Epidemic Graph Inference

Analysis of Graph Reconstruction from a Series of Cascades

Kevin Tang
California Institute of Technology
ktang@caltech.edu

Cody Han
California Institute of Technology
chhan@caltech.edu

ABSTRACT

Some epidemics spread on various types of real world networks which have unknown underlying structures. In our project, we analyzed the robustness and effectiveness of a greedy algorithm suggested by Netrapalli et al.[8] in reconstructing graphs from infection times. We implemented a framework for generating graphs, simulating epidemics on those graphs, solving for the original graph structure given the results of the simulations. We analyzed efficacy of our algorithms by comparing the inferred graph to the original. Our greedy algorithm applied to epidemics from a SIR model yields trees and tree-like graphs that accurately represent the original graphs that the simulations were performed on. Although the algorithm does not always achieve perfect recall of the original graph, we are able to reconstruct the majority of the graph structure with a reasonable number of cascades. We are successfully able to produce the theoretical results proposed by Netrapalli et al.

Keywords

Graph Inference, Epidemic, Greedy Algorithm

1. INTRODUCTION

1.1 Background information about Epidemics

We live in a connected world. Often, it is importance to model the connections between entities, such as the roads between buildings, the layout of a power grid, or the connections within a social network. We can model these connections between peoples as a network, represented by a graph with nodes and edges. Often, we not only want to model the connections, but how things spread through the network. For example, we may wish to model the spread of disease through a country, the propagation of viral advertising through a social network, or how viruses spread through computer networks. These self propagating things are known as epidemics.

The study of how epidemic and memetic spread through networks is a new and upcoming field. Much research has been done on modeling these epidemics and deducing central nodes in a graph given the graph structure. Different types of epidemics spread in different ways. and thus are modeled differently. A well known model stemming from epidemiology is SIR model, or Susceptible, Infected, Recovered. In this model, nodes are either non-infected but non active (susceptible), active and capable of infecting other nodes (infected), or non-active but has been infected in the past (recovered). This model is effective in modeling the spread of diseases. Extensions of this model also exist, such as the SI (susceptible, infected) model and the SIS (susceptible, infected, susceptible) [1]. In the SI model, the nodes do not recover and remain virulent. This model is a better description for incurable but infectious diseases such as AIDS, as well as Internet memes. The SIS model, on the other hand, better describes frequent diseases such as the common cold or the flu. It is worth noting that the commonly known Independent Cascade (IC) model is equivalent to the SIR model.

The main model that we focused on was the SIR/IC model, which can be formally described on a graph as follows: given an initial set of active nodes A_0, at each time step an activated node u is given a chance to activate each of its neighbors v with some probability $p_{u,v}$. After a node attempts to activate its neighbors, it becomes inactive (recovered) and cannot try to activate across its outgoing edges again. In the SI model, active nodes do not recover and become inactive, they remain active and attempt to activate their neighbors in each subsequent time-step. In the SIS model, each active node u becomes inactive, but can be reactivated through one of its incoming edges.

1.2 Introducing Graph Inference

Epidemic models give us an idea of when nodes are infected as well as which nodes are the most virulent or at risk. However, there are times where the opposite problem is important, such as knowing which nodes are infected at what times but don't know the structure of the graph. If we monitor all nodes in a graph in discrete intervals, we can deduce changes in the states of the nodes known as a cascade. What if we can observe a series of cascades, but don't know how these nodes are connected? Can we somehow use the models of graph epidemics to suggest how the nodes are interconnected?

This problem of graph inference can be modeled mathematically as follows. We wish to find out the structure of an

unknown graph $G(V, E)$. We define a cascade to be a vector of size $|V|$ containing the first infection times for each node. Given a set of cascades on G, we seek to create graph $G'(V, E')$ such that it reconstructs $G(V, E)$ as well as possible.

1.3 Graph Similarity Methods

Quantifying how accurately a graph reconstruction is a nontrivial matter. Many different similarity measures have been proposed to measure this. All similarity measures attempt to be constrained in several ways [6]. Firstly, the similarity score should be bounded within a range and comparison of any graph to itself should yield the maximum value of this bound. Similarly, each node within the graph should be most similar to itself. Finally, similarity scores should be meaningful in absolute terms, meaning that similarity scores from different graphs can be directly comparable. Thus, a good similarity measure is one that includes all of these things and also captures additional information about the similarities of the structure of the graph. A naive similarity measure is to compute the percentage of edges that are guessed correctly. This measure passes all of the requirements for similarity measures. As a fraction of correct edges, it is bounded from 0 to 1, and a graph would be completely correct with itself. However, the naive similarity measure does not capture any information about type two errors, or whether a guess was a good guess or not. Furthermore, it does not give information about whether or not the structure of the graph is similar or not. Therefore, other similarity methods should be explored.

One method of similarity that compares node to node similarity is first discussed by Kleinberg for the purpose of finding authoritative sources in an hyperlinked environment[3] and later formalized by Blondel et al. for the purposes of finding synonyms in text[2]. Melnik et al. also contributed to further solutions to the same similarity measure as applied to schema matching.[5] As such, this similarity measure has been used in many cases. This similarity computes a similarity matrix S between nodes in the two graphs, where $s(a, b)$ is the similarity between node a in the original matrix and b in the generated graph. This uses an iterative method of spreading the similarity through the Cartesian graph product of graphs A and B. This allows neighbors to spread similarity measure to each other. We can generalize this node similarity to graph similarity in two ways. First, we can take the trace of the similarity matrix and thus compare the similarity of nodes that should correspond to each other. Secondly, we potentially can find the set of n similarities that have unique rows and columns that maximizes the sum. This tells us whether or not we can interpret the new graph in such a way that nodes may correspond to a different node instead. A higher value of maximization similarity compared to the trace similarity would imply that the exact edges may be wrong, but the structure of the graph is very on point, or potentially isomorphic. Although using the trace is trivial to implement, the maximization similarity measure is a problem that is NP-hard and reduces to the maximum bipartite matching problem, and thus not truly viable to compute. However, a simple greedy approximation algorithm may be suffice to get just an idea of the graph's similarity. These two similarity metrics, however, fail to pass some of the necessary features of similarity, such as the similarity matrix is not always diagonally dominant and does not translate well as to a global comparison of the two graphs [2].

1.4 Current Inference Methods

Current approaches to this problem typically involve definition of novel node centralities along with their maximum likelihood (ML) estimators. Performing the ML estimation is computationally difficult, so the challenge involves showing that the MLE problem is convex or deriving an alternative formulation, for example as a combinatorial problem. Netrapalli [8] takes the former approach, whereas Shah takes the latter [9]. Netrapalli derives a change of variables that turns the problem of finding the graph that best explains a set of cascades into n convex problems, one for each node. Specifically, the graph is reconstructed from parental neighborhoods (set of all nodes that could have been a particular node's activator) for each node. The algorithm is locally efficient (number of times a node has to be infected scales with $d_i^2 \log |V|$ for each node i) and scales well because of its distributed nature.

The two approaches mentioned above also tackle different problems; Netrapalli looks for edges in the underlying graph, whereas Shah evaluates a "rumor centrality" and identifies a seed node for the epidemic. Prakash developed the NetSleuth algorithm [7], which can identify cases where there are multiple seed nodes. Each approach sheds light on the underlying network in some way; [8] learns the actual graph, [9] determines a single source, and [7] can determine multiple sources of epidemics.

Netrapalli [8] also gives a greedy algorithm with theoretical bounds proven on trees; this algorithm was the main subject of our study this term. We implemented it and tested it on trees and tree-like graphs to assess its graph recovery ability and robustness.

2. APPROACH

2.1 Overall code structure

Our project is written in Python 2.7 and uses Git for version control. We decided to use Python because we would be able to very quickly test and try different algorithms out, as well as have access to multiple graph libraries such as NetworkX, graphviz, and igraph. We had previous experience working with NetworkX in CS144, which made the experience a bit more pleasant. Our code documentation is aided by Sphinx and we used pytest for unit tests.

We designed our project as a Python package, such that accessing a function is as easy as typing `graph_interference.solver.greedysolver`. Our design to modularize our code heavily had several reasons. First of all, this forces us to standardize our inputs and outputs for our code, meaning that it would be simpler for us to link up bits of code together. Secondly this allows us to conceptualize the project into smaller, more manageable chunks in which one of us can take main responsibility for. Thirdly, modularizing the code proved to be especially useful while writing test scripts and generating figures, as it made the test code easy and fluid to write. We took care to document our code such that function inputs and outputs are clear, as well as used

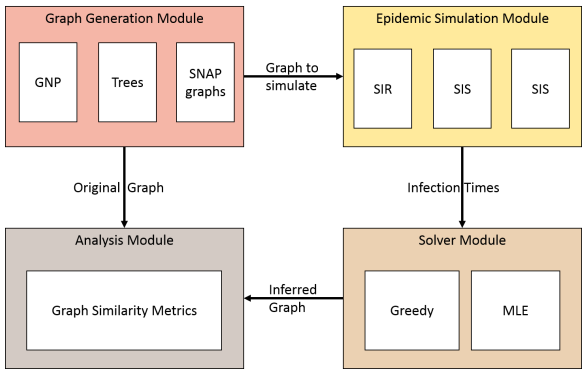

Figure 1: A diagram describing our code modules and how they communicate with each other

robust argument parsing that allowed us to be flexible with how to pass arguments to other parts of code. For example, we allowed the passing of one graph from one module to another via a NetworkX object, or by writing to and reading from files.

2.2 Individual modules

Graph Generation The graph generation model is responsible for generating graphs in the correct format for later programs to use. All graph generation modules are based off of a basic graph generator that we wrote, and simply overloads the basic graph generator's functions to add functionality. This forced us to maintain a standard of function names and also helped standardize variable name convention. We required each graph to generate a NetworkX graph, typically a directed graph. Each graph must contain nodes that are indexed by consecutive integers starting at 0. We accompanied each graph with information about what type of graph was generated and with what parameter, including an auto generated descriptor sting for easy debugging and testing. All graphs have ability to export itself to file as a GraphML file. One benefit of this type of organization is that future graphs that we may add may not need to use NetworkX at all, as long as it can be exported as a standardized format.

We implemented several types of graph generators, namely GNP (Erdos-Renyi) graphs, tree graphs, graphs edge lists as supplied by the SNAP repository[4], and graphs with a specified degree distribution. The GNP graph, importing from edge lists, and degree distribution graphs are written as wrappers over NetworkX functions that ensure that the resulting graphs and graph wrapper followed specifications. We made sure functions exposed enough details, but not enough to become unclear to use. We attempted to create trees based off of a function in NetworkX that generated power-law distribution trees. Unfortunately, this NetworkX module did not work very well for larger trees as the function would have a hard time converging, so we implemented our own tree generation algorithm. At each time-step, each node has a probability of growing an offshoot. This allows for very nice looking trees with varying degree distributions that has a flavor of preferential attachment.

Simulation The simulation module takes a graph, simulates epidemics on it, and outputs a set of infection time vectors that are the inputs of our graph inference algorithm. We implemented three common epidemic models: SI, SIR, SIS. They were extended from a base simulation module that defined shared functions between all epidemic models. The shared functions included graph loading, single epidemic steps, and full cascade runs that generate a set of cascades as final output. Each epidemic model also has a different definition of stability, that is, criteria for when the cascade has finished. The SIR model has the simplest definition of stability; it is finished when no nodes are active. The SI/SIS models require far more iterations because each node can go through multiple rounds of infecting its neighbors. One definition that we worked with for the stability condition in the SI and SIS models was the idea of the infected set of nodes staying constant for some k consecutive time-steps. Another more rigorous (and slower) way of determining stability, specifically for the SI model, would be to compute the set of all nodes that can possibly be infected given a specific set of seed nodes and run until all of those nodes are infected.

The simulation code can load edge weights from GraphML files stored on disk. If any edge does not have a weight specified in the graph, it is assigned a weight from a uniform $(0, 1)$ distribution. For each cascade, we choose a set of seed nodes by iterating through the nodes and activating each one with some probability α. The cascades are simulated in software and returned as `numpy` arrays of arrays in memory.

Solver The solver module takes a set of cascades generated from simulations and outputs the edges that most adequately explain the cascades it received. The algorithms we attempted to implement were all able to be separated into per-node optimization problems, so we defined a function that gives a particular node's parental neighborhood in the base class. A simple wrapper around the per-node solver was written to reconstruct the original graph from the results of the node-solver output on each node, and the inferred graph can be returned in memory as a NetworkX directed graph or written to disk in GraphML.

We implemented the greedy algorithm in [8] that works well for the SIR infection model and can be easily extended to the SI and SIS models. We ran the algorithm on trees and non-treelike graphs, verifying the results in the original paper and experimenting with the algorithm on different graphs. This algorithm can only return a tree as its final output by design, so we were somewhat limited in the graphs that we could run cascades on. Some bugs at the solver/simulation level made generating results with the greedy algorithm take longer than expected; subsequently, we did not manage to implement a working maximum likelihood solver.

Analysis Our analysis module takes two graphs and displays various statistics about the graphs. We first implemented a few very basic measure for comparing how well we did. We wrote functions to count the number

of edges we predicted correctly, failed to predict, and predicted false positives. We also counted the difference in number of total edges.

We also implemented a few centrality measures. First, we implemented a degree count comparison where we measured how different the predicted and actual degree was for that node. We then implemented this, but for comparing the degree distributions, where we compared the highest degree nodes to each other and so on. We also made an attempt at implementing the simple similarity measure that took the trace of the similarity scores. This iteratively calculated a similarity matrix s between two graphs of size $|V|$. The iterations were calculated as

$$s_{k+1} = \frac{(A \otimes B + A^T \otimes B^T)s_k}{\|(A \otimes B + A^T \otimes B^T)s_k\|}$$

where $\otimes$ is the Kronnecker product. This turned out less than satisfactory, however, as the results gave very little indication that one graph was more similar than another and failed several of the centrality tests. It would be a future endeavor to try to fix this similarity measure and also to explore the literature for more that may highlight other features of our graph reconstruction.

3. RESULTS

Our first analysis was looking graphically at the results of our graph reconstruction on generated tree graphs figure 2, and GNP (Erdös Renyi) graphs in figure 3. We can clearly see that we can almost perfectly reconstruct the original graph on trees, although the graph reconstruction is a lot worse when it comes down to a GNP graph. Graphically, we can see that although the graph has one misplaced edge, all nodes in the graph that had an original edge are accounted for. The resulting graph in figure 2 is actually isomorphic to the original graph. This suggests that for tree graphs, the greedy algorithm is extremely good at identifying the graph structure. This was evident in all runs on tree graphs.

The simulation on the GNP graph fared less as well. Roughly 2/3 of the original edges are predicted correctly, and the other one third incorrectly predicted (both type 1 and type 2 errors). However, the algorithm on GNP is often capable of producing the correct number of total edges and the correct number of connected components.

We then wanted to figure out how robust the algorithm was depending on the number of cascades. We generated a graph, and ran the simulation and inference modules with different numbers of cascades. For each number of cascades, we performed this sequence 10 times and collected the average reconstruction rate. We defined the reconstruction rate to be the fraction of the original graph that was correctly identified, and thus this does not measure type 2 errors. We measured the recovery rates for generated tree graphs in figure 4 and GNP graphs in 5. This shows that in both cases, the algorithm quickly converges almost exponentially to the maximum recovery rate. For tree graphs, near 100% recovery is expected for anything past $10n$ cascades, and 90% recovery requires only about $4n$ cascades. GNP graphs plateau at a recover rate of about 60%, but also converges quickly.

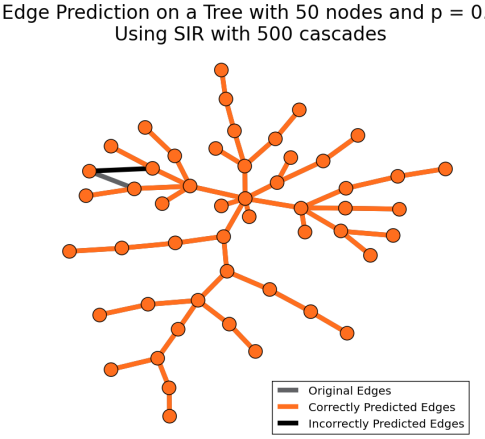

Figure 2: This tree graph was generated and then a simulation of 500 SIR cascades were performed. The original graph and generated graph's edges are overlaid, with orange edges being correctly identified edges, and black and grey edges being type 1 and type 2 errors respectively

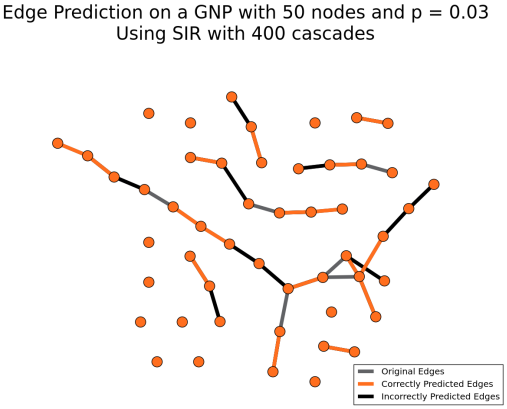

Figure 3: This GNP graph was generated and then a simulation of 500 SIR cascades were performed. The original graph and generated graph's edges are overlaid, with orange edges being correctly identified edges, and black and grey edges being type 1 and type 2 errors respectively

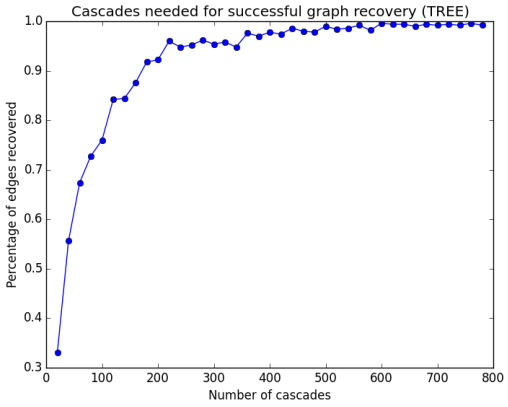

Figure 4: Recovery rate as a function of number of cascades using the same tree graph with 50 nodes. The recovery rate was averaged over ten runs for each data point

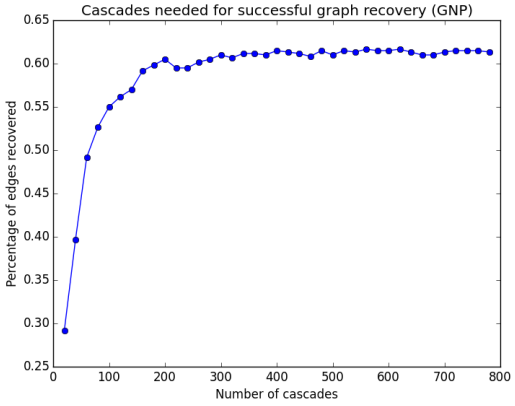

Figure 5: Recovery rate as a function of number of cascades using the same GNP graph with 50 nodes. The recovery rate was averaged over ten runs for each data point

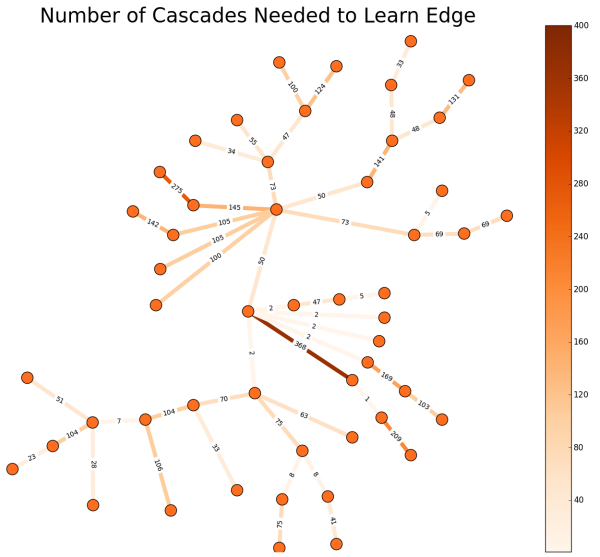

Figure 6: Number of Cascades needed to first identify the edge as a possible edge on a tree graph with 50 nodes

Our final analysis is to see how quickly does the algorithm learn nodes. We generated a tree graph of 50 nodes and ran the simulations to generate a total of 2000 cascades. We ran the greedy algorithm on the first k cascades, varying k from 1 to 2000. We recorded the first time that the greedy algorithm identified an edge correctly for each edge. The results are not surprising: the majority of the nodes are learned early, and by $2n$ cascades, about 3/4 of the edges have already been found. A colorized graph of the original edges of one of the trees is shown in figure 6. Here, lighter edges means that the edge was found earlier. Here, all edges were found in just 368 cascades.

4. CONCLUSIONS

4.1 Summary

We investigated the problem of reconstructing a graph's edges given vectors of (first) infection times on its nodes. We reviewed the literature and implemented graph generation, epidemic simulation, graph recovery, and graph analysis modules to test the greedy algorithm in [8]. We investigated its performance on tree and non-treelike graphs and discovered that it recovers trees accurately in an amount of cascades within the bounds given in the original paper. Moreover, we ran the algorithm on GNP graphs with $p \approx 1/n$ (tree-like); the algorithm's performance on GNP graphs was poorer than its performance on trees, but not by too much.

4.2 What's Next

The greedy algorithm we implemented, as is, assumes that a node v can only be infected by another node u that was infected exactly one time-step before v was infected. This assumption works when we are using the IC/SIR epidemic models, but it is clear that it does not hold for the SI and SIS models (active nodes are given multiple chances over different time-steps to infect their neighbors), so the algorithm

needs a little bit of tweaking when we change our epidemic model. One next step would be to loosen the assumption that only nodes infected at time $t-1$ could have infected nodes at time t. We can achieve this by assigning parents based on nodes that were active in the k steps before v was activated, where the case with $t = 1$ is the simple greedy algorithm once again. This modification will have a non-trivial effect on the algorithms running time, but will likely yield better results on cascades that came from SI and SIS simulation models.

Our algorithm can be parallelized for efficiency, as to solve for a parental neighborhood of any node we only need read-access to the set of infection times. Our current code solves for the incoming edges to each node one at a time, so we could observe a large speedup from exploiting concurrent programming.

Further steps that can be taken to advance the literature would be to analyze the two algorithms discussed (greedy, MLE) on a different cascade model. We have proposed a way to modify the greedy algorithm for the SI/SIS models, but the MLE reformulation may be trickier.

As discussed in the background, the methods used to capture graph similarity in this research are suboptimal to get more information. Graph recovery percentage does not tell us information about whether or not the structures of the graph are similar, and the node similarity matrix doesn't let us compare graphs to each other on a global scale. Thus, it would be worth some time to look further into the literature for graph similarity metrics that may add new insight as to how our algorithms perform.

Most of the graphs that we ran simulations on were machine-generated. We built the capacity to load and analyze graphs from files into our codebase, but the graphs that we attempted to analyze were too large or too non-treelike so that our algorithm did not output meaningful results. We also ran into efficiency issues here; on large graphs with more than tens of thousands of nodes our algorithm did not finish running in a reasonable amount of time. One way to address this may be to implement the graph solving (slowest part) modules in Cython or C and link them into the Python code.

Ultimately, we would like to apply graph inference to real world scenarios. There are a lot of ways in which graph inference can be used. We can attempt to figure out hidden connections between people by certain signals between them, such as disease or memetic expressions. With a virus, we can map out a dark net effectively without knowing exactly where the virus is going but just getting snapshots of where the virus has spread to. There are tons of projects available that can utilize graph inference, and it would be a great endeavor to pursue these topics.

4.3 What We Learned

Through this project, we dived head first into the field of graphs and networks. We were surprised by both the amount of done and being done, as well as the amount of questions that have yet to be answered. We learned not only much about epidemic spread, but also about other cutting edge areas of the study of networks and how these concepts were being applied to tackle real world problems. Our research brought us to see problems ranging from synonym finding in text to identifying super spreaders in viral epidemics.

We also learned that we need to pursue realistic and well explained research goals. We started out the project with a different idea relating to spreading network influence to regions that resisted influence due to the graph structure or due to the intrinsic nature of the node. We later realized that this topic is not very covered in the literature, and that we didn't have a concrete hold of what we would be doing to pursue this research idea. Having concrete research goals where success can be easily quantified is important for having a successful project as you have something to work towards. This is also useful for intermittent progress in the project where we have short term goals to work towards.

This was also a good experience in software development; neither of us have worked with large codebases very much, so we built our project from the ground up in an organic rather than heavily structured way. One benefit of our approach is that it allowed us to prototype quickly and write tests into our modules as we implemented them. We learned a lot about working on a project as a team.

5. REFERENCES

[1] L. J. Allen. Some discrete-time si, sir, and {SIS} epidemic models. *Mathematical Biosciences*, 124(1):83 – 105, 1994.

[2] V. D. Blondel, A. Gajardo, M. Heymans, P. Senellart, and P. V. Dooren. A measure of similarity between graph vertices: Applications to synonym extraction and web searching. *SIAM Review*, 46(4):647–666, 2004.

[3] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5):604–632, 1999.

[4] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[5] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 117–128. IEEE, 2002.

[6] M. Nikolic. Measuring similarity of graph nodes by neighbor matching.

[7] B. A. Prakash, J. Vrekeen, and C. Faloutsos. Spotting culprits in epidemics: How many and which ones? 1991.

[8] Praneeth Netrapalli and Sujay Sanghavi. Learning the graph of epidemic cascades. *ACM SIGMETRICS Performance Evaluation Review*, 40(1):211–222, June 2012.

[9] D. Shah and T. R. Zaman. Rumors in a network: Who's the culprit? *Information Theory, IEEE Transactions*, 57(8):5163–5181, 2011.

6. ACKNOWLEDGMENTS

We'd like to thank Adam Wierman for being a great mentor! We couldn't have done this project without him.