# Recovering Social Networks from Contagion Information

Sucheta Soundarajan[⋆] sucheta@cs.cornell.edu

John Hopcroft[⋆] jeh@cs.cornell.edu

**Abstract.** Many algorithms for analyzing social networks assume that the structure of the network is known, but this is not always a reasonable assumption. We wish to reconstruct an underlying network given data about how some property, such as disease or information, has spread through the network. Properties may spread through a network in different ways: for instance, an individual may learn information as soon as one of his neighbors has learned that information, but political beliefs may follow a different type of model. We create algorithms for discovering underlying networks that would give rise to the diffusion in the above two types of models. We first consider the case in which a vertex adopts a property as soon as one of its neighbors has adopted the property. After that, we consider the case in which a vertex adopts a property after at least half of its neighbors have adopted the property. Finally, we list future directions for this work.

**Keywords:** Social Networks, Diffusion, Contagion, Graph Algorithms

## 1 Introduction

The area of social network analysis raises many interesting and important questions. For instance: If a virus is spreading through a population, but there is a limited amount of vaccine, which people are the best to vaccinate[7]? If a toxin is spreading through a water network, where should sensors be placed to detect contamination[5]? Questions like this are becoming increasingly common, and their discussion generally assumes that the structure of the underlying network is known. In some cases, this is a legitimate assumption. For example, if we wish to trace the spread of some trend through an online social network such as Facebook, then it's believable that the structure of the network is known. In other cases, such as with a criminal network, the assumption is less plausible. However, in such cases, even though the structure of the network may be unknown, we may have some information about the spread of something through the network. For instance, the police may not know the routes and network connecting various drug dealers, but they might be able to pinpoint the times at which some new drug appears in various cities. If they know that this new drug first appeared in some city X, and then in some other cities Y and Z, and so on, can they use this knowledge to recreate the underlying network?

Although much work has been done to study the general structure of criminal networks[8][6], algorithms for analyzing social networks can often be very sensitive to the exact, as opposed to general, structure of the network[3]. Consider the example of the Mexican drug war, in which thousands of lives and billions of dollars have been lost in attempts to fight the drug cartels[2]. It seems that the authorities could simply capture the leaders of these cartels, but such actions often create a power vacuum, causing more bloodshed as rival cartels fight to take power. There may be algorithms to help the authorities analyze these networks and determine how to safely eliminate members of the cartels, but in order for them to apply such algorithms, the specific structure of the network must be known. Clearly, the authorities cannot simply survey criminals to determine their connections within the network, and their work is further complicated by the fact that many high ranking police officers may collaborate with the cartels. Algorithms to help the authorities recreate the network, or even parts of the network, would help them tremendously.

The work presented here will be vital to the creation of practical algorithms for such applications. In this paper, we will investigate the situation in which some property (e.g., illness, opinion, information, etc.) is spreading through a population, and that we know the times at which each individual adopted the property. We

[⋆]Dept. of Computer Science, Cornell University, Ithaca, NY 14850

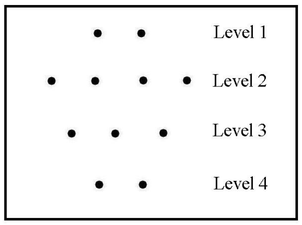

**Fig. 1.** Vertex Levels

also assume that we know how the property spreads. For instance, information spread may follow the rule that an individual obtains information as soon as one of his neighbors obtains that information, but other properties like political belief might follow the rule that an individual adopts a belief after some proportion of his neighbors have adopted that belief. First, we will define the problem precisely, and then briefly describe an algorithm for constructing a graph under the model of information spread. After that, we consider the case when a vertex adopts a property after at least half of its neighbors have adopted the property.

The model in which every vertex adopts a property after half of its neighbors have adopted it is applicable to both real and theoretical situations[4]. Consider the case of workers planning to go on strike[3]- each worker wishes to determine whether the strike will succeed or fail, and a worker may reasonably determine that if at least half of her colleagues are striking, then the strike is likely to succeed, and so she is willing to go on strike. One half is a reasonable threshold for any situation in which each individual desires to be in the majority part of the population: for example, when determining which social event to attend, an individual would like to attend the event that the majority of his friends are attending. We may, of course, encounter more complicated cases, such as when every vertex has a different threshold. We believe that algorithms for such cases will be largely based upon the algorithms in this paper, and the algorithms described here are vital in understanding the difficulties and intricacies involved in solving more complicated cases. This paper describes algorithms that are useful for real applications and will provide a foundation upon which further work can be built.

## 2 The General Problem

We wish to solve the problem of reconstructing a network given information about how some property has spread through the network, and the times at which each vertex adopted that property. We are given the vertices, the time at which each vertex adopts the property, and a description of how the property spreads.

**2.1 Model of Contagion** A model of contagion describes how some property spreads through a network. We consider the case of one property spreading through the network. We also assume that once a vertex adopts a property, it does not 'unadopt' that property. A simple model of contagion is: "A vertex adopts the property in the time interval after at least one of its neighbors adopts the property." This sort of model might correspond to the spread of a very contagious disease (e.g., a person gets sick after one of his friends gets sick) or information. A more complicated model of contagion is: "A vertex adopts the property in the time interval after at least $p$ fraction of its neighbors adopt the property." This type of model could correspond to the spread of opinion: for instance, while it's unlikely that a person would adopt a new political belief as soon as just one of his friends adopts it, it's more believable that a person adopts such a belief after a majority of his friends adopt it[3]. Although other models of contagion might allow each vertex to have a different value of $p$, this paper will only consider models of contagion in which each vertex has the same value of $p$.

**2.2 Terminology** A *time vector* is a vector of positive integers where the $i$th element of the vector is the time at which vertex $i$ adopts the property. The smallest integer in the vector is 1 (vertices which adopt the property at time 1 are the first ones to adopt the property) and the vector contains all integers up to its maximum value. In other words, no times can be 'skipped.'

If vertex $x$ adopts the property at time $t$, then $x$ is at *level* $t$. In this paper, we depict vertices sorted

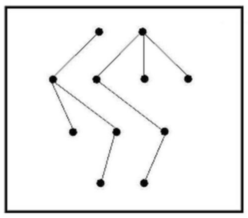

**Fig. 2.** Finding a Solution Graph for the Simple Model of Contagion

into levels so that vertices in level 1 are at the top, and vertices in the last level are at the bottom (Fig. 1).

For a particular model of contagion, a graph $G$ *solves* a time vector if, after introducing the property to all vertices which adopt the property at time 1, all others adopt the property at the appropriate time.

An edge $(w, x)$ is *incoming* to $x$ if $w$ is in a level prior to $x$, and *outgoing* from $x$ otherwise. Note that despite this terminology, edges in this graph are not directed. Although the graph itself is not directed, we use such terminology because we can treat the property as 'flowing' from one vertex to another in some direction.

**2.3 Problem Statement** Given a set of vertices, a time vector, and a model of contagion, create a graph $G$ over the vertices so that $G$ solves the time vector. In other words, if we know how the vertices are sorted into levels, we need to find a graph over the vertices so that when the property is introduced to the vertices in level 1 and spreads in accordance with the model of contagion, every vertex adopts the property at the appropriate time.

**2.4 Modified Problem Statement.** We also consider a variation on this problem, in which some edges may be *mandatory* (must be included in the solution graph) or *forbidden* (cannot be included in the solution graph).

## 3 Algorithm for a Simple Model of Contagion

In order to help the reader develop intuition about this problem, we will first consider a simple model of contagion in which a vertex adopts the property in the time interval immediately after a neighbor adopts the property. A model like this may be appropriate to explain the spread of information, for example. Constructing a graph to solve a time vector for this model of contagion is trivial. As an example, consider a vertex in level 4. This vertex adopts the property at time 4, so it must have an edge to some vertex in time 3. However, if it has an edge to vertices in times 1 or 2, then it would adopt the property before time 4. Thus, each vertex in level 2 or later must be connected to at least one edge to a vertex in the previous level, and cannot have any edges to vertices in levels earlier than that. So to construct a graph, simply connect each level 2 vertex to some level 1 vertex, then connect each level 3 vertex to some level 2 vertex, and so on. It is easy to see that a very similar method can also be used to solve the Modified Problem. For example, consider the set of vertices shown in Fig. 2. Each vertex in levels 2 and later is connected to a vertex in the previous level. There are many different solution graphs for this set of vertices.

## 4 Algorithms for a More Complicated Model of Contagion

A more complicated model of contagion is when a vertex adopts the property in the time interval immediately after some fixed proportion $p$ of its neighbors adopt the property. The algorithms here are for the case $p = \frac{1}{2}$. This model is relevant to situations in which each member of the population wishes to be a part of the majority, such as when individuals are deciding which of two conflicting social events to attend.

For $G$ to solve a time vector under this model of contagion, every vertex in levels 2 and higher must have at least one edge from the previous level. Call this requirement the *recency condition,* and an edge fulfilling this requirement a *recency edge.* Additionally, a vertex in level 2 or later cannot have more outgoing edges than incoming edges; if it did, then more than half its neighbors would be in a level later than it, but for the vertex

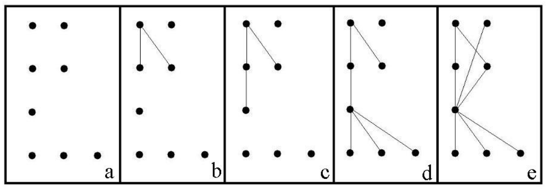

**Fig. 3.** Finding a Solution Graph for the More Complicated Model of Contagion

to adopt the property at the correct time, half of its neighbors must have already adopted the property by that time. Call this requirement the *balance condition.* Vertices in level 1 can have arbitrarily many outgoing edges, since we assume that they always adopt the property at time 1. A vertex is *satisfied* if it adopts the property at the correct time.

**4.1 Problem Intuition** Solving this problem requires that each vertex has a recency edge incoming from the previous level. Consider vertices in level 2. Since level 1 vertices can send out as many outgoing edges as desired, creating recency edges for level 2 vertices is trivial: simply connect each level 2 vertex to some level 1 vertex. Now consider vertices in level 3. Each level 3 vertex needs an edge from some level 2 vertex. So far, each level 2 vertex only has one incoming edge, but if there are more level 3 vertices than level 2 vertices, then some level 2 vertices will need to acquire more incoming edges to allow the addition of these mandatory outgoing edges, or the balance condition would be violated. Similarly, if there are more level 4 vertices than level 3 vertices, some level 3 vertices will need to acquire additional incoming edges. Each vertex can get an incoming edge from every level 1 vertex, and these extra incoming edges can be used to support any necessary outgoing edges. But sometimes, even more incoming edges might be necessary. To solve this problem, we will proceed through the levels in order, obtaining recency edges for each vertex while ensuring that the balance condition is not violated.

Suppose we are finding a solution graph for the vertex set shown in Fig. 3a. Each of the two level 2 vertices must obtain a recency edge from a level 1 vertex, so we begin by adding two such edges (Fig. 3b). Note that instead of having two outgoing edges from one level 1 vertex, we could have had one outgoing edge from each of the level 1 vertices. Next, the level 3 vertex must obtain a recency edge from a level 2 vertex, so we add this edge to the network (Fig. 3c). The first vertex in level 2 now has one outgoing edge, but since it also has one incoming edge, the balance condition has not been violated. Next, the three vertices in level 4 must obtain recency edges from the vertex in level 3 (Fig. 3d). Since the vertex in level 3 only has one incoming edge, but three outgoing edges, the balance condition has been violated. To remedy this, the vertex in level 3 must obtain two additional incoming edges (Fig. 3e).

**4.2 A Flow-Based Algorithm** The balance condition requirement strongly resembles a flow network, where outgoing flow is no more than incoming flow. We will create a flow network and find a satisfying flow[1], and from this determine which edges to put in the graph. Consider first what a solution graph will look like. Each vertex in levels 2 or later must have at least one recency condition edge from some vertex in the previous level. If a vertex has any outgoing edges, then it must have the same number of incoming edges. In our flow network, most edges will have a maximum capacity of 1, and all capacities will be integers. Thus, we will be able to find an integral flow. If in this flow, an edge has a flow of 1, then we will add that edge to the solution graph. If it has a flow of 0, we will not.

The flow network will contain all edges except those between vertices in the same level or vertices in adjacent levels. The edges are directed from earlier levels to later levels, and have a maximum capacity of 1. Although edges between vertices in the same level can certainly exist in a real social network, these edges are not a necessary part of a minimal solution graph, and so are not included in the flow network. Edges between vertices in adjacent level are potential recency edges, so must be handled in a special way.

Each vertex in levels 2 and later must obtain a recency edge from a vertex in the previous level. To satisfy this requirement, we will use a special structure to represent edges between vertices in adjacent levels. These

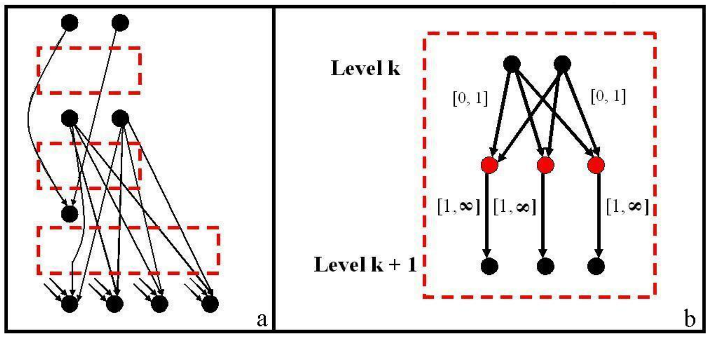

**Fig. 4.** Constructing the Flow Network

edges all have a maximum capacity of 1. See Fig. 4a. The unended edges at level 4 originate at level 1, and the dashed box represents the special structure.

To construct the special structure between vertices in adjacent levels, between each pair of adjacent levels $k$ and $k+1$ add a row of vertices. In this intermediate row, there is one vertex for each vertex in level $k+1$. There will be a edge from each vertex in level $k$ to each vertex in the intermediate row, each with a maximum capacity of 1. There will be an edge from each vertex in the intermediate row to the corresponding vertex in level $k+1$. These edges have a *minimum* capacity of 1 and unlimited maximum capacity. See Fig. 4b. Because these edges have a minimum capacity of 1, each vertex in the intermediate row must have an edge from at least one vertex in level $k$.

The vertices in level 1 can send as many outgoing edges as necessary, so these vertices will serve as sources of unlimited flow. We will also need a sink vertex. Consider an individual vertex in levels 2 or later: generally, a vertex will have an equal number of incoming and outgoing edges, and will not need to send any flow to a sink vertex. However, some vertices, particularly those in the last level, must have at least one incoming edge from the previous level, but will not have any outgoing edges. There may be other vertices in earlier levels like this as well- each vertex must have an incoming edge from the previous level, but they are not required to have any outgoing edges. In such cases, vertices may have an incoming flow of 1 and an outgoing flow of 0. Thus, each vertex in levels 2 or later should have an edge to the sink vertex with capacity 1.

Once this flow network has been created, find a satisfying flow. If an edge has a flow of 1, we add it to the solution graph.

We can easily solve the modified problem, in which some edges are mandatory or prohibited, by slightly changing the flow network. If an edge is mandatory, raise the minimum capacity along that edge from 0 to 1. If an edge is prohibited, remove it from the flow network.

**4.3 Generalizing the Flow Based Algorithm.** This algorithm works for the case $p = \frac{1}{2}$, where flow in is equal to flow out, but what about for other values of $p$? A natural case to consider is $p = \frac{2}{3}$, where for every outgoing edge, there are two incoming edges. Unfortunately, the flow-based algorithm cannot be generalized to this case.

Consider the general problem of finding a satisfying integer flow over an arbitrary network, where the flow out of a vertex is the floor of half the flow into the vertex. If such an algorithm existed, we could easily modify the above algorithm to solve the problem for $p = \frac{2}{3}$. Unfortunately, this general problem is NP-complete.

*Claim.* The problem of finding a satisfying integer flow over an arbitrary network, where the flow out of a vertex is the floor of half the flow into the vertex, is NP-complete.

*Proof.* We will reduce 3SAT to this problem. Fig. 5a represents one variable and Fig. 5b represents one clause. There is one sink vertex for each variable and one sink vertex for each clause. Each sink vertex has a demand of 1. The thick black lines carry flow from the source.

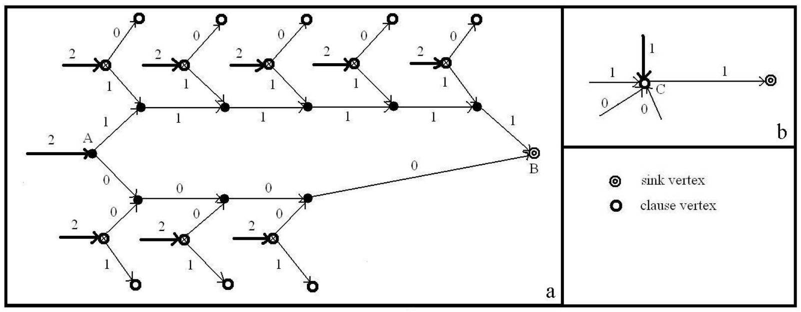

**Fig. 5.** Reducing 3SAT to a Flow Problem

Consider Fig. 5a, corresponding to one variable, first. Sink vertex B, on the right side, has a demand of 1. Source vertex A, on the left side, has an incoming flow of 2 and thus an outgoing flow of 1. Vertex A can send this outgoing flow across either the top row or bottom row of solid black vertices. In this example, the flow was sent across the top row. This corresponds to setting the variable to FALSE. In order for the flow across the top row to reach vertex B, each solid black vertex in the top row must receive one additional unit of flow. The vertices with a cross through the middle send out one unit of flow. This unit of flow can go to either a solid black vertex or to a clause vertex. In this case, since the top row of solid black vertices needs supplementary flow at each vertex, the clause vertices at the top receive no flow from this structure. The clause vertices at the bottom can each receive one unit of flow. In this example, there are 5 clauses containing the variable as TRUE and 3 clauses containing the variable as FALSE.

Now consider Fig. 5b, corresponding to one clause. Vertex C, in the middle, has an edge incoming from the source and three edges incoming from either the top row or the bottom row of the structures representing the three variables contained in the clause. Vertex C must receive 2 units of flow to send 1 unit to the sink vertex. Vertex C receives 1 unit of flow from the source, so it must also get 1 more unit of flow from one of the variable structures. There is a satisfying flow for this problem if and only there is a solution for the 3SAT problem. □

Note though that although the flow problem over a general network is NP-complete, perhaps the flow-based algorithm for our problem does not need the full power of a flow algorithm, and using a flow algorithm masks too much of the detail in our algorithm. Following is a different algorithm for solving for the case $p = \frac{1}{2}$, in which we make some of the details more explicit.

**4.4 Simplified Problem.** We will create an algorithm for a simplified version of the original problem, and then show that the algorithm for this simplified problem also solves the original problem.

Given a set of vertices and a time vector, create a graph $G$ over the vertices such that for every vertex $x$, $x$ has at least one neighbor in the previous level and at least half of $x$'s edges are incoming. In the following algorithm and proof, the term *solves* will refer to this simplified problem. Notice that a graph which solves the simplified problem does not necessarily solve the original problem; although a vertex has at least half of its edges incoming, if too many edges are incoming from much earlier levels, it may still adopt the property too early.

**4.5 A Backtracking Algorithm** In this algorithm we will construct a graph to solve the simplified problem. We proceed through the vertices in order, adding recency-condition edges to each vertex. If the vertex being asked to provide the outgoing recency-condition edge does not have sufficiently many incoming edges to send out another outgoing edge, then we attempt to modify the graph so that we can safely add the recency-condition edge. This section will not explain how to implement this method; rather, it presents a general description of how the algorithm works. Section 4.7 contains a specifc implementation of the backtracking algorithm.

Construct a graph $G$ by first connecting each vertex in level 2 to a vertex in level 1. Then progressing

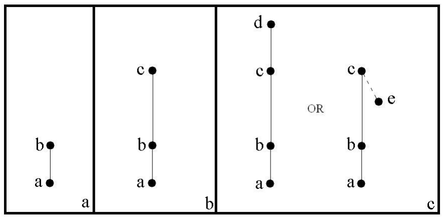

**Fig. 6.** A Backtracking Example

through the levels in order, for each vertex $v$ find a path from $v$ which goes through a vertex in the previous level and then proceeds back through earlier and earlier levels until it terminates at a vertex with more incoming edges than outgoing edges, such that no edge in the path is already in $G$. Call such a path a *satisfying sequence.* Such a sequence represents adding an edge to $v$ from a vertex in the previous level, but that vertex must now find a new incoming edge to allow the addition of the outgoing edge to $v$, and so on until a vertex with an excess of incoming edges is found. If a satisfying sequence can be found for $v$, then adding these edges to $G$ will satisfy $v$ while still leaving all previous vertices with at least as many incoming edges as outgoing edges.

It may be that no such sequence can be found unless $G$ is modified. For example, suppose we find a sequence from $v$ back to some vertex $w$, but then we are unable to proceed further because $w$ is unable to obtain any more incoming edges. Because $w$ is in a level prior to $v$, $w$ has already been satisfied and so there must be some edge incoming to $w$. This edge is currently being used to support some other satisfying sequence going through $w$ and eventually on to some later vertex $x$. If $x$ can find a different satisfying sequence which does not use $w$, then the edge incoming to $w$ can be used to support a satisfying sequence to $v$.

More precisely, the algorithm must find a path $(v, u_k), (u_k, u_{k-1}), ..., (u_2, u_1)$ such that:

1. $u_k$ is in the level immediately before $v$.
2. For all $i < k$, either:
   2a. $u_i$ is in a level previous to $u_{i+1}$ and $(u_i, u_{i+1})$ does not exist in $G$, or
   2b. $u_i$ is in a level after $u_{i+1}$ and $(u_i, u_{i+1})$ does exist in $G$.
3. If $u_1$ is in a level previous to $u_2$, then $u_1$ is either in level 1 or has more incoming than outgoing edges in $G$. If $u_1$ is in a level after $u_2$, then $u_1$ has more incoming edges than outgoing edges in $G$ and the edge $(u_1, u_2)$ was not necessary to satisfy the recency condition for $u_1$.
4. No vertex appears more than twice in the path.

Condition 2a represents vertex $u_{i+1}$ obtaining an additional incoming edge, and 2b represents vertex $u_{i+1}$ removing an outgoing edge to $u_i$. We will use the term *backtracking sequence* to describe such a path. Note that although a backtracking sequence is represented as a series of edges, it actually represents a series of proposed actions. If an edge in the sequence does not yet exist in the graph, then its presence in the sequence represents a proposal to add that edge to the graph. If an edge in the sequence already exists in the graph, then its presence in the sequence represents proposing removing that edge from the graph.

Another way to interpret this method is to consider an individual edge at a time rather than entire satisfying sequences. If $v$ is attempting to obtain an edge from a vertex $w$ in a previous level, then $w$ must have more incoming than outgoing edges. If that is not currently the case, then $w$ can either obtain a new incoming edge or remove an outgoing edge. If we remove an edge outgoing from $w$ and incoming to some vertex $x$, then $x$ may now have fewer incoming edges than outgoing edges, and so $x$ must either obtain a new incoming edge or remove an outgoing edge, and so on. If the edge removed was necessary to satisfy the recency condition for $x$, then $x$ must obtain a new incoming edge which satisfies the recency condition.

Consider Fig. 6. In 6a, vertex $a$ must find a recency edge, so we attempt to add an edge from vertex $b$ in the previous level. If vertex $b$ does not have enough incoming edges to add this edge, then $b$ must obtain a new incoming edge. Suppose that vertex $b$ asks vertex $c$ in some earlier level for an edge (6b). If vertex $c$ does not have enough incoming edges to send an outgoing edge to vertex $b$, then there are two possible options (6c). Vertex $c$ may attempt to get a new incoming edge from a vertex in a previous level, such as vertex $d$, or vertex $c$ may remove an existing outgoing edge to a later vertex, such as vertex $e$. We then apply this same procedure to either $d$ or $e$.

Note that although we originally intended for this algorithm to solve the simplified problem, as stated above, it actually solves the original problem as well. A vertex only obtains a non-recency condition incoming edge when it needs that edge to support some outgoing edge. Thus, a vertex will never have too many incoming edges relative to the number of outgoing edges, so the vertex will not adopt the property too early.

## 4.6 Proof of Correctness

*Claim.* Given a set of vertices and a time vector, then for the model of contagion in which a vertex adopts the property after half of its neighbors have adopted the property, the backtracking algorithm described above will create a minimal graph solving the time vector.

*Proof.* It is clear that any graph constructed by the algorithm solves the problem, so we must show that if there is a solution to a time vector, then this algorithm can find it. Suppose we have a solution graph $S$ to a time vector, and the algorithm has created a partial solution $G$ to that same time vector. We can assume without loss of generality that $S$ is minimal; that is, no edges can be removed from $S$ while leaving a graph that still solves the time vector. Let $v$ be the first vertex in $G$ that is not satisfied. We will create a backtracking sequence for $v$.

Since $v$ is satisfied in $S$, $v$ in $S$ has an edge incoming from some vertex $u$ in the previous level. This edge doesn't exist in $G$, since if it did, then $v$ would be satisfied. This edge is the beginning of the backtracking sequence. If $u$ in $G$ has enough incoming edges to send an outgoing edge to $v$ right away, then we are done. If not, then $u$ in $G$ either doesn't have enough incoming edges or has too many outgoing edges, as compared to $u$ in $S$. So either $u$ in $S$ has some incoming edge that it doesn't have in $G$, or $u$ in $G$ has some outgoing edge that it doesn't have in $S$. This edge is next in the backtracking sequence.

Now the next vertex in the backtracking sequence is in the same position as $u$ was in the last paragraph. We might be able to immediately add or remove the edge from $G$, but if not, then the next vertex must obtain a new incoming edge or remove an outgoing edge. Continue applying this argument. To show that this process terminates, we must show that the backtracking sequence is of finite length. Note that every edge in the sequence represents either an edge in $G$ or $S$, and there are only finitely many such edges. So we only need to show that each edge is added to the sequence at most once.

Above, we argued that if exactly one outgoing edge is requested from some vertex, then that vertex can either obtain a new incoming edge or remove an existing outgoing edge. This argument can be extended if multiple outgoing edges are requested from some vertex. If multiple outgoing edges are requested from a vertex, then that vertex can either obtain that many new incoming edges, or remove that many existing outgoing edges, or a combination. The same is true if multiple incoming edges are deleted from the vertex. (See the first lemma in the appendix for a proof.) Because of this, every time an outgoing edge is requested or an incoming edge is deleted from a vertex, the edge it adds or deletes in response can be chosen in such a way that there are no repetitions. We are never forced to put the same edge into the sequence multiple times, so the process must terminate.

After this process terminates, some vertices may appear multiple times. See the second lemma in the appendix for proof that we can eliminate cycles in the sequence so that each vertex appears at most twice. Now the sequence satisfies all the requirements for the backtracking sequence, so the algorithm can continue.□

**4.7 A Recursive Closure Algorithm** The recursive closure algorithm is an implementation of the backtracking algorithm. Going through the vertices in order of level, for each vertex we recursively create a set that contains vertices in a potential backtracking sequence. Initially, the set contains all vertices in the previous level, since these vertices can provide a recency edge. Next, we will add to the set those vertices from which current members of the set can either obtain a new incoming edge or remove an existing outgoing edge. We keep doing this recursively until we find some vertex which is able to satisfy the request without further searching. More precisely:

First, connect every vertex in level 2 to some vertex in level 1. Then starting with vertices in level 3 and progressing through the vertices in order of level, for each vertex $x$, do the following:

Recursively create a set of vertices, initially containing only $x$. First, add all vertices from which $x$ can receive a recency-condition edge. At each successive step, expand the set to contain vertices from which newly added members can either obtain a new incoming edge or remove an outgoing edge. Continue until a vertex with an excess of incoming edges is found. If no such vertex is found, then there is no backtracking sequence for $x$. Some bookkeeping is necessary to keep track of paths within the set; this can be done by associating each vertex with a pointer to the vertices from which it tried to obtain an incoming edge or remove an outgoing edge.

Some care must be taken, because if a vertex is deprived of a recency edge, it must obtain a new recency edge, rather than obtaining any new incoming edge or removing any existing outgoing edge. To deal with this, associate a 'flag' with each vertex in the set. If a vertex is not flagged, it may obtain any new incoming edge or remove an existing outgoing edge. If it is flagged, it can only obtain a new recency edge.

More precisely, create a set $S$ of vertices. Initially, $S$ contains only a flagged version of $x$, since $x$ must obtain a new incoming edge from the previous level. Then at each step, for each vertex $y$ in $S$, do the following:

If $y$ is not flagged, then it can add any incoming edge or remove an outgoing edge, so add to $S$ a copy of all vertices from which $y$ can obtain a new incoming edge or remove an existing outgoing edge. If $y$ wishes to remove an outgoing edge from a vertex which needed that edge in order to satisfy its recency condition, then add a flagged copy of that vertex to $S$. Otherwise, add an unflagged copy of the vertex.

If $y$ is flagged, then add to $S$ an unflagged copy of all vertices in the level previous to $y$ from which $y$ can obtain a new incoming edge.

During this process, if $S$ currently contains a flagged copy of some vertex, but we wish to add an unflagged copy of that vertex to $S$, then simply change the original copy from flagged to unflagged. Otherwise, if some vertex is encountered multiple times, only add it to $S$ once. Continue this process until we find an unflagged vertex with an excess of incoming edges. (Finding a flagged vertex with an excess of incoming edges isn't enough, since it was deprived of it's recency-condition edge.) If we cannot find such a vertex, then there is no backtracking sequence for $x$. Because each vertex is added to or modified in the set at most two times, the algorithm runs in polynomial time.

Like the flow-based algorithm, we can easily change the backtracking algorithm to solve the modified problem. If an edge is mandatory, we include it in the solution graph from the beginning, and never remove it. If an edge is forbidden, we simply do not consider it.

**4.8 Generalizing the Backtracking Algorithm.** The backtracking algorithm is similar to the flow-based algorithm, but each step is explicit. However, even though we do not use the full power of a flow algorithm here, the backtracking algorithm runs in exponential time for $p = \frac{2}{3}$. For $p = \frac{1}{2}$, the recursive closure implementation of the backtracking algorithm runs in polynomial time because each vertex is added to or modified in $S$ at most twice. Once a vertex says that it is able to fulfill a request, then the current iteration of the algorithm has essentially completed. For instance, if the algorithm requests that a certain vertex obtain a new incoming edge, and the vertex is able to obtain that edge, then that iteration is complete. Similarly, if the vertex replies that it cannot find a new non-recency incoming edge, then the algorithm will not again ask that vertex to try to find a new non-recency incoming edge in that iteration. This observation does not hold for $p = \frac{2}{3}$. Suppose that during the course of backtracking, some vertex $w$ is asked to provide an outgoing edge. To do that, vertex $w$ must find two new incoming edges. Suppose that $w$ attempts to find incoming edges from vertices $v$ and $x$. Vertex $v$, in turn, requests incoming edges from vertices $y$ and $z$. Suppose that $y$ and $z$ can both send edges to $v$, and so $v$ can send an edge to $w$, but vertex $x$ is unable to send an edge to $w$. Suppose further that although vertex $w$ can obtain an edge from $v$, it cannot obtain a new incoming edge from any other vertex. Since $w$ is unable to find two new incoming edges, the algorithm does not add edges $(y, v)$ and $(z, v)$ to the graph, even though it is possible to add them. During the same iteration of the algorithm, we might again ask vertex $v$ to obtain two new incoming edges. But at this point, we do not know if vertices $y$ and $z$ can still provide these edges, so we must query them again. This can happen many times, so we cannot put a polynomial bound on the running time of the algorithm.

## 5 Conclusion and Future Directions

In this paper, we have described a new problem: reconstructing social networks given information about how a property has spread through the network. To develop intuition about this problem, we considered a simple case corresponding to a property like information, where a vertex adopts the property after one of its neighbors adopts the property. We next considered a more difficult case in which a vertex adopts the property after at least $p = \frac{1}{2}$ of its neighbors adopt the property. We first described a flow-based algorithm that worked well for the case $p = \frac{1}{2}$ , but could not be generalized to other values of $p$. We thus considered the backtracking algorithm, in which details were made explicit. Finally, we discussed some difficulties in extending the backtracking algorithm to other models.

The model of contagion in which $p = \frac{1}{2}$ is relevant and important on its own: for instance, such a model would apply to networks in which each individual wishes to be in the majority, such as when people decide which social event to attend. However, there are certainly other models, and the strategies and difficulties described in this paper will be very important in solving these other problems. The next step is to find algorithms for other values of $p$. A promising observation suggests that for $p = \frac{1}{2}$, we may be able to limit the depth of the backtracking tree. Generalizing this to other values of $p$ may help us find an algorithm. Also, the problem as described cannot distinguish between vertices that adopt the property simultaneously. Thus, one extension is to consider multiple properties spreading through the network with different starting points or models of contagion. Instead of having a time vector, we would have a time matrix, where each vertex has a column of times, one for each property. We could also associate weights with edges, so that the heavier edges are more likely than the lighter edges. If we are considering the spread of a disease, then people who live far from one another may have a light edge between them, but people who are nearby might have a heavier edge between them. A different class of problems acknowledges that many different solution graphs may be possible, and then asks questions about the set of all solution graphs, such as: do any edges appear in all the graphs? which edges are the most likely to appear? do some edges appear in none of the graphs? When trying to reconstruct, for instance, a criminal network, we may not be able to recover the entire network with complete confidence, but knowing even parts of the network could prove useful for law enforcement officials. It is likely that many versions of this problem are NP-complete, and for those versions, we can attempt to create approximation algorithms.

There is much left to be done in this area, and we believe that the algorithms and issues presented here will provide a solid foundation for future work. Understanding the work described here is crucial for continued research.

## References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, “26. Maximum Flow” in *Introduction to Algorithms*, MIT Press and McGraw Hill, (2009), pp. 708-766.

[2] J. E. Gould, *Calderon Rejects ‘Absurd’ Reports on Mexico Drug War*, (2009). Bloomberg Database. 10 February, 2009.

[3] M. Granovetter, *Threshold Models of Collective Behavior*, Am. J. Sociol., 83.6 (1978), pp. 1420-1443.

[4] J. Kleinberg, “24. Cascading Behavior in Networks: Algorithmic and Economic Issues” in *Algorithmic Game Theory*, Cambridge Universisty Press, (2007).

[5] A. Krause, J. Leskovec, C. Guestrin, J. VanBriesen, and C. Faloutsos, *Efficient Sensor Placement Optimization for Securing Large Water Distribution Networks*, J. Water Res. Pl-ASCE, 134.6 (2008), pp. 516-526.

[6] V. E. Krebs, *Mapping Networks of Terrorist Cells*, Connections 24.3 (2001), pp. 43-52.

[7] J. C. Miller, J. M. Hyman, *Effective Vaccination Strategies for Realistic Social Networks*, Physica A, 386.2 (2007), pp. 780-785.

[8] J. Xu, H. Chen, *Criminal Network Analysis and Visualization*, Communications of the ACM, 48.6 (2005), pp. 100-107.

## Appendix

**Lemma 1.** *When showing that during the course of the backtracking algorithm there is always a valid backtracking sequence, if multiple outgoing edges are requested or incoming edges are deleted from a vertex, then it is possible for that vertex to obtain that many new incoming edges or delete that many existing outgoing edges, or some combination thereof.*

*Proof.* For Lemma 1, we need to show that it is possible to create the backtracking sequence above without adding any edge to the sequence multiple times. During the course of creating the backtracking sequence, some vertex $x$ may be encountered multiple times. Each time, an edge outgoing from $x$ may be requested by some later vertex, or an edge incoming to $x$ may be removed by some earlier vertex. The former type of edges are outgoing from $x$ in $S$ but not $G$, and the latter type of edges are incoming to $x$ in $G$ but not $S$. In response to either of these actions, $x$ must either find a new incoming edge or remove an existing outgoing edge. The edges that $x$ can request as a new incoming edge are those which are incoming to $x$ in $S$ but not $G$, and the outgoing edges that $x$ can remove are those outgoing from $x$ in $G$ but not $S$. We need to show that there are sufficiently many edges of the latter two types that we can respond to edges of the first two types without ever repeating an edge.

Define the following variables: Let $a_G$ be the number of edges incoming to $x$ in $G$ but not $S$, and $a_S$ be the number of edges incoming to $x$ in $S$ but not $G$. $a_{GS}$ is the number of edges incoming to $x$ in both $G$ and $S$. Define $b_G$, $b_S$, and $b_{GS}$ similarly, but for edges outgoing from $x$. Using these definitions and the argument in the previous paragraph, we need to show that $a_G + b_S \leq a_S + b_G$.

Consider the possible values of these variables. We have assumed that $S$ is minimal, so either $x$ in $S$ has the same number of incoming and outgoing edges, or it has exactly one incoming edge and zero outgoing edges. If neither of these is the case, then some edge incoming to $x$ could be removed and $S$ would still solve the time vector. The same holds for $G$. Any time an edge incoming to $x$ in $G$ is created, it is either to satisfy $x$'s recency condition or to allow the addition of an edge outgoing from $x$ on to some later vertex. In the first case, $x$ will have exactly one incoming edge and zero outgoing edges, and in the second case, $x$ has an equal number of incoming and outgoing edges. Thus, there are four cases we need to consider, based on the various combinations of the above possibilities for $x$ in $G$ and $S$.

**Case 1: $x$ has an equal number of incoming and outgoing edges in $G$ and $S$**
We need to show that $a_G + b_S \leq a_S + b_G$. Since vertex $x$ has an equal number of incoming and outgoing edges in both $G$ and $S$, we have $a_G + a_{GS} = b_G + b_{GS}$ and $a_S + a_{GS} = b_S + b_{GS}$. Subtracting the second equation from the first, we get $a_G - a_S = b_G - b_S$, and rearranging gives us the desired result $a_G + b_S = a_S + b_G$.

**Case 2: $x$ has an equal number of incoming and outgoing edges in $S$, $x$ has one incoming edge and zero outgoing edges in $G$**
We again need to show that $a_G + b_S \leq a_S + b_G$. In this case, vertex $x$ has an excess of incoming edges in $G$, so if an outgoing edge is ever requested from $x$ the backtracking sequence will be complete and $x$ does not need to find a new incoming edge or remove an outgoing edge in response. Because of this, we can assume that $b_S = 0$, so we need to show that $a_G \leq a_S + b_G$.

There are two sub-cases here: first, that $a_S = 1$ and $a_{GS} = 0$, and second, that $a_S = 0$ and $a_{GS} = 1$. Suppose the first sub-case holds, so $a_S = 1$ and $a_{GS} = 0$. We know that $a_G + a_{GS} = b_G + b_{GS}$, and have assumed that $a_{GS} = 0$. Because $x$ has no outgoing edges in $S$, $b_{GS} = 0$. Thus $a_G = b_G$, so certainly $a_G \leq a_S + b_G$. Suppose the second sub-case holds, so $a_S = 0$ and $a_{GS} = 1$. Since $a_S = 0$, we need to show that $a_G \leq b_G$. Since $a_{GS} = 1$ and $b_{GS} = 0$, and $a_G + a_{GS} = b_G + b_{GS}$, we get that $a_G + 1 = b_G$, so $a_G \leq b_G$.

**Case 3: $x$ has an equal number of incoming and outgoing edges in $G$, $x$ has one incoming edge and zero outgoing edges in $S$**
As in the last case, there are two sub-cases: first, that $a_S = 1$ and $a_{GS} = 0$, and second, that $a_{GS} = 1$ and $a_S = 0$. Suppose the first sub-case holds, so $a_S = 1$ and $a_{GS} = 0$. We need that $a_G + b_S \leq a_S + b_G$. Since $b_S = 0$, we need that $a_G \leq 1 + b_G$. Since $b_{GS} = 0$ and $a_{GS} = 0$, and $a_G + a_{GS} = b_G + b_{GS}$, we

get that $a_G = b_G$, so certainly $a_G \leq 1 + b_G$ and $a_G + b_S \leq a_S + b_G$. Now suppose we are in the second sub-case, so $a_{GS} = 1$ and $a_S = 0$. We again need that $a_G + b_S \leq a_S + b_G$. Since $b_S = 0$ and $a_S = 0$, we need that $a_G \leq b_G$. We know that $a_G + a_{GS} = b_G + b_{GS}$, so $a_G + 1 = b_G$, so then $a_G \leq b_G$ and so $a_G + b_S \leq a_S + b_G$.

**Case 4: $x$ has one incoming edge and zero outgoing edges in $G$ and $S$**
We need that $a_G + b_S \leq a_S + b_G$. In this case, $b_G$, $b_{GS}$, and $b_S$ are all zero, so we need that $a_G \leq a_S$. Since $x$ has one incoming edge in both $G$ and $S$, either $a_G = 1$, $a_S = 1$, and $a_{GS} = 0$, or $a_G = 0$, $a_S = 0$, and $a_{GS} = 1$. In either case, $a_G \leq a_S$, so $a_G + b_S \leq a_S + b_G$.

In each of the four cases, we showed that $a_G + b_S \leq a_S + b_G$. Thus, we are never forced to put an edge into the backtracking sequence multiple times. □

**Lemma 2.** *It is possible to eliminate cycles the backtracking sequence so that each vertex appears at most twice.*

*Proof.* For Lemma 2, we need to show that we can modify the backtracking sequence so that each vertex appears at most twice. Although the proof so far has described the backtracking sequence as a sequence of edges, here we will describe it as the equivalent sequence of vertices.

Suppose that a vertex $x$ appears multiple times in the sequence. There are two possibilities for each occurrence of $x$: first, $x$ can either obtain a new incoming edge or remove an existing outgoing edge, and second, if $x$ was deprived of its recency-condition edge, then $x$ must obtain a new incoming edge from the previous level. Now consider the first and last occurrences of $x$. There are four possibilities here.

**Case 1: In both the first and last occurrences, $x$ can either obtain a new incoming edge or remove an existing outgoing edge**
In this case, the edge added or removed in response to the last occurrence of $x$ could have been added or removed at the first occurrence of $x$, so we can simply remove this cycle from the sequence, leaving only one occurrence of $x$.

**Case 2: In both the first and last occurrences, $x$ must obtain a new incoming edge from the previous level**
This case is the similar to the last, and we can eliminate the cycle.

**Case 3: At the first occurrence of $x$, $x$ can obtain any new incoming edge or remove an existing outgoing edge, but at the last occurrence of $x$, $x$ must obtain a new incoming edge from the previous level**
In this case, the edge that the last occurrence of $x$ finds from the previous level could have been added at the first occurrence of $x$, so we can eliminate this cycle and leave one occurrence of $x$.

**Case 4: At the first occurrence of $x$, $x$ must obtain a new incoming edge from the previous level, but at the last occurrence of $x$, $x$ can obtain any new incoming edge or remove an outgoing edge.**
It's possible that at the last instance of $x$, $x$ did something other than finding a new incoming edge from the previous level, and this would not have been a valid action at the first occurrence of $x$. We cannot necessarily eliminate this cycle, so in this case, we may need to allow two occurrences of $x$.

If there are only two occurrences of $x$, we are done. Suppose there are more than two. Find the latest occurrence of $x$ which must obtain a new incoming edge from the previous level. Eliminate the portion of the backtracking sequence between this occurrence of $x$ and the first occurrence of $x$ (at which $x$ also must obtain a new incoming edge from the previous level), as we did in Case 2. Once this cycle is eliminated, there is only one occurrence of $x$ which must obtain a new incoming edge from the previous level.

In this new backtracking sequence, find the earliest occurrence of $x$ at which $x$ can either obtain any new incoming edge or remove an outgoing edge, and eliminate the portion of the backtracking sequence between this occurrence and the last occurrence of $x$ (at which $x$ can obtain any new incoming edge or remove an outgoing edge), as we did in Case 1. Now the backtracking sequence contains one occurrence of $x$ which must obtain a new incoming edge from the previous level, and one occurrence of $x$ which can obtain any new incoming edge or remove an outgoing edge, so there are two occurrences of $x$.

Thus, in all cases, we can modify the sequence so that $x$ appears at most twice. $\square$