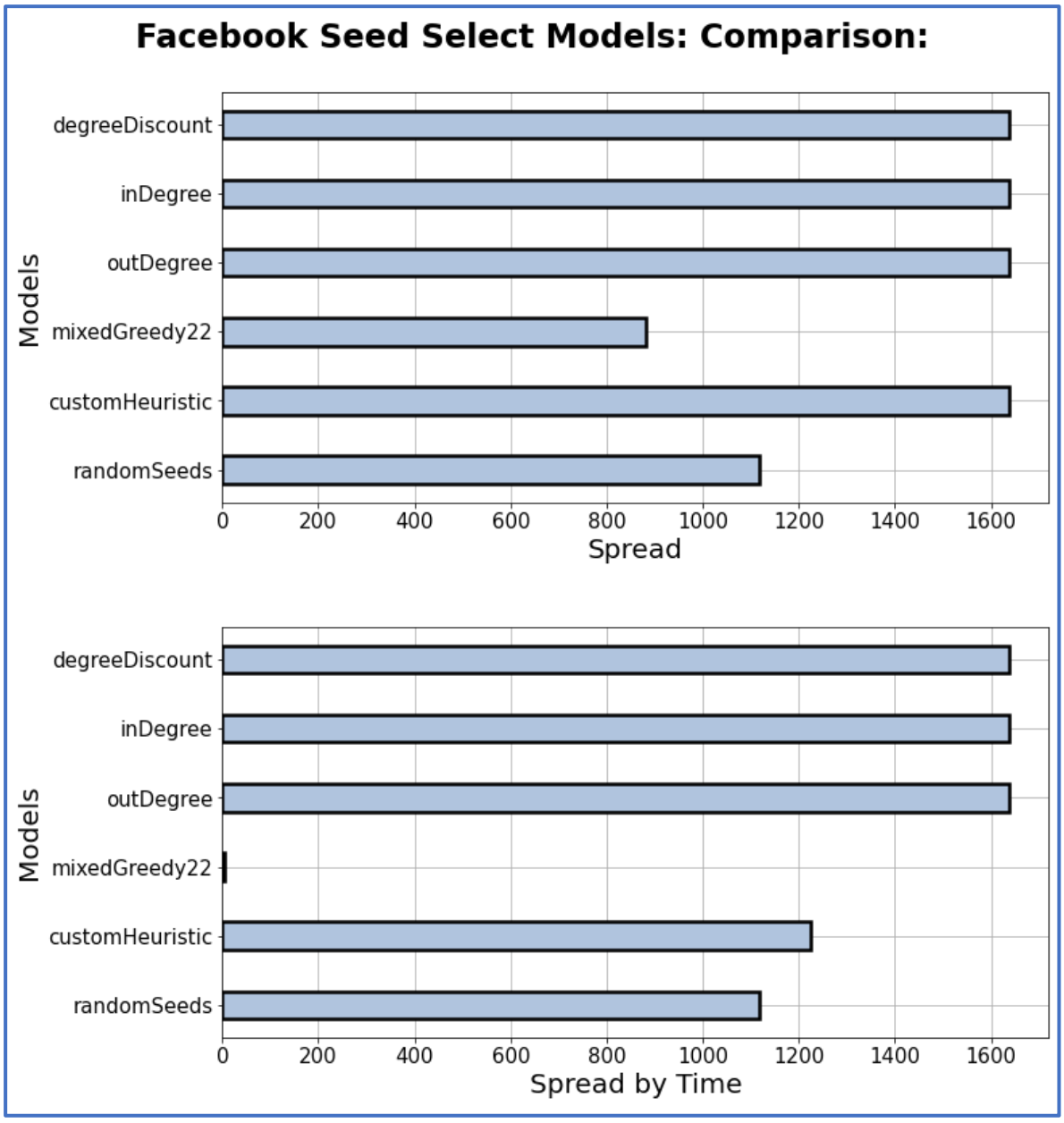
(zoomed out below)


ogGreedy
celf
impGreedy
degDisc
inDeg
outDeg
closeC
info
btwnC
approxCfBtwnC
loadC
subG
harmC
voteRank
pageRank
Hubs
Auth
mixedGreedy11
mixedGreedy12
mixedGreedy21
mixedGreedy22
customHeuristic
random
Models
0
2
4
6
8
10
12
Spread


ogGreedy
celf
impGreedy
degDisc
inDeg
outDeg
closeC
info
btwnC
approxCfBtwnC
loadC
subG
harmC
voteRank
pageRank
Hubs
Auth
mixedGreedy11
mixedGreedy12
mixedGreedy21
mixedGreedy22
customHeuristic
random
Models
0
2
4
6
8
10
12
Spread by Time

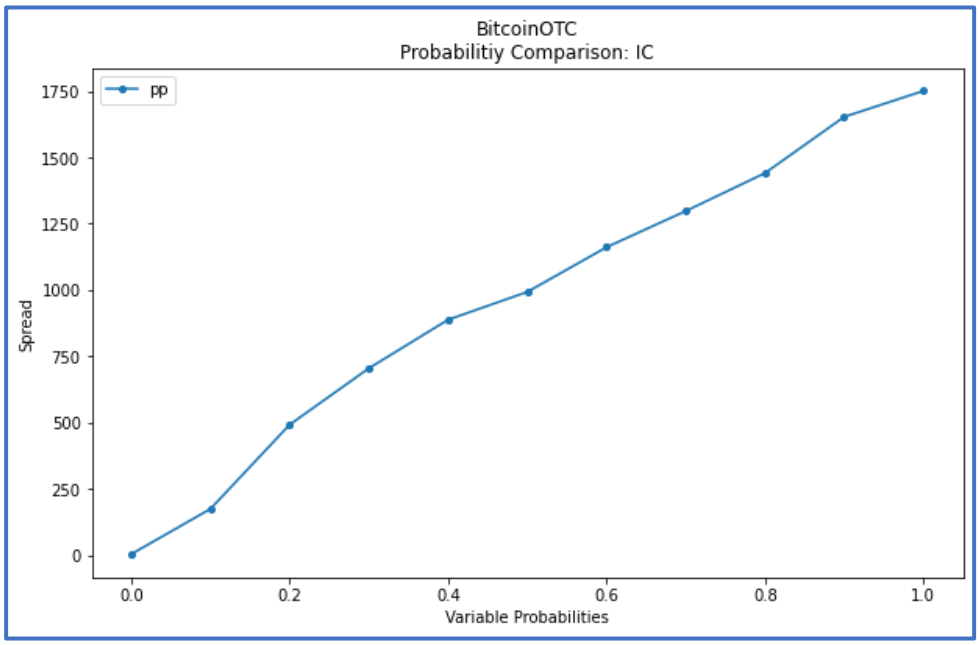

ogGreedy
celf
impGreedy
degDisc
inDeg
outDeg
closeC
info
btwnC
approxCfBtwnC
loadC
subG
harmC
voteRank
pageRank
Hubs
Auth
mixedGreedy11
mixedGreedy12
mixedGreedy21
mixedGreedy22
customHeuristic
random
Models
0
1
2
3
4
5
6
7
Spread
ogGreedy
celf
impGreedy
degDisc
inDeg
outDeg
closeC
info
btwnC
approxCfBtwnC
loadC
subG
harmC
voteRank
pageRank
Hubs
Auth
mixedGreedy11
mixedGreedy12
mixedGreedy21
mixedGreedy22
customHeuristic
random
Models
0
1
2
3
4
5
6
7
Spread by Time

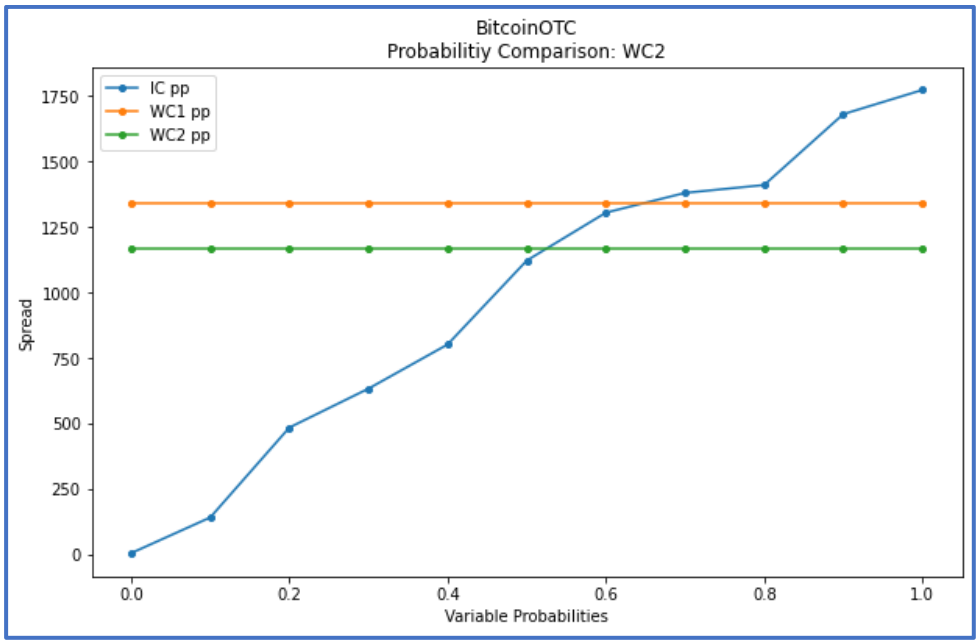

BitcoinOTC Seed Select Models: Comparison:
degreeDiscount
inDegree
outDegree
mixedGreedy22
customHeuristic
randomSeeds
Models
0
100
200
300
400
500
Spread
degreeDiscount
inDegree
outDegree
mixedGreedy22
customHeuristic
randomSeeds
Models
0
100
200
300
400
500
Spread by Time


Facebook Seed Select Models: Comparison:
Models
degreeDiscount
inDegree
outDegree
mixedGreedy22
customHeuristic
randomSeeds
0
200
400
600
800
1000
1200
1400
1600
Spread
Models
degreeDiscount
inDegree
outDegree
mixedGreedy22
customHeuristic
randomSeeds
0
200
400
600
800
1000
1200
1400
1600
Spread by Time

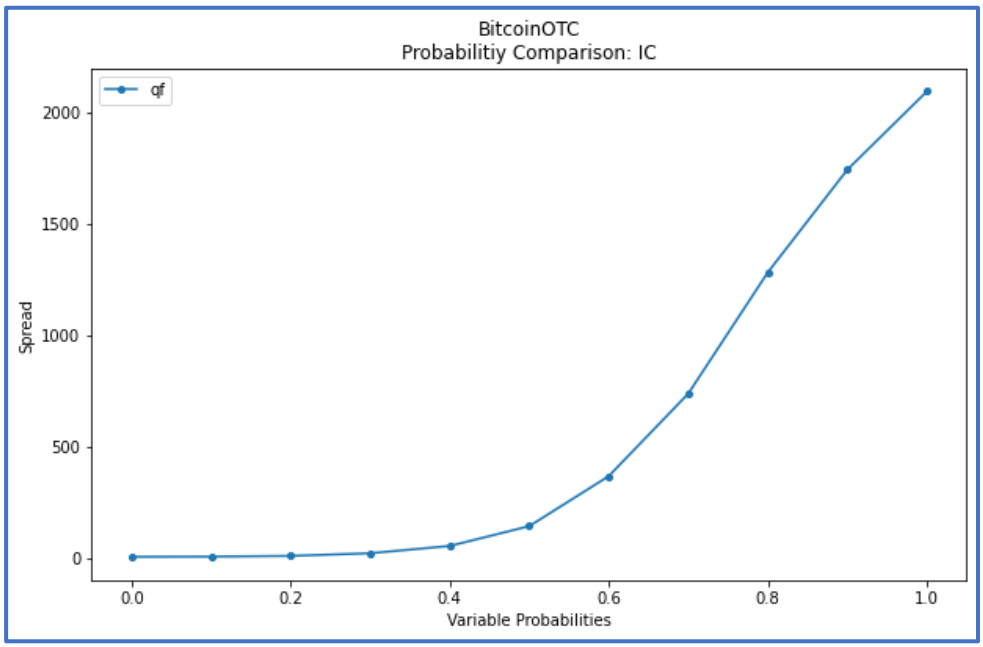
D.8 Propagation Parameter Comparison

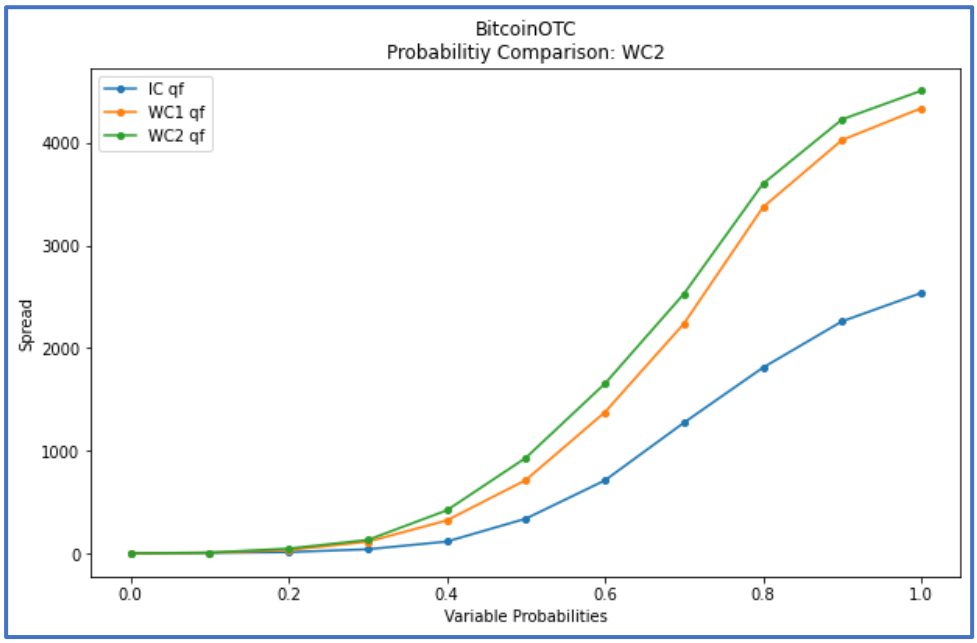
Propagation Probability



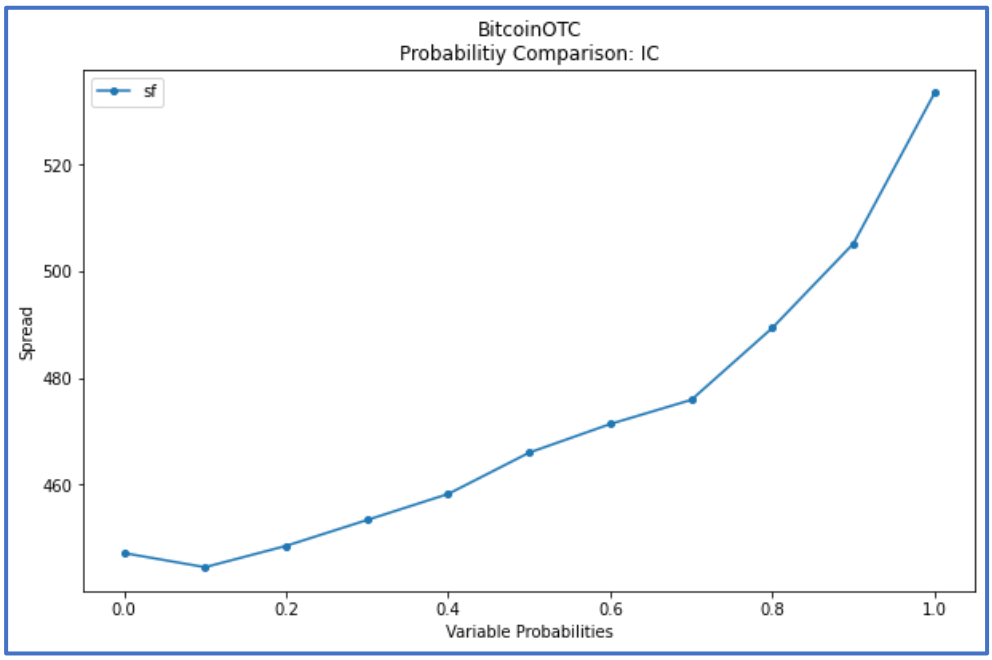


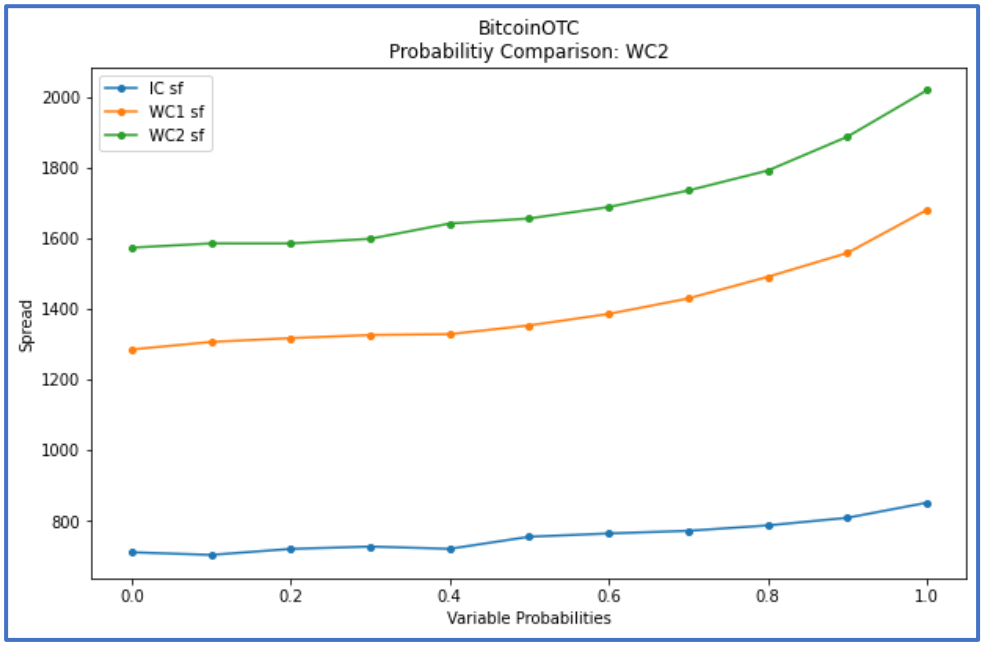
Quality Factor



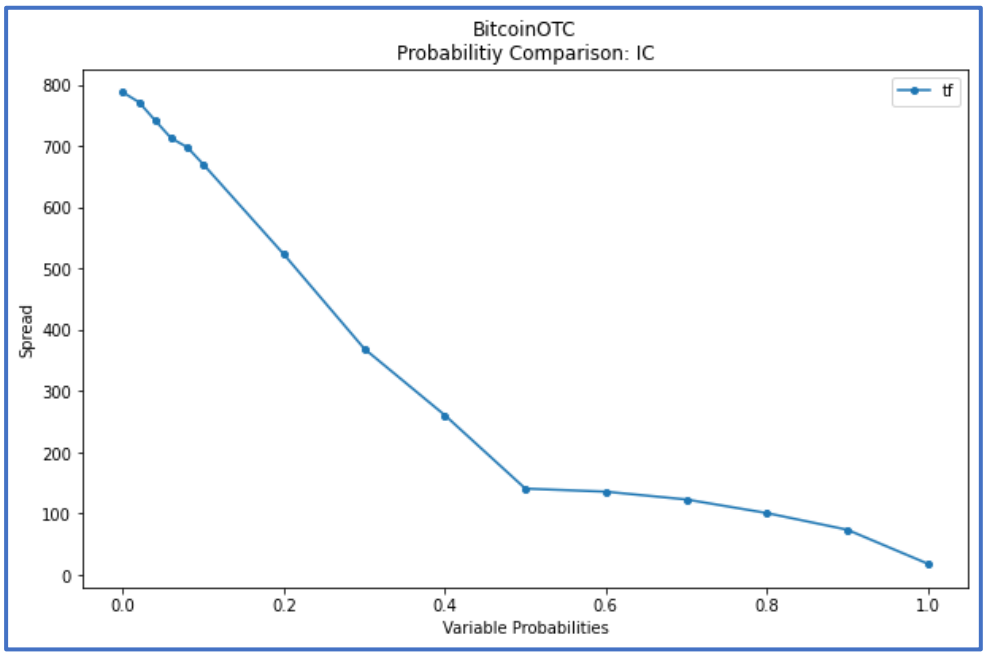


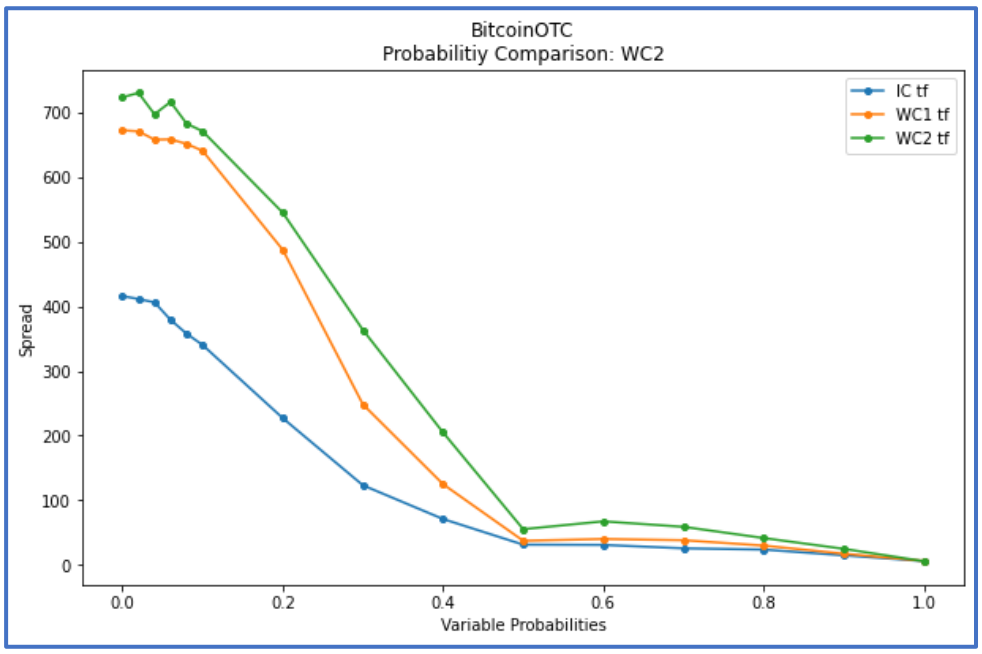
Switch Factor



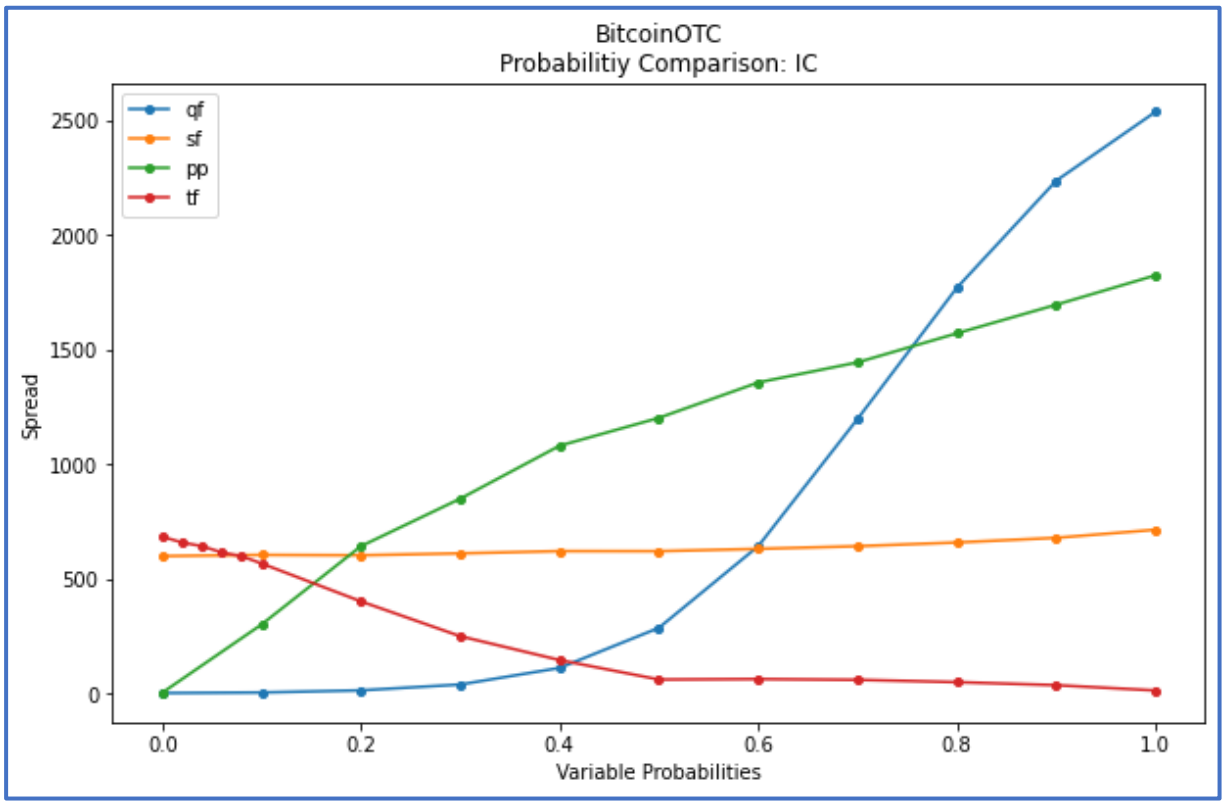


Time Factor


BitcoinOTC
Probabilitiy Comparison: IC
tf
Spread
Variable Probabilities
0
100
200
300
400
500
600
700
800
0.0
0.2
0.4
0.6
0.8
1.0


BitcoinOTC
Probabilitiy Comparison: WC2
IC tf
WC1 tf
WC2 tf
Spread
Variable Probabilities
0
100
200
300
400
500
600
700
0.0
0.2
0.4
0.6
0.8
1.0

All Parameters



Appendix E: Repo URL

Github repo link: {https://github.com/ssulls/Influence-Maximization-project.git}