UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

MEASURING AND OPTIMIZING INFLUENCE FOR RESILIENT COMMUNITY NETWORKS

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

LAUREN YEAGLE
Norman, Oklahoma
2018

MEASURING AND OPTIMIZING INFLUENCE FOR RESILIENT COMMUNITY NETWORKS

A THESIS APPROVED FOR THE
GALLOGLY COLLEGE OF ENGINEERING

BY

To my family and friends
Thank you for your love and encouragement.

Acknowledgements

Thank you to my advisor, Dr. Kash Barker for helping me to finish this thesis in such a short period of time. Kash helped me develop ideas and guided me throughout this process. When I was unsure how to proceed, Kash was always supportive and directed me to sources for assistance. I appreciate your encouragement and confidence.

Thank you to the other members of my committee, Dr. Ziho Kang and Dr. Shima Mohebbi, for your guidance and direction throughout various research projects and classes during my time at the University of Oklahoma. I truly value your feedback and am thankful for your support.

Last, I would like to thank the professors and friends I've had in the School of Industrial and Systems Engineering. I appreciate the ways you have challenged me and further developed my ability to solve problems during my time in college. To my ISE friends, thank you for being available during long hours and arduous projects during our time together.

Table of Contents

List of Tables

List of Figures

Abstract

Social networks and the use of technology allow communities to be connected, creating opportunities for individuals to spread information and influence others. This communication is critical when disruptions, such as natural disasters, occur. Finding these influencers, and subsequently maximizing their spread of influence in the network, is key for mitigating the effects of these disasters and restoring communities as quickly as possible. The proposed model seeks to first maximize the spread of influence through the network and then to minimize the vulnerability of the network after the disruption occurs. Maximization of influence involves a mixed integer formulation while minimizing vulnerability requires a bi-level function based on maximizing these influence scores before and after a disruption. The model incorporates social vulnerability scores to ensure the most susceptible members of the community are reached when needed. The network is subjected to disruptions by removing influencers of the community, affecting the most vulnerable members of the population, and creating spatial disruptions to disconnect the network. The model may be used to locate influencers and can be used by decision-makers to determine areas that need more assistance to be resistant to disasters. The model is tested on a sample graph with 16 nodes and applied to a Twitter network to find the influencers before and after a disruption.

Chapter 1.0 Introduction and Motivation

Natural disasters, attacks, and other emergency situations necessitate the need for resilience, minimizing the extent of the disruption and recovering losses as quickly as possible. This concept of resilience has been studied and applied to numerous contexts, including its relationship to community structures (Cutter et al., 2008; Cutter et al., 2014; Ramirez-Marquez et al., 2018). Additionally, government agencies, such as the National Academics of Science, have shown interest in investing to strengthen resilience (2012). Likewise, the National Institute of Standards and Technology enables communities to be resilient—to "prepare for anticipated hazards, adapt to changing conditions, and withstand and recover rapidly from disruptions" (2015).

At the center of these communities, individuals communicate and influence each other. Influence can be described as an actor's ability to shape the intentions, thinking, feelings, and behavior of other people (Hamill et al., 2007). This type of communication is especially important during disruptions when influencers within the network can spread information useful in minimizing the vulnerability of the community and helpful to its recovery after the disaster. The spread of information using technology has been useful in mitigating the effects of several disasters, such as the Haitian relief effort (Zook et al., 2010; Gao et al., 2011), Hurricane Irene (Dailey and Starbird, 2014), and the Boston Marathon Bombing (Bagrow et al., 2018).

This research seeks to minimize the vulnerability of a social network by choosing influencers that reach the most susceptible members of the community. The network is subjected to different types of disruptions—removing the most influential, disrupting the more vulnerable nodes, and upsetting communities based on various spatial sizes. By obtaining the change in influence scores before and after the disruption, the network's vulnerability can be assessed. Likewise, establishing these influencers in the network can be useful for mitigating the effect of these disruptions.

The rest of this thesis is outlined as follows: Chapter 2 details the background and supporting literature for measuring and optimizing influence and community structures. Chapter 3 gives the methodological background for applying the linear threshold model, calculating social vulnerability scores, and applying a particle swarm algorithm for the bi-level optimization problem. In Chapter 4, a model is proposed to measure the spread of influence and the particle swarm algorithm is developed to solve for minimizing vulnerability. Chapter 5 provides illustrating examples of a small, random sample network and is then applied to a community in a larger Twitter network. Concluding remarks are found in Chapter 6.

Chapter 2.0 Literature Review

This next section details methods for measuring and optimizing influence, as well as demonstrating how influence has been applied in community structures. Analyses and applications of research performed on Twitter networks are also included.

2.1 Measuring Influence

Influence in a network can be understood in several different ways, including graph topological measures, dynamics-based measures, and approximation models for finding multiple influencers. Graph topological measures include degree, k-core, and eigenvector centrality, while dynamics-based measures comprise closeness, betweenness, Katz centrality, and PageRank (Pei et al., 2018). Some applications have proposed variations on these methods including LeaderRank (Lu et al., 2016) and employing the Google reduced matrix from PageRank (Zant et al., 2018). Models to find multiple influencers track the growth of influence through the network.

The spread of influence through a network can be exhibited through the linear threshold and independent cascade models. In each case, edges between nodes are assigned a propagation probability. A node's influence in the linear threshold model depends on all of the neighbors of the node. If the sum of the arcs connected to active nodes is above a certain threshold, the node is considered active. Conversely, the independent cascade model passes influence from one active node to another, depending on a single probability between the two nodes (Kempe et al., 2003). These approaches were further applied to marketing applications using stochastic propagation models to reflect the power-law in the size of the cascade to determine how the number of recommendations increases in a time period (Leskovec, Adamic, et al., 2007). In another approach, optimal percolation, nodes are defined with a probability that a node is not activated given that another node is removed. The giant component represents a fraction of the inactive nodes and is minimized in order to maximize spreading (Morone and Makse, 2015). Additionally, Dodds (2018) employs a Susceptible-Infectious-Recovered (SIR) model, defining a gain ratio as a measure of the expected newly infected edges from a single infected edge connected to an uninfected edge. Because networks are typically large and complex, algorithms are required to find influencers in a reasonable amount of time.

2.2 Algorithms to Find Influencers

Various algorithms have been used to measure the spread of influence through the network. Greedy hill-climbing, used by Kempe et al. (2003), was found to choose influencers more effectively than by random chance or by using graph measures, such as high degree and centrality. A cost-effective lazy forward selection algorithm (CELF) seeks to first calculate marginal improvements and then selects nodes based on decreasing order of improvement. If the top marginal improvement is invalid, it is recomputed and placed back in the order. The top element will exhibit minimal change, and therefore it can be recomputed without all marginal improvements needing to be recalculated (Leskovek, Krause, et al., 2007). Adaptations (Chen et al., 2009) and improvements (Goyal et al., 2011) on this model also are found to be effective.

Other approaches include iteratively building a hypergraph of the nodes that would influence a randomly chosen node (Borgs et al., 2014), finding the minimum fraction of nodes to fragment the network using a non-backtracking matrix (Morone and

Makse, 2015), and a collective influence algorithm measuring influence through a breadth-first search on a directed diffusion graph (Pei et al., 2018; Teng et al., 2016).

When communities are incorporated into the analysis, k-medoids created a large scope of influence because it considered the local and global topological structure of the network (Zhang et al., 2013). This relationship between influence and communities has been further explored to understand how the structure of communities impacts the spread of information.

2.3 Influence and Community Structures

Communities can help explain how information is passed to neighbors in a network. Leskovec, Adamic, et al. (2007) identified communities to understand how recommendations flowed through similar interests in a network. One approach by He et al. (2015) first builds communities based around "super" nodes with high modularity. The node with the highest degree is selected as the spreader. Then, spreader nodes from other communities that are not connected to the first super node are chosen as influencers. This method performed better than choosing nodes by high degree in different networks. To understand the virality of content, communities are formed around early adopters, defining adopter entropy and the spread of these users across their community (Hui et al., 2018). Wei et al. (2018) identified influential nodes based on overlapping communities and network structure. They developed a ranking method based on the propagation capacity (number of communities a node belongs to) and the propagation speed (based on the network constraint coefficient of structural holes) to locate influential nodes. These types of community structures can be beneficial when analyzing the connectivity of Twitter networks.

2.4 Influence in Twitter Networks

Research also has been conducted in understanding influence in Twitter networks. Kwak et al. (2010) ranked influential users by their number of followers, PageRank, and number of retweets, finding that the number of retweets produces different rankings than the other two measures. This illustrates the fast diffusion of information. Virality in Twitter networks is often very small as popularity is influenced most by the size of the largest broadcast of information; a larger broadcast causes more spreading (Goel et al., 2015). Additionally, weak ties between Twitter communities can still be as effective as strong ties in generating larger amounts of traffic (Weng et al., 2018).

Real scenarios have been analyzed to understand the spread of information in these contexts. Bagrow et al., (2018) examined Twitter data from the Boston Marathon Bombing. The data showed that the population within three kilometers of the blast site and within five hours after the explosion are mentioned more quickly than their direct tweeting—this is likely due to news media sources. Also, a stronger second spike in activity after the bombing was related to the arrest report of an individual witnessed by the initial population, showing that once engaged, the Twitter population was ready to forward information (this had stronger virality although the length of activity was shorter). Another application from Ferrara (2018) analyzed the role of Twitter bots in the 2016 presidential election. Twitter bots typically have a high frequency of retweets and a high connectivity, creating confusion in distinguishing bots from humans. These characteristics of Twitter networks are helpful in understanding the nature of how influence spreads.

Chapter 3.0 Methodological Background

This section provides background to how influence is measured through the linear threshold model, the relevance of social vulnerability measures, and employing the particle swarm optimization model to solve the bi-level optimization problem.

3.1 Linear Threshold Model

To model the spread of influence through the network, this analysis will use the linear threshold model. As stated earlier, this algorithm depends on the status of all the neighboring nodes to determine if a node is active. Specifically, a weight, $b_{v,w}$, is assigned between nodes v and w to represent the probability that influence is passed. The sum of these arcs into node v must be less than or equal to one, $\sum_{w\ neighbor\ of\ v} b_{v,w} \leq 1$. Every node then chooses a threshold, θ_v, randomly from the uniform distribution between [0,1]. This threshold signifies the number of active nodes required to influence v. The model runs iteratively, as nodes activated in the previous time period remain active for the rest of the simulation. Nodes are activated as their sum of the edges of their neighboring active nodes surpasses the threshold limit, $\sum_{w\ active\ neighbor\ of\ v} b_{v,w} \geq \theta_v$. The number of active nodes at the end of the cascade represents the influence score (Kempe et al., 2003).

Because this problem is NP-complete, it must be approximated for large network sizes. This is proven by considering a parallel case of the Vertex Cover problem. The complete proof, as well as the proof for submodularity that allows for a good approximation using greedy techniques, is shown in Kempe et al. (2003).

3.2 Social Vulnerability Scores

During a disruption, it is important for the most vulnerable members of a community to be reached and assisted. To ensure this occurs, the optimization model can be incentivized to reach these members of the community. Communities can be assessed for their level of social vulnerability to environmental and other types of disruptions based on their social status. Cutter et al. (2003) have developed much of this work into assigning social vulnerability scores to United States' counties based on social and place inequalities. This, of course, includes socioeconomic status as communities with more wealth and larger safety nets are able to recover quicker when disasters occur. It also can include factors such as race and ethnicity, age, housing and transportation status, and education (see Cutter et al. (2003) for a comprehensive list). The Centers for Disease Control and Prevention (CDC) manages a similar Social Vulnerability Index. The index ranks each county based on 15 variables similar to Cutter et al. (2003) and groups these variables into four main categories—socioeconomic status, household composition and disability, minority status and language, and housing and transportation (Flanagan et al., 2011).

The data based on the 2016 census will be used for this analysis (Centers for Disease Control, 2016). Social vulnerability scores will be used to prioritize influence to reach the most vulnerable members of the community by including vulnerability in the objective function.

3.3 Particle Swarm Optimization

Because networks are typically large in size, the optimization model can be difficult to implement. Therefore, approximation models become useful in finding good solutions in a reasonable amount of time. This analysis will apply a particle swarm optimization to solve for influencers in the network. The particle swarm optimization model was developed by Kennedy and Eberhart (1995) to solve multi-dimensional nonlinear problems. It stems from research in social sciences, following the patterns of a flock of birds or a school of fish. After initializing a swarm of particles, these particles are then allowed to search the solution space, taking information from each particle's best solution and the best global solution. In each iteration, the particles are updated with these cognitive and social factors to converge on the optimal solution. The particle swarm model has these roots in genetic algorithms and evolutionary programming (Kennedy & Eberhart, 1995).

In the formulation of this problem, each node is stored as a binary variable, indicating whether it is active or not. Active nodes are chosen randomly and evaluated by the linear threshold model. Then, based on the evaluation scores, the particles will update and move towards the optimal solution. The particles will update for a certain number of generations and will ideally converge during this time.

3.4 Particle Swarm Optimization in the Bi-Level Problem

In order to minimize the vulnerability of influence after a disruption, a bi-level programming problem is required. In these problem sets, the objective function and the constraints for the upper and lower levels are expressed as follows:

$$\begin{aligned} &\mathbf{min}_{x\in X} \ F(x,y) \\ &\mathbf{subject\ to} \ G(x,y) \le 0 \\ &\qquad \mathbf{max}_{y\in Y} \ f(x,y) \\ &\qquad \mathbf{subject\ to} \ g(x,y) \le 0 \end{aligned}$$

The variables, x and y correspond to the decision variables for the leader and follower, while $F(x, y)$ and $f(x, y)$ relate to their objective functions, respectively (Gao et al., 2011). The leader function will try to minimize the vulnerability, dependent upon maximizing influence before or after the disruption—the follower function.

Influencers before the disruption are chosen using the linear threshold model. Then, these influencers are removed from the network, and the influencers after the disruption are found. The difference in their influence scores corresponds to the vulnerability measure. Because minimizing vulnerability depends on first maximizing influence before and after the disruption, a bi-level formulation is necessary. Particle swarm optimization is effective in solving these types of problems (Parsopoulos et al., 2002; Gao et al., 2011; Kennedy & Eberhart, 1997; Kuo & Han 2011; Li et al., 2006; Sinha et al., 2018; Kuo & Huang et al., 2009). The formulation for the problem is shown in the following section.

Chapter 4.0 Proposed Model

In this section, a model is proposed to measure influence based on the social vulnerability scores of those influenced.

4.1 Single-Level Formulation

Individuals in the network are represented as a set of nodes, N, with an index of influencers, $i \in N^I$, and an index of those influenced, $j \in N^J$, such that $N^I \cup N^J = N$. Links exist between individuals, where $(i, j) \in L$. A propagation probability, p_{ij}, exists on each of these links. If the sum of the active propagation probabilities of the links leading into a node is above a threshold limit, the node is considered active. This problem is modeled like a linear threshold model combined with a set cover problem (Kempe et al., 2003).

Variables include the list of starting influencers, y_i, and the list of ending nodes influenced, s_j. Because the starting influencers should also count as influenced individuals, x_j will be the total number influenced, or the max of y_i and s_j.

Parameters include the propagation probability of edges, p_{ij}, the social vulnerability associated with each node, v_j, and a large number, M, to create the maximum bounds for total influenced, x_j.

$$\textbf{Maximize } \sigma = \sum_{j \in N^J} x_j \cdot v_j \tag{1}$$

Constraints set a node as active based on the threshold limit, limit the number of starting influencers, and find the ending number of influenced nodes. The first constraint sums the propagation probability, p_{ij}, on the arcs leading to each node. This depends on the status of the outgoing node, y_i. The variable, u_j, holds the summation.

$$u_j = \sum_{j \in N^J} p_{ij} \cdot y_i \quad \forall (i,j) \in L \tag{2}$$

The next set of constraints use a threshold, θ_j, to determine if the person is influenced, where s_j is a binary variable to determine if the node is active. Because the formulation is trying to maximize s_j, only equation (3) is needed to provide the lower bound of 0 if the node does not meet the threshold.

$$\theta_j s_j \leq u_j \qquad \forall j \in N^J \tag{3}$$

The third constraint limits the number of starting influencers to c.

$$\sum_{i \in N^I} y_i \leq c \quad \forall i \in N^I \tag{4}$$

The next set of constraints finds the total number of influenced nodes.

$$x_j \geq s_j \qquad \forall j \in N^J \tag{5}$$

$$x_j \geq y_j \qquad \forall j \in N^J \tag{6}$$

$$x_j \leq s_j \cdot M b_j \qquad \forall j \in N^J \tag{7}$$

$$x_j \leq y_j + M(1 - b_j) \quad \forall j \in N^J \tag{8}$$

$$b_j \in \{0,1\} \qquad \forall j \in N^J \tag{9}$$

Other binary constraints are shown below.

$$y_i \in \{0,1\} \quad \forall i \in N^I \tag{10}$$

$$s_j \in \{0,1\} \quad \forall j \in N^J \tag{11}$$

4.2 Bi-Level Particle Swarm Algorithm

The bi-level problem seeks to maximize the influence of the graph before a disruption, σ_0, and afterward, σ_1. This change in influence should be minimized in order to reduce the vulnerability of the network. A particle swarm optimization method, as shown in Gao et al. (2011), will be used. The formulation is shown below.

$$\begin{aligned} &\textbf{min}_{x\in X}\ \sigma_0 - \sigma_1 \\ &\textbf{subject to}\ \ G(x,y) \le 0 \\ &\qquad\qquad \textbf{max}_{y\in Y}\ \sigma_1 \\ &\qquad\qquad \textbf{subject to}\ g(x,y) \le 0 \end{aligned} \tag{12}$$

The constraints are the same as the single level formulation. Maximizing σ_1 will seek to maximize σ_0. The bi-level particle swarm algorithm, adapted from Gao et al. (2011), is shown in Table 1 on the next page.

Table 1: Particle Swarm Bi-Level Algorithm

```
Algorithm 1: A PSO based algorithm for minimizing network vulnerability
Step 1: Initialize swarm of particles, $K$, for number of nodes on starting graph, $g$
  Randomly set positions, $x_i$, for starting influencers to 1 and others to 0
  Assign velocities, $v_{x_i}$, to all particles between $[-1,1]$
  Evaluate using linear threshold model
Step 2: Initialize counter for leader's loop, $j_g = 0$
Step 3: For the k-th particle, get response from follower:
    Step 3.1: Remove arcs connected to influencer nodes, creating graph, $f$
    Step 3.2: Initialize counter for follower's loop, $j_f = 0$
    Step 3.3: Initialize ($y_i$, $v_{y_i}$) and evaluate particles on graph $f$ using linear threshold model, finding particle best, $p_{y_i}$, and global best, $y^*$
    Step 3.4: Update positions and velocities, where $w$ is the inertia, $c$, is the cognitive factor, $s$, is the social factor, and $r$ is a random number:
        $v_{y_i}^{k+1} = w v_{y_i}^K + (c r_1^K (p_{y_i} - y_i^K) + (s r_2^K (y^{*K} - y_i^k)$
        $y_i^{K+1} = y_i^K + v_{y_i}^{k+1}$
    Step 3.5: Update counter, $j_f$
    Step 3.6: Return $y^*$
Step 4: Find change in influence between graph $g$ and graph $f$. Record particle best, $p_{x_i}$, and global best, $x^*$.
Step 5: Update positions and velocities, where $w$ is the inertia, $c$ is the cognitive factor, $s$ is the social factor, and $r$ is a random number:
        $v_{x_i}^{k+1} = w v_{x_i}^K + (c r_1^K (p_{x_i} - x_i^K) + (s r_2^K (x^{*K} - x_i^k)$
        $x_i^{K+1} = x_i^K + v_{x_i}^{k+1}$
Step 6: Update counter, $j_g$
Step 7: Return $x^*$
                                                                        [end]
```

Chapter 5.0 Illustrative Examples

Two examples were created to test the algorithm. The first features 16 nodes and shows how the flow of influence can spread through the network. The second models a Twitter community and shows how information disseminates in a large, highly connected network.

5.1 Sample Network

To test the formulation, a random network with 16 nodes was created, as shown in Figure 1. The size of the node corresponds to the social vulnerability, where larger nodes are more vulnerable. Some of the most vulnerable nodes include nodes 1, 2, 8, 14, and 15. The model will try to reach these nodes as much as possible. Threshold limits on each node constrain its ability to receive influence. Nodes with high threshold limits, such as 1, 4, 7, 12, and 13, have greater difficulty activating. The social vulnerability and threshold values were randomly sampled. The edge thickness corresponds to the propagation probability. Two nodes (15% of the size of the network) were chosen to be influencers.

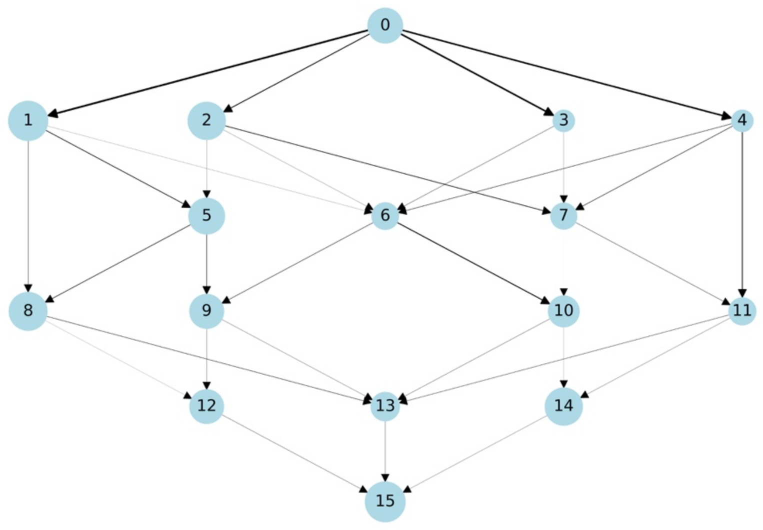

Figure 1: Sample network

Because this network is small, it can be run to find the optimal solution. To compare the performance of the algorithm with the optimal solution, only one iteration will be run. The optimal result chooses nodes 0 and 14 as influencers and results in a score of 4.217. This result is compared with the results of the particle swarm algorithm in Figure 2. First, the swarm size was varied while the number of generations was kept constant at 100. A swarm size of 50 was required to reach the optimal solution. Then, the swarm size was kept constant at 100 while the number of generations varied. This always produced the optimal result.

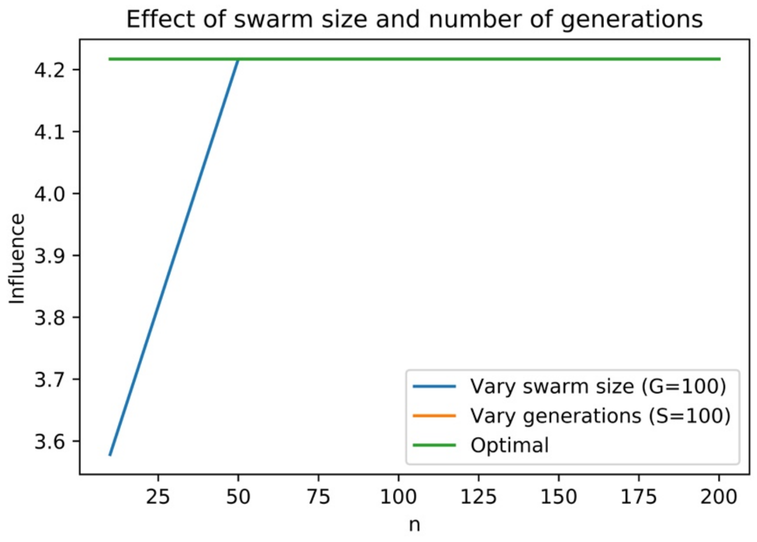

Figure 2: PSO parameter selection

However, when multiple iterations are allowed, and influence continues to pass, nodes 0 and 7 are chosen as influencers to produce a score of 4.781. This result, same as the before-disruption outcome, is shown in Figure 3 below. Influencers are shown in green and influenced in yellow. After removing the edges leading from the two influencers and rerunning the model, the score drops to 3.689, with nodes 4 and 5 chosen as influencers. The most vulnerable nodes are still reached as much as possible, even after the disaster.

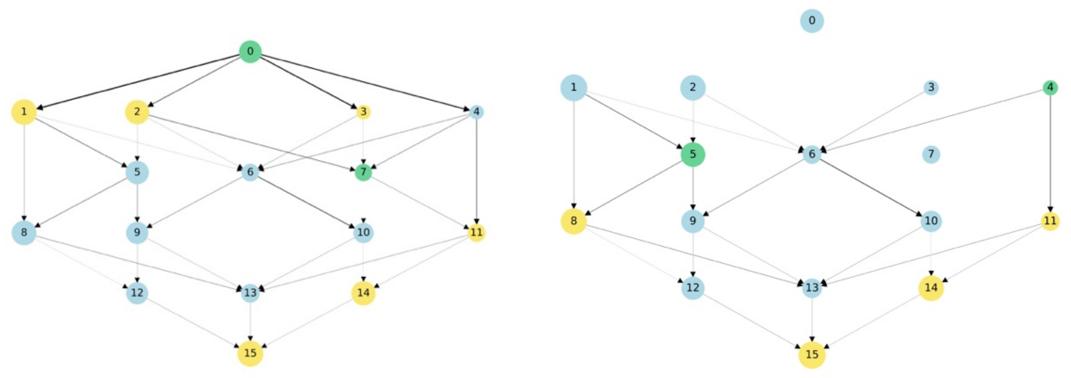

Figure 3: Influence results

5.2 Twitter Network

The Twitter network was created from Stanford's network dataset collection (Leskovek and Krevl, 2014). The network consists of 216 nodes and 3,507 edges. Most of the nodes are located within California or nearby states. The location of the user is based on either the geolocation of their last tweet or the location provided in the user's profile. This provides location data for 191 nodes.

5.2.1 Topological Measures

The average degree of the network is 16.236 and consists of 45 strongly connected components. This, along with a network diameter of six and an average path length of 2.433, illustrates the scale of large connectivity of the network. Figure 4 illustrates the connectivity of the network. The dense area near the center represents users in and near San Luis Obispo, California.

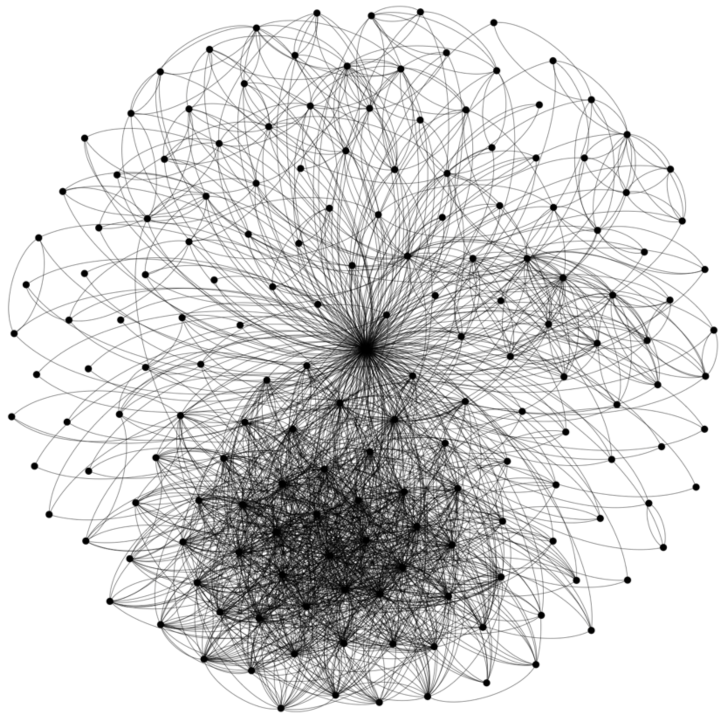

Figure 4: Twitter network connectivity

Because of this connectivity, influence should spread through these highly connected areas easily, as redundancy provides multiple opportunities for a node to be reached. The degree distribution is shown in Figure 5 below, where the scale-free distribution becomes evident. The power law-scaling demonstrates a node's ability to increase its connectivity at a much higher rate as its connections grow. Therefore, nodes in highly connected areas have a much greater distance in connectivity to those with a smaller degree (Barabasi & Albert, 1999).

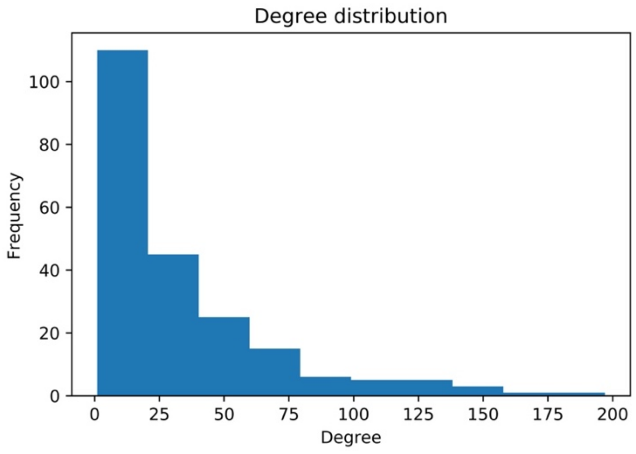

Figure 5: Degree distribution

Figure 6 shows the location of the Twitter users within the network. The highest density of users is located within the San Luis Obispo region along the coast of California (59 users). Figure 6 also shows the social vulnerability scores by county, where darker colors represent higher scores. Most users along the coast have a lower vulnerability, and vulnerability increases in further inland regions. This seems intuitive as users that live along the coast have a higher income and perhaps better opportunities, preparedness, and support during disruptions.

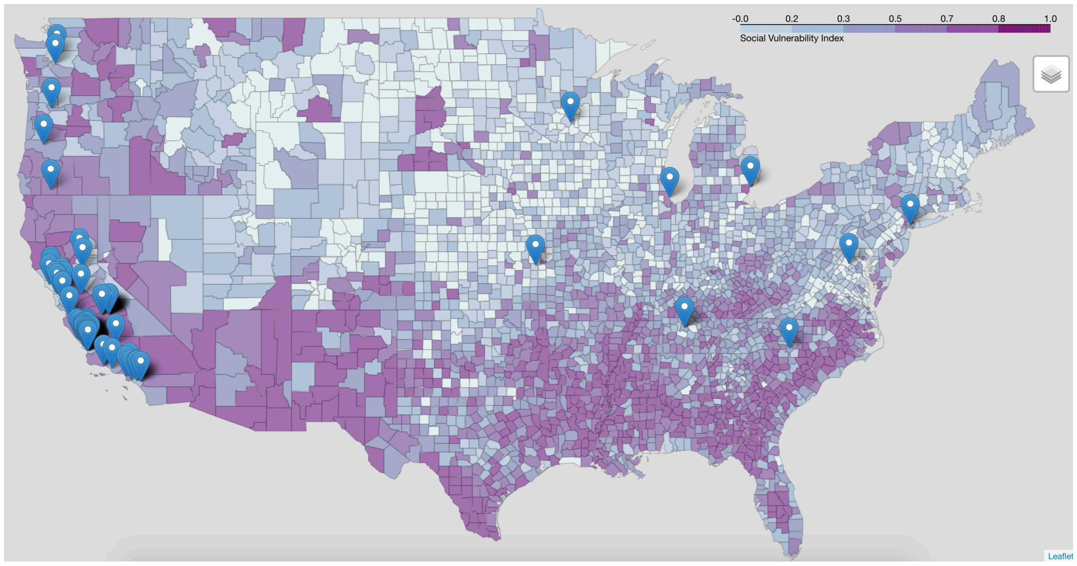

Figure 6: Twitter locations

5.2.2 Parameter Selection

Before running the model, parameters must be chosen for the number of influencers and, if using a constant threshold, the level to set for each node. For the particle swarm algorithm in the following examples, the swarm size is set to 100 and the number of generations to 10. Figure 7 below shows how influence changes as the percent of influencers increases. This follows a fairly consistent linear relationship that as the percent of influencers increases, the level of influence also increases. For the following plots and analysis, the percent of influencers will be set to 10%.

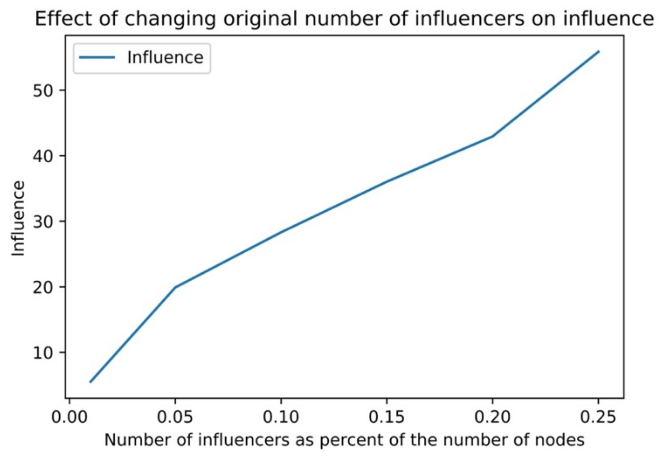

Figure 7: Altering number of influencers

The linear threshold model calls for each node to have its own random threshold value. However, it can be interesting to see how changing this threshold value affects the influence score. Figure 8 shows this change below. A significant drop occurs around 0.3, where at this point, it becomes difficult for the propagation probabilities to exceed the threshold limits. Nodes with thresholds above this limit are less likely to be reached.

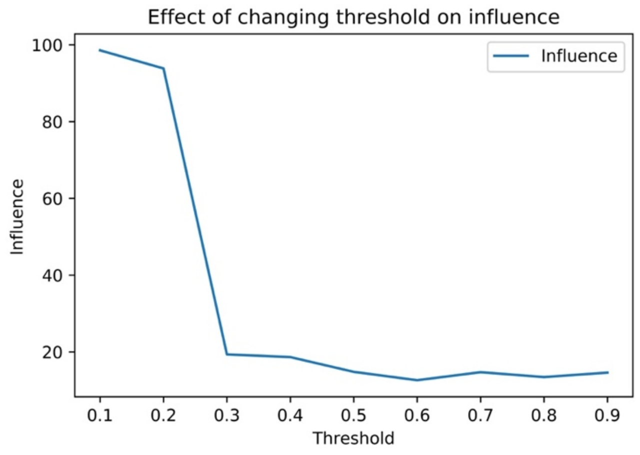

Figure 8: Altering threshold limits

5.2.3 Results

Several different methods exist to disrupt the network—based on influencers, vulnerability, and spatial locations. These different types of disruptions can vary the influence score in the network. Running the single-level particle swarm optimization before a disruption produces a score of 29.787. Each of the following plots illustrates the influence score after the disruption takes place. The first disruption type removes the more vulnerable nodes from the network. Figure 9 displays the resulting influence scores when a percentage of these nodes are removed. Removing nodes will, of course, lower the influence in the network. However, influence does not seem to drop quickly. One reason for this may stem from the high connectivity of San Luis Obispo and its low vulnerability. The nodes in the inland areas are removed but the influence along the coast remains.

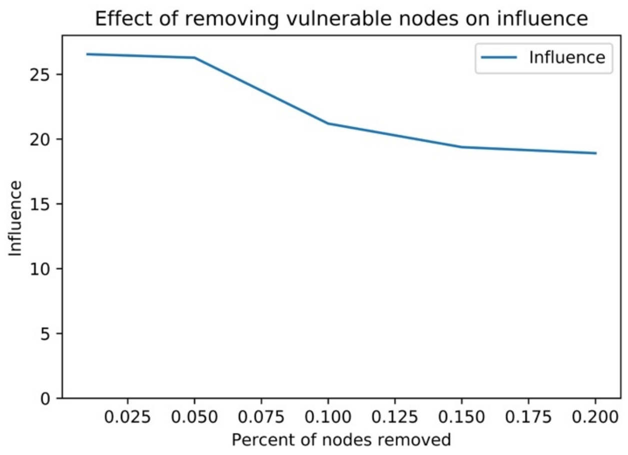

Figure 9: Disruption 1 - Affecting vulnerability

Additionally, disruptions can occur spatially. Three different areas were chosen for disruption, and the nodes in these areas were removed from the network. Figure 10 below shows each of the three boxes. The first centers around San Luis Obispo and the last extends to Los Angeles. Each of the three boxes disrupts 76, 119, and 131 nodes, respectively.

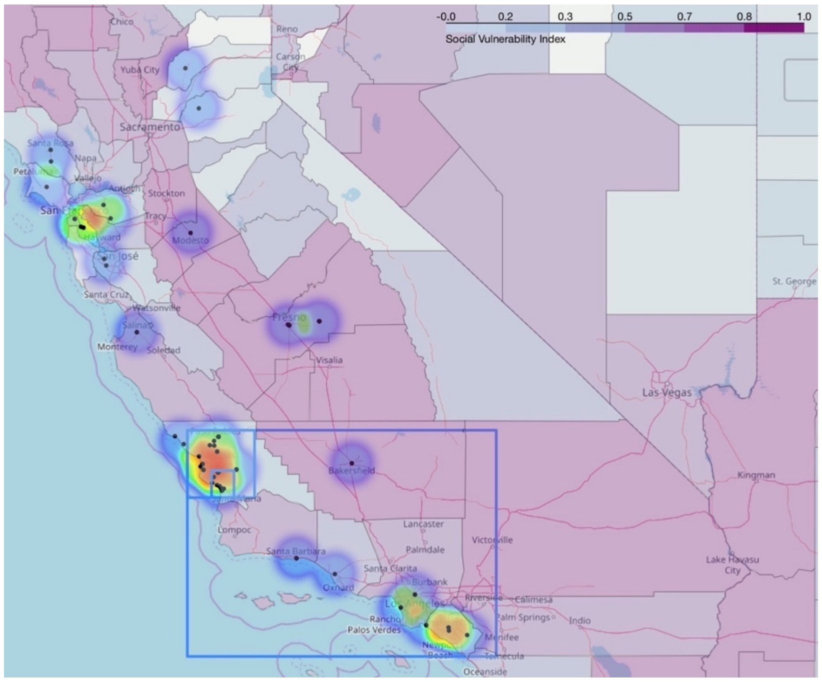

Figure 10: Spatial disruption map

Because of the densely packed users in these areas, the influence scores drop significantly after the first disruption. This region is highly susceptible to disruption and can severely impact influence in the rest of the network. Figure 11 shows the resulting scores.

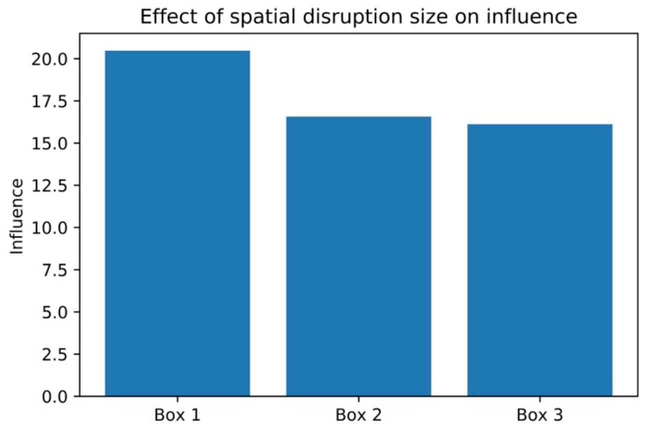

Figure 11: Disruption 2 - Affecting area

The last type of disruption explores removing the first set of influencers to disrupt the network. This problem requires the bi-level formulation as the result after the disruption depends on the influence from before the disruption. Because of the size of this problem, the algorithm only may return a solution that is close to optimal. Figure 12 represents the vulnerability of each of the four runs, resulting in an average score of 7.614. The maximum of each line holds the before-disruption score and falls to the after-disruption score. Run 4, while having a high influence score, also has the largest vulnerability. Run 2 illustrates the opposite case, where each run has a low influence score, but the vulnerability is minimized. A compromise between these, run 1, was chosen for the heat map analysis.

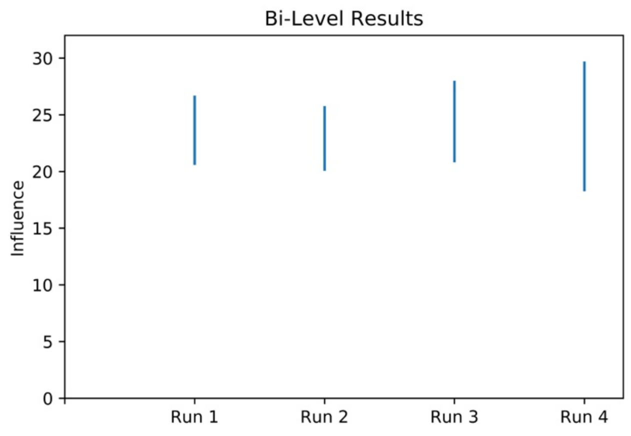

Figure 12: Disruption 3 - Affecting influencers

As Figure 13 shows, certain nodes were chosen as repeat influencers in all four of the runs. These are nodes 70 and 77 (while 13, 17, 68, and 127 occur three times) in the left plot. Node 43 in the right plot occurs three times. Because these nodes appear multiple times as influencers, they are likely important to spreading influence throughout the network.

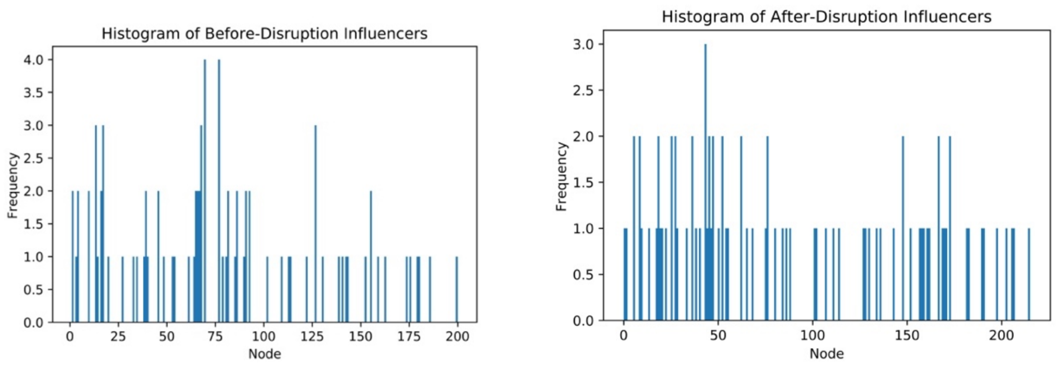

Figure 13: Repeat influencers

Figure 14 illustrates the spread of influence throughout the region. The results before the disruption are shown on the left, while the results afterward are on the right. Each point represents a user in the network, while the heat map demonstrates those users that are influenced. As the map shows, users in the San Luis Obispo remain the most influenced, while users in the other regions of California become less affected. Once again, the tight connectivity of San Luis Obispo is shown to help maintain influence in the community.

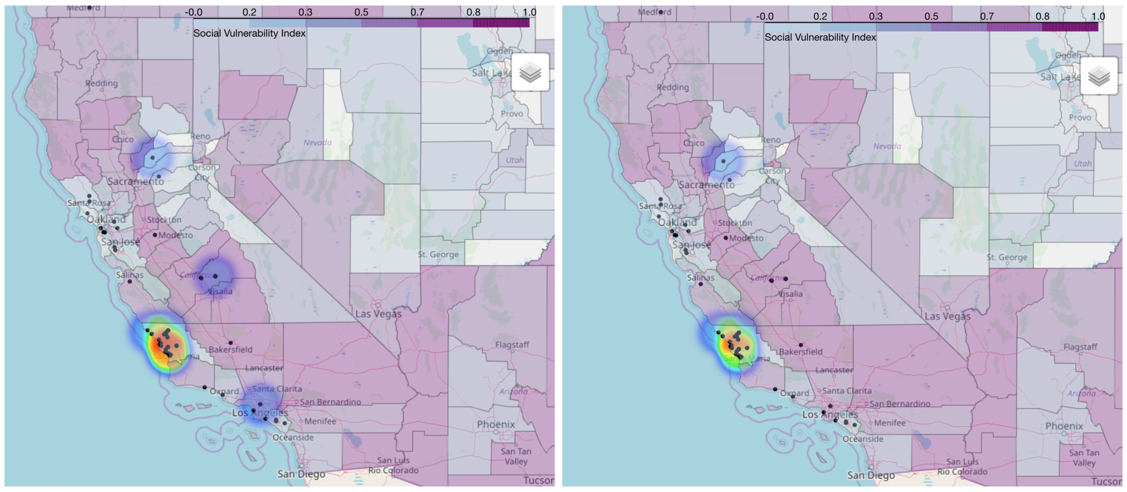

Figure 14: Heatmap of influencers

As the results have indicated, the spread of influence can be modeled to choose influencers that maximize this range and minimize the vulnerability. The high connectivity of San Luis Obispo made it a key target for the algorithm because it could easily activate a large number of nodes. As Figure 14 above shows, nodes in more vulnerable regions are still a target to reach before the disruption, but this becomes less significant after the disruption. These nodes could be further prioritized by applying a scaling factor to further separate the more vulnerable nodes and incentivize the algorithm to activate these nodes.

5.2.4 A Note on Complexity

In the sample problem with 16 nodes, the particle swarm algorithm was easily able to solve and find the optimal solution in the solution space with 100 particles. However, as the problem grows to include 216 nodes and 3,507 edges, it becomes much more difficult to find the optimal solution. Using 100 particles and 10 generations produce variable results and, while finding a good answer, may not find the best. Increasing the swarm size and number of generations, however, produces a much more time-intensive problem. The bi-level algorithm grows exponentially dependent on the product of the swarm size and number of generations. When 1,000 particles and 50 generations are used in the single-level particle swarm optimization (neglecting minimization of vulnerability), higher influence scores can be obtained. For example, a before-disruption score of 33.638 can be found (although this produces an after-disruption score of 24.370—not much higher than before). This is also a higher vulnerability score than previously obtained. Increasing these parameters in the bi-level problem may produce higher influence scores and reduced vulnerability; however, it comes at the cost of time.

Chapter 6.0 Conclusions

6.1 Summary

As technology continues to make society more connected, these tools can be utilized to spread information efficiently and quickly when needed. Natural disasters and other disruptions can severely impact society and leave people without the resources they need. By taking advantage of the technology in place, the extent of these disasters can hopefully be minimized as citizens know updates as they occur, receive information on evacuation areas, and find places to receive assistance. Identifying these influencers can lead to improved communication between members of the community and assist in spreading critical information when needed.

This thesis proposes a model that incorporates social vulnerability scores into influence and optimization models. Additionally, an algorithm for solving the bi-level problem to minimize vulnerability is proposed and tested on a Twitter network. As nodes are more connected, information is able to reach them easier and makes them more resistant to disruption.

6.2 Future Work

This work can be improved and expanded by analyzing a larger Twitter network that includes multiple communities and measures how influence travels within and among these communities. Additionally, increased knowledge into the vulnerability of smaller regions within the community can help identify the more vulnerable population in these areas. For example, susceptible populations in San Luis Obispo could be discovered and prioritized for influence in this highly connected area. Likewise, individuals that are currently influencing these vulnerable populations could be disrupted to see how this affects influence through the network.

This model also depends on a deterministic state where the structure of the network, the propagation probabilities, and the threshold limits are known. Incorporating uncertainty into these parameters more accurately reflects real scenarios and can be generalized for several networks. Similarly, the threshold could be included in the objective function to determine an optimal limit that allows influence to pass through neighbors. Finding ways to incentivize the threshold also can be beneficial to lowering it and activating more users in the community.

Further research could also extend to maximizing recovery of the system after the disruption, using the time to recover and amount recovered as indicators. Incorporating this with vulnerability leads to improving the resilience of these communities to disruption.

References

Bagrow, J. P. (2018). Information spreading during emergencies and anomalous events. *Complex Spreading Phenomena in Social Systems*, 269-286.

Barabási, A. L., & Albert, R. (1999). Emergence of scaling in random networks. *science*, *286*(5439), 509-512.

Borgs, C., Brautbar, M., Chayes, J., & Lucier, B. (2014, January). Maximizing social influence in nearly optimal time. *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, 946-957.

Centers for Disease Control and Prevention/ Agency for Toxic Substances and Disease Registry/ Geospatial Research, Analysis, and Services Program. Social Vulnerability Index 2016 Database US. (2016, May). Retrieved from https://svi.cdc.gov/data-and-tools-download.html.

Check if a geopoint with latitude and longitude is within a shapefile. (2015). Retrieved from https://stackoverflow.com/questions/7861196/check- if-a-geopoint-with-latitude-and-longitude-is-within-a-shapefile

Chen, W., Wang, Y., & Yang, S. (2009, June). Efficient influence maximization in social networks. *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, 199-208.

Cutter, S.L., Ash, K.D., & Emrich, C.T. (2014). The Geographies of Community Disaster Resilience. *Global Environmental Change*, *29*, 65-77.

Cutter, S.L., L. Barnes, M. Berry, C. Burton, E. Evans, E. Tate, and J. Webb. (2008). A Place-based Model for Understanding Community Resilience to Natural Disasters. *Global Environmental Change*, *18*(4), 598-606.

Cutter, S. L., Boruff, B. J., & Shirley, W. L. (2003). Social Vulnerability to Environmental Hazards*. *Social Science Quarterly, 84*(2), 242-261. doi:10.1111/1540-6237.8402002

Dailey, D. & Starbird, K. (2014). Journalists as Crowdsourcerers: Responding to Crisis by Reporting with a Crowd. *Journal of Computer Supported Cooperative Work*, *23*(4-6), 445-481.

Dodds, P. S. (2018). A simple person’s approach to understanding the contagion condition for spreading processes on generalized random networks. *Complex Spreading Phenomena in Social Systems*, 27-45.

Dodds, P. S. (2018). Slightly Generalized Contagion: Unifying Simple Models of Biological and Social Spreading. *Complex Spreading Phenomena in Social Systems*, 67-80.

Ferrara, E. (2018). Measuring social spam and the effect of bots on information diffusion in social media. *Complex Spreading Phenomena in Social Systems*, 229-255.

Flanagan, B.E., Gregory, E. W., Hallisey, E.J., Heitgerd, J. L., & Lewis, B. (2011). A Social Vulnerability Index for Disaster Management. *Journal of Homeland Security and Emergency Management*, *8*(1).

Gao, H., Barbier, G., Goolsby, R., & Zeng, D. (2011). Harnessing the Crowdsourcing Power of Social Media for Disaster Relief. *IEEE Intelligent Systems*, *26*(3), 10-14.

Gao, Y., Zhang, G., Lu, J., & Wee, H. M. (2011). Particle swarm optimization for bi-level pricing problems in supply chains. *Journal of Global Optimization*, *51*(2), 245-254.

Goel, S., Anderson, A., Hofman, J., & Watts, D. J. (2015). The structural virality of online diffusion. Management Science, *62*(1), 180-196.

Goyal, A., Lu, W., & Lakshmanan, L. V. (2011, March). Celf++: optimizing the greedy algorithm for influence maximization in social networks. *Proceedings of the 20th international conference companion on World wide web*, 47-48.

Hamill, J. T., Deckro, R. F., Wiley, V. D., & Renfro, R. S. (2007). Gains, losses and thresholds of influence in social networks. *International Journal of Operational Research*, *2*(4), 357-379.

He, J. L., Fu, Y., & Chen, D. B. (2015). A novel top-k strategy for influence maximization in complex networks with community structure. *PloS one*, *10*(12).

Hui, P. M., Weng, L., Shirazi, A. S., Ahn, Y. Y., & Menczer, F. (2018). Scalable Detection of Viral Memes from Diffusion Patterns. *Complex Spreading Phenomena in Social Systems*, 197-211.

Kempe, D., Kleinberg, J., & Tardos, É. (2003, August). Maximizing the spread of influence through a social network. *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 137-146.

Kennedy, J., & Eberhart, R.C. (1995). Particle swarm optimization. *Proceedings of IEEE International Conference on Neural Networks IV, 1000*.

Kennedy, J., & Eberhart, R. C. (1997, October). A discrete binary version of the particle swarm algorithm. *Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation., 1997 IEEE International Conference, 5,* 4104-4108.

Kuo, R. J., & Han, Y. S. (2011). A hybrid of genetic algorithm and particle swarm optimization for solving bi-level linear programming problem–A case study on supply chain model. *Applied Mathematical Modelling*, *35*(8), 3905-3917.

Kuo, R. J., & Huang, C. C. (2009). Application of particle swarm optimization algorithm for solving bi-level linear programming problem. *Computers & Mathematics with Applications*, *58*(4), 678-685.

Kwak, H., Lee, C., Park, H., & Moon, S. (2010, April). What is Twitter, a social network or a news media?. *Proceedings of the 19th international conference on World wide web*, 591-600.

Leskovec, J., Adamic, L. A., & Huberman, B. A. (2007). The dynamics of viral marketing. *ACM Transactions on the Web (TWEB)*, *1*(1), 5.

Leskovec, J., Krause, A., Guestrin, C., Faloutsos, C., VanBriesen, J., & Glance, N. (2007, August). Cost-effective outbreak detection in networks. *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 420-429.

Leskovec, J., & Krevl, A. (2014, June). SNAP Datasets: Stanford Large Network Dataset Collection. Retrieved from http://snap.stanford.edu/data/ego-Twitter.html

Li, X., Tian, P., & Min, X. (2006, June). A hierarchical particle swarm optimization for solving bilevel programming problems. *International Conference on Artificial Intelligence and Soft Computing*, 1169-1178. Springer, Berlin, Heidelberg.

Lü, L., Chen, D., Ren, X. L., Zhang, Q. M., Zhang, Y. C., & Zhou, T. (2016). Vital nodes identification in complex networks. *Physics Reports*, *650*, 1-63.

Morone, F., & Makse, H. A. (2015). Influence maximization in complex networks through optimal percolation. *Nature*, *524*(7563), 65.

National Academies of Science. (2012). *Disaster Resilience: A National Imperative*. Washington, DC: National Academies Press.

National Institute of Standards and Technology. (2015). *Community Resilience Planning Guide for Buildings and Infrastructure Systems*.

Parsopoulos, K. E., & Vrahatis, M. N. (2002, March). Particle swarm optimization method in multiobjective problems. *Proceedings of the 2002 ACM symposium on Applied computing*, 603-607.

Patel, A. (2018, February 27). How To Get Twitter Follower Data Using Python And Tweepy. Retrieved from https://labsblog.f-secure.com/2018/02/27/how-to-get-twitter-follower-data-using-python-and-tweepy/

Pei, S., Morone, F., & Makse, H. A. (2018). Theories for influencer identification in complex networks. *Complex Spreading Phenomena in Social Systems*, 125-148.

Ramirez-Marquez, J. E., Rocco, C. M., Barker, K., & Moronta, J. (2018). Quantifying the resilience of community structures in networks. *Reliability Engineering & System Safety*, *169*, 466-474.

Sinha, A., Malo, P., & Deb, K. (2018). A review on bilevel optimization: from classical to evolutionary approaches and applications. *IEEE Transactions on Evolutionary Computation*, *22*(2), 276-295.

Teng, X., Pei, S., Morone, F., & Makse, H. A. (2016). Collective influence of multiple spreaders evaluated by tracing real information flow in large-scale social networks. *Scientific reports*, *6*, 36043.

Wei, H., Pan, Z., Hu, G., Zhang, L., Yang, H., Li, X., & Zhou, X. (2018). Identifying influential nodes based on network representation learning in complex networks. *PloS one*, *13*(7).

Weng, L., Karsai, M., Perra, N., Menczer, F., & Flammini, A. (2018). Attention on Weak Ties in Social and Communication Networks. *Complex Spreading Phenomena in Social Systems*, 213-228.

Zant, S. E., Jaffrès-Runser, K., Frahm, K. M., & Shepelyansky, D. L. (2018). Interactions and influence of world painters from the reduced Google matrix of Wikipedia networks. *arXiv preprint arXiv:1807.01255*.

Zhang, X., Zhu, J., Wang, Q., & Zhao, H. (2013). Identifying influential nodes in complex networks with community structure. *Knowledge-Based Systems*, *42*, 74-84.

Zook, M., Graham, M., Shelton, T., & Gorman, S. (2010). Volunteered Geographic Information and Crowdsourcing Disaster Relief: A Case Study of the Haitian earthquake. *World Medical & Health Policy*, *2*(2), 7-33.

Appendix A: AMPL Code – Model File

```
reset;
option solver cplex;
option randseed 19;

# Groups of people, influencers, and connections between them
set PEOPLE;
set INFLUENCERS;
set ARCS within {INFLUENCERS, PEOPLE};

# Parameters
#param prop {ARCS} := Uniform01(); # Assign propagation probability to the arcs
param prop{ARCS};                # Assign propagation probability to the arcs
param t{PEOPLE};                 # Threshold for each person
param M := 1000;                 # For finding maximum
param sv{PEOPLE};                # Social vulnerability for each person

# Variables
var y{INFLUENCERS} binary;       # Starting influencers
var s{PEOPLE} binary;            # Ending influenced people
var u{PEOPLE};                   # Hold summation of arcs

var x{PEOPLE};                   # Used to find maximum between start and end
var b{PEOPLE} binary;            # Used to find maximum


# Optimize spread: Influenced node * social vulnerability score
maximize spread: sum{j in PEOPLE} x[j] * sv[j];

# Find sum of arcs of active nodes
subject to active {j in PEOPLE}: u[j] = sum{(i,j) in ARCS} y[i] * prop[i,j];

# Determine if node meets threshold limit
subject to threshold {j in PEOPLE}: t[j]*s[j] <= u[j];

# Determine number of starting influencers
subject to limit: sum {i in INFLUENCERS} y[i] <= 2;

# Constraints to find the ending maximum influenced
subject to max1 {j in PEOPLE}: x[j] >= s[j];
subject to max2 {j in PEOPLE}: x[j] >= y[j];
subject to max3 {j in PEOPLE}: x[j] <= s[j] + M*b[j];
subject to max4 {j in PEOPLE}: x[j] <= y[j] + M*(1-b[j]);

# Read in data and solve
data network.dat;
solve;
display y, x;
```

Appendix B: AMPL Code – Data File

```
set PEOPLE := 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15;
set INFLUENCERS := 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15;

set ARCS := (0, 1) (0, 2) (0, 3) (0, 4) (1, 5) (1, 6) (1, 8) (2, 5) (2, 6)
            (2, 7) (3, 6) (3, 7) (4, 6) (4, 7) (4, 11) (5, 8) (5, 9) (6, 9)
            (6, 10) (7, 10) (7, 11) (8, 12) (8, 13) (9, 12) (9, 13) (10, 13)
            (10, 14) (11, 13) (11, 14) (12, 15) (13, 15) (14, 15);

param sv :=
0 0.6957279716157805
1 0.9160960662545845
2 0.8164649890317078
3 0.028749761066347967
4 0.028041893921005423
5 0.7438086041530414
6 0.28855253318726903
7 0.25719517445052975
8 0.8508915655407895
9 0.650396427594281
10 0.49948586935849015
11 0.30653814079710606
12 0.6409787445759747
13 0.37651206017437044
14 0.8145728613758114
15 0.9453578017199536;

param t :=
0 0.4706810530434994
1 0.6979085881001218
2 0.23278514744401813
3 0.17396749189309824
4 0.9033647822607421
5 0.5021209861281979
6 0.5790873096413065
7 0.6273477338307205
8 0.2654174490273189
9 0.21802956966098364
10 0.38932199142757884
11 0.026846232825811223
12 0.6555561416786693
13 0.8620716020188668
14 0.0622859279293565
15 0.05652409688228199;
```

```
param prop :=
[0, 1] 0.895348163654844
[0, 2] 0.4212214974801205
[0, 3] 0.7958670533798993
[0, 4] 0.7746444551066404
[1, 5] 0.3246788210216138
[1, 6] 0.06662423643153312
[1, 8] 0.15668111503910137
[2, 5] 0.07525015228085663
[2, 6] 0.0848674226258275
[2, 7] 0.3211691249636829
[3, 6] 0.14562662430172618
[3, 7] 0.057795688790242584
[4, 6] 0.2090475007813697
[4, 7] 0.23307272605709883
[4, 11] 0.3665782545329316
[5, 8] 0.2757846236180138
[5, 9] 0.21682203466433164
[6, 9] 0.20994751324388922
[6, 10] 0.4599796758880993
[7, 10] 0.0033687985167396017
[7, 11] 0.1639780615694167
[8, 12] 0.0545023404376328
[8, 13] 0.20001477431154954
[9, 12] 0.10223165955573477
[9, 13] 0.11085158734693767
[10, 13] 0.1264766357720927
[10, 14] 0.04351434104468721
[11, 13] 0.16547335532032353
[11, 14] 0.1081189592616596
[12, 15] 0.1269944909233354
[13, 15] 0.10900143406071675
[14, 15] 0.10004572201850995;
```

Appendix C: Python code to create graph

```
import tweepy
from tweepy import OAuthHandler
import json
import time

import sys
import os
import io
import re
import pandas as pd
import random
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from random import Random

""" Useful Functions """
# Code adapted from https://labsblog.f-secure.com/2018/02/27/how-to-
get-twitter-follower-data-using-python-and-tweepy/
# Patel (2018)

# Save json files
def save_json(variable, filename):
    with io.open(filename, "w", encoding="utf-8") as f:
        f.write(str(json.dumps(variable, indent=4,
ensure_ascii=False)))

# Load saved json files
def load_json(filename):
    ret = None
    if os.path.exists(filename):
        try:
            with io.open(filename, "r", encoding="utf-8") as f:
                ret = json.load(f)
        except:
            pass
    return ret

# Save result of functions as json files
def try_load_or_process(filename, processor_fn, function_arg):
    load_fn = None
    save_fn = None
    if filename.endswith("json"):
        load_fn = load_json
        save_fn = save_json
    else:
        load_fn = load_bin
        save_fn = save_bin
    if os.path.exists(filename):
        print("Loading " + filename)
        return load_fn(filename)
    else:
        ret = processor_fn(function_arg)
        print("Saving " + filename)
        save_fn(ret, filename)
        return ret

# Get user data and latest tweet from user id
def get_user_objects(follower_ids):
    batch_len = 100
    num_batches = len(follower_ids) / 100
    batches = (follower_ids[i:i+batch_len] for i in range(0,
len(follower_ids), batch_len))
    all_data = []
    for batch_count, batch in enumerate(batches):
        sys.stdout.write("\r")
        sys.stdout.flush()
        sys.stdout.write("Fetching batch: " + str(batch_count) + "/" +
str(num_batches))
        sys.stdout.flush()
        try:
            users_list = api.lookup_users(user_ids=batch)
        except tweepy.RateLimitError:
            print ('Rate limited. Sleeping for 15 minutes.')
            time.sleep(15 * 60 + 15)
            continue
        users_json = (map(lambda t: t._json, users_list))
        all_data += users_json
    return all_data

# Get tweet from the tweet id
def get_geo_objects(tweet_ids):
    batch_len = 100
    num_batches = len(tweet_ids) / 100
    batches = (tweet_ids[i:i+batch_len] for i in range(0,
len(tweet_ids), batch_len))
    all_data = []
    for batch_count, batch in enumerate(batches):
        sys.stdout.write("\r")
        sys.stdout.flush()
        sys.stdout.write("Fetching batch: " + str(batch_count) + "/" +
str(num_batches))
        sys.stdout.flush()

        try:
            users_list = api.statuses_lookup(batch)
        except tweepy.RateLimitError:
            print ('Rate limited. Sleeping for 15 minutes.')
            time.sleep(15 * 60 + 15)
            continue

        users_json = (map(lambda t: t._json, users_list))
        all_data += users_json
    return all_data

# Return the geo data for each user - works with return of geo_objects
def clean(tweet_data):
    entry = {}
    for tweet in tweet_data:
        #id_str = tweet["user"]["id_str"]
        try:
            id_str = tweet["id_str"]

            if tweet['status']['place'] is not None:
                if tweet['status']['place']['bounding_box'] is not
None:
                    geo =
tweet['status']['place']['bounding_box']['coordinates'][0][0]
                    entry[id_str] = geo
                else:
                    entry[id_str] = None
            else:
                if tweet['location'] is not None:
                    entry[id_str] = tweet["location"]
                else:
                    entry[id_str] = None
        except KeyError:
            print(id_str)
    return entry

# Return list of follower ids
def get_follower_ids(target):
    return api.followers_ids(target)

# Create edges based on return of get_follower_ids
def create_edges(users):
    for target in users:
        print('Pulling followers for ', target)
        edges = []
        filename = "dbo/" + str(target) + "_edges.json"
        try:
            followers = get_follower_ids(target)
            for f in followers:
                edges.append([int(target), f])
            save_json(edges, filename)
        except tweepy.RateLimitError:
            print ('Rate limited. Sleeping for 15 minutes.')
            time.sleep(15 * 60 + 15)
            continue
        except tweepy.TweepError:
            pass

############################################

""" Data pull """

consumer_key = ""
consumer_secret = ""
access_token = ""
access_secret = ""

auth = OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_secret)

api = tweepy.API(auth)

target = "13179562"
filename = target + "_geo.json"
tweets  = pd.read_csv('data/13179562.edges', header=None, sep='
').values

g = nx.DiGraph()
g.add_edges_from(tweets)

nodeList = []
for n in g.nodes():
    nodeList.append(n)

geo_objects = try_load_or_process(filename, get_user_objects, nodeList)
cleaned = clean(geo_objects)
save_json(cleaned, '13179562_clean.json')


############################################

# Run Google API to get longitude and latitude for string locations
from geopy.geocoders import GoogleV3
import geopy
geolocator = GoogleV3(api_key = '’)
```

```
for k, v in cleaned.items():
    if isinstance(v, list) and v is not "":
        temp = v[0]
        v[0] = v[1]
        v[1] = temp

save_json(cleaned, '13179562_clean.json')
# some edits
cleaned = load_json('13179562_clean.json')

coordinates = {}
for k, v in cleaned.items():
    if isinstance(v, str) and v is not "":
        try:
            coordinates[k] = geolocator.geocode(v, timeout=3)
        except geopy.exc.GeocoderQueryError:
            print(k)
            pass
        except geopy.exc.GeocoderTimedOut:
            print(k)
            pass

filename = target + "_coord.json"

coord = {}
for k, v in cleaned.items():
    if isinstance(v, str) and v is not "":
        coord[k] = coordinates[k][1]
    else:
        coord[k] = cleaned[k]

save_json(coord, '13179562_coord.json')


############################################

""" ogr """
# Determine the social vulnerability factor of each user based on
longitude and latitude
# Code adapted from https://stackoverflow.com/questions/7861196/check-
if-a-geopoint-with-latitude-and-longitude-is-within-a-shapefile
# Check if a geopoint with latitude and longitude is within a
shapefile. (2015)

import ogr
from IPython import embed
import sys

drv = ogr.GetDriverByName('ESRI Shapefile') #We will load a shape file
ds_in = drv.Open("SVI2016_US_COUNTY/SVI2016_US_COUNTY.shp")    #Get the
contents of the shape file
lyr_in = ds_in.GetLayer(0)    #Get the shape file's first layer

#Put the title of the field you are interested in here
idx_reg = lyr_in.GetLayerDefn().GetFieldIndex("RPL_THEMES")

#If the latitude/longitude we're going to use is not in the projection
#of the shapefile, then we will get erroneous results.
#The following assumes that the latitude longitude is in WGS84
#This is identified by the number "4326", as in "EPSG:4326"
#We will create a transformation between this and the shapefile's
#project, whatever it may be
```

```
geo_ref = lyr_in.GetSpatialRef()
point_ref=ogr.osr.SpatialReference()
point_ref.ImportFromEPSG(4326)
ctran=ogr.osr.CoordinateTransformation(point_ref,geo_ref)

def lookup_sv(d):
    for k, v in d.items():
        if v is not "":
            lat = v[0]
            lon = v[1]

            #Transform incoming longitude/latitude to the shapefile's
projection
            [lon,lat,z]=ctran.TransformPoint(lon,lat)

            #Create a point
            pt = ogr.Geometry(ogr.wkbPoint)
            pt.SetPoint_2D(0, lon, lat)

            #Set up a spatial filter such that the only features we see
when we
            #loop through "lyr_in" are those which overlap the point
defined above
            lyr_in.SetSpatialFilter(pt)

            #Loop through the overlapped features and display the field
of interest
            for feat_in in lyr_in:
                d[k].append(feat_in.GetFieldAsDouble(idx_reg))
    return d

lookup_sv(coord)
save_json(coord, "13179562_sv.json")


############################################
# Fill in social vulnerability to nodes
sv = load_json("13179562_sv.json")

sv_val = []
for k, v in sv.items():
    if len(v) == 3:
        sv_val.append(v[2])

sv_vals = np.array(sv_val)
mean = sv_vals.mean()

# Put in mean if no location for user
for n in g.nodes():
    if str(n) not in sv:
        g.nodes[n]['sv'] = mean
    elif len(sv[str(n)]) < 3:
        g.nodes[n]['sv'] = mean
    else:
        g.nodes[n]['sv'] = sv[str(n)][2]

# Make sure it worked
nx.get_node_attributes(g, 'sv')

# Change twitter ids to integers to store in lists easier
g = nx.convert_node_labels_to_integers(g, label_attribute='name')
nx.get_node_attributes(g, 'name')

# Assign propagation probability to nodes
```

```
degrees = [d for n, d in g.in_degree()]
p = np.zeros((nodes,nodes))
for e in g.edges():
    n = e[1]
    if degrees[n] > 0:
        p[e[0],e[1]] = random.uniform(0,1/degrees[n])
    else:
        p[e[0],e[1]] = random.random()

# Make sure sum of probabilities <= 1
psum = p.sum(axis=0)

# Assign to edges
for e in g.edges():
    g.edges[e]['p'] = p[e[0],e[1]]

# Assign threshold to nodes
for n in g.nodes():
    g.nodes[n]['t'] = random.random()

# Assign coordinates to the nodes
for n in g.nodes():
    x = str(name[n])
    if x in coord:
        g.nodes[n]['loc'] = coord[x]
    else:
        g.nodes[n]['loc'] = ''

# Save network
nx.write_gpickle(g, "13179562_network.gpickle")
```

Appendix D: Python code for finding influencers

```
import pandas as pd
import random
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib import collections  as mc
from random import Random
import sys
import json
import os
import io
import re
import collections
import pylab as pl
import folium
from folium.plugins import HeatMap
from shapely.geometry import Point
from shapely.geometry.polygon import Polygon

# Read in graph
g = nx.read_gpickle('13179562_network.gpickle')

# Starting parameters
degree_sequence = sorted([d for n, d in g.degree()], reverse=True)
med_degree = np.median(degree_sequence)
nodes = g.number_of_nodes()

seed = 5113
myPRNG = Random(seed)

# Plot histogram of degree
plt.hist(degree_sequence)
plt.xlabel("Degree")
plt.ylabel("Frequency")
plt.title("Degree distribution")
plt.savefig('savefig/degree.jpg',dpi=1000)

############################################
# Run the linear threshold model
def ltm(x, graph):
    """
    :param x: the solution to run the linear threshold model on
    :param graph: the graph to test the solution on
    :return: a list - the influenced nodes
    """
    # Copy influence list
    influenced = x[:]

    # Get thresholds
    dt = nx.get_node_attributes(graph,'t')
    sort = collections.OrderedDict(sorted(dt.items()))
    threshold = list(sort.values())

    level = np.zeros((nodes,nodes))

    # Find the propagation of active nodes
    for i in range(0,len(influenced)):
        if influenced[i] == 1:
            for n in graph.neighbors(i):
                level[i,n] = graph.edges[(i,n)]['p'] * influenced[i]

    # Find sum into each node
    level_sum = level.sum(axis=0)

    # If above threshold, mark as active
    for i in range(0,len(level_sum)):
        if level_sum[i] >= threshold[i] and influenced[i] == 0:
            influenced[i] = 1

    # If no updates made, return influenced list
    if np.sum(influenced) == np.sum(x):
        return influenced
    # Otherwise run again
    else:
        return ltm(influenced,graph)

# Evaluate the solution
def evaluate(x, graph):
    """
    :param x: the solution to be evaluated
    :param graph: the graph to run the solution on
    :return: the solution's score and the solution
    """
    sv = nx.get_node_attributes(graph, 'sv')
    sort = collections.OrderedDict(sorted(sv.items()))
    sv= list(sort.values())

    # Run linear threshold model
    sol = ltm(x, graph)

    # Calculate score
    score = 0
    for i in range(0, len(sol)):
        score = score + (sol[i] * sv[i])

    return score, sol


""" Particle Swarm Optimization """
# Function to update the velocity
def updateVelocity(vel,pos,pBest,gBest,inertia):
    """
    Update the velocity
    :param: vel: velocity of the particles
    :param: pos: position of the particles
    :param: pBest: the best position for an individual particle
    :param: gBest: the best position of all particles
    :param: inertia: the amount of weight the previous velocity will
have
    :return: a list of the particles' velocities
    """

    cognitive = 1
    social = 3
    vmax = 10

    cogVel = []
    socVel = []

    for i in range(nodes):
        r1 = myPRNG.random()
        r2 = myPRNG.random()

        cogVel.append(cognitive * r1 * (pBest[i] - pos[i]))
        socVel.append(social * r2 * (gBest[i] - pos[i]))
        vel[i] = inertia * vel[i] + cogVel[i] + socVel[i]
        if vel[i] > vmax:
            vel[i] = vmax

        if vel[i] < -vmax:
            vel[i] = -vmax

    return vel

# Function to update the position
def updatePosition(pos,vel):
    """
    Update the position
    :param: pos: position of the particles
    :param: vel: velocity of the particles
    :return: a list of the particles' new positions
    """

    update = []
    num = int(ni * nodes)
    for i in range(nodes):
        update.append(pos[i] + vel[i])

    update = np.array(update)
    updateList = np.argsort(update)[-num:]

        # Make sure update is within feasible space
    for i in range(nodes):
        if i in updateList:
            pos[i] = 1
        else:
            pos[i] = 0

    return pos

def inertiaWeight(j,Generations):
    """
    Find the inertia (linearly decreasing)
    :param: j: the current iteration
    :param: Generations: how many times the optimization will run
    :return: float of inertia weight
    """
    return j/Generations

# Intialize swarm
def initializeSwarm(nodes, swarmSize, graph):
    """
    Create anititial swarm
    :param: nodes: the number of nodes in the problem
    :param: swarmSize: the size of the swarm
    :return: the position, velocity, current value, pBest, and pBestVal
of swarm
    """
    #the swarm will be represented as a list of positions, velocities,
values, pbest, and pbest values

    pos = [[] for _ in range(swarmSize)]      #position of particles --
will be a list of lists
    vel = [[] for _ in range(swarmSize)]      #velocity of particles --
will be a list of lists

    pos2 = [[] for _ in range(swarmSize)]      #position of particles -
- will be a list of lists
```

```
    vel2 = [[] for _ in range(swarmSize)]      #velocity of particles -
- will be a list of lists

    curValue = [] #value of current position  -- will be a list of real
values
    pBest = []    #particles' best historical position -- will be a
list of lists
    pBestVal = [] #value of pbest position  -- will be a list of real
values

    #initialize the swarm randomly
    for i in range(swarmSize):
          s = []
          for j in range(0,int(ni * nodes)):
              s.append(myPRNG.randint(0,nodes))
          for j in range(nodes):
                #if np.sum(pos[i]) < int(ni * nodes):
                if j in s:
                    #pos[i].append(myPRNG.randint(0,1))    #assign
random value between -500 and 500
                    pos[i].append(1)
                else:
                    pos[i].append(0)

                vel[i].append(myPRNG.uniform(-1,1))        #assign
random value between -1 and 1

          curValue.append(evaluate(pos[i], graph))   #evaluate the
current position

    pBest = pos[:]  # initialize pbest to the starting position
    pBestVal = curValue[:]  # initialize pbest to the starting position

    return pos, vel, curValue, pBest, pBestVal

# Particle Swarm Optimization
def PSO(nodes, swarmSize, Generations, graph):
    """
    Run the Particle Swarm Optimization
    :param: nodes: the number of nodes in the problem
    :param: swarmSize: the size of the swarm
    :param: Generations: the number of times to iterate through the
optimization
    :param: graph: the graph to run the optimization on
    :return: the best value and its score
    """

    # Initialize the swarm
    pos,vel,curValue,pBest,pBestVal = initializeSwarm(nodes, swarmSize,
graph)

    # Find best
    gBestIndex = pBestVal.index(max(pBestVal))
    gBest = pos[gBestIndex]
    gBestVal = max(pBestVal)

    # run the optimization
    for j in range(Generations):

        # Find pbest and gbest
        for k in range(swarmSize):
            curValue[k] = evaluate(pos[k], graph)

            if curValue[k] > pBestVal[k]:
                pBest[k] = pos[k]
                pBestVal[k] = curValue[k]

            if curValue[k] > gBestVal:
                gBest = pos[k]
                gBestVal = curValue[k]

        inertia = inertiaWeight(j,Generations)

        # Update position and velocity
        for k in range(swarmSize):
            vel[k] =
updateVelocity(vel[k],pos[k],pBest[k],gBest,inertia)
            pos[k] = updatePosition(pos[k],vel[k])

    return gBest, gBestVal

# Function to remove influencers
def removeInfluencers(graph, pos):
    """
    :param: graph: the graph to remove nodes from
    :param: pos: the nodes to remove
    :return: f: new graph
    """
    f = graph.copy()
    toRemove = []
    t = []
    for i in range(0, len(pos)):
        if pos[i] == 1:
            toRemove.append(i)
            t.append(graph.nodes[i]['t'])

    f.remove_nodes_from(toRemove)

    for i in range(0, len(toRemove)):
        n = toRemove[i]
        f.add_node(n)
        f.nodes[n]['sv'] = 0.0001
        f.nodes[n]['t'] = t[i]
    return f

# Function to remove vulnerable nodes
def vulnerable(graph, pct):
    """
    :param: graph: the graph to remove nodes from
    :param: pos: the nodes to remove
    :return: f: new graph
    """
    sv = nx.get_node_attributes(graph, 'sv')
    f = graph.copy()
    num = int(pct*nodes)

    v = np.array(list(sv.values()))
    vList = np.argsort(v)[-num:]

    t = []
    for i in vList:
        t.append(graph.nodes[i]['t'])

    f.remove_nodes_from(vList)

    for i in range(0, len(vList)):
        n = vList[i]
        f.add_edge(n,n)
```

```
            f.edges[n,n]['p'] = 0
            f.nodes[n]['sv'] = 0.0001
            f.nodes[n]['t'] = t[i]

        return f

# Function to remove nodes spatially
def boundary(graph, corners):
    """
    :param: graph: the graph to remove nodes from
    :param: pos: the nodes to remove
    :return: f: new graph
    """
    f = graph.copy()

    loc = nx.get_node_attributes(graph,'loc')
    polygon = Polygon(corners)
    toRemove = []
    t = []

    for i in range(0, len(loc)):
        if loc[i] is not '':
            point = Point(loc[i])
            if polygon.contains(point):
                toRemove.append(i)
                t.append(graph.nodes[i]['t'])

    f.remove_nodes_from(toRemove)

    for i in range(0, len(toRemove)):
        n = toRemove[i]
        f.add_edge(n,n)
        f.edges[n,n]['p'] = 0
        f.nodes[n]['sv'] = 0.0001
        f.nodes[n]['t'] = t[i]

    return f

def nestedPSO(nodes, swarmSize, Generations, graph):
    """
    Run the bi-level Particle Swarm Optimization
    :param: nodes: the number of nodes in the problem
    :param: swarmSize: the size of the swarm
    :param: Generations: the number of times to iterate through the
optimization
    :param: graph: the graph to run the optimization on
    :return: the best value and its score
    """

    # Initialize swarm
    pos,vel,curValue,pBest,pBestVal = initializeSwarm(nodes, swarmSize,
graph)

    # Intitialize best values
    gBestIndex = pBestVal.index(max(pBestVal))
    gBest = pos[gBestIndex]
    gBestVal = max(pBestVal)
    gBestAfter = []

    # Loop through generations
    for j in range(Generations):
        sys.stdout.write("\r")
        sys.stdout.flush()
        sys.stdout.write("Generation: " + str(j+1) + "/" +
str(Generations))
        sys.stdout.flush()

        for k in range(swarmSize):
            # Remove influencers
            f = removeInfluencers(graph, pos[k])

            # Run PSO on disrupted graph
            after = PSO(f.number_of_nodes(), swarmSize, Generations, f)

            # Evaluate the position
            curValue[k] = evaluate(pos[k], graph)

            # If the after value is higher than before, give it to
before
            if after[1][0] > curValue[k][0]:
                pos[k] = after[0]
                pBest[k] = pos[k]
                pBestVal[k] = after[1]

                if pBestVal[k][0] > gBestVal[0]:
                    gBest = pBest[k]
                    gBestVal = pBestVal[k]

            # If the difference is smaller than pbest, update pbest
            if (curValue[k][0] - after[1][0]) < pBestVal[k][0] and
(curValue[k][0] - after[1][0]) > 0:
                pBest[k] = pos[k]
                pBestVal[k] = curValue[k]

            # If the difference is smaller than gbest, update gbest
            if (curValue[k][0] - after[1][0]) < gBestVal[0] and
(curValue[k][0] - after[1][0]) > 0:
                gBest = pos[k]
                gBestVal = curValue[k]
                gBestAfter = after

        # Update inertia
        inertia = inertiaWeight(j,Generations)

        # Update position and velocity
        for k in range(swarmSize):
            vel[k] =
updateVelocity(vel[k],pos[k],pBest[k],gBest,inertia)
            pos[k] = updatePosition(pos[k],vel[k])

    return gBest, gBestVal, gBestAfter

############################################
# Plot growth of problem
x = np.array(range(1,1000))
y = x**2
plt.plot(x,y)
plt.ylabel("Complexity")
plt.xlabel("Size")
plt.title("Growth of Bi-Level Particle Swarm Algorithm")
plt.savefig("savefig/complexity.jpg", dpi=1000)

swarmSize = 100
Generations = 10
nodes = g.number_of_nodes()
```

```
############################################
""" Effectiveness of algorithm """
ni = 0.15
ss = [10,50,100,200]
Generations = 100
result_ss = {}
for swarmSize in ss:
    result_ss[swarmSize] = PSO(g.number_of_nodes(), swarmSize,
Generations, g)[1][0]

swarmSize = 100
gg = [10,50,100,200]
result_gg = {}
for Generations in gg:
    result_gg[Generations] = PSO(g.number_of_nodes(), swarmSize,
Generations, g)[1][0]

# Save results in data frame
ss = pd.DataFrame(list(result_ss.items()), columns=["n","Vary swarm
size (G=100)"])
gg = pd.DataFrame(list(result_gg.items()), columns=["n","Vary
generations (S=100)"])
gg['Optimal'] = 4.216969451

# Plot results
ax = ss.plot(0,1)
ax2 = gg.plot(0,1,ax=ax)
gg.plot(0,2,ax=ax2)
plt.title("Effect of swarm size and number of generations")
plt.ylabel("Influence")
plt.xlabel("n")
plt.savefig("savefig/ssgg.jpg", dpi=1000)

############################################
""" Using the same threshold for all nodes at varying levels """
T = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
ni = 0.1
result = {}
for t in T:
    result[t] = PSO(g.number_of_nodes(), swarmSize, Generations,
g)[1][0]

varyT = pd.DataFrame(list(result.items()), columns=["t", "Influence"])
varyT.plot(0,1)
plt.title("Effect of changing threshold on influence")
plt.ylabel("Influence")
plt.xlabel("Threshold")
plt.savefig("savefig/t.jpg", dpi=1000)

############################################
""" Using random threshold and varying number of influencers """
numI = [0.01, 0.05, 0.1, 0.15, 0.2,0.25]
result = {}
for ni in numI:
    result[ni] = PSO(g.number_of_nodes(), swarmSize, Generations,
g)[1][0]

varyN = pd.DataFrame(list(result.items()), columns=["n", "Influence"])
varyN.plot(0,1)
plt.title("Effect of changing original number of influencers on
influence")
plt.ylabel("Influence")
plt.xlabel("Number of influencers as percent of the number of nodes")
plt.savefig("savefig/ni.jpg", dpi=1000)

#############################################
""" Varying vulnerable node removal """
ni = 0.1
result = {}
percent = [0.01, 0.05, 0.1, 0.15, 0.2]
for pct in percent:
    f = vulnerable(g, pct)
    result[pct] = PSO(f.number_of_nodes(), swarmSize, Generations,
f)[1][0]

varyPCT = pd.DataFrame(list(result.items()), columns=["Percent",
"Influence"])
varyPCT.plot(0,1)
plt.ylim(0,28)
plt.title("Effect of removing vulnerable nodes on influence")
plt.ylabel("Influence")
plt.xlabel("Percent of nodes removed")
plt.savefig("savefig/pct.jpg", dpi=1000)

#############################################
""" Varying spatial disruptions """
ni = 0.1
slo = [(35.303218,-120.684299), (35.303218,-120.626450), (35.263562,-
120.626450), (35.263562,-120.684299)] #59

box1 = [(35.297194,-120.745239),(35.297194,-120.460968),(35.027747,-
120.460968),(35.027747,-120.745239)] #76
box2 = [(35.706377,-121.030884),(35.706377,-120.212402),(35.027747,-
120.212402),(35.027747,-121.030884)] #119
box3 = [(35.706377,-121.030884),(35.706377,-117.246094),(33.422272,-
117.246094),(33.422272,-121.030884)] #131
boxes = [box1,box2,box3]

result = []
for corners in boxes:
    f = boundary(g, corners)
    #result[Polygon(corners).area] = PSO(f.number_of_nodes(),
swarmSize, Generations, f)[1][0]
    result.append(PSO(f.number_of_nodes(), swarmSize, Generations,
f)[1][0])

b = ('Box 1', 'Box 2', 'Box 3')
y_pos = np.arange(len(b))
plt.bar(y_pos, result, align='center')
plt.xticks(y_pos, b)
plt.ylabel("Influence")
plt.title("Effect of spatial disruption size on influence")
plt.savefig("savefig/area.jpg", dpi=1000)

varyArea = pd.DataFrame(result)
varyArea.plot(0,1)
plt.title("Effect of spatial disruption size on influence")
plt.ylabel("Influence")
plt.xlabel("Area")
plt.savefig("savefig/area.jpg", dpi=1000)
```

```
############################################
“”” Run model “””
swarmSize = 100
Generations = 10
ni = 0.1

result = nestedPSO(g.number_of_nodes(), swarmSize, Generations, g)


""" Save """
gBest = result[0]
gBestVal = result[1]
gBestAfter = result[2]

def get_twitter_id(lyst):
    a = []
    for i in range(0, len(lyst)):
        if lyst[i] == 1:
            a.append(i)
    b = []
    for n in g.nodes():
        if n in a:
            b.append(g.nodes[n]['name'])
    return a,b

influencers = get_twitter_id(gBest)
before = get_twitter_id(gBestVal[1])
influencers_after = get_twitter_id(gBestAfter[0])
after = get_twitter_id(gBestAfter[1][1])

gBestVal[0]
gBestAfter[1][0]

with open('result.txt', 'w') as f:
    for item in influencers[0]:
        f.write("%s, " % item)
    f.write("\n")
    f.write("%s\n" % gBestVal[0])
    for item in before[0]:
        f.write("%s, " % item)
    f.write("\n")
    f.write("\n")
    for item in influencers_after[0]:
        f.write("%s, " % item)
    f.write("\n")
    f.write("%s\n" % gBestAfter[1][0])
    for item in after[0]:
        f.write("%s, " % item)

############################################
“”” Plot “””
lines = [[(1,26.69074114285715),(1,20.57512514285715)],
        [(2,25.766334285714294),(2,20.060527428571437)],
        [(3,28.001036571428585),(3,20.809234285714293)],
        [(4,29.7033342857143),(4,18.259329714285712)]]

lc = mc.LineCollection(lines)
fig, ax = pl.subplots()
ax.add_collection(lc)
ax.autoscale()
ax.margins(0.1)
plt.ylim(0,32)
b = ('','Run 1', 'Run 2', 'Run 3', 'Run 4')
y_pos = np.arange(len(b))
```

```
plt.xticks(y_pos,b)
plt.ylabel("Influence")
plt.title("Bi-Level Results")
plt.savefig("savefig/results.jpg", dpi=1000)

before =
[26.69074114285715,25.766334285714294,28.001036571428585,29.70333428571
43]
after =
[20.57512514285715,20.060527428571437,20.809234285714293,18.25932971428
5712]
before = np.array(before)
after = np.array(after)
np.mean(before - after)
before-after

beforeNodes = [1, 16, 17, 33, 35, 48, 65, 67, 68, 70, 77, 102, 113,
122, 130, 141, 153, 155, 163, 180, 200,
1, 4, 10, 13, 17, 39, 53, 64, 65, 66, 70, 77, 86, 91, 109, 114, 127,
155, 159, 174, 179,
3, 4, 13, 14, 38, 40, 46, 54, 61, 67, 68, 70, 77, 79, 82, 91, 93, 127,
139, 142, 186,
10, 13, 16, 17, 20, 27, 39, 46, 66, 68, 70, 77, 81, 82, 85, 86, 90, 93,
127, 143, 176]

afterNodes = [5, 13, 43, 46, 47, 50, 52, 54, 62, 80, 84, 86, 111, 127,
143, 148, 152, 158, 162, 183, 190,
0, 5, 9, 27, 28, 33, 38, 40, 52, 55, 62, 68, 107, 130, 148, 161, 173,
182, 191, 207, 215,
8, 17, 18, 20, 22, 25, 27, 36, 43, 44, 45, 76, 101, 114, 136, 157, 167,
169, 170, 173, 206,
1, 8, 18, 19, 25, 36, 43, 45, 47, 65, 75, 76, 88, 102, 128, 134, 159,
167, 171, 198, 203]

plt.hist(beforeNodes,216)
plt.title("Histogram of After-Disruption Influencers")
plt.xlabel("Node")
plt.ylabel("Frequency")
plt.savefig("savefig/afternodes.jpg", dpi=1000)

############################################
""" Complexity  """
swarmSize = 1000
Generations = 50
result = PSO(g.number_of_nodes(), swarmSize, Generations, g)
f = removeInfluencers(g,result[0])
result2 = PSO(g.number_of_nodes(), swarmSize, Generations, f)

result[1][0]
result2[1][0]

############################################
“”” Create folium map “””
coord = load_json('13179562_coord.json')

# Read in shapefiles
svi_county = os.path.join('SVI2016_US_COUNTY.geojson')
scores = os.path.join('SVI2016_US_COUNTY.csv')
scores_data = pd.read_csv(scores)

# Influenced nodes
before = [1, 15, 16, 17, 23, 33, 35, 48, 65, 67, 68, 70, 71, 72, 76,
77, 83, 84, 99, 100, 102, 110, 112, 113, 118, 122, 130, 136, 137, 138,
141, 144, 151, 153, 154, 155, 163, 165, 166, 170, 174, 176, 177, 178,
180, 182, 189, 192, 195, 200, 204, 207, 209, 211, 212, 213]
after = [5, 8, 13, 15, 23, 43, 46, 47, 50, 52, 54, 55, 62, 71, 72, 80,
83, 84, 86, 99, 100, 110, 111, 112, 127, 136, 138, 143, 144, 148, 152,
154, 158, 162, 166, 175, 177, 178, 183, 190, 198, 204, 215]

loc = nx.get_node_attributes(g, 'loc')
name = nx.get_node_attributes(g, 'name')

heat_data = []
for i in range(0,len(before)):
    if loc[i] is not '':
        heat_data.append(loc[i])

c = folium.Map(location=[38, -120.6], zoom_start=6)
HeatMap(heat_data).add_to(c)

for k, v in coord.items():
    if v is not "":
        folium.Circle(
            radius=100,
            location=v,
            popup=k,
            color='black',
            fill=False,
        ).add_to(c)

c.choropleth(
    geo_data=svi_county,
    name='choropleth',
    data=scores_data,
    columns=['OBJECTID', 'RPL_THEMES'],
    key_on='feature.id',
    fill_color='BuPu',
    fill_opacity=0.4,
    line_opacity=0.2,
    legend_name='Social Vulnerability Index'
)
folium.LayerControl().add_to(c)
c.save('before.html')

############################################
""" Sample graph """
"""
([1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
 (4.780702766311823, [1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1]),
 ([0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
  (3.689210867507707, [0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1,
1])))
 """
# Position for sample graph
pos = {}
pos[0] = np.array([0,0])
pos[1] = np.array([-2,-1])
pos[2] = np.array([-1,-1])
pos[3] = np.array([1,-1])
pos[4] = np.array([2,-1])
pos[5] = np.array([-1,-2])
pos[6] = np.array([0,-2])
pos[7] = np.array([1,-2])
pos[8] = np.array([-2,-3])
pos[9] = np.array([-1,-3])
pos[10] = np.array([1,-3])
pos[11] = np.array([2,-3])
```

```
pos[12] = np.array([-1,-4])
pos[13] = np.array([0,-4])
pos[14] = np.array([1,-4])
pos[15] = np.array([0,-5])

# Create sample graph
g = nx.DiGraph()
nodeList = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
g.add_nodes_from(nodeList)
g.add_edge(0,1)
g.add_edge(0,2)
g.add_edge(0,3)
g.add_edge(0,4)

g.add_edge(1,5)
g.add_edge(1,6)
g.add_edge(1,8)

g.add_edge(2,5)
g.add_edge(2,6)
g.add_edge(2,7)

g.add_edge(3,6)
g.add_edge(3,7)

g.add_edge(4,6)
g.add_edge(4,7)
g.add_edge(4,11)

g.add_edge(5,8)
g.add_edge(5,9)

g.add_edge(6,9)
g.add_edge(6,10)

g.add_edge(7,10)
g.add_edge(7,11)

g.add_edge(8,12)
g.add_edge(8,13)

g.add_edge(9,12)
g.add_edge(9,13)

g.add_edge(10,13)
g.add_edge(10,14)

g.add_edge(11,13)
g.add_edge(11,14)

g.add_edge(12,15)
g.add_edge(13,15)
g.add_edge(14,15)

ni = 0.15
swarmSize = 100
Generations = 10
nodes = g.number_of_nodes()
```

```
############################################
# Run bi-level problem
result = nestedPSO(g.number_of_nodes(), swarmSize, Generations, g)

# Set sizes for plotting
sv = []
for n in g.nodes():
    sv.append((g.nodes[n]['sv']+1)**2 * 100)

p = []
for e in g.edges():
    p.append((g.edges[e]['p']))

# Draw first graph
nx.draw(g, pos=pos, with_labels=True, node_size=sv, width=p,font_size =
8, node_color='lightblue')
plt.savefig('savefig/plot.jpg', dpi=1000)

influencers =[0,7]
influenced =[1,2,3,11,14,15]

# Set colors
color = []
for n in g.nodes():
    if n in influencers:
        color.append('#67d394')
    elif n in influenced:
        color.append('#fae86d')
    else:
        color.append('lightblue')

nx.draw(g, pos=pos, with_labels=True, node_size=sv, width=p, font_size
= 8, node_color=color)
plt.savefig('savefig/before.jpg', dpi=1000)

# Remove nodes for second plot
f = g.copy()
toRemove =[0,7]

# Remove node
f.remove_nodes_from(toRemove)

# Add back without edges
for e in toRemove:
    f.add_node(e)

# Assign the node the sv score
f.nodes[0]['sv'] = g.nodes[0]['sv']
f.nodes[7]['sv'] = g.nodes[7]['sv']

# New influencers
influencers =[4,5]
influenced = [8,11,14,15]

sv = []
for n in f.nodes():
    sv.append((f.nodes[n]['sv']+1)**2 * 100)

p = []
for e in f.edges():
    p.append((f.edges[e]['p']))

# Update color
color = []
```

```
for n in f.nodes():
    if n in influencers:
        color.append('#67d394')
    elif n in influenced:
        color.append('#fae86d')
    else:
        color.append('lightblue')


nx.draw(f, pos=pos, with_labels=True, node_size=sv, font_size = 8,
width=p, node_color=color)
plt.savefig('savefig/after.jpg', dpi=1000)
```