Optimally Learning Social Networks with Activations and Suppressions

Dana Angluin[a], James Aspnes[a,1], Lev Reyzin[a,*,2]

[a]*Department of Computer Science, Yale University, New Haven, Connecticut, 06511*
`{angluin,aspnes}@cs.yale.edu, lev.reyzin@yale.edu`

Abstract

In this paper we consider the problem of learning hidden independent cascade social networks using exact value injection queries. These queries involve activating and suppressing agents in the target network. We develop an algorithm that optimally learns an arbitrary social network of size n using $O(n^2)$ queries, matching the information theoretic lower bound we prove for this problem. We also consider the case when the target social network forms a tree and show that the learning problem takes $\Theta(n \log(n))$ queries. We also give an approximation algorithm for finding an influential set of nodes in the network, without resorting to learning its structure. Finally, we discuss some limitations of our approach, and limitations of path-based methods, when non-exact value injection queries are used.

Key words: value injection queries, social networks, active learning

1. Introduction

Social networks are used to model interactions within populations of individuals. These interactions can include distributing information, spreading a disease, or passing trends among friends. Viral marketing is often used as an example of a process well modeled by social networks. A company may want to virally market a product to its potential clients. The idea is to carefully choose some influential people to target. This can be done, for instance, by giving these people a free sample of the product. The targeted people have relationships in their population, and the hope is that they will virally spread interest in this product to their friends, and so on.

There are many different models of social networks, and these models (imperfectly) approximate complicated real world phenomena. One of the most basic and well-studied models is the **independent cascade model** [7, 10, 11],

*Corresponding author
[1]Supported in part by NSF grant CNS-0435201.
[2]This material is based upon work supported under a National Science Foundation Graduate Research Fellowship.

and it is the one we consider in this paper. Informally, in the independent cascade model, each individual, or agent, has some probability of influencing each other agent. When an agent is targeted with a product, he becomes activated, and then attempts to influence each of his neighbors, and so on. This model is called independent cascade because each agent's success probability in attempting to influence another agent is independent of the history of previous activation attempts in the network.

Social networks belong to the wider class of probabilistic networks. Probabilistic networks are circuits whose gate functions specify, for each combination of inputs, a probability distribution on the output. In the case of social networks, these gates compute rather simple functions of their inputs.

A natural question to ask is: what can we learn about the structure of these networks by experimenting with their behavior? Given access to a pool of agents in our network, one intuitive way in which we could experiment on this network would be to artificially excite some set of agents, for example by sending them political brochures in support of some measure, and then observe the consequences of the experiment. Furthermore, we will allow for the possibility of suppressing agents; when an agent is suppressed, he cannot be excited by another agent. To make things more realistic, and theoretically more interesting, we will not assume that we can observe the entire network. We will instead have an output agent, whose state at the end of this process we can see, e.g. the probability the President supports the measure.

Thus, in this paper we consider the setting where we can **inject values** into the network; we fix the states (or values) of any subset of agents in the target network and observe only the state of some specified agent, whom we think of as the output of the network. This is the value injection query model.

The idea of value injection queries was inspired both from hardness results in learning circuits by manipulating only the inputs [6, 9, 12] and by models of gene suppression and gene overexpression in the study of gene interaction networks [1, 8] and was proposed by Angluin et al. [4]. They show that acyclic deterministic boolean circuits with constant fan-in and $O(\log n)$ depth are learnable in polynomial time with value injection queries. Angluin et al. [3] extend these results to circuits with polynomial-size alphabets. They show that transitively reduced acyclic deterministic circuits that have polynomial-size alphabets, constant fan-in, and no depth bound are learnable in polynomial time with value injection queries. Then, Angluin et al. [2] extend this work to probabilistic circuits. They show that constant fan-in acyclic boolean probabilistic circuits of $O(\log(n))$ depth can be approximately learned in polynomial time, but that this no longer necessarily holds once the alphabet becomes larger than boolean.

However, unlike in previous work on the value injection model, we allow our target social networks to have cycles. In many classes of networks, allowing for cycles would make the problem ill-defined in the value injection model, as the values on the nodes of the network may not be stable. In the social networks case, the values of the nodes in the network converge. Also, unlike in previous work, our learnability results do not require a degree bound on the target network. This gives us a nice theoretical model whose properties are interesting to explore.

In Section 2 we formally define the model, value injection queries, and learning criteria. In Section 3 we develop an algorithm that learns any social network in $O(n^2)$ queries and prove a matching lower bound for this problem. In Section 4 we show that in the special case when the network comes from the class of trees, learning the network takes $\Theta(n \log(n))$ queries. In Section 5 we show some limitations of using path-based methods for learning social networks when value injection queries do not return exact probability distributions of value of the output node, which is the case in real-world settings. In Section 6 we give an approximation algorithm for learning influential sets of nodes in a social network.

For a preliminary version of this paper, we direct the reader to [5].

2. Model

2.1. Social Networks

We consider a class of circuits that represent social networks. We are specifically interested in a variant of the model of deterministic circuits defined in [3, 4]. Social networks have no distinguished inputs – instead, value-injection experiments may be used to override the values on any subset of the agents.

An **independent cascade social network** S consists of a finite nonempty set of independent excitation agents A, one of which is designated as the **output agent**. Agents take values from a boolean alphabet $\Sigma = \{0, 1\}$, corresponding to the states *waiting* and *activated*, respectively. The size of the social network is $n = |A|$.

An **independent excitation** agent function f on k inputs is defined by k parameters: the probabilities $p_1, \ldots, p_k$. If the inputs to the agent are $(b_1, \ldots, b_k) \in \{0, 1\}^k$, then the probability that $f(b_1, \ldots, b_k)$ is 0 is

$$\prod_{i=1}^{k} (1 - p_i)^{b_i}.$$

We define $0^0 = 1$.

If we are told, in an arbitrary order, which inputs to f are 1, then we may sample from the correct output distribution for f as follows. Initially the output is 0. Given that $b_i = 1$, then with probability p_i we set the output to 1 and with probability $(1 - p_i)$ we leave it unchanged. This corresponds to our intuitive notion of the behavior of social networks; when a neighbor of an agent is activated, the agent has some probability of becoming activated as well, and an agent will remain inactive if it was not activated by any of its neighbors.

2.2. Graphs of Social Networks

The weighted **network graph** of the social network has vertices A and a directed edge (u, v) if agent u is one of the inputs of agent v. If u is an input to v with activation probability $p_{(u,v)}$, then the edge has weight $p_{(u,v)}$. We say an edge exists if it has positive weight. The weighted network graph of a social network captures all relevant information about the social network. Therefore, we will often refer to a social network in terms of its graph. The **depth** of a node in the network is the number of edges in the shortest path from the node to the output. The depth of the network is the maximum over the depths of all the nodes in the network. The network is **acyclic** if the network graph contains no directed cycles. Unlike in previous work on value injection queries, in this paper we consider networks that may have cycles.

2.3. Experiments

The behavior of a social network consists of its responses to all possible value-injection experiments. In an experiment, some agents are fixed to values from $[0, 1]$ and others are left free. Fixing an agent to a 1 corresponds to **activating** or **firing** the agent, fixing to a 0 corresponds to **suppressing** the agent, and leaving an agent **free** allows it to function as it normally would. Fixing an agent to a value c between 0 and 1 corresponds to firing the agent with probability c and suppressing it with probability $1 - c$.

Formally, a **value-injection experiment** (or just experiment) e is a mapping from A to $\{[0, 1] \cup \{*\}\}$. If $e(g)$ is $*$, then the experiment e leaves agent g **free**; otherwise g is **fixed** to the value $e(g) \in [0, 1]$. If e is any experiment and $a \in [0, 1] \cup \{*\}$, the experiment $e|_{w=a}$ is defined to be the experiment e' such that $e'(w) = a$ and $e'(u) = e(u)$ for all $u \in A$ such that $u \neq w$.

We can define the behavior of a social network S as a function of a value-injection experiment in two different ways. The first is a percolation model. For each edge (u, v), we leave it "open" with probability $p_{(u,v)}$ and "closed" with probability $(1 - p_{(u,v)})$. For each node w in S, such that $e(w) = c$ for some $c \in [0, 1]$, we make node w fired with probability c and suppressed with probability $1 - c$. We let the indicator variable $I = 1$ if there is direct path using open edges from some fired node to the output node via free nodes, and we let $I = 0$ otherwise. This determines a probability distribution on assignments of 0 and 1 to I. We define the output $S(e)$ to be $E(I)$.

The following process, equivalent to the percolation model, defines the behavior of social network as a function of a value-injection experiment e. It is also the process that will guide the intuition and proofs in this paper. Initially every node is tentatively assigned the value 0. There is a queue of nodes to be assigned values, which initially contains the nodes fixed to values > 0 by e. The assignments are complete when the queue becomes empty. While the queue is nonempty, its first node v is dequeued. If $e(v) = *$, v is assigned the value 1. If $e(v) \neq *$, v is assigned a 1 with probability $e(v)$, and 0 with probability $(1 - e(v))$. If v is assigned a 1, for every node u such that v is an input to u, do the following.

1. If u is fixed to any value, or already assigned 1 or present in the queue, do nothing.
2. Otherwise, with probability $p_{(v,u)}$ add u to the queue, and with probability $(1 - p_{(v,u)})$ do nothing.

This process determines a joint probability distribution on assignments of 0 and 1 to the nodes of the social network S. In this case, the output $S(e)$ is the expected value of the output node given by e.

2.4. *Example:* S_1

We give a simple example of a social network. We define a network S_1 of 4 agents. We give the adjacency matrix of the network graph, labeling the agents associated with the nodes. Figure 1 shows the social network defined by the adjacency matrix.

$$\begin{array}{c|cccc} & a_1 & a_2 & a_3 & a_4 \\ \hline a_1 & - & .5 & 0 & 0 \\ a_2 & 0 & - & .5 & 1 \\ a_3 & 1 & 0 & - & .5 \\ a_4 & 0 & 0 & .3 & - \end{array}$$

The output agent is a_4.

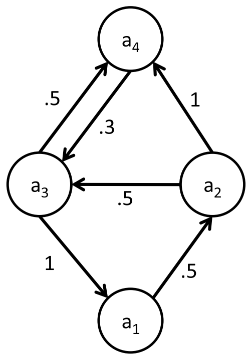

Figure 1: An illustration of the circuit S_1.

We first make an observation. Edge (a_4, a_3) has weight .3, meaning that a_4, when activated has a probability of .3 of activating a_3, if a_3 has not already been activated. However, because a_4 is the output, the weight of (a_4, a_3) does not affect any value injection experiment. It follows that no sequence of injection queries can learn the weight of (a_4, a_3).

We now consider the experiment e that leaves a_4 and a_2 free, suppresses a_3 (sets $a_3 = 0$) and activates a_1 ($a_1 = 1$). We wish to compute the output distribution $S_1(e)$. Because a_1 has only one outgoing edge of weight .5 to a_2, he activates agent a_2 with probability .5. a_2 has an edge to a_3, but a_3 is suppressed, so that edge has no effect. a_2 also has an edge of weight 1 to a_4, the output. So, whenever a_2 is active, he will activate a_4, and we have observed a_2 is active with probability $\frac{1}{2}$. So the output distribution $S_1(e)$ is an unbiased coin flip.

2.5. Behavior and Equivalence

The **behavior** of a network is the function that maps experiments e to output excitation probabilities $S(e)$. Two social networks S and S' are **behaviorally equivalent** if they have the same set of agents, the same output agent, and the same behavior, that is, if for every value-injection experiment e, $S(e) = S'(e)$. We also define a concept of approximate equivalence. For $\epsilon \geq 0$, S is **ϵ-behaviorally equivalent** to S' if they contain the same agents, the same output agent and for every value-injection experiment e, $|S(e) - S'(e)| \leq \epsilon$.

2.6. Queries

The learning algorithm gets information about the target network by repeatedly specifying an experiment e and observing the value assigned to the output node. Such an action is termed a **value injection query**. A value-injection query does not return $S(e)$, but instead returns a $\{0, 1\}$ value selected according to the probability $S(e)$. This means that the learner must repeatedly sample to approximate $S(e)$. To separate the effects of this approximation from the inherent information requirements of this problem, we define an **exact value injection query** to return $S(e)$. The focus of this paper is on exact value injection queries.

2.7. The Learning Problem

The learning problem we consider is: by making exact value injection queries to a target network S drawn from a known class of social networks, find a network S' that is behaviorally equivalent to S. The inputs to the learning algorithm are the names of the agents in S and the name of the output agent.

To help with terminology, let S be a social network. Let S' be any social network that differs only in edge (u, v). We say edge (u, v) is **discoverable** for S if there exists an experiment e such that $S(e) \neq S'(e)$. Otherwise we say that the edge is not discoverable. We could also view the learning problem in terms of finding the discoverable edges and their probabilities.

2.8. A Note on the Generality of this Model

The model introduced in this section allows for the observation of the network by looking at the output of one selected node. However, this model is surprisingly general. One may wish to consider, for example, the ability to observe the number of nodes to fire as a result of an experiment. Such a scenario could be simulated in our model – given any social network, one could make a new output node that is activated by each node with some fixed, chosen probability. Now the probability the output is activated corresponds to the number of network nodes that are activated in an experiment.

One could also imagine networks where some nodes spontaneously fire with some probability. We can again simulate this in the model we introduced. We add a node that is fired with probability 1 whenever any node in the network fires (all other nodes have 1-edges to the new node), and the new node can have edges to each node in the network, with probabilities corresponding to the desired spontaneous firing probabilities of the network nodes.

3. General Social Networks

In this section we prove the following theorem.

Theorem 1. *Any social network with n agents can be learned up to behavioral equivalence with $O(n^2)$ exact value injection queries and time polynomial in the number of queries.*

Before considering the case of arbitrary social networks, we begin by developing an algorithm that learns social networks that do not have edges of weight 1, to behavioral equivalence.

3.1. No Probability 1 *Edges*

First, we develop excitation paths, which are a variant of test paths, a concept central in previous work on learning deterministic circuits [3, 4]. An **excitation path** for an agent a is a value-injection experiment in which a subset of the free agents form a simple directed path[3] in the circuit graph from (but not including) a to the output agent. All agents not on the path with inputs into agents (excluding a) on the path are fixed to 0. A **shortest excitation path** is an excitation path of length equal to the depth of a.

Let G be the network graph of A. In G, the **up edges** are edges from nodes of larger depth to nodes of smaller depth, **level edges** are edges among nodes of the same depth, and **down edges** are edges from nodes of smaller depth to nodes of larger depth. An edge (u, v) is a **shortcut edge** if there exists a directed path in G of length at least two from u to v.

Lemma 2. *Let e be a shortest excitation path for node a and π be the nodes on the path. Let $p_1 \cdots p_k$ be the weights of the up edges in $\pi \cup a$. Then for $0 \leq c \leq 1$*

$$S(e|_{a=c}) = c\prod_{i=1}^{k} p_i.$$

Proof. In a shortest excitation path, if some node on the path does not activate, no node at smaller depth will activate, because a shortest excitation path cannot have shortcuts to nodes further along the path. Hence, all up edges must fire to fire the output. This happens exactly with probability $\prod_{i=1}^{k} p_i$. $\square$

We note that Lemma 2 still holds when a takes probability c, not only when it is set to c by e.

Lemma 3. *Let e be an excitation path experiment for node v and let $\pi = v_k, \ldots, v_0$ be the nodes along π in order from v to the output (with v_0 being the output node), such that there are no shortcut edges (v_i, v_j) for $j < i$ along π. Let $u \notin \pi$ be a node such that all edges from u to nodes on π are known and have weights < 1. Let $e' = e|_{v=*,u=1}$. Then, given $S(e')$ we can compute $p_{(u,v)}$.*

[3]A path with no repeated vertices is called simple.

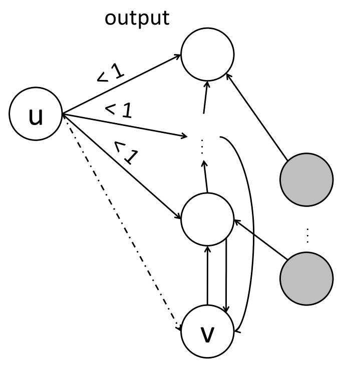

Figure 2: An illustration for Lemma 3. The shaded nodes are suppressed. The solid edges are known and the dashed edge is the edge to be computed.

Proof. We can see this situation illustrated in Figure 2. We observe that because there are no shortcuts along π, no node v_i will activate in e' unless either u activated it, or v_{i+1}. Hence, any edge (v_j, v_k) where $j < k$ does not affect $S(e')$. Therefore, we can compute $S(e')$ by summing over all the ways v_0 can activate. Either u activates it directly with probability $p_{(u,v_0)}$, or if not (with probability $1 - p_{(u,v_0)}$) we look at the probability u activates v_1 and the probability of v_1 firing the output, and so on. These quantities can be computed using the logic of Lemma 2. For the calculation below, we rename node v to v_{k+1}.

$$S(e') = \sum_{i=0}^{k+1} \left(p_{(u,v_i)} \prod_{j<i} (1 - p_{(u,v_j)})(p_{(v_{j+1},v_j)}) \right)$$

This equation is linear in $p_{(u,v_{k+1})}$, which we can solve for because the other quantities are known. □

We present an algorithm for learning social networks that do not contain edges of weight 1, Algorithm 1. We then show the conditions for an edge in the network to be learnable and analyze the running time of the algorithm.

The subroutine **Find-Up-Edges** builds a leveled graph of S. Let **level** i be the set of all nodes at depth i. Find-Up-Edges assigns each node to a level and finds all up edges in the graph. Starting at the top level and proceeding downward, for each pair of nodes u and v, such that u is one level deeper than v, Find-Up-Edges finds a shortest excitation path for u that goes through v to learn $p_{(u,v)}$. This experiment leaves the path free and suppresses all other nodes in the graph. We show correctness by induction on the level. For the base case, the edges from nodes at depth 1 form the paths. Considering nodes at depth i we assume we know all up edges on the induced subgraph at depths 0 to $i-1$. Therefore, for each node at depth $i-1$ we have a shortest excitation path to the root. Thus, for each node u not yet assigned a level, we can try experiments with excitation paths via each node v at depth $i-1$. Let e be such

Algorithm 1 Learning Social Networks without Edges of Weight 1

```
Let $S$ be the target social network.
Initialize $G$ to have the agents as vertices and no edges.
Run **Find-Up-Edges** to learn the leveled graph of $S$.
Add learned weighted edges to $G$.
for Each level in the graph do
  Run **Find-Level-Edges** to learn all level edges.
  Add the learned weighted edges to $G$
end for
Let the complete set $C = \emptyset$
for Each level $i$, from the deepest to the output do
  Run **Find-Down-Edges($G$,$C$,$i$)** to learn the down edges from that level.
  Add all nodes at the current level to $C$.
  Add the learned weighted edges to $G$.
end for
Output $G$ and halt.
```

an experiment with π as the excitation path. And let $p_1 \cdots p_{i-1}$ be the weights of the up edges in π. By Lemma 2 we can compute

$$p_{(u,v)} = \frac{S(e|_{u=1})}{\prod_{j=1}^{i-1} p_j}.$$

If $p_{(u,v)} > 0$ we assign node u to level i.

The subroutine **Find-Level-Edges** finds edges among nodes at the same depth. It again uses the notion of shortest excitation paths. Let nodes u and v be at depth i. To find $p_{(u,v)}$, the algorithm first finds any shortest excitation path from v to the output; suppose it passes through node w at depth $i-1$. Let e_1 be that experiment. Let $e_2 = e_1|_{u=1,v=*}$. From Find-Up-Edges, we know $p_{(u,w)}$ and $p_{(v,w)}$, and because all nodes on the shortest excitation path from w are at depth $\leq i-2$, we know e_2 is a shortest excitation path for w. Let $p_1 \ldots p_k$ be the weights of the up edges on this path. By performing $S(e_2)$, by Lemma 2, we can compute p_w, the probability that $w = 1$

$$p_w = \frac{S(e_2)}{\prod_{i=1}^{k} p_i}.$$

Because u (fired) and v (free) are the remaining unsuppressed nodes in S, given $p_{(u,w)}$ and $p_{(v,w)}$ we can compute $p_{(u,v)}$[4]

$$p_w = p_{(u,w)} + (1 - p_{(u,w)}) p_{(u,v)} p_{(v,w)}$$

$$p_{(u,v)} = \frac{p_w - p_{(u,w)}}{p_{(v,w)}(1 - p_{(u,w)})}.$$

[4]We note that this computation is a special case of Lemma 3.

Finally, the subroutine **Find-Down-Edges** finds down edges in the graph. By this point, the graph has the entire set of up and level edges. The idea of Find-Down-Edges is to find all down edges, with their sources starting from the deepest nodes, working up towards the root. The algorithm keeps a *complete set* C of nodes, among which all discoverable edges are known. Let the deepest node in the network be at depth d, we say that the C has height i if contains all nodes at depth greater than $d - i$. Find-Down-Edges grows the complete set, one level at a time, towards the root.

Algorithm 2 The Subroutine Find-Down-Edges from Algorithm 1 (Current Graph G, Complete Set C, Level i)

```
for Each node $u$ at the current level $i$ {find all down edges to C} do
  Sort each node in C by distance to the root in $G - \{u\}$.
  Let $v_1, \dots, v_k \in C$ sorted by increasing distance.
  Let $\pi_1, \dots, \pi_k$ be shortest paths for $v_1, \dots, v_k$ resp. in $G - \{ u\}$.
  for Node $v_j$ from $v_1$ to $v_k$ do
    Perform experiment $e_j$ of firing $u$, leaving $\pi_j$ free, and suppressing the
    rest of the nodes.
    Query $S(e_j)$. Compute by Lemma 3 the weight of $p_{(u,v)}$.
    Add $(u, v)$ to $G$ if $p_{u,v)} > 0$.
  end for
end for
```

We restate the algorithm and give an inductive proof of its correctness. We do induction on the height of the complete set. The base case contains all nodes at depth d. They, by definition, cannot have down edges, and since we know all of their level edges from Find-Level-Edges, they form a complete set. For the inductive step, we assume all nodes at depth $> i$ form a complete set, and the goal is to find all down edges to them from nodes at depth i. Let L be the set of nodes on level i. Let u be a node in L. For each node v_j in the complete set, the algorithm first finds the distance from v_j to the root in $G - \{u\}$ and π_j the corresponding shortest path, Let $v_1 \cdots v_k$ be the vertices in the complete set, sorted smallest to largest by this distance. Now, the edges $(u, v_1), (u, v_2), \cdots, (u, v_k)$ can be found in that order. We show this by induction.

We first show the inductive step. To test for the existence of (u, v_j), we perform the experiment e_j of firing u, leaving π_j free, and suppressing the rest of the nodes. We note that all nodes on π_j have a smaller depth than v_j, so all down edges from u to π_j are known by the time the algorithm gets to v_j, and all up edges along π are known. Because π_j is a shortest path in $G - \{u\}$, it clearly has no shortcuts. Hence, by Lemma 3 the weight of (u, v_j) can be computed given $S(e_j)$. The base case is done similarly. This completes the inductive proof of finding down edges, which completes the proof of growing the complete set.

We can now summarize the conditions for finding an edge. Find-Up-Edges and Find-Level-Edges discover all up and level edges as long as they are connected to the output. Find-Down-Edges finds all down edges that have a path to the output that doesn't use the source node. If every path from u to the output agent that starts with edge (u, v) goes through u, then edge (u, v) is not discoverable. We can see this because if the edge (u, v) activates v, it must mean that u has already fired, and because all paths from v go through u, the edge firing will not affect the output. Therefore, the edges this algorithm does not learn are not discoverable.

3.2. Arbitrary Social Networks

We now extend the ideas in Algorithm 1 to allow for edges of weight 1, giving us Algorithm 3. This algorithm is similar to Algorithm 1, except that Find-Level-Edges and Find-Down-Edges are combined into Find-Remaining-Edges. Algorithm 3 first builds a leveled graph of the social network as before, and the justification for Find-Up-Edges can be found in Section 3.1.

Algorithm 3 Learning Arbitrary Social Networks

```
Let $S$ be the target social network.
Initialize $G$ to have the agents as vertices and no edges.
Run **Find-Up-Edges** to learn the leveled graph of $S$.
Add learned weighted edges to $G$.
Let $C = \emptyset$ be the complete set.
**for** Each level $i$ in the graph, from the deepest level to the output node **do**
  Run **Find-Remaining-Edges**($G$,$C$,$i$) to learn all level and down edges.
  Add all nodes at the current level to $C$.
**end for**
Output $G$ and halt.
```

After Find-Up-Edges is run, the remaining edges that need to be found are down and level edges. The subroutine **Find-Remaining-Edges**, shown in Algorithm 4, accomplishes this task. Find-Remaining-Edges is similar to Find-Down-Edges. The algorithm once again keeps a complete set C in which all discoverable edges are known. C starts at the largest level and grows toward smaller levels. Find-Remaining-Edges finds all discoverable edges from the level it is on to the complete set. It also finds all discoverable edges between nodes at the level it is on. Then, that level is added to C.

Let L be the set of nodes on level i. To find down and level edges from nodes in L, Find-Remaining-Edges keeps a table T, with an entry for each possible edge originating from a node in L. Each entry is initially set to 1. After determining whether an edge is present, its corresponding entry becomes marked 0. The potential edges whose corresponding entries are marked 1 we call "unprocessed."

For each unprocessed edge (u, v), we find the set of all nodes we know are guaranteed to be activated when u is fired. This is the set of nodes reachable by edges of weight 1 from u in G. We call this set A_u. Now, we find the shortest path $\pi_{u,v}$ (if one exists) in $G - \{A_u\}$ from v to the output. If no unprocessed edge has such a path, then Find-Remaining-Edges terminates and the algorithm proceeds to the next level.

Otherwise, we take an edge (u, v) that minimizes the distance from v to the output in $G - \{A_u\}$. Let e be the experiment where u is fired, all nodes along $\pi_{u,v}$ are left free, and the rest of the nodes are suppressed. We will show that $S(e)$ is enough to determine $p_{(u,v)}$. Then, the entry for this edge is marked to 0 in the table, and if it is present, is added to G. Then the algorithm continues, recomputing the sets A_u for the remaining unprocessed edges.

Algorithm 4 Find-Remaining-Edges(Current Graph G, Complete Set C, Level i)

```
Let $L$ be the set of nodes at the current level $i$.
Let $M = L \cup C$.
Let $\Pi$ be a collection of paths.
Keep an $|L|$ by $|M|$ table $T$. $\forall\ w_i \in L$, $x_j \in M$ s.t. $w_i \neq x_j$, $T(w_i, x_j) = 1$.
loop
  Set $\Pi = \emptyset$.
  for Each node $w_i \in L$ do
    Find $A_{w_i}$, the set of all nodes reachable from $w_i$ by 1-edges (incl. $w_i$) in $G$.
    for Each node $x_j \in M$ where $T(w_i, x_j) = 1$ do
      Find the shortest path $\pi_{w_i,x_j}$ in $G - A_{w_i}$ from $x_j$ to the root.
      $\Pi = \Pi \cup \pi_{w_i,x_j}$.
    end for
  end for
  if $\Pi = \emptyset$ then return.
  Let $w_i, x_j$ minimize the length of $\pi_{w_i,x_j} \in \Pi$.
  Let experiment $e$ fire $w_i$, leave $\pi_{w_i,x_j}$ free, and suppress the rest of the nodes.
  Query $S(e)$ and compute $p_{(w_i,x_j)}$ by Lemma 3.
  Set $T(w_i, x_j) = 0$.
  If $p_{(w_i,x_j)} > 0$ then add $(w_i, x_j)$ to $G$.
end loop
```

We now show that the value of $S(e)$, as defined above, is sufficient to learn edge (u, v). All edges from u to π are either up edges or have already been processed by the time edge (u, v) is considered, otherwise there would be an unprocessed edge from u to a node on π with a shorter distance to the root in $G - A_u$. All edges on π in $G - C$ are known from Find-Up-Edges, and the rest of the edges are known because they are in C. Hence, by Lemma 3, we can compute the weight of edge (u, v), and add it to G if its weight is positive.

Find-Remaining-Edges returns when all remaining unprocessed down and level pairs of nodes u, v do not have a path from v to the root in $G - A_u$. The algorithm does not attempt to learn these edges. We will argue that when an execution of Find-Remaining-Edges terminates, all of the unprocessed edges are not discoverable. Let u, v be such a pair. Let S be the graph of the complete social network and B_u be the set of nodes reachable by edges of weight 1 in S. If there is no path from v to the root in $S - B_u$, edge (u, v) is clearly not discoverable. We note that $A_u \subseteq B_u$.

By way of contradiction, we will assume there exist vertices u (on level i) and v (on level $\geq i$) such that there is a path of discoverable edges from v to the root in $S - B_u$ but not in $G - A_u$ at the time Find-Remaining-Edge exits. Once this path reaches level $i - 1$ in G, then the path can be continued by following up edges to the root. By assumption, G has all discoverable edges among the complete set C, which contains all nodes at levels $> i$. Hence, there must be some smallest set of edges U going from nodes at level i, that are in S but not in G, such that if they were added to G, then then there would be a path from v to the root node in $G - A_u$. All of the edges in U must lie on a path π. Let edge $(x, y) \in U$ be the unprocessed edge closest to the root along the path. Because edge (x, y) was unprocessed, there was a path of 1 edges from x to a node in π above y; otherwise, there would be a path from y to the root in $G - A_x$ and (x, y) would have been processed. But taking the path of 1 edges from x to a node in π gives a path from v to the root in $G - A_u$ using one fewer unprocessed edge. This contradicts that U was the smallest set of edges that, if added to G, would make a path from v to the root in $G - A_u$. This contradicts our assumption that a discoverable edge exists that Find-Remaining-Edges does not find.

To analyze number of queries used, we observe that every query either confirms the absence of an edge or discovers one. Hence, Algorithm 3 performs at most $O(n^2)$ queries.

3.3. A Matching $\Omega(n^2)$ Lower Bound

We show an information theoretic lower bound for learning social networks that matches the bound of the algorithm.

Theorem 4. *$\Omega(n^2)$ queries are required to learn a social network.*

Proof. We give an information theoretic lower bound. We consider the following class of graphs on vertices $\{v_1, \ldots, v_{2n+1}\}$. We let v_{2n+1} be the output. The edges $(v_{n+1}, v_{2n+1}), (v_{n+2}, v_{2n+1}), \ldots, (v_{2n}, v_{2n+1})$ all have weight 1. The edges (v_1, v_{n+1}), (v_2, v_{n+2}), $\ldots$, (v_n, v_{2n}) also all have weight 1. For $1 \leq i \leq n$, $n + 1 \leq j \leq 2n$, and $j \neq i + n$, each edge (v_i, v_j) is either present with weight 1 or absent. The rest of the edges are absent. There are $2^{\Omega(n^2)}$ such graphs and the answer to every exact value injection query is 1 bit because all present edges have weight 1. Algorithm 3 differentiates all graphs in this class because all edges in this class of graphs are up edges and are therefore discoverable. Hence, by an information theoretic lower bound, at least $\log 2^{\Omega(n^2)} = \Omega(n^2)$ queries are needed. $\square$

4. Trees

In this section, we will consider the special case in which the target social networks come from the class of trees. A **tree social network** is a social network whose edges are up edges that form a tree.

Theorem 5. *Learning a social network tree takes* $\Theta(n \log n)$ *exact value injection queries.*

Proof. We first show the lower bound. Consider a directed path of nodes, with the output node at an endpoint. All edges along the path have probability 1. The only unknown is the ordering of the nodes along the path. Let u and v be two nodes. We can test which of the two nodes has a smaller distance to the root by the experiment that fires u and suppresses v. If this fires the output, then u is closer to the root; otherwise, v is closer. Hence, all orderings can be distinguished. Because all edges have probability one, the result of any experiment is deterministically a 1 or 0, a 1-bit answer. There are $n!$ orderings of nodes. This gives an $\Omega(\log(n!)) = \Omega(n \log(n))$ information-theoretic lower bound.

We now develop an algorithm that meets this bound for trees. Let T be the target tree social network. In a tree, an **ancestor** of node u is any node on the path from u to the output. We can test whether node v is an ancestor of node u by firing u and suppressing v. If the result is > 0, then v is not an ancestor of u. In general, to test whether there exists some node in V that is an ancestor of u, we can fire u and suppress all nodes in V. This allows us to find all k ancestors of a given node u by binary search in $O(k \log(n))$ queries. Because the ancestors of u form a path, we can sort them by their depth using $O(k \log(k))$ queries (an ancestor test involving two nodes provides a comparator) to get a directed path from u to the output.

Now, we will use the observation above to make an algorithm for reconstructing trees. We keep a graph T' that is a connected subgraph of T that we build up by adding new nodes until T' contains all the vertices in T. In attaching a new node u to T', we first determine v, u's deepest ancestor in T'. We can do this by recursively by splitting the nodes in T' into roughly equal halves H_1 and H_2 such that no node in H_2 is an ancestor of a node in H_1. In one query we can test whether v is in H_1 by suppressing all nodes in H_2 and firing u; thus, we can find v in $\log(n)$ queries. We then find, by binary search, the set of all ancestors of u in T that are not in T', and we sort them by their distance to the root in T. This gives a path of vertices from u to v that we can append to T' and continue this process until all the vertices are added to T'.

In adding a new node u to T' we spend $O(\log(n))$ queries to find its deepest ancestor in T', and $O(k \log(n))$ queries to add u's $k \geq 0$ newly found ancestors to T'. This costs us an amortized $O(\log(n))$ queries per node, giving an $O(n \log(n))$ algorithm for learning the structure of the tree. We note that the structure is learned using just zero/non-zero information from the queries.

Finally, to learn the weights of the edges in the tree, because we have a shortest excitation path for each node, the edge weights can be discovered in n

queries by Lemma 2. □

5. Limitations of Excitation Paths

In this section, we construct a family of social networks in which there exists a node, that when fired, activates the output node with high probability, but any excitation path experiment for that node has an exponentially small probability of activating the output. Namely, we will prove the following theorem.

Theorem 6. *There exists a family of social networks S for which there exists a node $v \in S$ and an experiment e where only v is fired, such that for any excitation path experiment e_π for v,*

$$S(e) = 2^{\Omega(\sqrt{n})} S(e_\pi)$$

Proof. Let $\{v_1, \cdots, v_n\}$ be a set of nodes in this network, with v_1 the output node. For all $1 < i < n-1$, let $p_{(i,i+1)} = 1$; we call these **back edges**. For all $i, j > 0$ such that $i + j \leq n$, create a new node w_{ij} and let $p_{(w_{ij}, v_i)} = 1$ and **forward edges** $p_{(v_{i+j}, w_{ij})} = 2^{-j/\sqrt{n}}$. This is illustrated in Figure 3.

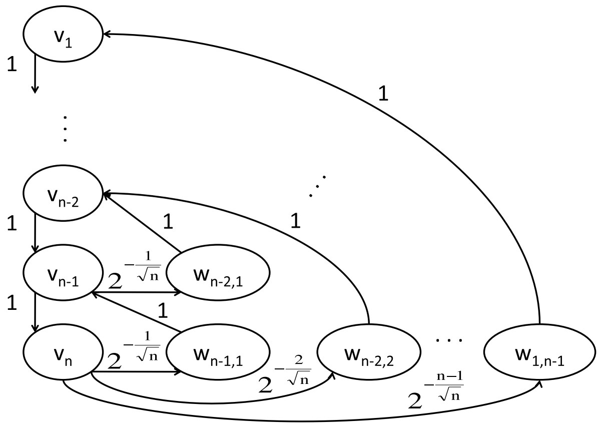
v1
1
1
1
1
1
1
vn-2
vn-1
vn
wn-2,1
wn-1,1
wn-2,2
w1,n-1
$2^{-\frac{1}{\sqrt{n}}}$
$2^{-\frac{1}{\sqrt{n}}}$
$2^{-\frac{2}{\sqrt{n}}}$
$2^{-\frac{n-1}{\sqrt{n}}}$

Figure 3: The social network S showing the limitations of excitation paths.

Let e_1 be an excitation path experiment for v_n, where v_n is fired. Let $S(e_1)$ be the probability all edges along the path fire. If e_1 uses k forward edges that decrease the distance to the output by $f_1, \ldots, f_k$, respectively (we note that $\sum f_i \geq n-1$), then

$$\begin{aligned} S(e_1) &= \prod_{i=1}^{k} 2^{-f_i/\sqrt{n}} \\ &= 2^{-\frac{\sum f_i}{\sqrt{n}}} \\ &= 2^{-\Omega(\sqrt{n})}. \end{aligned}$$

Let e_2 be the experiment where v_n is fired and the remaining agents are set free. We will show there exists a constant $c > 0$ such that $S(e_2) \geq c$.

We consider e_2. The probability that v_n does not fire any other nodes is $\prod_{i=1}^{n-1}\left(1 - 2^{-i/\sqrt{n}}\right)$. Now, we can bound the probability of the root firing. Let $T(i)$ be the probability the root becomes activated given v_i has fired. We set up a recurrence

$$\begin{aligned} T(1) &= 1 \\ T(n) &\geq \left(1 - \prod_{i=1}^{n-1}\left(1 - \frac{1}{2}^{i/\sqrt{n}}\right)\right) T(n-1) \end{aligned}$$

where we have an inequality above because if v_n activates any other node, then v_{n-1} becomes activated due to the back edges.

Thus, $T(n) \geq \left(1 - \frac{1}{2}^{\sqrt{n}}\right) T(n-1)$ because the first $\sqrt{n}$ terms of the product above are $\leq 1/2$. Unraveling the recurrence, we get

$$T(n) \geq \prod_{i=1}^{n}\left(1 - \frac{1}{2}^{\sqrt{i}}\right).$$

We know $\lim_{n\to\infty} T(n) > 0$ if $\sum_{i=1}^{\infty} \frac{1}{2}^{\sqrt{i}}$ converges. By the Cauchy Condensation Test, $\sum_{i=1}^{\infty} \frac{1}{2}^{\sqrt{i}}$ converges if and only if $\sum_{i=1}^{\infty} 2^n \frac{1}{2}^{\sqrt{2^n}}$ converges [15]. The ratio test easily tells us that $\sum_{i=1}^{\infty} 2^n \frac{1}{2}^{\sqrt{2^n}}$ converges. Therefore, there exists a constant $c > 0$ such that $\forall\ n\ T(n) \geq c$. $\square$

This example shows that many paths, each of which has an exponentially small effect on the output, can add up to have a detectable effect on the output. When using non-exact value injection queries, the goal is to learn a circuit to approximate behavioral equivalence. Yet this example shows us that if the learner has access only to non-exact value injection queries, then to learn this circuit by only path based methods like our algorithms do, one would need an exponential number of experiments to detect the effect on the output. This implies that for non-exact value injection queries, either the circuits would need a depth limitation, or non path-based algorithms would need to be developed.

6. Finding Small Influential Sets of Nodes

We now examine a seemingly easier problem. Instead of learning the entire social network, we consider the task of finding a small set of influential nodes. More formally, let $I \subset V$ such that $v_n \notin I$, and let e_I be the experiment where all nodes in I are fired and the rest are left free. I has **influence** p if $S(e_I) \geq p$; we call such a set **influential**. We first show that it is NP-Hard to find the smallest set of certain influence, even if the structure of the network is known.

Theorem 7. *Given a social network S of size n and a threshold probability p, it is NP-Hard to approximate the size of the smallest set of nodes having influence p within $o(\log(n))$.*

Proof. We reduce from Set Cover. Take an instance of Set Cover with points $\{x_1, \ldots, x_k\}$ and sets $\{X_1, \ldots, X_l\}$. In the social network S, we create a nodes $\{v_1, \ldots, v_k\}$ for the points and $\{w_1, \ldots, w_l\}$ for the sets in the original Set Cover instance. If point x_i belongs to set X_j, we make an edge from w_j to v_i with associated probability of 1. We set the influence threshold parameter p to $\frac{1}{2}$. We run edges from all nodes v_i to the output, all with associated probability $= 1 - \frac{1}{2}^{1/k}$. Activating a node w_i corresponds to choosing the set X_i and activating a node v_i corresponds to choosing an arbitrary set X_j that contains x_i. The output will fire with probability $\geq \frac{1}{2}$ only if all of the v_i's fire. This completes the reduction. Because Set Cover is NP-Hard to approximate to within $o(\log n)$ [14], so is approximating the size of the smallest influential set. $\square$

Theorem 8. *Let S be a social network of size n and let I be the smallest set of nodes having influence p, where $m = |I|$. We can find a set of nodes of size $m \log(p/\epsilon)$ of influence $(p - \epsilon)$ using $O(nm \log(p/\epsilon))$ exact value injection queries.*

Proof. Consider Algorithm 5.

Algorithm 5 An Algorithm for Finding a Set of Influential Nodes

```
Let $S$ be the target social network.
Let $p$ be the threshold probability.
Let $\epsilon$ be the error tolerance.
Let $I = \emptyset$.
Let $e_I$ be the experiment where all nodes in $I$ are fired, and the rest are left free.
while $S(e_I) < p - \epsilon$ do
  Let $v = \arg\max_{v_j \in V} S(e_I|_{v_j=1})$
  $I = I \cup \{v\}$
end while
Return $I$
```

Assume the optimal solution X, where $S(e_X) \geq p$, has size m. We claim that at any stage of the algorithm, if $S(e_I) < p - \epsilon$, greedily adding one more node w to I makes

$$S(e_{I \cup \{w\}}) \geq S(e_I) + \frac{p - S(e_I)}{m}.$$

We can see this by noting that there exists a set of at most m nodes, namely X, that will get the probability all the way to p. By Lemma 9, some node will get us at least $\frac{1}{m}$th of the way there.

Let k be the number of rounds this algorithm is run. We look at the difference between p and $S(e_I)$ after k rounds. By the observation above, we can compute the number of rounds to get the difference to within ϵ. For

$$p\left(1-\frac{1}{m}\right)^k < \epsilon$$

it suffices that

$$e^{-\frac{k}{m}} < \frac{\epsilon}{p}$$

or

$$k > m\log\left(\frac{p}{\epsilon}\right).$$

Therefore, after $m\log(\frac{p}{\epsilon})$ rounds, $S(e_I)$ is within ϵ of p. We check $O(n)$ nodes each round, making for $O(nm\log(\frac{p}{\epsilon}))$ queries. $\square$

We now reconcile the algorithm and the hardness of approximation result. Given a social network created by the Set Cover reduction from Theorem 7, we can try to learn the influential nodes using Algorithm 5. If we set

$$\begin{aligned}
\epsilon &= \frac{1}{2}^{\frac{1}{n}} - \frac{1}{2}^{\frac{1}{n-1}} \\
&= \left(1 + \frac{\ln(1/2)}{n} + \frac{1}{2}\left(\frac{\ln(1/2)}{n}\right)^2 + \dots\right) - \left(1 + \frac{\ln(1/2)}{n-1} + \frac{1}{2}\left(\frac{\ln(1/2)}{n-1}\right)^2 + \dots\right) \\
&= (1-1) + \left(\frac{\ln(1/2)}{n} - \frac{\ln(1/2)}{n-1}\right) + \frac{1}{6}\left(\left(\frac{\ln(1/2)}{n}\right)^2 - \left(\frac{\ln(1/2)}{n-1}\right)^2\right) + \dots \\
&= \Theta\left(\frac{1}{n^2}\right),
\end{aligned}$$

this makes ϵ small enough to force the algorithm to cover all of the v_i's. It would find a set of

$$(m\log(pn^2)) = O(m\log(n))$$

nodes having influence p, which gives a $O(\log(n))$ approximation and matches the lower bound. It is worth noting that the greedy algorithm for Set Cover also matches its hardness of approximation lower bound [16].

We will use Lemma 9, a version of which is derived in [11]. A function f is **submodular** if $f(A \cup \{x\}) - f(A) \geq f(B \cup \{x\}) - f(B)$ whenever $A \subseteq B$.

Lemma 9. *$S(e_I)$ is a positive monotone, submodular function of I.* [11]

Corollary 10. *If p is the maximum influence of any k node set in the network, then Algorithm 5, with a threshold of 1, terminated after k steps, produces a set with influence $\geq (1-\frac{1}{e})p$.*

Proof. Nemhauser et al. [13] show that greedily maximizing a non-negative, monotone, submodular function on sets gives a $(1-\frac{1}{e})$ approximation to the function on k-element sets. Hence, this follows from Lemma 9. $\square$

7. Open Problems

We leave open a number of interesting and challenging problems. Our results rely on exact value injection queries. While these queries are theoretically elegant, in real-world applications learners would normally have access only to non-exact value injection queries, and for such queries our algorithms would need to be modified, mainly because we look for shortest paths, not necessarily the paths least diluted by the multiplication of probabilities. The Angluin et al. [2] results on probabilistic networks adapt an exact value injection query algorithm to work in a non-exact setting, yet we see no clear way of similarly modifying our algorithms. Furthermore, in moving to the non-exact setting, because of the results from Section 5, either target network depth would need to be limited, or algorithms would have to be invented that do not rely on excitation paths.

Another interesting topic to explore is what other classes of cyclic networks can be learned using similar algorithms? Our algorithms rely on the independence assumption in the independent cascade social network model. However, there are other more general models of social networks, like the **decreasing cascade model** [11]. It would be worthwhile exploring their learnability as well.

In the real world, one also rarely has the ability to activate or suppress so many nodes at once. It is an open question under what restrictions on query size social networks are learnable. Another option to explore is to make the learner pay a higher cost for using larger queries.

Finally, is often the case that graph algorithms run faster on sparse graphs. It would be interesting to design an algorithm for learning social networks whose query complexity was a function of the size of the edge set in the target graph.

References

[1] T. Akutsu, S. Kuhara, O. Maruyama, and S. Miyano. Identification of genetic networks by strategic gene disruptions and gene overexpressions under a boolean model. *Theor. Comput. Sci.*, 1(298):235–251, 2003.

[2] D. Angluin, J. Aspnes, J. Chen, D. Eisenstat, and L. Reyzin. Learning acyclic probabilistic circuits using test paths. In *COLT '08: 21st Annual Conference on Learning Theory*, pages 169–180, 2008.

[3] D. Angluin, J. Aspnes, J. Chen, and L. Reyzin. Learning large-alphabet and analog circuits with value injection queries. *Machine Learning*, 72(1-2):113–138, 2008.

[4] D. Angluin, J. Aspnes, J. Chen, and Y. Wu. Learning a circuit by injecting values. *J. Comput. Syst. Sci.*, 75(1):60–77, 2009.

[5] D. Angluin, J. Aspnes, and L. Reyzin. Optimally learning social networks with activations and suppressions. In *ALT '08: 19th International Conference on Algorithmic Learning Theory*, pages 272–286, 2008.

[6] D. Angluin and M. Kharitonov. When won't membership queries help? *J. Comput. Syst. Sci.*, 50(2):336–355, 1995.

[7] J. Goldenberg, B. Libai, and E. Muller. Using complex systems analysis to advance marketing theory development: Modeling heterogeneity effects on new product growth through stochastic cellular automata. *Academy of Marketing Science Review*, 2001.

[8] T. Ideker, V. Thorsson, and R. Karp. Discovery of regulatory interactions through perturbation: Inference and experimental design. In *Pacific Symposium on Biocomputing 5*, pages 302–313, 2000.

[9] M. Kearns and L. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *J. ACM*, 41(1):67–95, 1994.

[10] D. Kempe, J. Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 137–146, New York, NY, USA, 2003. ACM.

[11] D. Kempe, J. M. Kleinberg, and É. Tardos. Influential nodes in a diffusion model for social networks. In *ICALP '05: 32nd International Colloquium on Automata, Languages and Programming*, pages 1127–1138, 2005.

[12] M. Kharitonov. Cryptographic hardness of distribution-specific learning. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 372–381, New York, NY, USA, 1993. ACM Press.

[13] G. Nemhauser, L. Wolsey, and F. M. An analysis of the approximations for maximizing submodular set functions. *Mathematical Programming*, 14:265–294, 1978.

[14] R. Raz and S. Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 475–484, New York, NY, USA, 1997. ACM.

[15] W. Rudin. *Principles of Mathematical Analysis.* International Series in Pure and Applied Mathematics. McGraw-Hill, 1976.

[16] P. Slavík. A tight analysis of the greedy algorithm for set cover. In *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 435–441, New York, NY, USA, 1996. ACM.