Abstract

Active Learning of Interaction Networks

Lev Reyzin

2009

From molecular arrangements to biological organisms, our world is composed of systems of small components interacting with and affecting each other. Scientists often learn the structure of such systems by tampering with them and making observations. In this thesis, we develop methods for automating this process from an active learning perspective, a setting where the learner is not restricted to making passive observations, but can choose to query the data.

First, we consider the setting of learning hidden graphs with queries. Each query type is motivated by a real-world problem, from genome sequencing to evolutionary tree reconstruction. We give new algorithms for learning graphs and also consider the problem of verifying the results of the learning task.

Next, we turn to value injection queries, which model experiments used to identify gene regulatory networks. We analyze the complexity of learning large alphabet and analog circuits with value injection queries. We then apply this model to social networks, allowing the learner to activate and suppress agents in the network, and we give an optimal algorithm and matching lower bound for this problem. Finally, we examine the passive learner, who watches the output of agents in a social network and must deduce the most likely underlying network.

Last, we consider a classical problem in query learning: learning finite automata, which themselves are networks of connected states. We introduce label queries as a generalization of the well studied membership queries. We give algorithms for learning automata using label queries and analyze other models for learning automata.

Active Learning of Interaction Networks

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Lev Reyzin

Dissertation Director: Dana Angluin

December 2009

to my parents

Acknowledgments

I feel I have been very fortunate during my time at Yale, and it is in no small part because I had a great advisor, Dana Angluin. I am continually inspired by the dedication and humility Dana approaches everything she does. She has shown me how sustained effort often pays off in the end. She has taught me how to persist in tackling obstacles when progress seems hopeless – how to break problems down into smaller and simpler parts, with the faith the pieces come together in the end. Dana has shown me patience in our weekly meetings, and given me invaluable advice and guidance in my research. I leave Yale hoping to emulate these qualities in my career ahead.

I also thank Jim Aspnes for our frequent collaborations. Meetings with Jim were never for a moment dull, and the enthusiasm with which Jim approaches research is infectious. From him, I have learned that research should always be fun, and it is a lesson I hope never to forget.

My decision to go to graduate school in the first place was inspired by three professors at Princeton. Brian Kernighan supervised my first research project. Moses Charikar exposed me to theory research and encouraged my initial attempts at it. Rob Schapire introduced me to machine learning and supervised a project that resulted in my first publication. My choice of research focus is mostly due to him.

My friends at Yale made the many long days and nights of work enjoyable, and

I have learned a lot from them. I thank Pradipta Mitra, Aaron Johnson, Ronny Dakdouk, Edo Liberty, Rich Alimi, Kevin Chang, Hong Jiang, Nick Ruozzi, Sam Daitch, and Xueyuan Su. I especially thank Nikhil Srivastava for great collaborations on research, much of which appears in this dissertation.

I also want to acknowledge my sources of funding. This dissertation is based upon work supported under a National Science Foundation Graduate Research Fellowship. My graduate studies have also been funded by a Yahoo! Research Kern Family Scholarship and by a Yale University Fellowship.

Most importantly, I want to thank my family. It would be hard to imagine my time at Yale without Marina's love and constant support – I am very lucky. I am also indebted to my parents for fostering my love of learning and always encouraging me. I marvel at the challenges they overcame so that I could pursue my dreams.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

First of all, there have to be many bodies present – many angels, many electrons, many atoms, many molecules, many people, etc. Secondly, for there to be a problem, these bodies have to interact with each other.

–Richard P. Feynman

Atoms bond with one another: they interact, they share electrons, and they form molecules. Some molecules are simple. Some are intricate and complex and form the building blocks of life. Complex molecular interactions make cells and organisms. Organisms reproduce, evolve, fight, cooperate, trade, communicate, and form even more complex networks that have their own patterns of interaction.

An important goal of science is to learn about these processes and the rules governing them. Often, scientists do not have access to these structures. They need to run experiments, look at their output, and interpret the results properly to make conclusions about the physical world.

It is seldom clear what experiment is best to run, especially when experiments are costly and time is limited. Moreover, it may be unclear what to do with the results of the experiments. In this dissertation, we seek to automate parts of this process of discovery. We show that many problems fit nicely into the task of learning the structure of hidden interaction networks and that the world provides ways of probing these hidden structures. Thus, we can model many problems as computational learning problems and apply machine learning methods to tackle them.

1.1 The Origins of Active Learning

Researchers in machine learning and learning theory have developed numerous models to study learning in a computational framework. One of the first paradigms for studying learning appeared in a 1967 paper of Gold, where he defined **learning in the limit** [46]. In this paradigm, the learner is presented with an infinite sequence of examples labeled by an unknown concept. After seeing each example, the learner outputs a hypothesis for the concept. The learner is said to learn a concept class in the limit if, for every concept in the class, after some point, it converges on a hypothesis that correctly classifies all examples.

Learning in the limit and variants of it are still studied, but this model was not suitable for capturing the phenomenon of learning that happens in our world. Gold's model assumes almost nothing about how the examples are generated, does not take computational efficiency into account, and insists that the learner find a hypothesis that makes no errors. Little progress was made on finding a good paradigm for studying computational learning until in 1984, in a seminal paper [85], Valiant introduced **PAC** (Probably Approximately Correct) learning.

In the PAC learning model, training examples are given to the learner from a fixed, arbitrary distribution and labeled by an unknown concept. A concept class is said to PAC learnable if for any concept in the class and any distribution, a learner can produce, with probability $(1-\delta)$, a hypothesis that has ϵ error on future examples, after using a number of training examples polynomial in the relevant parameters of the problem and running in time polynomial in the number of examples.

The PAC model is possibly the most ubiquitous model of learning, but it suffers from some drawbacks in capturing certain situations. In PAC learning, the learner acts as an observer. It passively sees whatever examples the world chooses to give it and does its best afterward. The PAC model fails to take into account the ability of learners in certain settings to choose their training data.

This is the gap query learning fills in. In 1987, Angluin [9, 10] introduced a model of learning where the learner creates its own examples and can **query** an **oracle**.[1] The learner can ask the oracle to label an example according to the unknown concept to be learned; this is called a **membership query**. In Angluin's model, the learner is also able to perform **equivalence queries** by asking the oracle whether its current concept is the correct one.[2] Angluin's idea encompasses the ability of scientists to run experiments of their own choosing, and the output of the oracle corresponds to the results of an experiment. Because no underlying distribution generates the data in this model, the learner is expected to learn the concept exactly.

However, Angluin's model also presented some difficulty for real world applications. While in many settings, the power to ask membership queries is justified, in some domains this ability makes little sense. Not every example is sensible enough to be labeled one way or another. For example, while most articles written in major newspapers can be classified into categories such as "Sports" or "Business," not every piece of arbitrarily chosen text can so easily subject itself to classification.

Modern **active learning** addresses some problems with the original framework. It encompasses the query learning model[3] and also many variants. One model by Cohn *et al.* [35] is called selective sampling. In this framework unlabeled data are

[1]In [46], Gold considered queries within the context of learning in the limit, but did not emphasize them because queries did not change learnability in his model.

[2]An equivalence query can be simulated in the PAC model by choosing a hypothesis and waiting sufficiently long to see how it performs on future data.

[3]Sometimes "query learning" and "active learning" are used interchangeably.

generated by a distribution, and the data selected by the learner are labeled for a price. Other active learning models include PAC with membership queries, which puts the learner in the PAC setting, but gives it power to query arbitrary examples. Pool based active learning [71] models the learner being given large amounts of unlabeled data and having to choose queries from the pool of unlabeled data. Many strategies for active learning have been studied, including query-by-committee [45, 82], uncertainty sampling [70, 71], and variance reduction [36]. A survey of active learning appears in [81]. We will be using the active learning framework for most of this dissertation.

Another important model, which is used in most of Chapter 6, is **online learning** [74]. Here, examples present themselves one at a time, and the learner must produce labels on the spot. Usually, the dataset is assumed to be so large that the learner cannot store the data it has seen. In prediction with expert advice [75], which is closely tied to online learning, the learner is given access to a pool of experts, and it must label data online, using their advice. To analyze a learner's performance, we compare its error rate with that of the best hypothesis (or expert) in hindsight. This is called **competitive analysis**.

Machine learning also covers many other learning phenomena. Reinforcement learning concerns itself with agents choosing policies for their actions [57]. Semi-supervised learning looks at how to take advantage of unlabeled data [33]. In learning with random noise, labels on the training examples are randomly flipped [24]. Agnostic learning makes no assumption on the origin of the labels of the data [61]. Statistical queries allow the learner to get estimates of the sample space [59]. Yet, these are only a small collection of machine learning models.

1.2 Active Learning of Interaction Networks

In this dissertation we create and analyze models inspired by real world problems in learning **interaction networks**: finite populations of elements whose state may change as a result of interacting with other elements according to specific rules. We now look at some of the interaction networks and query models that we study in this dissertation. We do not formally define the models here, but rather give an idea of the problems we tackle.

Evolutionary trees form a fairly basic interaction network. In an evolutionary tree an edge between two species represents an genetic linkage often taking millennia to establish. In Chapter 2 we study the problem of evolutionary tree reconstruction. Every species has a genetic distance to every other species, and these distances can be recovered by performing an experiment that compares their DNA. Due to the process of evolution, these distances are tree-realizable, and the task is to reconstruct the evolutionary tree by querying the distances. Because these experiments are costly, an important goal is to use as few queries as possible in reconstructing an evolutionary tree, and this can be studied from a computational standpoint.

Next we turn to the inner workings of some of these genetic tests. One of the first steps in analyzing a species, or rather an organism from a species, is to sequence its DNA. A common approach is to find contigs – contiguous regions of the genome – and to place them in proper order. Multiplex PCR (Polymerase Chain Reaction) is a tool for testing contigs for adjacent regions. Multiplex PCR experiments are modeled by edge counting queries. Edge counting queries take a subset of contigs, and the result of a query is a count of the number of connections contained in the queried subset. The goal of the learner is to reconstruct the adjacency graph, and edge counting queries are among the main queries studied in Chapter 3. In that chapter we also look at other models of other interaction networks inspired by settings such as reacting chemicals, circuit boards, and Internet topology.

After examining genome sequencing, we look at learning the mechanisms behind gene interaction by examining gene regulatory networks. Learning with value injection queries [14] models learning gene regulatory networks by disrupting and overexpressing genes and looking at the output of the network – the expressed phenotype – using techniques currently employed by biologists. The value injection query model represents gene regulatory networks as hidden acyclic circuits and allows the learner to override values on the wires of the circuit to learn the function it computes. In Chapter 4 we study these queries when they are applied to analog and large-alphabet circuits.

We can also take value injection queries and apply them to different settings. Social networks represent connections of individual agents in a population. Social networks are used to model diffusion of diseases, fads, or any form of information through a population. Independent cascade social networks describe the strength of influence of agents on each other, and they are modeled by weighted directed graphs, with edge weights representing the probabilities of agents activating one another. In Chapter 5 we look at how a learner can discover the strength of these connections by activating and suppressing agents in a social network and observing the resulting series of activations.

In Chapter 6 we again examine learning social networks, but from the point of view of a passive learner. The learner is able to observe connected parts of the social network – for example, in the case of the spread of disease, the learner might be shown groups of infected persons. We consider the problem of inferring the most likely social network given access to such data.

Finally, in Chapter 7 we take a new look at one of the most well studied problems in active learning: learning finite state automata. We study this problem mainly from theoretical interest, but it also has applications to the problem of robot localization.

In automata, the interaction between states happens as agents move from state to state, possibly changing their actions given what they see. We prove learnability results for new models of learning in this classic framework. In the new model, a teacher sprinkles labels over the states of an automata. The answer to querying a string on the automata includes not only an accept/reject, but also the label of the state the string leads to.

1.3 Preliminaries

This dissertation involves a theoretical study of interaction networks. Hence, we assume basic knowledge of probability, linear algebra, combinatorics, and graph theory, as well as some comfort with computer science theory, including basic complexity and asymptotic notation. Other concepts are defined where needed. In this section, we present some recurring ideas and definitions used in this thesis.

1.3.1 Graphs: Notation and Definitions

Notation

A graph $G = (V, E)$ consists of a set V of n vertices and a set E of m edges. Each edge $e \in E$ consists of a unique pair $\{u, v\}$ of vertices $u, v \in V$ with $u \neq v$. If there is a $\{u, v\}$-edge, u and v are said to be adjacent. In a **directed graph**, the pairs (u, v) are ordered pairs, and we say that there is a directed edge from u to v. In a **weighted graph**, each edge $e \in E$ has weight (or cost) w_e. In an **unweighted graph** all weights $w_e = 1$.

Definitions

The degree of a vertex is the number of edges that contain it. A graph has **degree** d of all of its vertices have degree $\leq d$. A graph has average degree d if the average over the degrees of the vertices is $\leq d$. A path in a graph is a sequence of vertices such that each vertex is adjacent to the next one in the sequence. A path is **simple** if all vertices along the path are unique. A cycle is a path where the start and end vertex are the same. A graph is **connected** if there is a path from every vertex to every other vertex. A **tree** graph is a connected graph with $n-1$ edges. A **leaf** in a tree is a node of degree 1. A **path** graph is a tree with exactly 2 leaves. A **star** graph is a tree with one vertex that is adjacent to the rest of the vertices. A **matching** is a set of edges without common vertices, and a vertex is said to be matched if it incident to an edge in the matching. A **perfect matching** is a matching in which every vertex is matched.

A path in a directed graph is a sequence of vertices such that each vertex has a directed edge to the next one in the sequence. A directed graph is **strongly connected** if there is a directed path from every vertex to every other vertex.

A Note on Random Graphs

An Erdös Rényi **random graph**, denoted $G(n,p)$, is a graph on n vertices in which every possible edge occurs independently with probability p. It has $p\binom{n}{2}$ edges in expectation. If $p = .5$ each of the $2^{\binom{n}{2}}$ graphs occurs equiprobably. The connectivity properties of random graphs exhibit certain threshold phenomena. If $p \geq \frac{(1+\epsilon)\ln(n)}{n}$ then the graph is almost surely connected [41], that is as $n \to \infty$, the probability $G(n,p)$ is connected approaches 1. One can also use a more general model for generating random graphs, with each edge having a possibly different probability of occurring – this, of course, applies to both directed and undirected graphs.

1.3.2 Information Theoretic Lower Bounds

In this dissertation, we often rely on information theoretic arguments to establish lower bounds on the number of queries a learner must perform. An **information theoretic lower bound** uses the following observation: the answer to every query gives the learner a limited amount of information, and the learner must ask enough queries to be able to specify any output.

Suppose it requires n bits to specify any output, and the learner only receives b bits on every query. This gives us a lower bound of $\Omega(n/b)$ queries.

For an example, we now apply an information theoretic argument for a familiar problem: sorting n numbers with comparisons. We assume a comparison of two numbers a and b can have 3 different results: $a > b$, $a < b$, or $a = b$. The result of a comparison gives the algorithm at most $2 = O(1)$ bits of information. There are $n!$ possible orderings of n numbers, so it takes $\Omega(\log(n!)) = \Omega(n \log(n))$ bits to specify an ordering. So an information theoretic lower bound tells us the known result that $\frac{\Omega(n \log(n))}{O(1)} = \Omega(n \log(n))$ comparisons are required for sorting n numbers.

1.3.3 Learners and Adversaries

A algorithm is an **adaptive learner** if it can ask queries one at a time, deciding each subsequent query only after seeing the answer to the previous queries. A learning algorithm is **non-adaptive** if it must choose its queries all at once, before seeing the results of any query. Information theoretic lower bounds apply against both adaptive and non-adaptive learners.

In analyzing the performance of an algorithm, we can view the oracle answering the learner's queries as an adversary. An **adaptive adversary** must answer each future query consistently with its previous answers, but can otherwise change its target concept arbitrarily. An adaptive adversary is used when measuring the worst-case performance of an algorithm. An **oblivious adversary** must commit to a concept after seeing the learner's algorithm and cannot change it afterward. Because the adversary has access to the learner's algorithm, an oblivious adversary is just as powerful as an adaptive adversary if the learning algorithm is deterministic. If the learner uses randomness, we analyze the expected performance of the learning algorithm.

1.4 A Note to the Reader

Much of the work in this thesis has appeared in journal papers or conference proceedings, and has been done jointly with collaborators. Chapter 2 is based on the paper "On the Longest Path Algorithm for Reconstructing Trees from Distance Matrices" [79], which is joint work with Nikhil Srivastava. Chapter 3 is based on the paper "Learning and Verifying Graphs Using Queries with a Focus on Edge Counting" [78], which is joint work with Nikhil Srivastava. Chapter 4 is based on the paper "Learning Large-Alphabet and Analog Circuits with Value Injection Queries" [12, 13], which is joint work with Dana Angluin, James Aspnes, and Jiang Chen. Chapter 5 is based on the paper "Optimally Learning Social Networks with Activations and Suppressions" [15, 16], which is joint work with Dana Angluin and James Aspnes. Chapter 6 is based on joint work with Dana Angluin and James Aspnes. Chapter 7 is based on the paper "Learning Finite Automata Using Label Queries" [17], which is joint work with Dana Angluin, Leonor Becerra-Bonache, and Adrian Horia Dediu.

Chapter 2

The Longest Path Algorithm

In this chapter, we focus on analyzing a well known algorithm for reconstructing evolutionary trees. This short chapter also serves as an introduction to the types of algorithms and analysis used in this thesis.

2.1 Introduction and Background

In [37], Culberson and Rudnicki consider the problem of reconstructing a degree d weighted tree given query access to it. If the tree has n vertices, a **shortest path query** on two vertices, $\mathbf{SP}(i, j)$, returns d_{ij}, the weight of the (unique) path between vertices i and j. They describe an algorithm which finds the longest path in the tree, divides the remaining vertices into subtrees according to where they connect to this path, and then recurses on the subtrees. Their algorithm relies on three key ideas:

1. The longest path in a subtree can be computed in linear time. Simply pick an arbitrary vertex r and find the distances from r to every other vertex. Let u be the farthest vertex from r, and repeat to find the farthest vertex v from u. Then π_{uv} is the longest path in the tree, and we have performed only $2n$ queries.

2. Given a longest path π_{uv} and distances computed in step (1), every other vertex z can be placed either on π_{uv} or on a subtree rooted at a known vertex w (a "hub") on π_{uv}, with no additional **SP** queries. To be precise, z is in the subtree rooted at w if and only if $d_{zu} - d_{zv} = d_{wu} - d_{wv}$.

3. No further **SP** queries involving any hub w need be made, since for every vertex z in w's subtree, we have $d_{zw} = d_{zu} - d_{wu}$. This means that we can effectively forget about vertices that occur on the longest path, as far as **SP** queries concerned.

The algorithm presented in [37] is equivalent to what is described above, with minor simplifications. We will call this algorithm **Longest Path**.

In their analysis, Culberson and Rudnicki refer to **SP** queries in step (1) as "hub computations" and establish that the running time is dominated by them. They claim that for unweighted trees[1], the running time of the algorithm is $O(dn \log_d n)$. Their claim rests on the following argument:

> *Since once a vertex... is located on a path in the tree it no longer participates in such computations in other partitions, the number of computations is maximized when the longest path in every subtree is minimized. When the maximum degree is restricted, this leads to balanced trees where all internal vertices are of maximum degree.*

So according to [37], the worst case for degree four is the kind of tree shown in Figure 2.1.

Yet, this proof is incorrect. In section 2.2, we construct an unweighted tree on which the algorithm takes $\Omega(n^{3/2}\sqrt{d})$ time, contradicting the claim in [37]. In section 2.3 we present a revised analysis of Longest Path, showing that $\Theta(n^{3/2}\sqrt{d})$ is tight for the unweighted case.

[1]Culberson and Rudnicki call unweighted trees "topological trees."

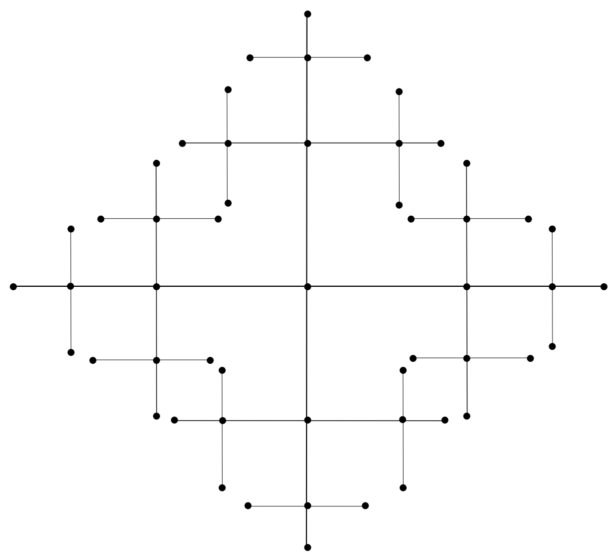


Figure 2.1: A tree that minimizes the longest path in every subtree, from [37].

It is worth noting that other algorithms exist that achieve the $O(dn \log_d n)$ bound, which was also shown to be a lower bound in [65]. A quite different algorithm called **Deepest Point**[2], discovered by Hein [52], reconstructs both general and unweighted trees in $O(dn \log_d n)$ time. Further, variants of this problem have been studied in the experiment model for constructing evolutionary trees by Kannan *et al.* [58], Brodal *et al.* [30], and Lingas *et al.* [72]. Notwithstanding, Longest Path is one of the first algorithms claimed to run in sub-quadratic time for tree reconstruction and also one of the simplest. In this chapter we present a correct analysis of it.

2.2 A Counterexample

While the tree in Figure 2.1 does minimize the lengths of the longest paths, it also "nicely" splits the remaining vertices. Here, we take the opposite approach and consider a family of trees where removing the longest path always leaves the re-

[2]In constructing Deepest Point [52], Hein mentions in passing that an algorithm recursively placing longest paths would run in $\Omega(n^{\frac{3}{2}})$

maining vertices in a single subtree, as shown in Figure 2.2. Notice that G_i has $1+3+\ldots.(2i-1) = i^2$ vertices and a longest path of length $2i-1$ (perpendicular to the horizontal "stem") – that is, the tree on n vertices has a longest path of length $2i-1 = O(\sqrt{n})$ and a stem of length $i = O(\sqrt{n})$. Removing the longest path which is centered on the stem from G_i gives G_{i-1}.

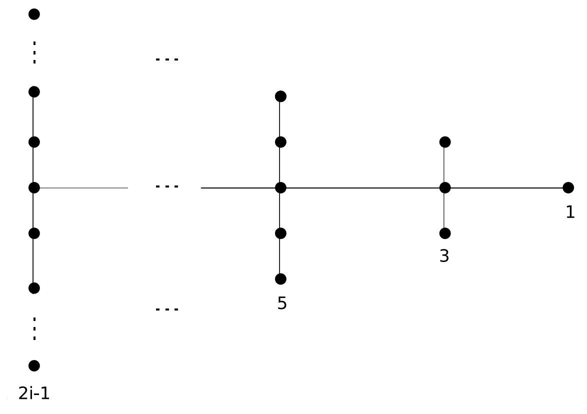


Figure 2.2: A counterexample to the $O(dn \log_d n)$ analysis of [37], the tree G_i.

For a tree of size n, finding a longest path lying vertically along the spine takes $\Omega(n)$ **SP** queries and leaves a subtree of size $\Omega(n - \sqrt{n})$. Hence, the total running time is at least

$$T(n) \geq n + T(n - \sqrt{n})$$

$$T(1) = 0.$$

The recurrence is solved easily by substitution to get $T(n) = \Omega(n^{3/2})$, showing that the $O(dn \log_d n)$ analysis of [37] is incorrect.

All vertices in the above example are of degree at most 3. For the general degree d case, we consider trees G'_i that have $\frac{d}{2} - 1$ copies of each path on the stem. Observe that $|G'_i| = \Omega(d|G_i|)$ (since a linear number of vertices are copied $\Omega(d)$ times), so that longest paths in G'_i are of length $O(\sqrt{\frac{n}{d}})$. Since all longest paths of a given length intersect at only one vertex (which is on the stem), Longest Path does not split them into smaller parts by placing them in separate subtrees. This gives the following recurrence:

$$T(n) \geq n + T\left(n - \sqrt{\frac{n}{d}}\right)$$

$$T(1) = 0$$

which has the solution $T(n) = \Omega(n^{3/2}\sqrt{d})$.

2.3 Analysis of the Unweighted Case

Say that a subtree is at *phase* j if it is obtained by recursing j times. Consider all subtrees $T_1 \ldots T_k$ at a particular phase, with a total of n vertices. The number of **SP** queries used to find a longest path in T_i is just $O(|T_i|)$, so the total number of queries used in the phase is $\sum_i O(|T_i|) = O(n)$, i.e., *linear* in the total number of nodes. Thus we can bound the running time of Longest Path by $n\times$(number of phases).

The correct analysis of Longest Path relies on following observation:

Lemma 2.3.1. *Suppose π is a longest path in a subtree T, and S is the set of longest paths in T that are edge-disjoint with π. Then all paths in $S \cup \{\pi\}$ share a single vertex.*

Proof. Suppose π_1 and π_2 are longest paths in T. Assume they do not intersect. Find the shortest path π' that connects π_1 and π_2 (we can do this since T is a tree). But now we can construct a path longer than π_1: take the longer halves of π_1 and π_2 and join them with π'.

Thus every two longest paths in T intersect. Moreover, if π_1 and π_2 are edge-disjoint, they must intersect at a unique vertex: they cannot intersect at two consecutive vertices because then they would share an edge, and they cannot intersect at two non-consecutive vertices because that would imply a cycle in T.

Suppose $\pi' \in S$, and let π and π' intersect at v. Suppose $\pi'' \in S$ does not pass through v; then, it must intersect π at a single vertex $w \neq v$. Also, π'' intersects π' at some vertex u not on π by the claim above. But now u, v, w lie on a cycle, which is impossible. □

Lemma 2.3.2. *If the longest path at a phase has length l, then there are at most dl phases remaining before the algorithm terminates.*

Proof. Suppose the longest path π chosen by Longest Path has length l. Any longest paths that share an edge with π will be split into smaller paths, each of length at most $l-1$, for the next phase. By lemma 2.3.1 all longest paths that are edge-disjoint from π must pass through a single vertex, which has degree at most d. Thus there are at most d such paths, and after d phases, the longest path remaining will be of length at most $l - 1$. Therefore, after dl phases, all longest paths are of length 0, and Longest Path terminates. □

Theorem 2.3.3. *Longest Path runs in $O(n^{3/2}\sqrt{d})$ time.*

Proof. If at any phase the longest path is of length at most $\sqrt{\frac{n}{d}}$, then by lemma 2.3.2 the number of phases remaining is $O(d\sqrt{\frac{n}{d}})$. Since each phase takes linear time, the total time starting from such a phase is $O(nd\sqrt{\frac{n}{d}}) = O(n^{3/2}\sqrt{d})$.

So assume the input has a longest path of length $l > \sqrt{\frac{n}{d}}$. How many phases can go by without l falling below $\sqrt{\frac{n}{d}}$? If $l \geq \sqrt{\frac{n}{d}}$ then at least $\sqrt{\frac{n}{d}}$ vertices are removed in each phase; thus, the number of such phases is at most $n/\sqrt{\frac{n}{d}} = \sqrt{dn}$. Again, each phase takes linear time, so we reach a phase with $l \leq \sqrt{\frac{n}{d}}$ in time $O(n^{3/2}\sqrt{d})$, as desired. □

Chapter 3

Learning and Verifying Graphs with Queries

3.1 Introduction

Graph learning appears in many different contexts for learning interaction networks. In this chapter, we explore models for graph learning inspired by various real-world problems.

Suppose we are presented with a circuit containing a set of chips on a board. We can test the resistance between two chips with an ammeter. In as few measurements as possible, we want to learn whether the entire circuit is connected, or whether we need to power the components separately. This can be seen as a graph learning problem, in which the chips are vertices of a hidden graph and the ammeter measurements are queries into the graph, which tell whether a pair of vertices is connected by a path. If we are given a strong enough ammeter to tell not only whether two chips are connected, but also how far apart they are in the underlying circuit, we get the stronger **shortest path** queries – a concept familiar to us from Chapter 2, where shortest path queries are used in the context of evolutionary tree reconstruction.

In a different setting [18], testing which pairs of chemicals react in a solution is modeled by **edge detecting** queries. Here, vertices correspond to chemicals, edges designate chemical reactions, and a set of chemicals 'reacts' if and only if it induces an edge. Applications of this model extend to bioinformatics, where learning a hidden matching [6] turns out to be useful in analyzing PCR, a technique for DNA sequencing. As we discuss in Section 1.2, PCR can be modeled as graph learning problems, and a focus of this chapter is studying **edge counting** queries, which are modeled after multiplex PCR.

Our goal is to explore several graph-learning problems and queries. We consider the following types of queries, defined on graphs $G = (V, E)$:

- **Edge detecting query (ED)**: Check if there is edge between any two vertices in $S \subseteq V$. *This model has applications in genome sequencing and was studied in [3, 6, 18, 19, 49].*

- **Edge counting query (EC)**: Return the number of edges in the subgraph induced by $S \subseteq V$. *Learning graphs with edge counting queries is called the* ***additive model*** *in bioinformatics, where it has extensive uses, especially for analyzing multiplex PCR, and was studied in [29, 34, 50].*

- **Shortest path query (SP)**: Return the length of shortest path in G between two vertices; if no path exists, return ∞. *This is the canonical model in the evolutionary tree literature; see [52, 65] and Chapter 2.*

The second kind of task we consider is graph verification. Suppose we are interested in learning the structure of some protein networks, and after months of careful measurement, we complete our learning task. If we then find out there is a small chance we made a mistake in our measurements or if we have reason to believe our equipment may have been broken during experimentation, can we verify the structures we've learned more efficiently than learning them over again? More concretely, we are interested in how efficiently we can decide whether a graph presented to us is indeed the "true graph." This is a natural question to ask, especially since real world data is often noisy, or we sometimes have reason to mistrust results we are given. Every learning problem induces a new verification problem.

We consider different classes of graphs for our learning and verification tasks. The first class is **arbitrary graphs**, where there are no restrictions on the topology of the graph. Any algorithm that learns or verifies an arbitrary graph can also be used for more restricted settings. We also consider learning **trees**, where we know the graph we are trying to learn is a tree, but we are not aware of its topology. This is a natural setting for learning structures that we know do not have underlying cycles, for example evolutionary trees. Finally, we consider the problem of learning the **partition** of a graph, where the learner must determine the connected components of the hidden graph without necessarily learning its edges. Here, we do not restrict the underlying class of graphs, but instead relax the learning problem. This is a natural question in settings where different partitions represent qualitative differences, for example in electrical networks, a power generator in one partition cannot power any nodes outside its own partition. Note that this also subsumes the natural question of whether or not a graph is connected.

In this chapter we fill in some gaps in the literature on these problems and introduce the verification task for these queries. We also introduce the problem of learning partitions and present results in the **EC** query case. We then show what problems remain open. After presenting a summary of the past work done on these problems, we divide our results into two sections: Graph Learning and Graph Verification.

3.2 Previous Work

Some of the earliest work in graph discovery was for the evolutionary tree reconstruction problem we study in Chapter 2. While the longest path algorithm fails at solving this task optimally, Hein [52] tackles the problem of learning a degree d restricted tree with **SP** queries. He describes an $O(dn \lg n)$ algorithm that builds the tree by inserting one node at a time, in a carefully chosen order under which each insertion takes $O(d \lg n)$ queries. Among other results, King *et al.* [65] provide a matching lower bound for this problem.

Angluin and Chen [18] show that $O(\lg n)$ adaptive **ED** queries per edge are sufficient to learn an arbitrary hidden graph. Their algorithm repeatedly divides the graph into independent subgraphs (i.e., it colors the graph), so as to eliminate interference to **ED** queries from previously discovered edges, and uses a variant of binary search to find new edges within each subgraph. It is worth noting that this is not far from an information-theoretic lower bound of $\Omega(\epsilon \lg n)$ **ED** queries per edge for the family of graphs with $n^{2-\epsilon}$ edges. A later paper [19] generalizes these results to hypergraphs using different techniques.

The work of Angluin and Chen is preceded by a few papers [3, 6, 49] that tackle learning restricted families of graphs, such as stars, cliques, and matchings. Alon *et al.* [6] provide lower and upper bounds of $.32\binom{n}{2}$ and $(1/2 + o(1))\binom{n}{2}$ respectively on learning a matching using nonadaptive **ED** queries, and a tight bound of $\Theta(n \lg n)$ **ED** queries in expectation if randomization is allowed. Alon and Asodi [3] prove similar bounds for the classes of stars and cliques. Grebinski and Kucherov [49] study reconstructing Hamiltonian paths with **ED** queries. It turns out that many of these results are subsumed by those of [18] if we ignore constant factors.

Grebinski and Kucherov [50] also study the problem of learning a graph using **EC** queries and give tight bounds of $\Theta(dn)$ and $\Theta(n^2/\lg n)$ nonadaptive queries for d-

degree-bounded and general graphs respectively. They also prove tight $\Theta(n)$ bounds for learning trees. Their constructions make heavy use of separating matrices. In [34], Choi and Kim show that graph reconstruction can be done using $O\left(\frac{|E|\log(n)}{\log(|E|)}\right)$ queries, but their algorithm takes time exponential in n. In [29], Grebinski and Kucherov present a survey on learning various restricted cases of graphs, including Hamiltonian cycles, matchings, stars, and k degenerate graphs, with **ED** and **EC** queries.

In the graph verification setting, Beerliova *et al.* [26] consider the problem of discovering and verifying networks using distance queries. In this setting, which models discovering nodes on the Internet, the learner can query a vertex, and the answer to the query is the set of all edges whose endpoints have different distances from the query vertex. They show there is no $o(\log n)$ competitive algorithm unless $P = NP$.

Both the learning and verification tasks also bear some relation to the field of Property Testing, where the object is to examine small parts of the adjacency matrix of a graph to determine a global property of the graph. For a survey of this area, see [48].

3.3 Graph Learning

We first note that **EC** queries are at least as strong as **ED** queries and that the problem of learning an arbitrary graph is at least as hard as learning trees or partitions. Hence, in this chapter, any lower bounds for stronger queries and easier targets apply to weaker queries and harder target classes. Conversely, any upper bounds we establish for weaker queries and harder problems apply for stronger queries and more restricted classes.

We first establish that $\Theta(n^2)$ **SP** and **ED** queries is essentially tight for learning arbitrary graphs and partitions.

Proposition 3.3.1. $\Omega(n^2)$ **SP** *queries are needed to learn the* ***partition*** *of a hidden graph on* n *vertices.*

Proof. We prove this by an adversarial argument; the adversary simply answers '∞' (i.e., not connected) for all pairs of vertices i, j. If fewer than $\binom{n}{2}$ queries are made, then some pair i, j is not queried, and the algorithm cannot differentiate between the graph with no edges and the graph with a single edge $\{i, j\}$ (for which $\mathbf{SP}(i, j) = 1$). But these graphs have different partitions. □

If k is the number of components in a graph, there is an obvious algorithm that does better for $k < n$, even without knowledge of k:

Proposition 3.3.2. $O(nk)$ **SP** *queries are sufficient to determine the* ***partition*** *of a hidden graph on* n *vertices, if* k *is the number of components in the graph.*

Proof. We use a simple iterative algorithm:

- Step 1: Place 1 in its own component.[1]
- Step $i > 1$: Query $\mathbf{SP}(i, w)$ for an item w from each existing component; if $\mathbf{SP}(i, w) \neq \infty$, place i in the corresponding component and move to the next step. Otherwise, create a new component containing i and move to the next step.

Correctness is trivial. For complexity, note that there at most k components at any step (since there are at most k components at phase n and components are never destroyed); hence n vertices take at most nk queries. □

Proposition 3.3.3. $\Omega(n^2)$ **ED** *queries are needed to learn the* ***partition*** *of a hidden graph on* n *vertices.*

[1]We use numbers $1, 2, \ldots, n$ to represent the vertices of the graph.

Proof. Consider the class of graphs on n vertices consisting of two copies of $K_{\frac{n}{2}}$, which we will call C_1 and C_2, and one possible edge between C_1 and C_2. If there is an edge, all the vertices are in a single component; otherwise there are two components. Any algorithm that learns the partition must distinguish between the two cases. Observe that an **ED** query on a set S containing more than one vertex from either C_1 or C_2 will not yield any information since an edge is guaranteed to be present in S and any such query will be answered with a 'yes'. Hence, all informative queries must contain one vertex from C_1 and one vertex from C_2. An adversary can keep on answering 'no' to all such queries, and unless all possible pairs are checked, an edge may be present between C_1 and C_2. Hence, the algorithm cannot tell whether the graph has one component or two until it asks all $\approx (\frac{n}{2})^2 = \Omega(n^2)$ queries. □

It turns out that **EC** queries are considerably more powerful than **ED** queries for this problem.

Proposition 3.3.4. $\Omega(n)$ **EC** *queries are needed to learn the* ***partition*** *of a hidden graph on n vertices.*

Proof. We use an information-theoretic argument. The number of partitions of an n element set is given by the Bell number B_n; according to de Bruijn [38]:

$$\ln B_n = \Omega(n \ln n)$$

Since each **EC** query gives a $\lg(\binom{n}{2}) = 2\lg n$ bit answer, we need $\Omega(\frac{\lg(B_n)}{2\lg n}) = \Omega(\frac{n \lg n}{\lg n}) = \Omega(n)$ queries. □

Theorem 3.3.5. $O(n \lg n)$ **EC** *queries are sufficient to learn the* ***partition*** *of a hidden graph on n vertices.*

Proof. Consider the following n phase algorithm, in which the components of $G[1 \ldots i]$ are determined in phase i.

- *Phase 1*: Set $\mathcal{C} = \{c_1\}$ with $c_1 = \{1\}$. $\mathcal{C}$ will keep track of the components $c_1, c_2, \ldots$ known at any phase, and we will let $\mathcal{C} + v$ denote $\{v\} \cup \bigcup_{c_i \in \mathcal{C}} c_i$.

- *Phase* $(i+1)$: Let $v = (i+1)$, and query $\mathbf{EC}(\mathcal{C} + v)$. If $\mathbf{EC}(\mathcal{C} + v) = \mathbf{EC}(\mathcal{C})$ (i.e., there are no edges between v and $\mathcal{C}$), add a new component $c = \{v\}$ to $\mathcal{C}$.

  Otherwise, split $\mathcal{C}$ into roughly equal halves $\mathcal{C}_1$ and $\mathcal{C}_2$ and query $\mathbf{EC}(\mathcal{C}_1 + v), \mathbf{EC}(\mathcal{C}_2 + v)$. Pick any half $h \in \{1, 2\}$ for which $\mathbf{EC}(\mathcal{C}_h + v) > \mathbf{EC}(\mathcal{C}_h)$ and repeat recursively until $\mathbf{EC}(\{c_j\} + v) > \mathbf{EC}(c_j)$ for a single component $c_j \in \mathcal{C}$[2]. This implies that there are edges between c_j and v; we will call c_j a *live* component.

  Repeat on $\mathcal{C} \backslash \{c_j\}$ to find another live component $c_{j'}$, if it exists; repeat again on $\mathcal{C} \setminus \{c_j, c_{j'}\}$ and so on until no further live components remain (or equivalently, no new edges are found). Remove all live components from $\mathcal{C}$ and add a new component $\{v\} \cup \bigcup_{\text{live } c_j} c_j$.

Correctness is simple, by induction on the phase: we claim that $\mathcal{C}$ contains the components of $G[1 \ldots i]$ at the end of phase i. This is trivial for $i = 1$. For $i > 1$, suppose $\mathcal{C} = \{c_1, \ldots, c_m\}$ at the beginning of phase i, and by the inductive hypothesis $\mathcal{C}$ contains precisely the components of $G[1 \ldots (i-1)]$. The components that do not have edges to v are unaffected by its introduction in $G[1 \ldots i]$, and these are not changed by the algorithm. All other components are connected to v and therefore to each other in $G[1 \ldots i]$; but these are marked 'live' and subsequently merged into a single component at the end of the phase. This completes the proof.

To analyze complexity, we use a "potential argument." Let Δ_i denote the increase in the number of components in $\mathcal{C}$ during phase i. There are three cases:

[2]Notice that this is essentially a binary search.

- $\Delta_i = 1$: There are no live components (v has no edges to any component in $\mathcal{C}$), and this is determined with a single $\mathbf{EC}(\mathcal{C} + v)$ query.
- $\Delta_i = 0$: There is exactly 1 live component (v connects to exactly one member of $\mathcal{C}$). Since there are at most n components to search, it takes $O(\lg n)$ queries to find this component.
- $\Delta_i < 0$: There are $k > 1$ live components with edges to v, bringing the number of components down by $k - 1$.[3] Finding each one takes $O(\lg n)$ queries, for a total of $O(k \lg n) = O((-\Delta_i + 1) \lg n)$.

The total number of queries is

$$\sum_{i:\Delta_i=1} 1 + \sum_{i:\Delta_i=0} (\lg n) + \sum_{i:\Delta_i<0} O((-\Delta_i + 1) \lg n)$$

The first two sums are bounded by $O(n \lg n)$ since there are n phases, and the last one becomes

$$O(n \lg n) + O(\lg n) \sum_{\Delta_i<0} (-\Delta_i).$$

But $\sum_{\Delta_i<0}(-\Delta_i)$, the total *decrease* in the number of components, cannot be greater than n since the total *increase* is bounded by n (one new component per phase) and the final number of components is nonnegative. So the total number of queries is $O(n \lg n)$, as desired.

To see that this analysis is tight, consider the case where G has exactly $n/2$ components, with $\Delta_i = 1$ for $i < n/2$, $\Delta_i = 0$ for $i \geq n/2$. The first $n/2$ phases take only $O(n/2)$ queries, but the remaining $n/2$ take $O(\lg(n/2))$ queries each, for a total of $O(n/2 \lg(n/2) + n/2) = O(n \lg n)$ queries. $\square$

Proposition 3.3.6. *$O(|E| \lg n)$* **EC** *queries are sufficient to learn a hidden* ***graph***

[3]The k components previously in $\mathcal{C}$ are replaced by a single component, hence $\Delta_i = -(k-1)$.

on n vertices.

Proof. The algorithm of Angluin and Chen ([18]) achieves this since **EC** queries are more powerful than **ED** queries, but we present a simpler method here that exploits the counting ability of **EC**. The key observation is that we can learn the degree of any vertex v in two queries:

$$d(v) = \mathbf{EC}(V) - \mathbf{EC}(V \setminus \{v\})$$

We use this to find all of the neighbors of v, using a binary search similar to that in the algorithm of theorem 3.3.5. Split $V \setminus \{v\}$ into halves V_1, V_2 and query $\mathbf{EC}(V_1 + v), \mathbf{EC}(V_2 + v)$. Pick a half such that $\mathbf{EC}(V_i + v) > \mathbf{EC}(V_i)$ and recurse until $\mathbf{EC}(w + v) > 0$ for some vertex w. This implies that w is a neighbor of v. Repeat the procedure on $V \setminus \{w, v\}$ to find more neighbors, and so on, until $d(v)$ neighbors are found.

We can reconstruct the graph by finding the neighbors of each vertex; this uses a total of

$$\sum_v d(v) \lg n = \lg n \sum_v d(v) = 2|E| \lg n = O(|E| \lg n)$$

queries, as desired. □

It follows from the above proof that the degree sequence of a graph can be computed in $2n$ queries, and consequently any property that is determined by it takes only linear queries.

Proposition 3.3.7. $\Omega(n^2)$ **SP** *queries are needed to learn a hidden* ***tree****.*

Proof. Consider a **quasi-star** – a graph on $2n+1$ vertices, which are of three kinds: a single center vertex s, n 'inner' vertices $x_1 \ldots x_n$, and n 'outer' vertices $y_1 \ldots y_n$. The center and inner vertices form a star (with edges $\{x_i, s\}$) and the outer vertices are

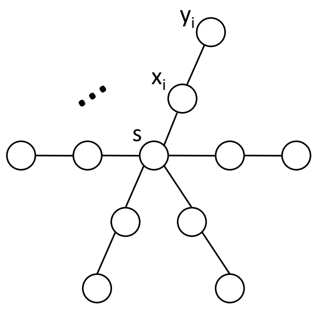


Figure 3.1: An illustration of a quasi-star.

matched with the inner vertices (for each y_i there is a unique x_{j_i} such that $\{x_{j_i}, y_i\}$ is an edge; no x_{j_i} is repeated). A quasi-star is pictured in Figure 3.1.

Suppose the learning algorithm knows that G is a quasi-star. There are only three kinds of **SP** queries: $\mathbf{SP}(s, x_i) = 1$, $\mathbf{SP}(s, y_i) = 2$, and

$$\mathbf{SP}(x_i, y_j) = \begin{cases} 1 & \text{if } \{x_i, y_j\} \text{ is an edge} \\ 3 & \text{otherwise} \end{cases}$$

The only query that gives any information is the last kind, and the problem reduces to that of learning a matching using **ED** queries, which we know by [6] takes $\Omega(n^2)$ queries. $\square$

Query	**partition**	**graph**	**tree**
ED	$\Theta(n^2)$	$\Theta(\lvert E\rvert \lg n), \Theta(n^2)$[18]	$\Theta(n \lg n)$
EC	$O(n \lg n)$ $\Omega(n)$	$O(\lvert E\rvert \lg n), O(\frac{n^2}{\lg n}), O(dn)$[18, 50] $\Omega(dn), \Omega(\frac{n^2}{\lg n})$[50]	$\Theta(n)$
SP	$\Theta(nk)$	$\Theta(n^2)$	$\Theta(n^2), \Theta(dn \log_d n)$ [52, 65]

Table 3.1: Summary of results for polynomial time algorithms.

Table 3.1 shows the known bounds for the problems we consider. We can see that tight asymptotic bounds exist for all of these learning problems, except for learning partitions with **EC**.

We note that learning a tree becomes significantly easier when the degrees of its vertices are restricted, and in many cases, knowing a bound on the degree of a graph can help with the learning problem.

3.4 Graph Verification

In this setting, a verifier is presented a graph $G(V, E)$ and asked to check whether it is the same as a hidden graph $G^*(V, E^*)$, given query access to G^*. In this section, we explore the complexity of graph verification using various queries. Mainly, we show that while verifying unrestricted graphs is hard using **SP** and **ED** queries, there is a fast randomized algorithm that uses **EC** queries.

Proposition 3.4.1. *Verifying an arbitrary* ***graph*** *takes* $\Theta(n^2)$ **SP** *queries and* $\Theta(n^2)$ **ED** *queries.*

Proof. Consider the problem of the verifying a clique when the hidden graph is really a clique with some edge (u, v) removed. $\mathbf{SP}(u', v') = 2$ if and only if $u' = u$ and $v' = v$. A simple adversarial argument shows that $\Omega(n^2)$ queries are necessary. Similarly, for **ED** queries, let $S = \{u, v\}$. The answer to query $\mathbf{ED}(U)$, where $|U| \neq 2$ is predetermined. Otherwise, $\mathbf{ED}(U) = 0$ if and only if $U = S$. There are $\binom{n}{2}$ choices for S such that $|S| = 2$; hence $\Omega(n^2)$ are needed. For both **SP** and **ED** queries the $O(n^2)$ algorithm of checking all pairs of vertices is obvious. $\square$

Given that **SP** queries are most often considered in evolutionary tree learning, we also consider the problem of verifying a tree with **SP** queries. In this setting, the verifier knows the hidden graph is a tree and is presented with a tree to verify.

Proposition 3.4.2. *Verifying a* ***tree*** *takes* $\Theta(n)$ **SP** *queries.*

Proof. Consider the problem of verifying a path graph (from the class of path graphs). This reduces to verifying that a given ordering of the vertices is correct. If the answers to each query are consistent with the graph to be verified, each query verifies at most two vertices in the ordering. An adversary can choose whether or not to swap any pair of vertices that have not been queried and either stay consistent with the input path graph or not until at least $n/2$ **SP** queries have been performed. Conversely, we can verify each edge individually in $n-1$ queries. □

We now consider the problem of verifying a graph with **EC** queries. Here, we see that **EC** queries are quite powerful for verifying arbitrary graphs.

Theorem 3.4.3. *Any* ***graph*** *can be verified by a randomized algorithm using* 1 **EC** *query, with success probability* $1/4$.

Proof. We define $\mathbf{EC}(V, G)$ to be the query $\mathbf{EC}(V)$ on graph G. The algorithm is simple. We let Q be a random subset of vertices of V, with each vertex chosen independently with probability $\frac{1}{2}$. We query $\mathbf{EC}(Q, G^*)$ and compute $\mathbf{EC}(Q, G)$. If the two quantities are not equal, we say G and G^* are different. Otherwise we say they are the same. We will show that if $G = G^*$ the algorithm always returns the correct answer, and otherwise gives the correct answer with probability at least $\frac{1}{4}$.

Consider the symmetric difference $S = (V, E\Delta E^*)$. Let $A = \{(u,v) \in E \setminus E^* : u, v \in Q\}$ and $B = \{(u,v) \in E^* \setminus E : u, v \in Q\}$. If $G = G^*$ then $|A| = |B| = 0$ and we are always right in saying the graphs are identical; otherwise $G \neq G^*$ and $E\Delta E^* \neq \varnothing$, so by the following lemma $|E\Delta E^*| = |A| + |B|$ is odd with probability $\frac{1}{4}$. But this immediately implies that $|A| \neq |B|$, as desired. □

Lemma 3.4.4. *Let $G(V, E)$ be a graph with at least one edge. Let $G'(V', E')$ be the subgraph induced by taking each vertex in G independently with probability $\frac{1}{2}$. If G is non-empty, the probability that $|E'|$ is odd is at least $\frac{1}{4}$.*

Proof. Fix an ordering $v_1 \dots v_n$ so that $(v_{n-1}, v_n) \in E$. Select each of $v_1 \dots v_{n-2}$ independently with probability $1/2$, and let H' be the subgraph induced by the selected vertices. Suppose the probability that H' contains an odd number of edges (i.e., $\texttt{parity}(H') = 1$) is p.

Let i (resp. j) be the number of edges between v_{n-1} and H' (resp. v_n and H'). Consider two cases:

- $i \equiv j \mod 2$ If both are chosen an odd number of edges is added to H' and $\texttt{parity}(H') = 1 - \texttt{parity}(G')$. This happens with probability $1/4$.

- $i \not\equiv j \mod 2$. Assume w.l.o.g. that i is odd and j is even. Then, if v_{n-1} is chosen and v_n is *not* chosen, an odd number of edges is added to H', and again $\texttt{parity}(H') = 1 - \texttt{parity}(G')$. This happens with probability $1/4$.

On the other hand, if neither v_{n-1} nor v_n is chosen then $\texttt{parity}(G') = \texttt{parity}(H')$, and this happens with probability $1/4$. So upon revealing the last two vertices, the parity of H' is flipped with probability at least $1/4$ and not flipped with probability at least $1/4$, independently of what happens in H'. Let F denote the event that it is flipped (i.e., that $\texttt{parity}(H') \neq \texttt{parity}(G')$. Then,

$$\begin{aligned}
\mathbb{P}[\texttt{parity}(G') = 1] &= \mathbb{P}[\texttt{parity}(G') = 1 | \texttt{parity}(H') = 1]\mathbb{P}[\texttt{parity}(H') = 1] \\
&\quad + \mathbb{P}[\texttt{parity}(G') = 1 | \texttt{parity}(H') = 0]\mathbb{P}[\texttt{parity}(H') = 0] \\
&= \mathbb{P}[\overline{F} | \texttt{parity}(H') = 1]p + \mathbb{P}[F | \texttt{parity}(H') = 0](1-p) \\
&= \mathbb{P}[\overline{F}]p + \mathbb{P}[F](1-p) \quad \text{by independence} \\
&\geq 1/4(p + 1 - p) = 1/4
\end{aligned}$$

as desired. □

This finishes the proof of Theorem 3.4.3. Since this result has 1-sided error, we can easily boost the $\frac{1}{4}$ probability to any constant, and Corollary 3.4.5 follows immediately.

Corollary 3.4.5. *Any graph can be verified by a randomized algorithm with error ϵ using $O(\log(\frac{1}{\epsilon}))$* **EC** *queries.*

3.4.1 Relation to Fingerprinting

Suppose A and B are $n \times n$ matrices over a field $\mathbb{F}$. It is known that if $A \neq B$, then for a vector $v \in \{0,1\}^n$ chosen uniformly at random we have

$$\mathbb{P}[Av \neq Bv] \geq 1/2.$$

This is Freivalds' fingerprinting technique [43]. It is was originally developed as a technique for verifying matrix multiplications, and can be used for testing for equality of any two matrices.

An easy extension of this method says that for vectors $v, w \in \{0,1\}^n$ chosen independently at random, if $A \neq B$ we have

$$\begin{aligned}\mathbb{P}[w^TAv \neq w^TBv] &= \mathbb{P}[w^TAv \neq w^TBv|Av = Bv]\mathbb{P}[Av = Bv] \\ &\quad + \mathbb{P}[w^TAv \neq w^TBv|Av \neq Bv]\mathbb{P}[Av \neq Bv] \\ &\geq 0 \times \mathbb{P}[Av = Bv] + \frac{1}{2} \times \frac{1}{2} \\ &= \frac{1}{4}\end{aligned}$$

This bears a strong resemblance to graph verification with **EC** queries. Let A and B be the incidence matrices of G and G^*, respectively. Then an **EC** query Q corresponds to multiplication on the left and right by the characteristic vector of Q, and the algorithm becomes: choose $v \in \{0,1\}^n$ uniformly at random and return 'same' if and only if $v^TAv = v^TBv$. By Theorem 3.4.3 if $A \neq B$ then $Pr[v^TAv \neq v^TBv] \geq \frac{1}{4}$.

This raises a natural question. For *arbitrary* $n \times n$ matrices A and B over a field, if $A \neq B$, then for a vector $v \in \{0,1\}^n$ chosen uniformly at random, is $\mathbb{P}[v^TAv \neq v^TBv] \geq 1/4$ (or some other constant > 0)?

This turns out not to be the case. Consider the two matrices

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \qquad B = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$A \neq B$, but it is not hard to check that for any vector $v \in \{0,1\}^n$, $v^TAv = v^TBv$. In fact, this holds true for adjacency matrices of 'opposite' directed cycles on > 3 vertices. A graph theoretic interpretation of this fact is that if the number of directed edges on any induced subset of the two opposite directed cycles is the same, then an **EC** query will always return the same answer for the two different cycles. Needless to say, this property is not limited to the adjacency matrices of directed cycles: in fact, it holds for any two matrices A and B such that $A + A^T = B + B^T$, since

$$v^T(A + A^T)v = v^TAv + v^TA^Tv = v^TAv + (v^TAv)^T = 2v^TAv$$

for all v, so that $v^TAv = v^TBv$ for all v.

Hence, we know that standard fingerprinting techniques do not imply Theorem 3.4.3. Furthermore, the proof to Theorem 3.4.3 generalizes easily to weighted graphs and a more general form of **EC** queries, where the answer to the query is the sum of the weights of its induced edges. Since any symmetric matrix can be viewed as an adjacency matrix of an undirected graph, we have the following fingerprinting technique for symmetric matrices.

Theorem 3.4.6. *Let A and B be $n \times n$ symmetric matrices over a field such that $A \neq B$,*[4] *then for v chosen uniformally at random from $v \in \{0,1\}^n$, $Pr[v^TAv \neq v^TBv] \geq \frac{1}{4}$.*

Proof. Let $C = A - B \neq 0$, and note that $v^TAv \neq v^TBv \iff v^TCv \neq 0$. Identify C with the weighted graph $G = (V, E)$, where $V = \{v_1 \dots v_n\}$ and $E = \{(u,v) : C(u,v) \neq 0\}$, and $\mathtt{wt}(u,v) = C(u,v)$. We proceed as in the proof of Lemma 3.4.4. Fix $v_1 \dots v_n$ so that $\mathtt{wt}(v_{n-1}, v_n) \neq 0$, and let H' be as before. Define:

$$\mathtt{wt}(H) = \sum_{(u,v) \in H} \mathtt{wt}(u,v); \quad \mathtt{wt}(w,H) = \sum_{(w,v)\in G, v \in H} \mathtt{wt}(w,v).$$

The first quantity is a generalization of `parity`, the second of the number of edges from a vertex to a subgraph. Let $T = \mathtt{wt}(v_{n-1}, H') + \mathtt{wt}(v_n, H') + \mathtt{wt}(v_{n-1}, v_n)$, and consider two cases:

- $T = 0$. Since $\mathtt{wt}(v_{n-1}, v_n) \neq 0$, we know that at least one of the other terms must be nonzero. Assume w.l.o.g. that this is $\mathtt{wt}(v_n, H')$. So choosing v_n but not v_{n-1} is will make $\mathtt{wt}(G') \neq \mathtt{wt}(H')$, and this happens with probability 1/4.

- $T \neq 0$. Choosing both v_n and v_{n-1} sets $\mathtt{wt}(G') = \mathtt{wt}(H') + T \neq \mathtt{wt}(H')$. This happens with probability 1/4.

Again, we choose *neither* vertex with probability 1/4, in which case $\mathtt{wt}(G') = \mathtt{wt}(H')$. Finally,

[4]Or, more generally, any matrices A and B with $A + A^T \neq B + B^T$.

$$\begin{aligned}
\mathbb{P}[\texttt{wt}(G') \neq 0] &= \mathbb{P}[\texttt{wt}(G') \neq 0 | \texttt{wt}(H') \neq 0]\mathbb{P}[\texttt{wt}(H') \neq 0] \\
&\quad + \mathbb{P}[\texttt{wt}(G') \neq 0 | \texttt{wt}(H') = 0]\mathbb{P}[\texttt{wt}(H') = 0] \\
&\geq \mathbb{P}[\texttt{wt}(G') = \texttt{wt}(H') | \texttt{wt}(H') \neq 0]\mathbb{P}[\texttt{wt}(H') \neq 0] \\
&\quad + \mathbb{P}[\texttt{wt}(G') \neq \texttt{wt}(H') | \texttt{wt}(H') = 0]\mathbb{P}[\texttt{wt}(H') = 0] \\
&= \mathbb{P}[\texttt{wt}(G') = \texttt{wt}(H')]\mathbb{P}[\texttt{wt}(H') \neq 0] \\
&\quad + \mathbb{P}[\texttt{wt}(G') \neq \texttt{wt}(H')]\mathbb{P}[\texttt{wt}(H') = 0] \qquad \text{by independence} \\
&\geq 1/4(\mathbb{P}[\texttt{wt}(H') = 0] + \mathbb{P}[\texttt{wt}(H') \neq 0]) = 1/4
\end{aligned}$$

as desired. □

3.5 Discussion

There is a tantalizing asymptotic gap of $O(\lg n)$ in our bounds for **EC** queries for learning the partition of the graph. It would also be interesting to know under which, if any, query models it is easier to learn the number of components than the partition itself. Another open question asked by Bouvel, Grebinski, and Kucherov [29], and one that we leave open, is whether a hidden graph of *average* degree d can be learned with $O(dn)$ **EC** queries.[5] This has since been answered by Choi and Kim [34], who show that any graph with m edges can be learned non-adaptively using $O\left(\frac{m \log(n)}{\log(m)}\right)$ queries. Their algorithm, however, is not polynomial time, and find finding an efficient algorithm that meets this bound remains an interesting problem.

Some other problems left to be considered are learning and verification problems for other restricted classes of graphs. For example, of theoretical interest is the

[5]Bouvel, Grebinski, and Kucherov [29] restrict themselves to a non-adaptive framework, where all queries must be asked simultaneously.

problem of verifying trees with **ED** queries. There is an obvious $O(n)$ brute-force algorithm, but it may be possible to do better. Also, other classes of graphs have been studied in the literature (see the Section 3.2) including Hamiltonian paths, matchings, stars, and cliques. It may be revealing to see the power of the queries considered herein for learning and verifying these restricted classes of graphs.

It would also be useful to look at this problem from a more economic perspective. Since edge counting queries are strictly more powerful than edge detecting queries, they ought to be more expensive in some natural framework. Taking costs into account and allowing learners to be able to choose queries with the goal of both learning the graph and minimizing cost should be an interesting research direction.

Finally, our work shows that graph verification is possible even for many classes of directed graphs. It would be interesting to redefine these queries for directed graphs and explore their power.

Chapter 4

Learning Analog Circuits by Injecting Values

4.1 Introduction

We consider learning large-alphabet and analog acyclic circuits in the value injection model introduced in [14]. In this model, we may inject values of our choice on any subset of wires, but we can only observe the one output of the circuit. However, the value injection query algorithms in that paper for boolean and constant alphabet networks do not lift to the case when the size of the alphabet is polynomial in the size of the circuit.

One motivation for studying the boolean network model is learning gene regulatory networks. In a boolean model, each node in a gene regulatory network can represent a gene whose state is either active or inactive. However, genes may have a large number of states of activity. Constant-alphabet network models may not adequately capture the information present in these networks. The motivates our interest in larger alphabets.

Akutsu *et al.* [2] and Ideker *et al.* [53] consider the discovery problem that models the experimental capability of gene disruption and overexpression. In such experiments, it is desirable to manipulate as few genes as possible. In the particular models considered in these papers, node states are fully observable – the gene expression data gives the state of every node in the network at every time step. Their results show that in this model, for bounded fan-in or sufficiently restricted gene functions, the problem of learning the structure of a network is tractable.

In contrast, there is ample evidence that learning boolean circuits solely from input-output behaviors may be computationally intractable. Kearns and Valiant [60] show that specific cryptographic assumptions imply that **NC1** circuits and **TC0** circuits are not PAC learnable in polynomial time. These negative results have been strengthened to the setting of PAC learning with membership queries [23], even with respect to the uniform distribution [64]. Furthermore, positive learnability results exist only for fairly limited classes, including propositional Horn formulas [21], general read once Boolean formulas [22], and decision trees [31], and those for specific distributions, including **AC0** circuits [73], DNF formulas [54], and **AC0** circuits with a limited number of majority gates [55].[1]

Thus, Angluin *et al.* [14] look at the relative contributions of full observation and full control of learning boolean networks. Their model of value injection allows full control and restricted observation, and it is the model we study in this chapter. Interestingly, their results show that this model gives the learner considerably more power than with only input-output behaviors but less than the power with full observation. In particular, they show that with value injection queries, **NC1** circuits and **AC0** circuits are exactly learnable in polynomial time, but their negative results show that depth limitations are necessary.

A second motivation behind our work is to study the relative importance of

[1]Algorithms in both [55] and [73] for learning **AC0** circuits and their variants run in quasi-polynomial time.

the parameters of the models for learnability results. The impact of alphabet size on learnability becomes a natural point of inquiry, and ideas from fixed parameter tractability are very relevant [40, 77].

In this chapter we show positive learnability results for bounded fan-in, large alphabet, arbitrary depth circuits given some restrictions on the topology of the target circuit. Specifically, we show that transitively reduced circuits and circuits with bounded shortcut width (as defined in section 4.2) are exactly learnable in polynomial time, and we present evidence that shortcut width is the correct parameter to look at for large-alphabet circuits. We also show that analog circuits of bounded fan-in, logarithmic depth, and small shortcut width that satisfy a Lipschitz condition are approximately learnable in polynomial time.

4.2 Preliminaries

4.2.1 Circuits

We give a general definition of acyclic circuits whose wires carry values from a set Σ. For each nonnegative integer k, a **gate function** of arity k is a function from Σ^k to Σ. A **circuit** C consists of a finite set of wires $w_1, \ldots, w_n$, and for each wire w_i, a gate function g_i of arity k_i and an ordered k_i-tuple $w_{\sigma(i,1)}, \ldots, w_{\sigma(i,k_i)}$ of wires, the **inputs** of w_i. We define w_n to be the **output wire** of the circuit. We may think of wires as outputs of gates in C.

The **unpruned graph** of a circuit C is the directed graph whose *vertices* are the wires and whose *edges* are pairs (w_i, w_j) such that w_i is an input of w_j in C. A wire w_i is **output-connected** if there is a directed path in the unpruned graph from that wire to the output wire. Wires that are not output-connected cannot affect the output value of a circuit. The **graph** of a circuit C is the subgraph of its unpruned graph induced by the output-connected wires.

A circuit is **acyclic** if its graph is acyclic. In this chapter we consider only acyclic circuits.[2] If u and v are vertices such that $u \neq v$ and there is a directed path from u to v, then we say that u is an **ancestor** of v and that v is a **descendant** of u. The **depth** of an output-connected wire w_i is the length of a longest path from w_i to the output wire w_n. The depth of a circuit is the maximum depth of any output-connected wire in the circuit. A wire with no inputs is an **input wire**; its **default value** is given by its gate function, which has arity 0 and is constant.

We consider the property of being transitively reduced [1] and a generalization of it: bounded shortcut width. Let G be an acyclic directed graph. An edge (u, v) of G is a **shortcut edge** if there exists a directed path in G of length at least two from u to v. G is **transitively reduced** if it contains no shortcut edges. A circuit is transitively reduced if its graph is transitively reduced. Note that in a transitively reduced circuit, for every output-connected wire w_i, no ancestor of w_i is an input of any descendant of w_i, otherwise there would be a shortcut edge in the graph of the circuit.

The **shortcut width** of a wire w_i is the number of wires w_j such that w_j is both an ancestor of w_i and an input of a descendant of w_i. (Note that we are counting wires, or vertices, not edges.) The **shortcut width** of a circuit C is the maximum shortcut width of any output-connected wire in C. A circuit is transitively reduced if and only if it has shortcut width 0. A circuit's shortcut width turns out to be a key parameter in its learnability by value injection queries.

[2]In Chapter 5 we consider a variant of this model for circuits that allow cycles.

4.2.2 Experiments on Circuits

Let C be a circuit. An **experiment** e is a function mapping each wire of C to $\Sigma \cup \{*\}$, where $*$ is not an element of Σ. If $e(w_i) = *$, then the wire w_i is **free** in e; otherwise, w_i is **fixed** in e. If e is an experiment that assigns $*$ to wire w, and $\sigma \in \Sigma$, then $e|_{w=\sigma}$ is the experiment that is equal to e on all wires other than w, and fixes w to σ. We define an ordering $\preceq$ on $\Sigma \cup \{*\}$ in which all elements of Σ are incomparable and precede $*$, and lift this to the componentwise ordering on experiments. Then $e_1 \preceq e_2$ if every wire that e_2 fixes is fixed to the same value by e_1, and e_1 may fix some wires that e_2 leaves free.

For each experiment e we inductively define the value $w_i(e) \in \Sigma$, of each wire w_i in C under the experiment e as follows. If $e(w_i) = \sigma$ and $\sigma \neq *$, then $w_i(e) = \sigma$. Otherwise, if the values of the input wires of w_i have been defined, then $w_i(e)$ is defined by applying the gate function g_i to them, that is, $w_i(e) = g_i(w_{\sigma(i,1)}(e), \ldots, w_{\sigma(i,k_i)}(e))$. Because C is acyclic, for any experiment this uniquely defines $w_i(e) \in \Sigma$ for all wires w_i. We define the value of the circuit to be the value of its output wire, that is, $C(e) = w_n(e)$ for every experiment e.

Let C and C' be circuits with the same set of wires and the same value set Σ. If $C(e) = C'(e)$ for every experiment e, then we say that C and C' are **behaviorally equivalent**. To define approximate equivalence, we assume that there is a metric d on Σ mapping pairs of values from Σ to a real-valued distance between them. If $d(C(e), C'(e)) \leq \epsilon$ for every experiment e, then we say that C and C' are **ϵ-equivalent**.

We consider two principal kinds of circuits. A **discrete circuit** is a circuit for which the set Σ of wire values is a finite set. An **analog circuit** is a circuit for which $\Sigma = [0, 1]$. In this case we specify the distance function as $d(x, y) = |x - y|$.

4.2.3 The Learning Problems

We consider the following general learning problem. There is an unknown target circuit C^* drawn from a known class of possible target circuits. The set of wires $w_1, \ldots, w_n$ and the value set Σ are given as input. The learning algorithm may gather information about C^* by making calls to an oracle that will answer value injection queries. In a **value injection query**, the algorithm specifies an experiment e and the oracle returns the value of $C^*(e)$. The algorithm makes a value injection query by listing a set of wires and their fixed values; the other wires are assumed to be free, and are not explicitly listed. The goal of a learning algorithm is to output a circuit C that is either exactly or approximately equivalent to C^*.

In the case of learning discrete circuits, the goal is behavioral equivalence and the learning algorithm should run in time polynomial in n. In the case of learning analog circuits, the learning algorithm has an additional parameter $\epsilon > 0$, and the goal is ϵ-equivalence. In this case the learning algorithm should run in time polynomial in n and $1/\epsilon$.

4.3 Learning Large-Alphabet Circuits

In this section we consider the problem of learning a discrete circuit when the alphabet Σ of possible values is of size $n^{O(1)}$. In Section 4.5 we reduce the problem of learning an analog circuit whose gate functions satisfy a Lipschitz condition to that of learning a discrete circuit over a finite value set Σ; the number of values is $n^{\Theta(1)}$ for an analog circuit of depth $O(\log n)$. Using this approach, in order to learn analog circuits of even moderate depth, we need learning algorithms that can handle large alphabets.

The algorithm Circuit Builder [14] uses value injection queries to learn acyclic discrete circuits of unrestricted topology and depth $O(\log n)$ with constant fan-in and constant alphabet size in time polynomial in n. However, the approach of [14] to building a sufficient set of experiments does not generalize to alphabets of size $n^{O(1)}$ because the total number of possible settings of side wires along a test path grows superpolynomially. In fact, we give evidence in Section 4.3.1 that this problem becomes computationally intractable for an alphabet of size $n^{\Theta(1)}$.

In turn, this negative result justifies a corresponding restriction on the topology of the circuits we consider. We first show that a natural top-down algorithm using value-injection queries learns transitively reduced circuits with arbitrary depth, constant fan-in and alphabet size $n^{O(1)}$ in time polynomial in n. We then give a generalization of this algorithm to circuits that have a constant bound on their shortcut width. The topological restrictions do not result in trivial classes; for example, every leveled graph is transitively reduced.

Combining these results with the discretization from Section 4.5, we obtain an algorithm using value-injection queries that learns, up to ϵ-equivalence, analog circuits satisfying a Lipschitz condition with constant bound, depth bounded by $O(\log n)$, having constant fan-in and constant shortcut width in time polynomial in n and $1/\epsilon$.

4.3.1 Hardness for Large-Alphabet Circuits with Unrestricted Topology

We give a reduction that turns a large-alphabet circuit learning algorithm into a clique tester. Because the clique problem is complete for the complexity class $W[1]$ (see [40, 77]), this suggests the learning problem may be computationally intractable for classes of circuits with large alphabets and unrestricted topology.

The Reduction. Suppose the input is (G, k), where $k \geq 2$ is an integer and $G = (V, E)$ is a simple undirected graph with $n \geq 3$ vertices, and the desired output is whether G contains a clique of size k. We construct a circuit C of depth $d = \binom{k}{2}$ as follows. The alphabet Σ is V; let v_0 be a particular element of V. Define a gate function g with three inputs s, u, and v as follows: if (u, v) is an edge of G, then the output of g is equal to the input s; otherwise, the output is v_0. The wires of C are $s_1, \ldots, s_{d+1}$ and $x_1, x_2, \ldots, x_k$. The wires x_j have no inputs; their gate functions assign them the default value v_0. For $i = 1, \ldots, d$, the wire s_{i+1} has corresponding gate function g, where the s input is s_i, and the u and v inputs are the i-th pair (x_ℓ, x_m) with $\ell < m$ in the lexicographic ordering. Finally, the wire s_1 has no inputs, and is assigned some default value from $V - \{v_0\}$. The output wire is s_{d+1}.

To understand the behavior of C, consider an experiment e that assigns values from V to each of $x_1, \ldots, x_k$, and leaves the other wires free. The gates g pass along the default value of s_1 as long as the values $e(x_\ell)$ and $e(x_m)$ are an edge of G, but if any of those checks fail, the output value will be v_0. Thus the default value of s_1 will be passed all the way to the output wire if and only if the vertex values assigned to $x_1, \ldots, x_k$ form a clique of size k in G.

We may use a learning algorithm as a clique tester as follows. Run the learning algorithm using C to answer its value-injection queries e. If for some queried experiment e, the values $e(x_1), \ldots, e(x_k)$ form a clique of k vertices in G, stop and output the answer "yes." If the learning algorithm halts and outputs a circuit without making such a query, then output the answer "no." Clearly a "yes" answer is correct, because we have a witness clique. And if there is a clique of size k in G, the learning algorithm must make such a query, because in that case, the default value assigned to s_1 cannot otherwise be learned correctly; thus, a "no" answer is correct. Then we have the following.

Theorem 4.3.1. *If for some nonconstant computable function $d(n)$ an algorithm using value injection queries can learn the class of circuits of at most n wires, alphabet size s, fan-in bound 3, and depth bound $d(n)$ in time polynomial in n and s, then there is an algorithm to decide whether a graph on n vertices has a clique of size k in time $f(k)n^{\alpha}$, for some function f and constant α.*

Proof. (Note that the function f need not be a polynomial.) On input (G, k), where G has n vertices, we construct the circuit C as described above, which has alphabet size $s' = \binom{n}{2}$, depth $d' = \binom{k}{2}$ and number of wires $n' = d' + k + 1$. We then evaluate $d(1), d(2), \ldots$ to find the least N such that $d(N) \geq n'$. Such an N may be found because $d(n)$ is a nonconstant computable function; the value of N depends only on k. We run the learning algorithm on the circuit C padded with inessential wires to have N wires, using C to answer the value injection queries. By hypothesis, because $d' \leq d(N)$, the learning algorithm runs in time polynomial in N and s'. Its queries enable us to answer correctly whether G has a clique of size k. The total running time is bounded by $f(k)n^{\alpha}$ for some function f and some constant α. □

Because the clique problem is complete for the complexity class $W[1]$, a polynomial time learning algorithm as hypothesized in the theorem for any non-constant computable function $d(n)$ would imply fixed-parameter tractability of all the problems in $W[1]$ [40, 77]. However, we show that restricting the circuit to be transitively reduced (Theorem 4.3.5), or more generally, of bounded shortcut width (Theorem 4.4.1), avoids the necessity of a depth bound at all.[3]

Remark. A natural question is whether a pattern graph less dense than a clique might avoid squaring the parameter k in the reduction. In fact, there is a polynomial-time algorithm to test whether a graph contains a path of length $O(\log n)$ [7]. A

[3]The target circuit C constructed in the reduction is of shortcut width $k - 1$.

reduction similar to the one above can be used to test for the presence of an arbitrary graph H on k vertices $\{1, \ldots, k\}$ as an induced subgraph in G. The gate with inputs x_ℓ and x_m tests for an edge in G (if (ℓ, m) is an edge of H) or tests whether the vertices are distinct and not an edge of G (if (ℓ, m) is not an edge of H.) Note that regardless of the number of edges in H, the all-pairs structure is necessary to verify that the distinctness of the vertices assigned to $x_1, \ldots, x_k$.

4.3.2 Distinguishing Paths

This section develops some properties of distinguishing paths, making no assumptions about shortcut width. Let C^* be a circuit with n wires, an alphabet Σ of cardinality s, and fan-in bounded by a constant k. An arbitrary gate function for such a circuit can be represented by a **gate table** with s^k entries, giving the value of the gate function for each possible k-tuple of input symbols.

Experiment e **distinguishes** σ from τ for w if e sets w to $*$ and

$$C^*(e|_{w=\sigma}) \neq C^*(e|_{w=\tau}).$$

If such an experiment exists, the values σ and τ are **distinguishable** for wire w; otherwise, σ and τ are **indistinguishable** for w.

A **test path** π for a wire w in C^* consists of a directed path of wires from w to the output wire, together with an assignment giving fixed values from Σ to some set S of other wires; S must be disjoint from the set of wires in the path, and each element of S must be an input to some wire beyond w along the path. The wires in S are the **side wires** of the test path π. The **length** of a test path is the number of edges in its directed path. There is just one test path of length 0, consisting of the output wire and no side wires.

We may associate with a test path π the partial experiment p_π that assigns $*$

to each wire on the path, and the specified value from Σ to each wire in S. An experiment e **agrees with** a test path π if e extends the partial experiment p_π, that is, p_π is a subfunction of e. We also define the experiment e_π that extends p_π by setting all the other wires to $*$.

If π is a test path and V is a set of wires disjoint from the side wires of π, then V is **functionally determining** for π if for any experiment e agreeing with π and leaving the wires in V free, for any experiment e' obtained from e by setting the wires in V to fixed values, the value of $C^*(e')$ depends only on the values assigned to the wires in V. That is, the values on the wires in V determine the output of the circuit, given the assignments specified by p_π. A test path π for w is **isolating** if $\{w\}$ is functionally determining for π. The following property is then clear.

Lemma 4.3.2. *If π is an isolating test path for w then the set V of inputs of w is functionally determining for π.*

We define a **distinguishing path** for wire w and values $\sigma, \tau \in \Sigma$ to be an isolating test path π for w such that e_π distinguishes between σ and τ for w. The significance of distinguishing paths is indicated by the following lemma, which is analogous to Lemma 10 of [14].

Lemma 4.3.3. *Suppose σ and τ are distinguishable for wire w. Then for any minimal experiment e distinguishing σ from τ for w, there is a distinguishing path π for wire w and values σ and τ such that the free wires of e are exactly the wires of the directed path of π, and e agrees with π.*

Proof. We prove the result by induction on the depth of the wire w; it clearly holds when w is the output wire. Suppose the result holds for all wires at depth at most d in C^*, and assume that w is a wire at depth $d+1$ and that e is any minimal experiment that distinguishes σ from τ for w. Every free wire in e must be reachable from w; using the acyclicity of C^*, let w' be a free wire in e whose only free input is w. Let $\sigma' = w'(e|_{w=\sigma})$ and $\tau' = w'(e|_{w=\tau})$. Because e is minimal, we must have $\sigma' \neq \tau'$.

Moreover, the minimality of e also implies that

$$C^*(e|_{w=\sigma,w'=\sigma'}) = C^*(e|_{w=\tau,w'=\sigma'})$$

and

$$C^*(e|_{w=\sigma,w'=\tau'}) = C^*(e|_{w=\tau,w'=\tau'}),$$

so we must have

$$C^*(e|_{w=\sigma,w'=\sigma'}) \neq C^*(e|_{w=\sigma,w'=\tau'}),$$

which means that the experiment $e' = e|_{w=\sigma}$ distinguishes σ' from τ' for w'. The experiment e' is also a minimal experiment distinguishing σ' from τ' for w'; otherwise, e would not be minimal. The depth of w' is at most d, so by induction, there is a distinguishing path π' for wire w' and values σ' and τ' such that the free wires of e' are exactly the wires of the directed path π', and e' agrees with π'.

We may extend π' to π as follows. Add w to the start of the directed path in π'. The side wires of π are the side wires of π' with their settings in π', together with any inputs of w' (other than w) that are not already side wires of π', set as in e. The result is clearly an isolating test path for w that distinguishes σ from τ. Also the wires in the directed path of π are precisely the free wires of e, and e agrees with π, which completes the induction. □

Conversely, a shortest distinguishing path yields a minimal distinguishing experiment, as follows. This does not hold for circuits of general topology without the restriction to a shortest path.

Lemma 4.3.4. *Let π be a shortest distinguishing path for wire w and values σ and τ. Then the experiment e obtained from p_π by setting every unspecified wire to an arbitrary fixed value is a minimal experiment distinguishing σ from τ for w.*

Proof. Because π is a distinguishing path, w is functionally determining for π, so e distinguishes σ from τ for w. If e is not minimal, then there is some minimal $e' \preceq e$ such that e' distinguishes σ and τ for w. By Lemma 4.3.3, there is a distinguishing path for w and values σ and τ whose path wires are the free wires of e'. This contradicts the assumption that π as a shortest path distinguishing σ from τ for w. □

4.3.3 The Distinguishing Paths Algorithm

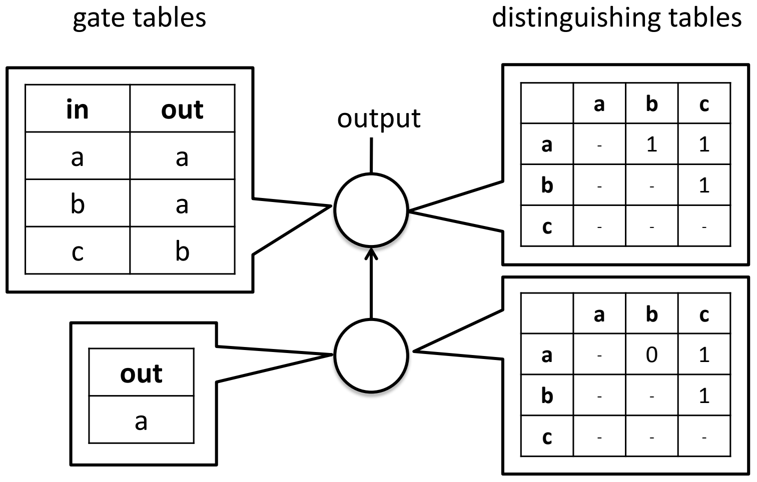
In this section we develop the Distinguishing Paths Algorithm.

Theorem 4.3.5. *The Distinguishing Paths Algorithm learns any transitively reduced circuit with n wires, alphabet size s, and fan-in bound k, with $O(n^{2k+1}s^{2k+2})$ value injection queries and time polynomial in the number of queries.*

Lemma 4.3.6. *If C^* is a transitively reduced circuit and π is a test path for w in C^*, then none of the inputs of w is a side wire of π.*

Proof. Every side wire u of π is an input to some wire beyond w in the directed path of wires, that is, to some descendant of w. If u were an input to w, then u would be an ancestor of w and an input to a descendant of w, contradicting the assumption that C^* is transitively reduced. □

The Distinguishing Paths Algorithm builds a directed graph G whose vertices are the wires of C^*, in which an edge (v, w) represents the discovery that v is an input of w in C^*. The algorithm also keeps for each wire w a **distinguishing table** T_w with $\binom{s}{2}$ entries, one for each unordered pair of values from Σ. The entry for (σ, τ)



	a	b	c
a	-	1	1
b	-	-	1
c	-	-	-

	a	b	c
a	-	0	1
b	-	-	1
c	-	-	-

Figure 4.1: The gate and distinguishing tables of a circuit over alphabet $\{a, b, c\}$.

in T_w is 1 or 0 according to whether or not a distinguishing path has been found to distinguish values σ and τ on wire w. Stored together with each 1 entry is a corresponding distinguishing path and a bit marking whether the entry is processed or unprocessed. An illustration of a circuit's gate and distinguishing tables appears in Figure 4.1.

At each step, for each distinguishing table T_w that has unprocessed 1 entries, we try to extend the known distinguishing paths to find new edges to add to G and new 1 entries and corresponding distinguishing paths for the distinguishing tables of inputs of w. Once every 1 entry in every distinguishing table has been marked processed, the construction of distinguishing tables terminates. Then a circuit C is constructed with graph G by computing gate tables for the wires; the algorithm outputs C and halts.

To extend a distinguishing path for a wire w, it is necessary to find an input wire of w. Given a distinguishing path π for wire w, an input v of w is **relevant** with respect to π if there are two experiments e_1 and e_2 that agree with π, that set the inputs of w to fixed values, that differ only by assigning different values to v, and are such that $C^*(e_1) \neq C^*(e_2)$. Let $V(\pi)$ denote the set of all inputs v of w that are relevant with respect to π. It is only relevant inputs of w that need be found, as shown by the following.

Lemma 4.3.7. *Let π be a distinguishing path for w. Then $V(\pi)$ is functionally determining for π.*

Proof. Suppose $V(\pi)$ is not functionally determining for π. Then there are two experiments e_1 and e_2 that agree with π and assign $*$ to all the wires in $V(\pi)$, and an assignment a of fixed values to the wires in $V(\pi)$ such that the two experiments e_1' and e_2' obtained from e_1 and e_2 by fixing all the wires in $V(\pi)$ as in a have the property that $C^*(e_1') \neq C^*(e_2')$.

Because π is a distinguishing path for w, the set V of all inputs of w is functionally determining for π. Thus, e_1' and e_2' must induce different values for at least one input of w (that cannot be in $V(\pi)$.) Let e_1'' be e_1' with all of the inputs of w fixed to their induced values in e_1', and similarly for e_2'' with respect to e_2'. Now $C^*(e_1'') = C^*(e_1') \neq C^*(e_2') = C^*(e_2'')$, and both e_1'' and e_2'' fix all the inputs of w. By changing the differing fixed values of the inputs of w one by one from their setting in e_1'' to their setting in e_2'', we can find a single input wire u of w such that changing just its value changes the output of the circuit. The resulting two experiments witness that u is an input of w relevant with respect to π, which contradicts the fact that u is not in $V(\pi)$. $\square$

Given a distinguishing path π for wire w, we define its corresponding **input experiments** E_π to be the set of all experiments e that agree with π and set up to $2k$ additional wires to fixed values and set the rest of the wires free. Note that each of these experiments fix at most $2k$ more values than are already fixed in the distinguishing path. Consider all pairs (V, Y) of disjoint sets of wires not set by p_π

such that $|V| \leq k$ and $|Y| \leq k$; for every possible way of setting $V \cup Y$ to fixed values, there is a corresponding experiment in E_π.

Find-Inputs. We now describe a procedure, Find-Inputs, that uses the experiments in E_π to find all the wires in $V(\pi)$. Define a set V of at most k wires not set by p_π to be **determining** if for every disjoint set Y of at most k wires not set by p_π and for every assignment of values from Σ to the wires in $V \cup Y$, the value of C^* on the corresponding experiment from E_π is determined by the values assigned to wires in V, independent of the values assigned to wires in Y. Find-Inputs finds all determining sets V and outputs their intersection.

Lemma 4.3.8. *Given a distinguishing path π for w and its corresponding input experiments E_π, the procedure Find-Inputs returns $V(\pi)$.*

Proof. First, there is at least one set in the intersection, because if V_w is the set of all inputs to w in C^*, then by Lemma 4.3.6 and the acyclicity of C^*, no wires in V_w are set in p_π. By Lemma 4.3.2, V_w is functionally determining for π and therefore determining, and, by the bound on fan-in, $|V_w| \leq k$, so V_w will be one such set V. Let V^* denote the intersection of all determining sets V.

Clearly, every wire in V^* is an input of w, because $V^* \subseteq V_w$. To see that each $v \in V^*$ is relevant with respect to π, consider the set $V' = V_w - \{v\}$ of inputs of w other than v. This set must not appear in V^*(because $v \in V^*$), so it must be that for some pair (V', Y) there are two experiments e_1 and e_2 in E_π that give the same fixed assignments to V' and different fixed assignments to Y, and are such that $C^*(e_1) \neq C^*(e_2)$. Then $v(e_1) \neq v(e_2)$, because $V' \cup \{v\}$ is functionally determining for π. Thus, if we take e_1' to be e_1 with v fixed to $v(e_1)$ and e_2' to be e_1 with v fixed to $v(e_2)$, we have two experiments that witness that v is relevant with respect to π. Thus $V^* \subseteq V(\pi)$.

Conversely, suppose $v \in V(\pi)$ and that V^* does not include v. Then there is some set V in the intersection that excludes v. Also, there are two experiments e_1 and e_2 that agree with π, set the inputs of w to fixed values and differ only on v, such that $C^*(e_1) \neq C^*(e_2)$. Let Y consist of all the inputs of w that are not in V; clearly $v \in Y$, none of the elements of Y are set in p_π and $|Y| \leq k$. There is an experiment $e_1' \in E_\pi$ for the pair (V, Y) that sets the inputs of w as in e_1 and the other wires of V arbitrarily, and another experiment $e_2' \in E_\pi$ for the pair (V, Y) that agrees with e_1 except in setting v to its value in e_2. These two experiments set the inputs of w as in e_1 and e_2 respectively, and the inputs of w are functionally determining for π, so we have $C^*(e_1') = C^*(e_1) \neq C^*(e_2) = C^*(e_2')$. This is a contradiction: V would not have been included in the intersection. Thus $V(\pi) \subseteq V^*$, concluding the proof. □

Find-Paths. We now describe a procedure, Find-Paths, that takes the set $V(\pi)$ of all inputs of w relevant with respect to π, and searches, for each triple consisting of $v \in V(\pi)$ and $\sigma, \tau \in \Sigma$, for two experiments e_1 and e_2 in E_π that fix all the wires of $V(\pi) - \{v\}$ in the same way, but set v to σ and τ, respectively, and are such that $C^*(e_1) \neq C^*(e_2)$. On finding such a triple, the distinguishing path π for w can be extended to a distinguishing path π' for v by adding v to the start of the path, and making all the wires in $V(\pi) - \{v\}$ new side wires, with values fixed as in e_1. If this gives a new 1 for entry (σ, τ) in the distinguishing paths table T_v, then we change the entry, add the corresponding distinguishing path for v to the table, and mark it unprocessed. We have to verify the following.

Lemma 4.3.9. *Suppose π' is a path produced by Find-Paths for wire v and values σ and τ. Then π' is a distinguishing path for wire v and values σ, τ.*

Proof. Because v is an input to w in C^*, prefixing v to the path from π is a path of wires from v to the output wire in C^*. Because v is an input of w, by Lemma 4.3.6, v

is not among the side wires S for π. The new side wires are those in $V(\pi) - \{v\}$, and because they are inputs of w, by Lemma 4.3.6 they are not already on the path for π nor in the set S. Thus, π' is a test path. The new side wires are fixed to values with the property that changing v between σ and τ produces a difference at the output of C^*. Because by Lemma 4.3.7, $V(\pi)$ is functionally determining for π, the test path π' is isolating for v. Thus π' is a distinguishing path for wire v and values σ and τ. □

The Distinguishing Paths Algorithm initializes the simple directed graph G to have the set of wires of C^* as its vertex set, with no edges. It initializes T_w to all 0's, for every non-output wire w. Every entry in T_{w_n} is initialized to 1, with a corresponding distinguishing path of length 0 with no side wires, and marked as unprocessed. The Distinguishing Paths Algorithm is summarized in Algorithm 1; the procedure Construct-Circuit is described below.

Algorithm 1 The Distinguishing Paths Algorithm

```
Initialize $G$ to have the wires as vertices and no edges.
Initialize $T_{w_n}$ to all 1's, marked unprocessed.
Initialize $T_w$ to all 0's for all non-output wires $w$.
while there is an unprocessed 1 entry $(\sigma, \tau)$ in some $T_w$ do
  Let $\pi$ be the corresponding distinguishing path.
  Perform all input experiments $E_\pi$.
  Use Find-Inputs to determine the set $V(\pi)$.
  Add any new edges $(v, w)$ for $v \in V(\pi)$ to $G$.
  Use Find-Paths to find extensions of $\pi$ for elements of $V(\pi)$.
  for each extension $\pi'$ that gives a new 1 entry in some $T_v$ do
    Put the new 1 entry in $T_v$ with distinguishing path $\pi'$.
    Mark this new 1 entry as unprocessed.
  Mark the 1 entry for $(\sigma, \tau)$ in $T_w$ as processed.
Use Construct-Circuit with $G$ and the tables $T_w$ to construct a circuit $C$.
Output $C$ and halt.
```

We now show that when processing of the tables terminates, the tables T_w are correct and complete. We first consider the correctness of the 1 entries.

Lemma 4.3.10. *After the initialization, and after each new* 1 *entry is placed in a distinguishing table, every* 1 *entry in a distinguishing table* T_w *for* (σ, τ) *has a corresponding distinguishing path* π *for wire* w *and values* σ *and* τ.

Proof. This condition clearly holds after the initialization, because the distinguishing path consisting of just the output wire and no side wires correctly distinguishes every distinct pair of values from Σ. Then, by induction on the number of new 1 entries in distinguishing path tables, when an existing 1 entry in T_w gives rise to a new one in T_v, then the path π from T_w is a correct distinguishing path for w. Thus, by Lemma 4.3.8, the Find-Inputs procedure correctly finds the set $V(\pi)$ of inputs of w relevant with respect to π, and by Lemma 4.3.9, the Find-Paths procedure correctly finds extensions of π to distinguishing paths π' for elements of $V(\pi)$. Thus, any new 1 entry in a table T_v will have a correct corresponding distinguishing path. □

A distinguishing table T_w is **complete** if for every pair of values $\sigma, \tau \in \Sigma$ such that σ and τ are distinguishable for w, T_w has a 1 entry for (σ, τ).

Lemma 4.3.11. *When the Distinguishing Paths Algorithm terminates,* T_w *is complete for every wire* w *in* C^*.

Proof. Assume to the contrary and look at a wire w at the smallest possible depth such that T_w is incomplete; assume it lacks a 1 entry for the pair (σ, τ), which are distinguishable for w. Note that w cannot be the output wire. Because the depth of w is at least one more than the depth of any descendant of w, all wires on all directed paths from w to the root have complete distinguishing tables. By Lemma 4.3.10, all the entries in all distinguishing tables are also correct.

Because σ and τ are distinguishable for w, by Lemma 4.3.3 there exists a distinguishing path π for wire w and values σ and τ. On this distinguishing path, w is followed by some wire x. The wires along π starting with x and omitting any side wires that are inputs of x is a distinguishing path for wire x and values σ' and τ', where σ' is the value that x takes when $w = \sigma$ and τ' is the value that x takes when $w = \tau$ in any experiment agreeing with π.

Because x is a descendant of w, its distinguishing table T_x is complete and correct. Thus, there exists in T_x a 1 entry for (σ', τ') and a corresponding distinguishing path π_x. This 1 entry must be processed before the Distinguishing Paths Algorithm terminates. When it is processed, two of the input experiments for π_x will set the inputs of x in agreement with π, and set w to σ and τ respectively. Thus, w will be discovered to be a relevant input of x with respect to π, and a distinguishing experiment for wire w and values σ and τ will be found, contradicting the assumption that T_w never gets a 1 entry for (σ, τ). Thus, no such wire w can exist and all the distinguishing tables are complete. □

Construct-Circuit. Now we show how to construct a circuit C behaviorally equivalent to C^* given the graph G and the final distinguishing tables. G is the graph of C, determining the input relation between wires. Note that G is a subgraph of the graph of C^*, because edges are added only when relevant inputs are found.

Gate tables for wires in C will keep different combinations of input values and their corresponding output. Since some distinguishing tables for wires may have 0 entries, we will record values in gate tables up to equivalence, where σ and τ are in the same equivalence class for w if they are indistinguishable for w. We process one wire at a time, in arbitrary order. We first record, for one representative σ of each equivalence class of values for w, the outputs $C^*(e_\pi|_{w=\sigma})$ for all the distinguishing paths π in T_w. Given a setting of the inputs to w (in C), we can tell which equivalence class of values of w it should map to as follows. For each distinguishing path π in T_w, we record the output of C^* for the experiment equal to e_π with the inputs of w set to the given fixed values and $w = *$. For this setting of the inputs, we set the output in w's gate table to be the value of σ with recorded outputs matching these outputs for all π. Repeating this for every setting of w's inputs completes w's gate table, and we continue to the next gate.

Lemma 4.3.12. *Given the graph G and distinguishing tables as constructed in the Distinguishing Paths Algorithm, the procedure Construct-Circuit constructs a circuit C behaviorally equivalent to C^*.*

Proof. Assume to the contrary that C is not behaviorally equivalent to C^*, and let e be a minimal experiment (with respect to $\preceq$) such that $C(e) \neq C^*(e)$. Using the acyclicity of C, there exists a wire w that is free in e and its inputs (in C) are fixed in e. Let σ be the value that w takes for experiment e in C, and let τ be the value that w takes for experiment e in C^*. Because e is minimal, $\sigma \neq \tau$.

Now $C(e) = C(e|_{w=\sigma})$ and $C^*(e) = C^*(e|_{w=\tau})$, but because e is minimal, we must have $C(e|_{w=\sigma}) = C^*(e|_{w=\sigma})$, so $C^*(e|_{w=\sigma}) = C(e) \neq C^*(e) = C^*(e|_{w=\tau})$ and e distinguishes σ from τ for w. Thus, because the distinguishing tables used by Construct-Circuit are complete and correct, there must be a distinguishing path π for (σ, τ) in T_w.

Consider the set V of inputs of w in C^*. If in the experiment e_π the wires in V are set to the values they take in e in C^*, then the output of C^* is $C^*(e|_{w=\tau})$. If V' is the set of inputs of w in C, then $V' \subseteq V$, and if in the experiment e_π the wires in V' are set to their fixed values in e, then the output of C^* is $C^*(e|_{w=\sigma})$, where σ is the representative value chosen by Construct-Circuit. Thus, there must be a wire $v \in V - V'$ relevant with respect to π, but then v would have been added to the circuit graph as an input to w when π was processed, a contradiction. Thus, C is behaviorally equivalent to C^*. □

We analyze the total number of value injection queries used by the Distinguishing Paths Algorithm; the running time is polynomial in the number of queries. To construct the distinguishing tables, each 1 entry in a distinguishing table is processed once. The total number of possible 1 entries in all the tables is bounded by ns^2. The processing for each 1 entry is to take the corresponding distinguishing path π and construct the set E_π of input experiments, each of which consists of choosing up to $2k$ wires and setting them to arbitrary values from Σ, for a total of $O(n^{2k}s^{2k})$ queries to construct E_π. Thus, a total of $O(n^{2k+1}s^{2k+2})$ value injection queries are used to construct the distinguishing tables.

To build the gate tables, for each of n wires, we try at most s^2 distinguishing path experiments for at most s values of the wire, which takes at most s^3 queries. We then run the same experiments for each possible setting of the inputs to the wire, which takes at most $s^k s^2$ experiments. Thus Construct-Circuit requires a total of $O(n(s^3+s^{k+2}))$ experiments, which are already among the ones made in constructing the distinguishing tables. Note that every experiment fixes at most $O(kd)$ wires, where d is the depth of C^*. This concludes the proof of Theorem 4.3.5.

4.4 Circuits with Bounded Shortcut Width

In this section we describe the Shortcuts Algorithm, which generalizes the Distinguishing Paths Algorithm to circuits with bounded shortcut width as follows.

Theorem 4.4.1. *The Shortcuts Algorithm learns the class of circuits having n wires, alphabet size s, fan-in bound k, and shortcut width bounded by b using a number of value injection queries bounded by $(ns)^{O(k+b)}$ and time polynomial in the number of queries.*

When C^* is not transitively reduced, there may be edges of its graph that are important to the behavior of the circuit, but are not completely determined by the behavior of the circuit. For example, the three circuits given in Figure 1 of [14] are behaviorally equivalent, but have different graphs; a behaviorally correct circuit cannot be constructed with just the edges that are common to the three circuit graphs. Thus, the Shortcuts Algorithm focuses on finding a **sufficient** set of experiments for C^*, and uses Circuit Builder [14] to build the output circuit C.

A gate with gate function g and input wires $u_1, \ldots, u_\ell$ is **wrong** for w in C^* if there exists an experiment e in which the wires $u_1, \ldots, u_\ell$ are fixed, say to values $u_j = \sigma_j$, and w is free, and there is an experiment e such that $C^*(e) \neq C^*(e|_{w=g(\sigma_1,\ldots,\sigma_\ell)})$, and is **correct** otherwise. The experiment e, which we term a **witness experiment** for this gate and wire, shows that no circuit C using this gate for w can be behaviorally equivalent to C^*. A set E of experiments is **sufficient** for C^* if for every wire w and every candidate gate that is wrong for w, E contains a witness experiment for this gate and this wire.

Lemma 4.4.2. *[14] If the input E to Circuit Builder is a sufficient set of experiments for C^*, then the circuit C that it outputs is behaviorally equivalent to C^*.*

The need to guarantee witness experiments for all possible wrong gates means that the Shortcuts Algorithm will learn a set of distinguishing tables for the restriction of C^* obtained by fixing $u_1, \ldots, u_\ell$ to values $\sigma_1, \ldots, \sigma_\ell$ for every choice of at most k wires u_j and every choice of assignments of fixed values to them.

On the positive side, we can learn quite a bit about the topology of a circuit C^* from its behavior. An edge (v, w) of the graph of C^* is **discoverable** if it is the initial edge on some minimal distinguishing experiment e for v and some values σ_1 and σ_2. This is a behaviorally determined property; all circuits behaviorally equivalent to C^* must contain all the discoverable edges of C^*.

Because e is minimal, w must take on two different values, say τ_1 and τ_2 in $e|_{v=\sigma_1}$ and $e|_{v=\sigma_2}$ respectively.. Moreover, $e|_{v=\sigma_1}$ must be a minimal experiment distinguishing τ_1 from τ_2 for w; this purely behavioral property is both necessary and sufficient for a pair (v, w) to be a discoverable edge.

Lemma 4.4.3. *The pair (v, w) is a discoverable edge of C^* if and only if there is an experiment e and values $\sigma_1, \sigma_2, \tau_1, \tau_2$ such that e is a minimal experiment distinguishing σ_1 from σ_2 for v, and $e|_{v=\sigma_1}$ is a minimal experiment distinguishing τ_1 from τ_2 for w.*

We now generalize the concept of distinguishing paths to leave potential shortcut wires unassigned. Assume that C^* is a circuit, with n wires, an alphabet Σ of s symbols, fan-in bound k, and shortcut width bound b. A **test path with shortcuts** π is a directed path of wires from some wire w to the output, a set S of **side wires** assigned fixed values from Σ, and a set K of **cut wires** such that S and K are disjoint and neither contains w, and each wire in $S \cup K$ is an input to at least one wire beyond w in the directed path of wires. One intuition for this is that the wires in K could have been set as side wires, but we are treating them as possible shortcut wires, not knowing whether they will end up being shortcut wires or not. As before, we define p_π to be the partial experiment setting all the wires in the directed path to $*$ and all the wires in S to the specified fixed values. Also, e_π is the experiment that extends p_π by setting every unspecified wire to $*$. The **length** of π is the number of edges in its directed path of wires.

Let π be a test path with shortcuts of nonzero length, with the directed path $v_1, v_2, \ldots, v_r$, side wires S, and cut wires K. The **1-suffix** of π is the test path π' obtained as follows. The directed path is $v_2, \ldots, v_r$, the side wires S' are all elements of S that are inputs to at least one of $v_3, \ldots, v_r$, and the cut wires K' are all elements of $K \cup \{v_1\}$ that are inputs to at least one of $v_3, \ldots, v_r$. If $t < r$, the t-suffix of π is obtained inductively by taking the 1-suffix of the $(t-1)$-suffix of π. A **suffix** of π is the t-suffix of π for some $1 \leq t < r$.

If π is a test path with shortcuts and V is a set of wires disjoint from the side wires of π, then V is **functionally determining** for π if for any experiment that agrees with π and fixes all the wires in V, the value output by C^* depends only on the values assigned to the wires in V. Then π is **isolating** if the set of wires $\{w\} \cup K$ is functionally determining for π. Note that if we assign fixed values to all the wires in K, we get an isolating test path for w.

Lemma 4.4.4. *Let π be an isolating test path with shortcuts. If π' is any suffix of π then π' is isolating.*

Proof. Let π have directed path $v_1, \ldots, v_r$, side wires S and cut wires K. Let π' be the 1-suffix of π, with side wires S' and cut wires K'. The values of v_1 and K determine the output of C^* in any experiment that agrees with π. The only wires in $\{v_1\} \cup K$ that are not in K' are inputs of v_2 that are not also inputs of some $v_3, \ldots, v_r$. Similarly, the only wires in $S - S'$ are inputs of v_2 that are not also inputs of some $v_3, \ldots, v_r$. By setting the value of v_2, we make these input wires irrelevant, so $\{v_2\} \cup K'$ are functionally determining for π'. $\square$

In this setting, what we want to distinguish are pairs of assignments to (w, B), where B is a set of wires not containing w. An **assignment** to (w, B) is just a function with domain $\{w\} \cup B$ and co-domain Σ. If a is an assignment to (w, B) and e is an experiment mapping w and every wire in B to $*$, then by $(e|_a)$ we denote the experiment e' such that $e'(v) = a(v)$ if $v \in \{w\} \cup B$ and $e'(v) = e(v)$ otherwise. If a_1 and a_2 are two assignments to (w, B), then the experiment e **distinguishes** a_1 from a_2 if e maps $\{w\} \cup B$ to $*$ and $C^*(e|_{a_1}) \neq C^*(e|_{a_2})$.

Let π be a **distinguishing path with shortcuts** with initial path wire w, side wires S and cut wires K. Then π is **distinguishing** for the pair (w, B) and assignments a_1 and a_2 to (w, B) if $K \subseteq B$, $B \cap S = \emptyset$, π is isolating and e_π distinguishes a_1 from a_2. If such a path exists, we say (w, B) is **distinguishable** for a_1 and a_2.

Note that this condition requires that π not set any of the wires in B. When $B = \emptyset$, these definitions reduce to the previous ones.

4.4.1 The Shortcuts Algorithm

Overview of algorithm. We assume that at most k wires $u_1, \ldots, u_\ell$ have been fixed to values $\sigma_1, \ldots, \sigma_\ell$, and denote by C^* the resulting circuit. The process described is repeated for every choice of wires and values. Like the Distinguishing Paths Algorithm, the Shortcuts Algorithm builds a directed graph G whose vertices are the wires of C^*, in which an edge (v, w) is added when v is discovered to be an input to w in C^*; one aim of the algorithm is to find all the discoverable edges of C^*.

Distinguishing tables. The Shortcuts Algorithm maintains a distinguishing table T_w for each wire w. Each entry in T_w is indexed by a triple, (B, a_1, a_2), where B is a set of at most b wires not containing w, and a_1 and a_2 are assignments to (w, B). If an entry exists for index (B, a_1, a_2), it contains π, a distinguishing path with shortcuts that is distinguishing for (w, B), a_1 and a_2. The entry also contains a bit marking the entry as processed or unprocessed.

Initialization. The distinguishing table T_{w_n} for the output wire is initialized with entries indexed by $(\emptyset, \{w_n = \sigma\}, \{w_n = \tau\})$ for every pair of distinct symbols $\sigma, \tau \in \Sigma$, each containing the distinguishing path of length 0 with no side wires and no cut wires. Each such entry is marked as unprocessed. All other distinguishing tables are initialized to be empty.

While there is an entry in some distinguishing table T_w marked as unprocessed, say with index (B, a_1, a_2) and π the corresponding distinguishing path with shortcuts, the Shortcuts Algorithm processes it and marks it as processed. To process it, the algorithm first uses the entry try to discover any new edges (v, w) to add to the graph G; if a new edge is added, all of the existing entries in the distinguishing table for wire w are marked as unprocessed. Then the algorithm attempts to find new distinguishing paths with shortcuts obtained by extending π in all possible ways. If an extension is found to a test path with shortcuts π' that is distinguishing for (w', B'), a'_1 and a'_2, if there is not already an entry for (B', a'_1, a'_2), or, if π' is of shorter length that the existing entry for (B', a'_1, a'_2), then its entry is updated to π' and marked as unprocessed. When all possible extensions have been tried, the algorithm marks the entry in T_w for (B, a_1, a_2) as processed.

In contrast to the case of the Distinguishing Paths Algorithm, the Shortcuts Algorithm tries to find a *shortest* distinguishing path with shortcuts for each entry in the table. When no more entries marked as unprocessed remain in any distinguishing table, the algorithm constructs a set of experiments E as described below, calls Circuit Builder on E, outputs the resulting circuit C, and halts.

Processing an entry. Let (B, a_1, a_2) be the index of an unprocessed entry in a distinguishing table T_w, with corresponding distinguishing path with shortcuts, π, where the side wires of π are S and the cut wires are K. Note that $K \subseteq B$ and $S \cap B = \emptyset$. Let the set E_π consist of every experiment that agrees with π, arbitrarily fixes the wires in K, and arbitrarily fixes up to $2k$ additional wires not in K and not set by p_π, and sets the remaining wires free. There are $O((ns)^{2k}s^b)$ experiments in E_π; the algorithm makes a value injection query for each of them.

Finding relevant inputs. For every assignment a of fixed values to K, the resulting path π_a is an isolating test path for w. We use the Find-Inputs procedure (in Section 4.3.3) to find relevant inputs to w with respect to π_a, and let $V^*(\pi)$ be the union of the sets of wires returned by Find-Inputs over all assignments a to K. For each $v \in V^*(\pi)$, add the edge (v, w) to G if it is not already present, and mark all existing entries in all the distinguishing tables for wire w as unprocessed.

Lemma 4.4.5. *The wires in $V^*(\pi)$ are inputs to w and the wires in $V^*(\pi) \cup K$ are functionally determining for π.*

Proof. This follows from Lemma 4.3.8, because for each assignment a to K, the resulting π_a is an isolating path for w, and any wires in the set returned by Find-Inputs are indeed inputs to w. Also, for each assignment a to K, the set $V(\pi_a)$ is functionally determining for π_a, and is contained in $V^*(\pi)$. $\square$

Additional input test. The Shortcuts Algorithm makes an additional input test if π distinguishes two assignments a_1 and a_2 such that there is a wire $w' \in K$ such that a_1 and a_2 agree on every wire other than w. Let π' be the distinguishing path obtained from π by fixing every wire in $K - \{w\}$ to its value in a_1. If there is an experiment e agreeing with π' and setting w to $*$ and fixing every element of $V(\pi')$, and two values σ_1 and σ_2 such that $C^*(e|_{v=\sigma_1}) \neq C^*(e|_{v=\sigma_2})$, and, moreover, for every $\tau \in \Sigma$, $C^*(e|_{w=\tau,v=\sigma_1}) = C^*(e|_{w=\tau,v=\sigma_2})$, then add edge (v, w) to G if it is not already present, and mark all the existing entries in the distinguishing table for wire w as unprocessed.

Lemma 4.4.6. *If edge (v, w) is added to G by this additional input test, then v is an input of w in C^*.*

Proof. Note that w must take two different values, say τ_1 and τ_2, in the experiments $e|_{v=\sigma_1}$ and $e|_{v=\sigma_2}$; thus, w must be a descendant of v. Moreover, $C^*(e|_{w=\tau_1,v=\sigma_1}) = C^*(e|_{w=\tau_1,v=\sigma_2})$ and $C^*(e|_{w=\tau_2,v=\sigma_1}) = C^*(e|_{w=\tau_2,v=\sigma_2})$, from which we conclude that $C^*(e|_{w=\tau_1,v=\sigma_1}) \neq C^*(e|_{w=\tau_2,v=\sigma_1})$.

If v is not an input of w, then let U be the set of all inputs of w. In e, if we set $v = \sigma_1$ and $w = *$ and the wires in U as induced by $e|_{v=\sigma_1}$, then $w = \tau_1$ and the output of C^* is $C^*(e|_{w=\tau_1,v=\sigma_1})$. If we then change the values on wires in U one by one to their values in $e|_{v=\sigma_2}$, because the final result have $w = \tau_2$ and output $C^*(e|_{v=\sigma_1,w=\tau_2})$, there must be an input u such that fixing the other inputs to w and changing u's value changes the output with respect to $e|_{v=\sigma_1}$. Thus, u is a relevant input with respect to the distinguishing path $\pi_{\{v=\sigma_1\}}$, and must be in the set $V(\pi)$. This is a contradiction, because wires in $V(\pi)$ are fixed in e, and u must change value from $e|_{v=\sigma_1}$ and $e|_{v=\sigma_2}$. Thus v must be an input of w. □

Extending a distinguishing path. After finding as many inputs of w as possible using π, the Shortcuts Algorithm attempts to extend π as follows. Let $I_G(w)$ be the set of all inputs of w in G. For each pair (w', K') such that $w' \in I_G(w)$ and K' is a set of at most b wires not containing w' such that $K' \subseteq I_G(w) \cup K$ and K' is disjoint from the path wires and side wires of π, we let $S_0 = (K \cup V^*(\pi)) - (\{w'\} \cup K')$. Note that the set of wires in $S_0 \cup K' \cup \{w'\}$ is functionally determining for π.

For each assignment a of fixed values to S_0, the algorithm extends π to π' as follows. It adds w' to the start of the directed path, adds S_0 to the set of side wires (fixed to the values assigned by a) and takes the cut wires to be K'. Note that every wire in K' is an input to some wire beyond w on the path. Because w' is an input of w, and all of the wires in $V^*(\pi) \cup K$ are accounted for among (w, K') and S', and all of the wires in S' are inputs to w or wires beyond w on the path, the result is an isolating test path with shortcuts for (w', K').

The algorithm then searches through all triples (B', a_1', a_2') where B' is a set of at most b wires not containing w', and a_1' and a_2' are assignments to (w', B'), to discover whether π' is distinguishing for (w', B'), a_1' and a_2'. If so, the algorithm checks the distinguishing table $T_{w'}$ and creates or updates the entry for index (B', a_1', a_2') as follows. If there is no such entry, one is created with π'. If there already is an entry and π' is shorter than the path in the entry, then the entry is changed to contain π'. If the entry is created or changed by this operation, it is marked as unprocessed.

When all possible extensions of π have been tried, the entry in T_w for (B, a_1, a_2) is marked as processed.

Correctness and completeness. We define the distinguishing table T_w to be **correct** if whenever π is an entry in T_w for (B, a_1, a_2), then π is a distinguishing path with shortcuts that is distinguishing for (w, B), a_1 and a_2. For each wire w, let $B(w)$ denote the set of shortcut wires of w in the target circuit C^*. If π is a distinguishing path with shortcuts such that every edge in its directed path is discoverable, we say that π is **discoverable**. The distinguishing T_w table is **complete** if for every pair a_1 and a_2 of assignments to $(w, B(w))$ that are distinguishable by a discoverable path, there is an entry in T_w for index $(B(w), a_1, a_2)$.

Lemma 4.4.7. *When Shortcuts Algorithm finishes the processing of the distinguishing tables, every distinguishing table T_w is correct and complete.*

Proof. The correctness follows inductively from the correctness of the initialization of T_{w_n} by the arguments given above. To prove completeness, we prove the following stronger condition about the distinguishing tables when the Shortcuts Algorithm finishes processing them: (1) for every wire w and every pair a_1 and a_2 of assignments to $(w, B(w))$ that are distinguishable by a discoverable path, the entry for $(B(w), a_1, a_2)$ is a shortest discoverable distinguishing path with shortcuts that is distinguishing for $(w, B(w))$, a_1 and a_2.

Condition (1) clearly holds for T_{w_n} after it is initialized, and this table does not change thereafter. Assume to the contrary that condition (1) does not hold and let w be a wire of the smallest possible depth such that T_w does not satisfy condition (1). Note that w is not the output wire.

There must be assignments a_1 and a_2 for $(w, B(w))$ that are distinguishable by a discoverable path such that in T_w, the entry for $(B(w), a_1, a_2)$ is either nonexistent or not as short as possible. Let π be a shortest possible discoverable distinguishing path with shortcuts that is distinguishing for $(w, B(w))$, a_1 and a_2. Let S be the side wires of π, with assignment a, and let K be the cut wires of π. Then we have $K \subseteq B$ and $S \cap B = \emptyset$. Because w is not the output wire, the directed path in π is of length at least 1. Let π' be the 1-suffix of π, with initial vertex w', side wires S' and cut wires K'. Note that (w, w') must be a discoverable edge and that π' is also discoverable. By Lemma 4.4.4, π' is isolating.

For any two assignments a_1' and a_2' to $(w', B(w'))$ such that $a_j'(u)$ is the value of u in $e_\pi|_{a_j}$ for each $u \in \{w'\} \cup K'$, we have that π' is distinguishing for $(w', B(w'))$, a_1' and a_2'. To see this, note that $\{w'\} \cup K'$ is functionally determining for π', so $C^*(e_{\pi'}|_{a_j'}) = C^*(e_\pi|_{a_j})$ for $j = 1, 2$, and these latter two values are distinct. Let a_j' denote the assignment to $(w', B(w'))$ induced by the experiment $e_\pi|_{a_j}$ for $j = 1, 2$; these two assignments have the required property.

Because the depth of w' is smaller than the depth of w, condition (1) must hold for $T_{w'}$, and the distinguishing table for $T_{w'}$ must contain an entry for $(B(w'), a_1', a_2')$ that is a shortest discoverable distinguishing path with shortcuts π'' that is distinguishing for $B(w')$, a_1' and a_2'. Note that the length of π'' is at most the length of π minus 1.

We argue that the discoverable edge (w, w') must be added to G by the Shortcuts Algorithm. This edge is the first edge on a minimal experiment e distinguishing σ_1 from σ_2 for w. This corresponds to a distinguishing path ρ with no cut edges distinguishing σ_1 from σ_2 for w, and every edge of this path is also discoverable. There are two cases, depending on whether w is a shortcut of w' on the path or not.

If w is not a shortcut edge of w' on the path, then the 1-suffix of ρ will be a discoverable distinguishing path with no cut edges that is distinguishing for w', τ_1, and τ_2, where these are the values w' takes in $e|_{w=\sigma_j}$ for $j = 1, 2$. Because condition (1) holds for $T_{w'}$, there will be an entry in $T_{w'}$ containing a distinguishing path with shortcuts for $(w', B(w'))$ that distinguishes the two assignments that set $B(w')$ as in

e and set w' to τ_1 and τ_2. Because w is a relevant input with respect to ρ, the edge (w, w') will be added to G if it is not already present when ρ is processed.

If w is a shortcut edge of w' on the path, then the 1-suffix of ρ will be a discoverable distinguishing path with cut edges $\{w\}$ that is distinguishing for the assignments $\alpha_1 = \{w = \sigma_1, w' = \tau_1\}$ and $\alpha_2 = \{w = \sigma_2, w' = \tau_2\}$. Because $w \in B(w')$ and $T_{w'}$ satisfies condition (1), there will be an entry ρ in $T_{w'}$ for $(w', B(w'))$ that distinguishes the two assignments to $(w', B(w'))$ that agree with α_1 and α_2 on w' and w, and set every other element of $B(w')$ as in e. When the entry ρ is processed, the additional input test will discover the edge (w, w') and add it to the graph G if it is not already present. In fact, this shows that every discoverable edge (v, w') will eventually be discovered by the algorithm because $T_{w'}$ is complete.

Thus, we can be sure that the entry π'' will be (re)processed when every discoverable edge (v, w') is present in G, including (w, w'). When this happens, the entry π'' will be extended to a distinguishing path with shortcuts that is distinguishing for $(w, B(w))$, a_1 and a_2 and has length at most that of π. To see that this holds, note that if v is a side wire of π'', then it cannot be an ancestor of w' because otherwise it is a shortcut wire of w' and in $B(w')$, which is disjoint from the side wires of π''. Thus, the side wires of π'' cannot include any input of w' or any wire in $B(w)$, because all these wires are ancestors of w'. Moreover, since all the discoverable inputs to w' have been added to G, one of the possible extensions of π'' will set (some of) the inputs of w' in such a way that moving from assignment a_1 to assignment a_2 to $(w, B(w))$ with the other side gate settings of π'' will move from a_1' to a_2' for $(w', B(w'))$,

Thus, the entry (B, a_1, a_2) will exist and be of length at most the length of π when the algorithm finishes processing the distinguishing tables. This contradiction shows that all the distinguishing tables must be complete.

□

Building a circuit. When all the entries in all the distinguishing tables are marked as processed, the Shortcuts Algorithm constructs a set E of experiments. For every table T_w and every distinguishing path π for (B, a_1, a_2) in the table such that $a_1(u) = a_2(u)$ for every $u \in B$, and every set V of at most k wires not set by p_π and every assignment a to V, add to E the experiment $e_\pi|_a$, that extends e_π by the assignment a. After iterating the above process over all possible choices of at most k wires $u_1, \dots, u_\ell$ and assignments to them, the algorithm takes the union of all the resulting sets of experiments E and calls Circuit Builder [14] on this union and outputs the returned circuit C and halts.

Lemma 4.4.8. *The circuit C is behaviorally equivalent to the target circuit C^*.*

Proof. We show that the completeness of the distinguishing tables implies that the set E of experiments is sufficient, and apply Lemma 4.4.2 to conclude that C is behaviorally equivalent to C^*. Suppose a gate g with inputs $u_1, \dots, u_\ell$ is wrong for wire w in C^*. Then there exists a minimal experiment e that witnesses this; e fixes all the wires $u_1, \dots, u_\ell$, say as $u_j = \sigma_j$ for $j = 1, \dots, \ell$, sets the wire w free and is such that $C^*(e) \neq C^*(e|_{w=g(\sigma_1,\dots,\sigma_\ell)})$.

Consider the iteration of the table-building process for the circuit C^* with the restriction $u_j = \sigma_j$ for $j = 1, \dots, \ell$. In this circuit, e distinguishes between $w = \sigma$ and $w = \tau$, where σ is the value w takes in C^* for e, and $\tau = g(\sigma_1, \dots, \sigma_\ell)$. Note that the free wires of e form a directed path of discoverable edges. Because the table T_w is complete, there will be a distinguishing path π with shortcuts for $(w, B(w))$ for assignments a_1 and a_2 where $a_1(w) = \sigma$ and $a_2(w) = \tau$, and $a_1(v) = a_2(v)$ for all $v \in B(w)$. For every input v of w in C^* that is not among $u_1, \dots, u_\ell$, π does not set v, because it only sets wires that are inputs to descendants of w, and any input of w that is an input of a descendant of w is a short cut wire of w and therefore in $B(w)$. However, π does not set any wires in $B(w)$. Thus, among the choices of sets of at most k wires and values to set them to, there will be one that sets just the inputs (in C^*) of w as in e. The corresponding experiment e' in E will be a witness experiment eliminating the gate g with inputs $u_1, \ldots, u_\ell$, so the set of experiments to Circuit Builder is sufficient for C^*. □

Running time. To analyze the running time of the Shortcuts Algorithm, note that there are $O(n^k s^k)$ choices of at most k wires and values from Σ to fix them to; this bounds the number of iterations of the table building process. In each iteration, there are $O(n^{b+1} s^{2b+2})$ total entries in the distinguishing tables. Each entry in a distinguishing table may be processed several times: when it first appears in the table, and each time its distinguishing path is replaced by a shorter one, and each time a new input of w is discovered, for a total of at most $n + k$ times. Thus, the total number of entry-processing events by the algorithm in one iteration is $O((n + k)n^{b+1} s^{2b+2})$. Each such event makes $O((ns)^{2k} s^b)$ value injection queries, so $O((n + k)n^{2k+b+1} s^{2k+3b+2})$ value injection queries are made by the algorithm in each iteration, for a total of $O((n+k)n^{3k+b+1} s^{3k+3b+2})$ value injection queries made by the Shortcuts Algorithm. The number of experiments given as input to Circuit Builder is $O(n^{2k+b+1} s^{2k+2b+2})$, because each final entry may give rise to at most $O(n^k s^k)$ experiments in E in each iteration. This concludes the proof of Theorem 4.4.1.

4.5 Learning Analog Circuits via Discretization

We first give a simple example of an analog circuit. We then show how to construct a discrete approximation of an analog circuit, assuming its gate functions satisfy a Lipschitz condition with constant L, and apply the large-alphabet learning algorithm of Theorem 4.4.1, to get a polynomial-time algorithm for approximately learning an analog circuit with logarithmic depth, bounded fan-in and bounded shortcut width.

4.5.1 Example of an Analog Circuit

For example, let $\wedge(x, y) = xy$ for all $x, y \in [0, 1]$ and let $\vee(x, y) = x + y - xy$ for all $x, y \in [0, 1]$. (Note that these are polynomial representations of conjunction and disjunction when restricted to the values 0 and 1.) Then $\wedge$ and $\vee$ are analog functions of arity 2, and we define a circuit with 6 wires as follows. Let g_1 be the constant function 0.1, g_2 be the constant function 0.6 and g_3 be the constant function 0.8. These functions assign default values to the corresponding wires. Let g_4 be the function $\vee$, and let its pair of inputs be w_1, w_2. Let g_5 be the function $\vee$, and let its pair of inputs be w_2, w_3. Finally, let w_6 be the function $\wedge$, and let its pair of inputs be w_4, w_5. If we consider the experiment e_0 that assigns $*$ to every wire, we calculate the values $w_i(e_0)$ as follows. Using their default values,

$$w_1(e_0) = 0.1, w_2(e_0) = 0.6, w_3(e_0) = 0.8.$$

Then, because the inputs to w_4 and w_5 have defined values,

$$w_4(e_0) = \vee(0.1, 0.6) = 0.64, w_5(e_0) = \vee(0.6, 0.8) = 0.92.$$

Because the inputs to w_6 now have defined values,

$$w_6 = \wedge(0.64, 0.92) = 0.5888.$$

If we consider the experiment e_1 that fixes the value of w_5 to 0.2 and assigns $*$ to every other wire, then as before,

$$w_1(e_1) = 0.1, w_2(e_1) = 0.6, w_3(e_1) = 0.8, w_4(e_1) = 0.64.$$

However, because the value of w_5 is fixed to 0.2 in e_1,

$$w_5(e_1) = 0.2, w_6(e_1) = \wedge(0.64, 0.2) = 0.128.$$

4.5.2 A Lipschitz Condition

An analog function g of arity k satisfies a Lipschitz condition with constant L if for all $x_1, \ldots, x_k$ and $x'_1, \ldots, x'_k$ from $[0, 1]$ we have

$$|g(x_1, \ldots, x_k) - g(x'_1, \ldots, x'_k)| \leq L \max_i |x_i - x'_i|.$$

For example, the function $\wedge(x, y) = xy$ satisfies a Lipschitz condition with constant 2. A Lipschitz condition on an analog function allows us to bound the error of a discrete approximation to the function. For more on Lipschitz conditions, see [56].

Let m be a positive integer. We define a discretization function D_m from $[0, 1]$ to the m points $\{1/2m, 3/2m, \ldots, (2m-1)/2m\}$ by mapping x to the closest point in this set (choosing the smaller point if x is equidistant from two of them.) Then $|x - D_m(x)| \leq 1/2m$ for all $x \in [0, 1]$. We extend D_m to discretize analog experiments e by defining $D_m(*) = *$ and applying it componentwise to e. An easy consequence is the following.

Lemma 4.5.1. *If g is an analog function of arity k, satisfying a Lipschitz condition with constant L and m is a positive integer, then for all $x_1, \ldots, x_k$ in $[0, 1]$, $|g(x_1, \ldots, x_k) - g(D_m(x_1), \ldots, D_m(x_k))| \leq L/2m$.*

4.5.3 Discretizing Analog Circuits

We describe a discretization of an analog gate function in which the inputs and the output may be discretized differently. Let g be an analog function of arity k and r, s

be positive integers. The (r, s)-**discretization** of g is g', defined by

$$g'(x_1, \ldots, x_k) = D_r(g(D_s(x_1), \ldots, D_s(x_k))).$$

Let C be an analog circuit of depth d_{max} and let L and N be positive integers. Define $m_d = N(3L)^d$ for all nonnegative integers d. We construct a particular discretization C' of C by replacing each gate function g_i by its (m_d, m_{d+1})-discretization, where d is the depth of wire w_i. We also replace the value set $\Sigma = [0, 1]$ by the value set Σ' equal to the union of the ranges of D_{m_d} for $0 \leq d \leq d_{max}$. Note that the wires and tuples of inputs remain unchanged. The resulting discrete circuit C' is termed the (L, N)-**discretization** of C.

Lemma 4.5.2. *Let L and N be positive integers. Let C be an analog circuit of depth d_{max} whose gate functions all satisfy a Lipschitz condition with constant L. Let C' denote the (L, N)-discretization of C and let $M = N(3L)^{d_{max}}$. Then for any experiment e for C, $|C(e) - C'(D_M(e))| \leq 1/N$.*

Proof. Define $m_d = N(3L)^d$ for all nonnegative integers d; then $M = m_{d_{max}}$. We prove the stronger condition that for every experiment e for C and every wire w_i, if d is the depth of w_i, we have

$$|w_i(e) - w_i'(D_M(e))| \leq 1/m_d,$$

where $w_i(e)$ is the value of wire w_i in C for experiment e and $w_i'(D_M(e))$ is the value of wire w_i in C' for experiment $D_M(e)$. Because the output wire is at depth $d = 0$, this will imply that $C(e)$ and $C'(D_M(e))$ do not differ by more than $1/N$.

Let e be an arbitrary experiment for C. We proceed by downward induction on the depth d of w_i. When this quantity is d_{max}, the wire w_i is at maximum depth and has no inputs. The wire w_i is fixed in e if and only if it is fixed in $D_M(e)$, and in either case, the values assigned to w_i agree to within $1/2M < 1/m_{d_{max}}$. Now consider w_i at depth d, assuming inductively that the condition holds for all wires at greater depth. If w_i is fixed in e then it is fixed in $D_M(e)$ and the values assigned to it differ by at most $1/2M$. If w_i is free in e then it is free in $D_M(e)$. Consider the input wires to w_i, say $w_{j_1}, \ldots, w_{j_s}$; these are all at depth at least $d+1$, so by the inductive hypothesis

$$|w_{j_r}(e) - w'_{j_r}(D_M(e))| \leq 1/m_{d+1},$$

for $r = 1, \ldots, s$.

Note that

$$w_i(e) = g_i(w_{j_1}(e), \ldots, w_{j_s}(e))$$

and

$$w'_i(D_M(e)) = D_{m_d}(g_i(y_1, \ldots, y_s)),$$

where $y_r = D_{m_{d+1}}(w'_{j_r}(D_M(e)))$ for $r = 1, \ldots, s$. Note that by the properties of the discretization function,

$$|y_r - w'_{j_r}(D_M(e))| \leq 1/(2m_{d+1}).$$

By the Lipschitz condition on the gate function g_i we have

$$|g_i(w_{j_1}(e), \ldots, w_{j_s}(e)) - g_i(y_1, \ldots, y_s)| \leq L(3/2)(1/m_{d+1}) = 1/(2m_d),$$

because

$$|w_{j_r}(e) - y_r| \leq 1/m_{d+1} + 1/(2m_{d+1}).$$

Discretizing the output of g_i by D_{m_d} adds at most $1/(2m_d)$ to the difference, so

$$|g_i(w_{j_1}(e), \ldots, w_{j_s}(e)) - D_{m_d}(g_i(y_1, \ldots, y_x))| \leq 1/m_d,$$

that is,

$$|w_i(e) - w_i'(D_M(e))| \leq 1/m_d,$$

which completes the induction. □

This lemma shows that if every gate of C satisfies a Lipschitz condition with constant L, we can approximate C's behavior to within ϵ using a discretization with $O((3L)^d/\epsilon)$ points, where d is the depth of C. For $d = O(\log n)$, this bound is polynomial in n and $1/\epsilon$.

Theorem 4.5.3. *There is a polynomial time algorithm that approximately learns any analog circuit of n wires, depth $O(\log n)$, constant fan-in, gate functions satisfying a Lipschitz condition with a constant bound, and shortcut width bounded by a constant.*

4.6 Discussion

In this chapter, we extended the results of Angluin *et al.* [14] to the large-alphabet setting under the value injection query model. We showed topological conditions under which large-alphabet circuits are efficiently learnable and gave evidence that the conditions for shortcut width that we consider are necessary. We also showed that analog circuits can be approximated by large alphabet circuits, and that they can be approximately learned given a restriction on their depth.

In making these algorithms more practical, a goal for future research is to find ways to minimize the number of gates fixed in any given value injection query, especially for the types of circuits that occur in real-world networks. In manipulating gene regulatory networks, biologists can often only override a few wires at a time, and this presents challenges for learning.

Given that small-alphabet, large-alphabet, and analog circuits have been studied under the value-injection model, Angluin *et al.* [11] extend these results to Bayesian circuits, where probabilities are attached to the gates. In Chapter 5 we apply these queries to social networks. One interesting direction to explore is possible implications of this work for complexity theory. For example, does the class of circuits that are efficiently learnable with value injection queries represent a natural class of problems?

Chapter 5

Active Learning of Social Networks

5.1 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** [47, 62, 63], and it is the one we consider in this chapter. Informally, in the independent cascade model, each individual, or agent, has some probability of influencing each other agent. When targeted with a product, an agent becomes activated, and then attempts to influence each neighbor, 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; a suppressed agent 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 chapter 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 [23, 60, 64] and by models of gene suppression and gene overexpression in the study of gene interaction networks [2, 53] and was proposed by Angluin *et al.* [14]. 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. In Chapter 4 we extend these results to circuits with polynomial-size alphabets, and show that bounded shortcut width 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.* [11] 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 5.2 we formally define the model, value injection queries, and learning criteria. In Section 5.3 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 5.4 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 5.5 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 5.6 we give an approximation algorithm for learning influential sets of nodes in a social network.

5.2 Model

5.2.1 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 Chapter 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.

5.2.2 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 Chapter 4, in this chapter we consider networks that may have cycles.

5.2.3 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 chapter. 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.

We note that in comparing this process with the percolation model, the probabilistic decision whether to add a node to the queue is equivalent to the decision to make an edge "open" or "closed." At any point in the queuing process, however, only those edges that can affect $S(e)$ are explored. Because edges can be considered in any order in the percolation model, in the queuing process nodes can also be added (or even removed) from the queue in any order.

5.2.4 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 5.1 shows the social network defined by the adjacency matrix.

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

The output agent is a_4.

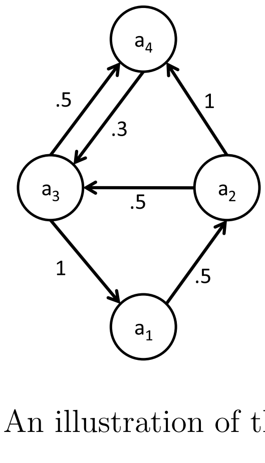


Figure 5.1: An illustration of the circuit S_1.

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, a_1 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, a_4 will become activated, 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.

5.2.5 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, $|S(e) - S'(e)| \leq \epsilon$.

5.2.6 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 chapter is on exact value injection queries.

5.2.7 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.

Let S be a social network, and 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.

5.2.8 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.

5.3 General Social Networks

In this section we prove the following theorem.

Theorem 5.3.1. *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.

5.3.1 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 in [14] and in our work in Chapter 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 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 5.3.2. *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 5.3.2 still holds when a takes probability c, not only when it is set to c by e.

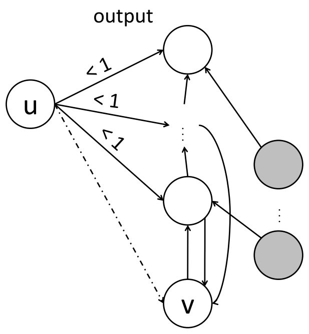


Figure 5.2: An illustration for Lemma 5.3.3. The shaded nodes are suppressed. The solid edges are known and the dashed edge is the edge to be computed.

Lemma 5.3.3. *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 $p_{(u,v)}$.

Proof. We can see this situation illustrated in Figure 5.2. 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 5.3.2. 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 2. 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

Algorithm 2 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 $G$
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 $G$.
Output $G$ and halt.
```

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 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 5.3.2 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 5.3.2, 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 $p_{(u,v)}$[1]

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

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

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 3 The Subroutine Find-Down-Edges from Algorithm 2 (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 $v_1, \ldots, v_k \in C$ sorted by increasing distance.
  Let $\pi_1, \ldots, \pi_k$ be shortest paths for $v_1, \ldots, v_k$ 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 5.3.3 the weight of $p_{(u,v)}$.
    Add $(u, v)$ to $G$ if $p_{u,v)} > 0$.
```

We restate the algorithm and give an inductive proof of its correctness. We do

[1]We note that this computation is a special case of Lemma 5.3.3.

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 5.3.3 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.

5.3.2 Arbitrary Social Networks

We now extend the ideas in Algorithm 2 to allow for edges of weight 1, giving us Algorithm 4. This algorithm is similar to Algorithm 2, except that Find-Level-Edges and Find-Down-Edges are combined into Find-Remaining-Edges. Algorithm 4 first builds a leveled graph of the social network as before, and the justification for Find-Up-Edges can be found in Section 5.3.1.

Algorithm 4 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 $C$.
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 5, 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 set 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 5 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.
            $\Pi = \Pi \cup \{\pi_{w_i,x_j}\}$.
    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 5.3.3.
    Set $T(w_i, x_j) = 0$.
    If $p_{(w_i,x_j)} > 0$ then add $(w_i, x_j)$ to $G$.
```

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 5.3.3, 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 4 performs at most $O(n^2)$ queries.

5.3.3 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 5.3.4. $\Omega(n^2)$ *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 4 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. □

5.4 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 5.4.1. *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 give 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 5.3.2. □

5.5 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 5.5.1. *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 5.3.

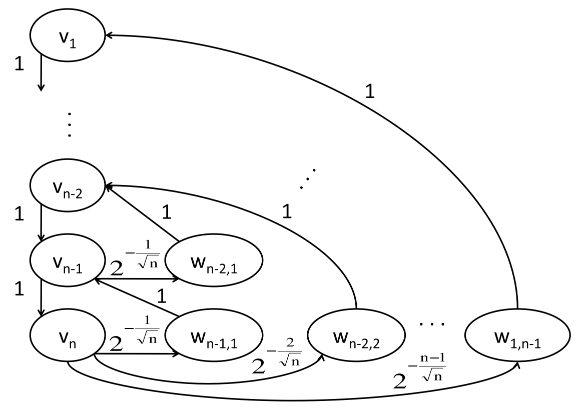


Figure 5.3: The social network S showing the limitations of excitation paths.

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 [80]. 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 $\forall\ n\ T(n) \geq c$. □

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.

5.6 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 5.6.1. *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 $o(\log n)$ [42], so is approximating the size of the smallest influential set. $\square$

Theorem 5.6.2. *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.*

Consider Algorithm 6. Assume the optimal solution X, where $S(e_X) \geq p$, has

Algorithm 6 An Algorithm for Finding a Set of Influential Nodes

```
Proof.  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 $v = \arg\max_{v_j \in V} S(e_I|_{v_j=1})$
    $I = I \cup \{v\}$
  Return $I$
```

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 5.6.3, 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 5.6.1, we can try to learn the influential nodes using Algorithm 6. If we set

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

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 [83].

We will use Lemma 5.6.3, a version of which is derived in [63]. A function f is **submodular** if $f(A \cup \{x\}) - f(A) \geq f(B \cup \{x\}) - f(B)$ whenever $A \subseteq B$.

Lemma 5.6.3. *$S(e_I)$ is a positive monotone, submodular function of I.* [63]

Corollary 5.6.4. *If p is the maximum influence of any k node set in the network, then Algorithm 6, with a threshold of 1, terminated after k steps, produces a set with influence $\geq (1 - \frac{1}{e})p$.*

Proof. Nemhauser *et al.* [76] 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 5.6.3. □

5.7 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. Noise in measurement also presents potential problems for real-world learners. The Angluin *et al.* [11] 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 5.5, 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** [63]. 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 larger queries more costly to the learner. Finally, graph algorithms often 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 of the target graph.

Chapter 6

Inferring Social Networks from Data

6.1 Introduction

In Chapter 5, we study inferring social networks in the active learning context. Yet, we often passively observe phenomena that hint at the structure of an underlying social network. In this chapter, we will consider a passive learner – one who must observe events as they unfold, without the ability to tamper with the network. Even though this dissertation focuses on learning in the active context, we can use social networks to compare the problem of network inference for the active and passive learners.

In the United States, the Centers for Disease Control and Prevention release various data[1] on persons affected by illnesses. Ideally, at least for the sake of learner, we would have information on exactly who is affected in each outbreak. If, as in Chapter 5, we consider persons as agents in a social network, we can try to learn the network's underlying structure from this information. In an idealized setting, we can

[1]For more on CDC statistics, we direct the reader to `www.cdc.gov/datastatistics/`.

consider persons infected during one outbreak as connected subsets in a population – for a disease spreads among persons in close proximity, and such data impose **constraints** on the topology of the network. Given such constraints, the problem would then be to find a maximum likelihood social network from the disease data. We note that unlike in Chapter 5, our task will be to learn the connections among the persons in the network, not the probability of them activating one another.

Thus, each set imposes a constraint on the network – namely that it be connected in its induced subgraph. The goal of the learner is to infer the most probable network that satisfies the connectivity requirements presented to it.

The learner can also have a prior belief about the probability each edge appears in the network. Let $p_{(u,v)}$ be the a priori probability of an edge appearing between nodes u and v. If the prior distribution on edges is independent and each edge appears with low probability, the goal of finding a maximum likelihood social network given the constraints is to find a set of edges E that satisfies all of the constraints, for which the quantity

$$\prod_{\{u,v\}\in E} p_{(u,v)} \prod_{\{u,v\}\notin E} \left(1 - p_{(u,v)}\right)$$

is maximized. If all quantities $p_{(u,v)}$ are sufficiently small, then this product is approximated by

$$\prod_{\{u,v\}\in E} p_{(u,v)}.$$

Taking the logarithm, we want a set of edges E that minimizes

$$\sum_{\{v,u\}\in E} - \log(p_{(u,v)}),$$

the sum of log-likelihood costs.

We can now think of the priors in terms of their log-likelihood **costs**. The goal of the learner becomes to construct the cheapest network (with respect to the prior costs) that satisfy the connectivity constraints.

This task presents various natural variations. We can consider what happens if the constraints are given to the learner in advance, and when the constraints arrive online. If they arrive online, they can be chosen adversarially or obliviously. We can imagine all edges in a network having the same cost, or that edges in a network have arbitrary costs. There are also cases when some information about the underlying social network is known, for example, that there exists a path that satisfies all the constraints.

6.1.1 Past Work

We note that this problem was considered by Korach and Stern [66] in another context where users in a **trusted set** in a network want to send messages among themselves without having the messages travel outside the group. Trusted sets of users can overlap, creating complicated structures, and these trusted sets form connectivity constraints in their subgraphs, imposing similar requirements to those in the social network inference problem.

In [66] Korach and Stern analyze the offline version of this problem for the case where the constraints can be satisfied by a tree. They give a polynomial time algorithm that finds the optimal solution in the tree-realizable case. In [67] Korach and Stern consider this problem for the case where the optimal solution forms a tree, and all of the connectivity constraints must be satisfied by stars. They pose as an open question the case of general graphs. Among our results, we answer their question in this chapter.

In a different line of work, Alon *et al.* [5] explore a wide range of network optimization problems, including generalized connectivity, cuts, facility location, and multicast. The connectivity problem they study involves ensuring a network with fractional edge weights has a flow of 1 over cuts specified by the constraints. In [4], Alon *et al.* also study approximation algorithms for the Online Set Cover problem, which has connections to Network Inference problems which we explore in this chapter. In [51] Gupta *et al.* also consider a network design problem for pairwise vertex connectivity constraints.

In the area of active learning, the problem of discovering networks from connectivity queries has been much studied in [3, 6, 20, 27, 49] and Chapter 3. In active learning of hidden networks, the object of the algorithm is to learn the network exactly. Our model is similar, except the algorithm has only the constraints it is given, and the task is to output the most likely network consistent with the constraints.

6.1.2 Preliminaries

In this chapter, we consider the following **Network Inference** problem. V is a set of vertices, and for each undirected edge $e = (v_i, v_j)$, c_e is the cost of constructing edge e. A collection of connectivity constraints $S = \{S_1, S_2, \ldots, S_r\}$ is given, where each S_i is a subset of V. The task is to construct a set E of edges between vertices of V such that for each i, the set S_i induces a connected subgraph of $G = (V, E)$. The quality of the solution is measured by comparing the sum of the costs of all the edges in E with the optimal cost of satisfying all the connectivity constraints.

In the offline version of the problem, the algorithm knows all of the constraints at the outset; in the **Online Network Inference** problem, the constraints are given to the algorithm one by one, and edges must be added to G to satisfy each new constraint. By default, we allow the edges to have **arbitrary costs**, but in the **uniform cost** version of the problem the edge costs are all equal to 1.

When we restrict the **underlying graph** in a problem to a smaller class of graphs, we mean that all constraints S_i can be satisfied (for the online case, in hindsight), by a graph from that class.

6.1.3 Our Results

In Section 6.2 we analyze the offline problem, where we show that the Uniform Cost Network Inference problem (and therefore the arbitrary cost one) has a hardness of approximation lower bound of $\Omega(\log(n))$ times the optimal solution. We give an algorithm that gives an $O(\log(n) + \log(r))$ approximation, where r is the number of constraints. This matches the lower bound when r is polynomial in n.

In Section 6.3, we look at the Online Network Inference Problem. First, in Subsection 6.3.1, we look at the case when the underlying uniform cost graph is a star or path. In both cases, we show that the optimal algorithm has an $\Omega(\log(n))$-competitive ratio and give a matching $O(\log(n))$-competitive algorithm. We also show that in the case when edges have costs, there is no cn-competitive algorithm for any $c < 1$, even when the underlying graph is a path.

Then, in Subsection 6.3.2 we consider the general case of Online Network Inference, where the topology of the underlying graph is unrestricted. There we give an $O(n\log(n))$-competitive algorithm for the arbitrary cost case, that almost matches our lower bound. For the Uniform Cost Network Inference problem, we give an $\Omega(\sqrt{n})$-competitive lower bound, and in the case of an oblivious adversary, we give an $O(n^{2/3}\log^{2/3}(n))$-competitive algorithm.

6.2 Offline Network Inference

We first examine the Uniform Cost Network Inference problem in the offline case.

Theorem 6.2.1. *If P≠NP, the approximation ratio for the* ***Uniform Cost Network Inference*** *problem on n nodes is* $\Omega(\log n)$.

Proof. We reduce from the Hitting Set problem. The inputs to Hitting Set are $U = \{v_1, v_2, \ldots, v_n\}$ and $\{C_1, C_2, \ldots, C_j\}$ with $C_i \subseteq U$. The **Hitting Set** problem is to minimize $|H|$, where $H \subseteq U$ such that $\forall C_i$, $H \cap C_i \neq \emptyset$. Let k be a constant. We define an instance of the Uniform Cost Network Inference problem with n^k by n vertices $v_{(i,j)}$, for all $1 \leq i \leq n^k$ (rows) and $1 \leq j \leq n$ (columns). For each i, the vertices in row i, $\{v_{(i,1)}, v_{(i,2)}, \ldots, v_{(i,n)}\}$, correspond to the elements $\{v_1, \ldots, v_n\}$ in the Hitting Set instance.

Now we define the connectivity constraints for the Uniform Cost Network Inference problem. First we enforce that all pairs of vertices in each row i are connected, by adding a connectivity constraint for each pair of vertices $\{v_{(i,j)}, v_{(i,k)}\}$. For each constraint C_i in the Hitting Set problem, we create $\binom{n^k}{2}$ connectivity constraints. Without loss of generality, let $C_i = \{v_1, v_2, \ldots, v_k\}$. For each pair $l \neq j$ such that $1 \leq l, j \leq n^k$ we add a connectivity constraint

$$S_{C_i}^{l,j} = \{v_{(l,1)}, v_{(l,2)}, \ldots, v_{(l,k)}, v_{(j,1)}, v_{(j,2)}, \ldots, v_{(j,k)}\} \tag{6.1}$$

in the Uniform Cost Network Inference problem. This enforces the Hitting Set constraints pairwise between the n^k rows of the network inference problem.

Each pair of rows in our new instance contains the original Hitting Set instance. First, the algorithm has no choice but to place a clique on each row. Then, let equation (6.1) be a constraint. To satisfy $S_{C_i}^{l,j}$, the algorithm must choose some edge between row l and row j among vertices $1, \ldots, k$. We observe that if the algorithm chooses an edge between two vertices corresponding to different elements in the two rows, it could do at least as well by choosing the edge going between two copies of one of the two elements. To see this, if edge $(v_{(l,x)}, v_{(j,y)})$, with $x \neq y$, is chosen to satisfy the constraint $S_{C_i}^{l,j}$, edge $(v_{(l,x)}, v_{(j,x)})$ would have also satisfied the constraint (and corresponds to choosing element x in the Hitting Set). Then, for any other constraint between the two rows, $(v_{(l,x)}, v_{(j,y)})$ will satisfy it only if $(v_{(l,x)}, v_{(j,x)})$ will. Hence an optimal algorithm may choose edges in one-to-one correspondence with the elements in the original Hitting Set instance.

Because Hitting Set is the complement of Set Cover, if P≠NP, its optimal approximation ratio is $\Omega(\log(n))$ [42], and there are $\Theta\binom{n^k}{2}$ pairs of Hitting Set instances (or rows). The optimal solution has $\left(n^k\binom{n}{2} + \mathrm{OPT}\binom{n^k}{2}\right)$ edges – the first term counts the pairwise constraints in each row. So unless P=NP, the best polynomial time algorithm will require $\left(n^k\binom{n}{2} + \Omega\left(\log(n)\mathrm{OPT}\binom{n^k}{2}\right)\right)$ edges. Because k can be chosen to be arbitrarily large, this gives us the result. □

Below, we give an algorithm that almost meets this lower bound, even in the arbitrary cost case.

Theorem 6.2.2. *There is an $O(\log(n)+\log(r))$-competitive polynomial time approximation algorithm for the **Network Inference** problem on n nodes and r constraints.*

Proof. The inputs are the vertices $\{v_1, v_2, \ldots, v_n\}$, the cost c_e of each edge $e = (v_i, v_j)$, and the constraints $\{S_1, S_2, \ldots, S_r\}$. Let C be a potential function that sums over all constraints S_i, the number of components S_i induces in G minus 1. As long as $C > 0$, the constraints are not satisfied. We have $C_0 = \sum_i (|S_i| - 1)$ because the graph initially has no edges, and each constraint S_i induces $|S_i|$ components in G. Now, consider the following greedy algorithm: until all constraints are satisfied (while $C > 0$), take the edge e that has the lowest ratio of c_e to ΔC, the amount by which e reduces C. If in adding one edge, we have reduced C by k, then we say we have done k "reductions," and paid a "price" per reduction of $c_e/\Delta C$. After C_0 reductions, we are done, so we consider the reductions in the order that they occurred.

Now we consider how much we paid to do the kth reduction. Because $(k-1)$ reductions had been done already, C_{k-1} was at most $(C_0 - k + 1)$. The optimal algorithm did all the reductions in OPT, so the average price it paid for all the remaining reductions is at most $\mathrm{OPT}/(C_0-k+1)$. However, because we are choosing greedily, for the kth reduction, we pay a price no more than the optimal algorithm. So we look at the total price we paid (which is the cost of our algorithm), and it is $\sum_{k=1}^{C_0} \mathrm{OPT}/(C_0 - k + 1) = \mathrm{OPT}\log(C_0)$, which we can bound from above by $\mathrm{OPT}\log(nr) = \mathrm{OPT}(\log(n) + \log(r))$. □

6.3 Online Network Inference

In the online setting, the collection of connectivity constraints $S_1, S_2, \ldots, S_r$ is now given one at a time, and we say that upon being presented S_i the algorithm is on **round** i. Also, let E_i be the edge set after the algorithm satisfies constraint S_i. To explore the worst-case performance of our algorithms, unless otherwise stated, we assume an **adaptive adversary**, meaning that the adversary can wait for the algorithm to satisfy constraint S_i before determining constraint S_{i+1}.

In this section, we are interested in competitive analysis. An algorithm is c-**competitive** if the cost of its solution is less than c times OPT, where OPT is the best solution in hindsight. In the case when we know the underlying graph is, for instance, a uniform cost path or star, we know that $\mathrm{OPT} = (n-1)$.

First, we prove a lemma helpful for analyzing online algorithms.

Lemma 6.3.1. *Let $n(G,S)$ be the number of connected components $S \subseteq V$ induces in G, and let $G_i = (V, E_i)$. For every algorithm for the* ***Online Network Inference*** *problem, there is an algorithm that performs at least as well and adds exactly $(n(G_i, S_{i+1}) - 1)$ edges on every round i.*

Proof. Let A be any algorithm for the online network inference problem. We can make a new algorithm called A_{lazy}, that on each round inserts only a subset of edges that A has inserted up to that round, enough to keep the constraints satisfied. Each edge that A inserts but A_{lazy} does not, A_{lazy} remembers as possible edges for future rounds and adds them as needed to satisfy future constraints. It is clear that A_{lazy} needs to put down a spanning tree on the components induced by constraint i, which is $(n(G_i, S_{i+1}) - 1)$ edges; any fewer edges would not satisfy the constraint. Thereby, A_{lazy} satisfies the constraints, and because A_{lazy} uses a subset of the edges of A, it performs at least as well. $\square$

6.3.1 Stars and Paths

First, we examine the case when the underlying graph is a star. This is a natural framework for Network Inference Problems because it corresponds to the case when the constraints can be satisfied by a network with one server (the star's center) and the rest clients (the leaves.)

Theorem 6.3.2. *The optimal competitive ratio for the* ***Uniform Cost Network Inference*** *problem on n nodes when the algorithm knows the underlying graph is a* ***star*** *is* $\Theta(\log(n))$.

Proof. We first prove the *lower bound* – that the competitive ratio for any algorithm is $\Omega(\log(n))$. The adversary maintains a partition of the vertices into two sets: C, the possible centers, and $D = (V - C)$, the non-centers. Initially C has $(n-1)$ vertices and D has one vertex, and the initial two constraints given to the algorithm are V and C. At every step, the adversary looks for a vertex $v \in C$ that maximizes the number of edges (u, v) with $u \in D$ given by the algorithm, and moves v from C to D, that is, $C' = C \setminus \{v\}$ and $D' = D \cup \{v\}$. The new constraints given by the adversary are $C' \cup u$ for all $u \in D'$. Thus, the algorithm must ensure at least one edge from each element of D' to some element of C'. The adversary continues until it has moved all but one vertex from C to D.

To analyze, we consider the edges from elements of D to the element v moved from C to D when $|C| = i$. Each element of D must have at least one edge to an element of C, so the maximum number of edges from D to one element of C is at least the average: $(n-i)/i$. These edges are all distinct, so the algorithm must produce at least $\sum_{i=2}^{n-1}(n-i)/i = \Omega(n\log(n))$ edges in all. Yet, all these constraints can be satisfied by a star with $(n-1)$ edges. This completes the proof of the lower bound.

For the *upper bound*, we give an $O(\log(n))$-competitive algorithm. The algorithm will keep track of a set C_i of potential centers and $D_i = V - C_i$ known non-centers at round i. Any node not appearing in some constraint cannot be a center. The algorithm keeps nodes in C_i connected by a path, and each node in D_i is connected to some node in C_i, such that the number of edges going into each node in C_i from D_i is no more than $\lceil(|D_i|)/|C_i|\rceil$, meaning that all nodes in C_i have close to the same degree. Initially, $C_0 = V$ and is connected by an arbitrary path (costing $O(n)$ edges). At any stage of the algorithm, when a constraint S_i comes in, if it does not eliminate any potential centers, it is easy to see S_i is already satisfied. Otherwise, we remove any potential centers $R_i \subset C_{i-1}$ that are now known to be non-centers from C_{i-1} (to form C_i), and we add them to D_{i-1} (to form D_i). Further, we ignore all edges to nodes in R_i. We re-stitch the path connecting nodes in C_i, which takes at most $|R_i|+1$ edges. Then, we connect (in such a way that keeps the degrees of the nodes in C_i about equal) all nodes in R_i to nodes in C_i, which takes $|R_i|$ edges, and also all nodes in D_{i-1} that became disconnected from C_i because were connected to nodes in R_i, which takes $O\left(\frac{|R||D_{i-1}|}{|C_{i-1}|}\right)$ edges. This clearly satisfies constraint S_i.

To see why this gives us the needed result, we notice at at most n centers can be removed from C, and therefore connections involving nodes in R_i take $\sum_{i=1}^{n} O(|R_i|) = O(n)$. The rest of the connections, by the analysis in the paragraph above, cost $\sum_i O\left(\frac{|R||D_{i-1}|}{|C_{i-1}|}\right) \le \sum_i O\left(\frac{|R_i|n}{|C_{i-1}|}\right)$. If we consider removing one center at a time (as opposed to in groups R_i), we can bound this from above by $O(n\sum_{i=1}^{n}\frac{1}{n-i}) =$

$O(n \log(n))$. □

Next, we examine another natural network structure – when the underlying network is a path. This is the case when the constraints can be realized by a serial network

Theorem 6.3.3. *The optimal competitive ratio for the* ***Uniform Cost Network Inference*** *problem on n nodes when the algorithm knows the underlying graph is a* ***path*** *is* $\Theta(\log(n))$.

Proof. First we prove the *lower bound*, that any algorithm has a competitive ratio of $\Omega(\log(n))$. We show an adversarial strategy that forces the algorithm to use $O(n \log(n))$ edges when the optimal solution in hindsight uses only $(n-1)$ edges. The adversary first shows all the nodes, which by Lemma 6.3.1 the optimal algorithm connects using $(n-1)$ edges. Then the adversary divides the nodes into two independent sets and presents each of them to the algorithm in arbitrary order. The optimal algorithm must connect the two subgraphs with trees (again by Lemma 6.3.1), and the adversary repeats this process recursively. We say that each depth in the recursion is a new level in this process. Because the algorithm puts down $O(n)$ edges per level, given this strategy for the adversary, the optimal algorithm needs to put down a path at each step so as to balance the sizes of two following independent sets and limit the algorithm to $O(\log(n))$ levels. Hence, the algorithm uses $\Omega(n \log(n))$ edges, but it is clear that knowing the sets in advance, one can satisfy the connectivity requirements using $O(n)$ edges - by simply connecting the smallest sets and then merging them accordingly into a path. This gives us the desired $\Omega(\log(n))$ gap.

Now we prove the *upper bound* by giving an $O(\log((n))$-competitive algorithm. We first observe that every constraint S_i is a sub-interval of the path, and the algorithm must put down enough edges to capture a permutation of the vertices consistent with the S_i's. The algorithm we introduce maintains a **pq-tree** – a data structure, introduced by Booth and Lueker in [28], that keeps track of all consistent orderings of nodes given contiguous intervals in a permutation. A pq-tree is a tree that consists of leaf nodes, p-nodes, and q-nodes. A **leaf node** is an element (or vertex in our case), A **p-node** (permutation node) has 2 or more children of any type, and its children form a contiguous interval, but can be ordered in any order. A **q-node** has 3 or more children of any type and its children form an interval in the given order or its reverse. Each new interval constraint updates the pq-tree, and then the algorithm adds edges to satisfy the new constraint.

We will show that the algorithm can satisfy the constraints using $O(n \log(n))$ edges by using a potential function to keep track of the evolution of the pq-tree. Let P be the set of p-nodes in a given tree and Q be the set of q-nodes. Also for any node p, let $c(p)$ count p's children. For constants a and b, our potential function is

$$\Phi = a \sum_{p \in P} ((c(p) - 1)(\log(c(p) - 1)) + b|Q|. \tag{6.2}$$

We observe that the pq-tree before any constraints arrive has one p-node at the root, and all its children are leaf nodes. This corresponds to an arbitrary permutation of the vertices. So at the beginning, $\Phi = \Theta(n \log(n))$. In comparison, when the permutation is specified, the root is a q-node and the rest of the nodes are leaves. In that case, $\Phi = \Theta(1)$.

Now we look at what happens when a constraint comes in. We will argue that the number of edges we need to insert into our graph is a lower bound on the drop in the potential function, and because it is always the case that $\Phi \geq 0$, this will complete the proof.

We first analyze the most common type of update to a pq-tree. A constraint comes in and splits a known interval into two, that is, it splits a p-node with m children into two p-nodes (one with at most $(k + 1)$ children and the other with at most $(m-k)$ children), and attaches them to a q-node parent. So the drop in the potential function is as follows (where $H(p)$ is the binary entropy function.)

$$\begin{aligned}
-\Delta\Phi &= a\left((m-1)\log(m-1) - ((k)\log(k) + (m-k-1)\log(m-k-1))\right) - b \\
&= a(m-1)\left(\log(m-1) - \frac{k}{m-1}\log(k) - \frac{m-k-1}{m-1}\log(m-k-1)\right) - b \\
&= a(m-1)\left(-\frac{k}{m-1}\log\left(\frac{k}{m-1}\right) - \frac{m-k-1}{m-1}\log\left(\frac{m-k-1}{m-1}\right)\right) - b \\
&= a(m-1)H\left(\frac{k}{m-1}\right) - b \\
&\geq a(m-1)\min\left(\frac{k}{m-1}, \frac{m-k-1}{m-1}\right) - b \\
&= a\min(k, m-k-1) - b.
\end{aligned}$$

Now, $2\min(k, m-k-1)$ is exactly how much is required in the worst case to stitch up a split interval – because we have to connect up all of the nodes in the smaller new interval, and patch at most as many gaps in the larger interval (similar to the reasoning in the proof of the lower bound). It takes at most 4 more edges to connect up the ends of the two new intervals to the rest of the graph, and this can be paid for if $a = 10$ and $b = 4$. We remember $\min(k, m-k-1) \geq 1$, so we spend 2 on splitting the p-node, 4 on re-stitching, and 4 on the new q-node, and thus $a = 10$.

Booth and Lueker in [28] characterized all of the possible updates to the pq-tree using 10 patterns: L, P1, P2, P3, P4, P5, P6, Q1, Q2, and Q3. They are given in Section 6.5. Neither L, Q1, nor P1 changes the number of p-nodes or q-nodes. P2-P6 split at most one p-node and create at most one q-node, and are covered by our analysis above. Q2 and Q3 require us to reconnect at most 2 pairs of endpoints (with 4 edges), but also reduce the number of q-nodes by 1 or 2 (this is why $b = 4$), and the edges are paid for by the drop in the potential function. $\square$

In the arbitrary cost case, the competitive ratio becomes considerably worse.

Theorem 6.3.4. *There is no (cn)-competitive algorithm for $c < 1$ for the* ***Online Network Inference*** *problem on n vertices, even when the underlying graph is a* ***path****.*

Proof. We let all edges among $(n-1)$ of the vertices have cost 0, and all edges from the remaining vertex, s, have cost 1. The adversary first tells the algorithm that all the vertices are connected. When the algorithm satisfies this constraint, the adversary excludes from the next constraint all vertices the algorithm has chosen to directly connect to s. This continues until the adversary forces the algorithm to use all the 1 edges. But because each constraint is a subset of the previous constraint, the optimal solution only needs to contain the final cost 1 edge, and can connect the remaining vertices using a path that goes through the vertices in the order they were excluded in the adversary's choice of constraints. Hence, the algorithm was forced to pay a cost of $(n-1)$, while the optimal solution pays a cost of 1. $\square$

6.3.2 General Graphs

We introduce the **Online Fractional Network Inference** problem, in which the algorithm is similarly given a set of vertices V and edge costs c_e for all $e = (v, w) \in V$, and sees a sequence of constraints $\{S_1, S_2, \dots, S_r\}$. The task is to assign fractional weights w_e to the edges (or pairs of vertices), such that for each i, the maximum flow between each pair of vertices in S_i is at least 1, given the weights w_e (to be interpreted as edge capacities). The quality of the solution is measured by comparing $\sum c_e w_e$ with the optimal cost of satisfying all the connectivity constraints. In the online problem, the algorithm may not decrease any edge weights from round to round.

Lemma 6.3.5. *There is an $O(\log(n))$-competitive polynomial time algorithm for the* ***Online Fractional Network Inference*** *problem on n nodes.*

Proof. We give Algorithm 7 for the Online Fractional Network Inference problem.

Algorithm 7 An $O(\log(n))$-competitive Algorithm for the Online Fractional Network Inference Problem

```
Let $|V| = n$ and $|E| = m$
Upon seeing first constraint, set all $w_e = \frac{1}{m^2}$
**for** each constraint $S$ **do**
  **for** each pair $v, w \in S$ **do**
    **if** the flow from $v$ to $w$ in $S$ is at least 1 **then**
      do nothing
    **else**
      **while** the flow from $v$ to $w$ in $S$ is less than 1 **do**
        compute a min-weight cut $C$ between $v$ and $w$ in $S$.
        for each edge $e \in C$, $w_e = w_e(1 + 1/c_e)$
```

Algorithm 7 is a modification of the algorithm in **3.1** of Alon *et al.* [5], and this proof closely follows their logic.

We say that the optimal solution OPT has cost α. We assume the value of α is known, and we can then assume all edges have cost between 1 and m.[2] We now follow the argument in Alon *et al.* [5], which works for Algorithm 7 almost without modification. First we note that the algorithm generates a feasible solution. This is clear from its termination condition.

Now we will prove that the number of weight augmentation steps performed during the run of the algorithm is $O(\alpha \log(m))$. Consider the potential function

$$\Phi = \sum_{e \in E} c_e w_e^* \lg(w_e),$$

where w_e^* is the weight of edge e in OPT. It is clear from the initial edge weights that the potential function begins as $\Phi_0 = -O(\alpha \lg(m))$. Because no edge gets weight more than 2, the potential function never exceeds 2α. And the increase in

[2]Alon *et al.* [5] argue that we can use all edges of cost less than α/m and stay within our bound, and we can ignore all edges with cost greater than α, and then rescale. They also show how to guess α to within a factor of 2, justifying the assumption that α is known in advance.

the potential function with each weight augmentation step is at least 1:

$$\begin{aligned}\Delta\Phi &= \sum_{e\in E} c_e w_e^* \lg(w_e(1+1/c_e)) - \sum_{e\in E} c_e w_e^* \lg(w_e)\\ &= \sum_{e\in E} c_e w_e^* \lg(1+1/c_e)\\ &\geq \sum_{e\in E} w_e^*\\ &\geq 1.\end{aligned}$$

Finally, we look at the cost of our solution, $\sum_{e\in E} w_e c_e$, (which begins at ≤ 1) and notice that in a weight augmentation step, it does not exceed $\sum_{e\in E} \frac{w_e}{c_e} c_e \leq 1$. So, whenever Φ increases by at least 1, the cost of our solution increases by no more than 1. This gives us an $O(\log(m)) = O(\log(n))$ approximation to the Online Fractional Network Inference problem. □

We can now use Lemma 6.3.5 to develop an algorithm that almost matches the lower bound from Theorem 6.3.4.

Theorem 6.3.6. *There is an $O(n\log(n))$-competitive polynomial time algorithm for the* ***Online Network Inference*** *problem on n nodes.*

Proof. We take the algorithm for solving the Online Fractional Network Inference problem from Lemma 6.3.5, and use it together with a rounding scheme similar to the one considered by Buchbinder [32] for solving linear programs, to get our result.

For each edge e, we choose $2n$ random variables $X(e,i)$ independently and uniformly from $[0,1]$. For each edge, we let threshold $T(e) = \min_{i=1}^{2n} X(e,i)$. Then we run the algorithm for the Online Fractional Network Inference problem, and whenever $w_e \geq T(e)$, we add e to our integral solution, and continue. Now we claim the following.

1. The integral solution has expected cost $O(n)$ times the fractional solution.

2. The integral solution satisfies all the constraints with high probability.

To prove the first claim, for any edge e, the probability that $X(e, i) < w_e$ is w_e. The probability that e is chosen to be in the integral solution is the probability that some $X(i, s) < w_e$ – we call this event A_i. Hence, the probability of $\cup_{i=1}^{2n} A_i$ is $2nw_e$, and by linearity of expectation, on every round, the expected cost of our solution is $O(n)$ times the fractional solution, which is a $O(\log(n))$ approximation of OPT. Hence our solution is an $O(n \log(n))$ approximation of OPT in expectation.

To prove the second claim, we pick a constraint S. The constraint S is satisfied if and only if for every cut $C \in S$, there exists an edge crossing C in our solution. We fix a cut C. The probability the cut is not crossed is the probability we have not chosen any edge crossing the cut. This probability is $\prod_{e \in C} (1 - w_e)^{2n} \leq e^{\left(-2n \sum_{e \in C} w_e\right)}$. And because the cut is crossed with a flow of 1 in the fractional solution (i.e. $\sum_{e \in C} w_e \geq 1$) at the time it is considered by the algorithm, we can bound this by $\frac{1}{e^{2n}}$. There are r constraints and at most 2^n cuts per constraint, so by the union bound, the probability our solution is not feasible is $\left(\frac{r 2^n}{e^{2n}}\right)$. Because $r < 2^n < e^n$, the probability our solution is not feasible tends to 0 as n increases, completing the proof. □

Alternate Proof. We now give an alternate proof of the theorem by reducing the Online Network Inference problem to Online Set Cover. In Online Set Cover, $X = \{1, 2, \ldots, n\}$ is a set of n elements and S is a family of m weighted subsets of X. S is given to the algorithm in advance, and elements of $X' \subseteq X$ arrive one at a time in arbitrary order online. While X is known to the algorithm, X' is not. The goal of the algorithm is to select a collection of sets from S of lowest weight, such that at any point in the algorithm, every element that has arrived is contained in some selected subset. Once a subset is selected, it cannot be unselected.

In reducing the Online Network Inference problem to Online Set Cover, we make the weighted edges correspond to the weighted sets given to the algorithm. For each constraint to arrive online, we make a set cover element element for each possible cut through the constraint. A set covers an element if its corresponding edge crosses the cut corresponding to the element. In the Online Network Inference problem, each cut in a constraint must be crossed by some edge, and if each cut is crossed, the constraint is satisfied. The cost of the optimal solution to both problems is the same.

Hence, if the original Online Network Inference problem has n nodes, then the Online Set Cover instance has $O(n^2)$ sets and $O(2^n)$ possible elements (or partitions of vertices). Alon *et al.* [4] and Buchbinder [32] provide algorithms for Online Set Cover that give an $O(\log(m)\log(n))$-competitive ratio if m is the number of sets and n is the number of elements. For the Online Network Inference problem, this gives an $O(n\log(n))$-competitive bound, completing the proof. □

We now make a simple observation for the uniform cost case.

Proposition 6.3.7. *There is an $O(n)$-competitive polynomial time algorithm for the* ***Online Uniform Cost Network Inference*** *problem on n nodes.*

Proof. Consider the algorithm that puts down a clique for each constraint presented to it. Let $q \leq n$ be the number of nodes that appear in at least one constraint. Our algorithm uses $O(q^2)$ edges, but the optimal algorithm must clearly use at least $\Omega(q)$ edges. □

We also present a lower bound for the Online Uniform Cost Network Inference problem.

Theorem 6.3.8. *The* ***Online Uniform Cost Network Inference*** *problem on n nodes has an $\Omega(\sqrt{n})$-competitive lower bound.*

Proof. We divide the vertices into two sets Q and R, with $|Q| = \sqrt{n}$ and $|R| = n - \sqrt{n}$. For each $v_i \in R$, the adversary does the following. At stage $t = 1$ the adversary sets

$Q_{(i,1)} = Q$. At stage t, the adversary gives the learner the constraint $S_{(i,t)} = Q_{(i,t)} \cup v_i$. Let $C_{(i,t)}$ be the set of vertices in Q which the learner connects to v_i in response to being presented $S_{(i,t)}$. The adversary sets $Q_{(i,t+1)} = Q_{(i,t)} \setminus C_{(i,t)}$ and continues to the next stage. The adversary stops when $Q_{(i,t)} = \emptyset$.

To analyze this strategy for the adversary, for each v_i, we order the edges from v_i to R by the stage in which the learner has placed them, breaking ties arbitrarily. It is clear that the last edge the learner places is sufficient to connect v_i to R for all constraints $S_{(i,t)}$. Hence, all of these constraints can be satisfied in retrospect by placing a clique on Q using $\binom{\sqrt{n}}{2} = O(n)$ edges and one edge per vertex in R, also using $O(n)$ edges. However, the learner places $\Omega(n)$ edges per vertex in Q, amounting to $\Omega\left(n\sqrt{n}\right)$ edges in total, giving the desired result. □

We now consider the Online Network Inference problem with an **oblivious adversary** – an adversary who commits to the constraints $\{S_1, S_2, \ldots, S_r\}$ before presenting any of them to the algorithm.

Theorem 6.3.9. *There is a randomized polynomial time algorithm for the* ***Online Uniform Cost Network Inference*** *problem on n nodes that gives an expected $O(n^{2/3}\log^{2/3}(n))$-competitive ratio against an oblivious adversary.*

Proof. We assume that the optimal solution has $m = \Omega(n)$ edges (that each vertex appears in some constraint). We then create an Erdös Rényi random graph on our graph G, by putting in edges independently with a specified probability. Random graph connectivity has a sharp threshold of $\frac{c\log(n)}{n}$ for $c > 1$ [41]. When $p = \frac{c\log^{2/3}(n)}{n^{1/3}}$, G has $O(n^{5/3}\log^{2/3}(n))$ edges in expectation. Now, our algorithm is simple – for each constraint S_i such that $|S_i| \geq n^{1/3}\log^{1/3}(n)$, because of our choice of p, S_i is already connected with high probability in G. Because we assume that there are only polynomially many constraints (even in the offline case, as in Theorem 6.2.2), for large enough c, all such constraints are satisfied in expectation. For every constraint

S_i of size $< n^{1/3}\log^{1/3}(n)$ that we see, we can put a clique with $O(n^{2/3}\log^{2/3}(n))$ edges on that constraint, and each time we do that, we are guaranteed to hit at least one edge in OPT. Hence, this costs us $O(n^{5/3}\log^{2/3}(n) + n^{2/3}\log^{2/3}(n)\text{OPT})$ edges in expectation, and because $m = \Omega(n)$, we have an $O(n^{2/3}\log^{2/3}(n))$ approximation ratio. $\square$

6.4 Discussion and Open Problems

In this paper we present a theoretical study of the Network Inference problem. This model allows us to estimate connections among people (or other populations) from data that exposes certain constraints. While in practical settings, we rarely have good prior estimates of the probability of connections among people, these algorithms should give some idea of the structure of actual social networks. One challenge in using this model to learn real-world networks is that often times, due to issues of privacy, data is anonymized (for example disease data), and it is hard to tell when the same person participates in multiple constraints. However, there are other settings where this would not be an issue. For network construction problems, our algorithms give network designers methods of optimizing costs while satisfying their users' constraints, especially in the offline setting.

We leave open some interesting questions. In the offline case, we give an $\Omega(\log(n))$ hardness of approximation lower bound and an $O(\log(n) + \log(r))$ approximation algorithm for both the arbitrary cost and uniform cost Network Inference problems. If r is polynomial in n these bounds match, but otherwise there is a gap. We also have a $\log(n)$ asymptotic gap for the Online Network Inference problem. For the Online Uniform Cost Network Inference problem, we have an $\Omega(\sqrt{n})$ adversarial lower bound and an $O(n^{2/3}/log^{1/3}(n))$ algorithm for the oblivious case. Improving these bounds is an important problem.

Another open problem is to find tight bounds for trees in the uniform cost case. For stars and paths, the bounds are tight, and our arguments can be adapted to give a $\Omega(\log(n))$-competitive lower bounds against an oblivious adversary. Perhaps an $O(\log(n))$-competitive algorithm can be found for trees in general, but our algorithms for paths and stars rely on their specific properties and do not immediately generalize. Finally, one can consider generalizations of the Network Inference Problem, for example constraints could require the vertices to be k-connected in the induced subgraphs.

6.5 Appendix: Updating a pq-tree

We briefly describe the patterns in [28] for updating pq-trees, as broken down into 10 cases. This can be used as a guide for tracking the changes in Equation 6.2.

L This pattern simply relabels some leaf nodes.

P1 This pattern simply relabels a p-node.

P2 This pattern moves some children of a p-node into their own p-node.

P3 This pattern moves some children of a p-node into their own p-node and creates a parent q-node.

P4 This pattern moves some children of a p-node to be children of a newly created p-node, whose parent is a q-node that is a child of the original p-node.

P5 This pattern moves some children of a p-node into their own p-node that is the child of the original p-node, which becomes transformed to a q-node.

P6 This pattern moves some children of a p-node to their own p-node that is moved to be the child of a newly created q-node formed by merging two q-nodes.

Q1 This pattern simply relabels a q-node.

Q2 This pattern deletes a q-node and moves its children to become children of its parent q-node.

Q3 This pattern deletes two q-nodes and merges their children to become children of their parent q-node.

Chapter 7

Learning Automata from Labels

7.1 Introduction

The problem of learning the behavior of a finite automaton has been considered in several domains, including language learning and environment learning by robots. Many interesting questions remain about the kinds of information that permit efficient learning of finite automata.

One basic result is that finite automata are not learnable using a polynomial number of membership queries. Consider a "password machine", that is, an acceptor with $(n + 2)$ states that accepts exactly one binary string of length n; the learner may query $(2^n - 1)$ strings before finding the one that is accepted. In this case, the learner gets no partial information from the unsuccessful queries.

However, Freund *et al.* [44] show that regardless of the topology of the underlying automaton, if its states are randomly labeled with 0 or 1, then a robot taking a random walk on the automaton can learn to predict the labels while making only a polynomial number of errors of prediction. Random labels on the states provide a rich source of information that can be used to distinguish otherwise difficult-to-distinguish pairs of states.

In a different setting, Becerra-Bonache *et al.* [25] introduced **correction queries** to model a kind of correction provided by a teacher to a learner when the learner's utterance is not grammatically correct. In their model, a correction query with a string w gives the learner not only membership information about w, but also, if w is not accepted, either the minimum continuation of w that is accepted, or the information that no continuation of w is accepted. In certain cases, corrections may provide a substantial amount of partial information for the learner. For example, for a password machine, a prefix of the password will be answered with the rest of the password. We may think of correction queries as labeling each state q of the automaton with the string r_q that is the response to any correction query w that arrives at q.

In both of these cases, labels on states may facilitate the learning of finite automata: randomly chosen labels in the work of Freund *et al.* and meaningfully chosen labels in the work of Becerra-Bonache *et al.* In this chapter we explore the general idea of adding labels to the states of an automaton to make it easier to learn. That is, we allow a teacher to prepare an automaton M for learning by adding labels to its states (either carefully or randomly chosen). When the learner queries a string, the learner receives not only the original output of M for that string, but also the label attached to that state by the teacher. In an extension of this idea, we also allow the teacher to "unfold" the machine M to produce copies of a state that may then be given different labels. These ideas are also relevant to automata testing [69] – labeling and unfolding automata can make their structure easier to verify.

Depending on how labels are assigned, learning may or may not become easier. If each state is assigned a unique label, the learning task becomes easy because the learner knows which state the machine reaches on any given query. However, if the labels are all the same, they give no additional information and learning may require an exponential number of queries (as in the case of membership queries.)

Hence we focus on questions of the following sort. Given an automaton, how can a teacher use a limited set of labels to make the learning problem easier? If random labels are sprinkled on the states of an automaton, how much does that help the learner? How few labels can we use and still make the learning problem tractable? Other questions concern the structure of the automaton itself. For example, we may consider changing the structure of the automaton before labeling it. We also consider the problem of learning randomly labeled automata with random structure.

7.2 Preliminaries

We consider finite automata with output, defined as follows. A finite automaton M has a finite set Q of states, an initial state $q_0 \in Q$, a finite alphabet X of input symbols, a finite alphabet Y of output symbols, an output function γ mapping Q to Y and a transition function τ mapping $Q \times X$ to Q. We extend τ to map $Q \times X^*$ to Q in the usual way. A finite acceptor is a finite automaton with output alphabet $Y = \{0, 1\}$; if $\gamma(q) = 1$ then q is an accepting state, otherwise, q is a rejecting state. In this chapter we assume that there are at least two input symbols and at least two output symbols, that is, $|X| \geq 2$ and $|Y| \geq 2$.

For any string $w \in X^*$, we define $M(w)$ to be $\gamma(\tau(q_0, w))$, that is, the output of the state reached from q_0 on input w. Two finite automata M_1 and M_2 are **output-equivalent** if they have the same input alphabet X and the same output alphabet Y and for every string $w \in X^*$, $M_1(w) = M_2(w)$.

If M is a finite automaton with output, then an **output query** with string $w \in X^*$ returns the symbol $M(w)$. This generalizes the concept of a membership query for an acceptor. That is, if M is an acceptor, an output query with w returns 1 if w is accepted by M and 0 if w is rejected by M. We note that Angluin's polynomial time algorithm to learn finite acceptors using membership queries and equivalence queries generalizes in a straightforward way to learn finite automata with output using output queries and equivalence queries [10].

If q_1 and q_2 are states of a finite automaton with output, then q_1 and q_2 are **distinguishable** if there exists a **distinguishing string** for them, namely, a string w such that $\gamma(\tau(q_1, w)) \neq \gamma(\tau(q_2, w))$, that is, w leads from q_1 and q_2 to two states with different output symbols. If M is minimized, every pair of its states are distinguishable, and M has at most one sink state.

If d is a nonnegative integer, the ***d*-signature tree** of a state q is the finite function mapping each input string z of length at most d to the output symbol $\gamma(\tau(q, z))$. We picture the d-signature tree of a state as a rooted tree of depth d in which each internal node has $|X|$ children labeled with the elements of X, and each node is labeled with the symbol from Y that is the output of the state reached from q on the input string z that leads from the root to this node. The d-signature tree of a state gives the output behavior in a local neighborhood of the automaton reachable from that state.

For any finite automaton M with output, we may consider its **transition graph**, which is a finite directed graph (possibly with multiple edges and self-loops) defined as follows. The vertices are the states of M and there is an edge from q to q' for each transition $\tau(q, a) = q'$. Properties of the transition graph are applied to M; that is, M is **strongly connected** if its transition graph is strongly connected. Similarly, the **out-degree** of M is $|X|$ for every node, and the **in-degree** of M is the maximum number of edges entering any node of its transition graph. For a positive integer k, we define an automaton M to be ***k*-concentrating** if there is some set Q' of at most k states of M such that every state of M can reach at least one state in Q'. Every strongly connected automaton is 1-concentrating.

7.2.1 Labelings

If M is a finite automaton with output, then a **labeling** of M is a function ℓ mapping Q to a set L of labels, the **label alphabet**. We use M to construct a new automaton M^ℓ by changing the output function to $\gamma'(q) = (\gamma(q), \ell(q))$. That is, the new output for a state is a pair incorporating the output symbol for the state and the label attached to the state. For the scenario of **learning with labels**, we assume that the learner has access to output queries for M^ℓ for some labeling ℓ of the hidden automaton M. For the scenario of **learning with unfolding and labels**, we assume that the learner has access to output queries for M_1^ℓ for some labeling ℓ of some automaton M_1 that is output-equivalent to M. In these two scenarios, the queries will be referred to as **label queries**. The goal of the learner in either scenario is to use label queries to find a finite automaton M' output-equivalent to M. Thus, the learner must discover the output behavior of the hidden automaton, but not necessarily its topology or labeling. We assume the learner is given both X and $|Q|$.

7.3 Learning with Labels

First, we show a lower bound on the number of label queries required to learn a hidden automaton M with n states and an arbitrary labeling ℓ.

Proposition 7.3.1. *Let L be a finite label alphabet. Learning a hidden automaton with n states and a labeling ℓ using symbols from L requires*

$$\Omega\left(\frac{|X| n \log(n)}{1 + \log(|L|))}\right)$$

label queries in the worst case.

Proof. Recall that we have assumed that $|X|$ and $|Y|$ are both at least 2; we consider

$|Y| = 2$. Domaratzki *et al.* [39] have shown that there are at least

$$(|X| - o(1))n2^{n-1}n^{(|X|-1)n}$$

distinct languages accepted by acceptors with n states. Because each label query returns one of at most $2 \cdot |L|$ values, an information theoretic argument gives the claimed lower bound on the number of label queries. As a corollary, when $|X|$ and $|L|$ are constants, we have a lower bound of $\Omega(n \log(n))$ label queries. □

7.3.1 Labels Carefully Chosen

In this section, we examine the case where a limit is placed on the number of different labels the teacher may use, and the teacher is able to label the states after examining the automaton. Moreover, the learning algorithm may take advantage of knowing the labeling strategy of the teacher. In this setting the problem takes on an aspect of coding, and indicates the maximum extent to which labeling may facilitate efficient learning. We begin with a simple proposition.

Proposition 7.3.2. *An automaton with n states, helpfully labeled using n different labels, can be learned using $|X|n$ label queries.*

Proof. The teacher assigns a unique integer label between 1 and n to each state. The learner asks a label query with the empty string to determine the output and label of the start state, and then explores the transitions from the start state by querying each $a \in X$. After querying an input string w, the label indicates whether this state has been visited before. If the state is new, the learner explores all the transitions from it by querying wa for each $a \in X$. Thus, after querying at most $|X|n$ strings, the learner knows the structure and outputs of the entire automaton. The lower bound shows that this is asymptotically optimal if the label set L has n

elements. □

We next consider limiting the teacher to a constant number of different labels: a polynomial number of label queries suffices in this case.

Theorem 7.3.3. *For each automaton with n states, there is a helpful labeling using $2^{|X|}$ different labels such that the automaton can be learned using $O(|X|n^2)$ label queries.*

Proof. Given an automaton M of n states, the teacher chooses an outward-directed spanning tree T rooted at q_0 of the transition graph of the automaton, and labels the states of M to communicate T to the learner as follows. The label of state q is the subset of X corresponding to the edges of T from q to other nodes. The label of q directs the learner to q's children. Using at most n label queries and the structure of T, the learner can create a set S of n input strings such that for each state q of M, there is one string $w \in S$ such that $\tau(q_0, w) = q$.

In [8], Angluin gives an algorithm for learning a regular language using membership queries given a live complete sample for the language. A live complete sample for a language L is a set of strings P, that for every state q (other than the dead state) of the minimal acceptor for L, contains a string that leads from the start state to q. Given a live complete sample P, a learner can find the regular language using $O(k|P|n)$ membership queries, where k is the size of the input alphabet. A straightforward generalization of this algorithm to automata with output shows that the set S and $O(|X|n^2)$ output queries can be used to find an automaton output equivalent to M. □

However, the number of queries, $O(n^2)$, does not meet the $\Omega(n \log n)$ lower bound, and the number of different labels is large. For a restricted class of automata, there is a helpful labeling with fewer labels that permits learning with an asymptotically optimal $O(n \log n)$ label queries. To appreciate the generality of Theorem 7.3.4, we note once more that every strongly connected automaton is 1-concentrating, and as we will see in Lemma 7.4.2, automata with a small input alphabet can be unfolded to have small in-degree.

Theorem 7.3.4. *Let k and c be positive integers. Any automaton in the class of c-concentrating automata with in-degree at most k can be helpfully labeled with at most $(3k|X| + c)$ labels so that it can be learned using $O(|X|n \log(n))$ label queries.*

Proof. We give the construction for 1-concentrating automata and indicate how to generalize it at the end of the proof. Given a 1-concentrating automaton M the teacher chooses as the *root* a node reachable from all other nodes in the transition graph of M. The depth of a node is the length of the shortest path from that node to the root. The teacher then chooses a spanning tree T directed inward to the root by choosing a parent for each non-root node. (One way to do this is to let the parent of a node q be the first node reached along a shortest path from q to the root.) The teacher assigns, as part of the label for each node q, an element $a \in X$ such that $\tau(q, a)$ is the parent of q.

The teacher now adds more information to the labels of the nodes, which we call color, using the colors yellow, red, green, and blue. The root is the unique node colored yellow. Let $t = \lceil \log n \rceil$; t bits are enough to give a unique identifier for every node of the graph. Each node at depth a multiple of $(t + 1)$ is colored red. For each red node v we choose a unique identifier of t bits $(c_1, c_2, \ldots, c_t)$ encoded as green and blue labels. Now consider the maximal subtree rooted at v containing no red nodes. For each level i from 1 to the depth of the subtree, all the nodes at level i of the subtree are colored with c_i (which is either blue or green.) The teacher has (so far) used $3|X| + 1$ labels – a direction and one of three colors per non-root node, and a unique identifier for the root.

Given this labeling, the learner can start from any state and reach a localization state whose identifier is known, as follows. The learner uses the parent component of the labels to go up the tree until it passes one red node and arrives at a second red node, or arrives at the root (whichever comes first), keeping track of the labels seen. If the learner reaches the root, it knows where it is. Otherwise, the learner interprets the labels seen between the first and second red node encountered as an identifier for the node v reached. This involves observing at most $(2t+2)$ labels. Thus, even if the in-degree is not bounded, a 1-concentrating automaton can be labeled so that with $O(\log(n))$ label queries the learner can reach a uniquely identified localizing state.

If each node of the tree T also has in-degree bounded by k, another component of the label for each non-root node identifies which of the k possible predecessors of its parent it is (numbered arbitrarily from 1 to at most k.) If the learner collects these values on the path from u to its localization node v, then we have an identifier for u with respect to v. Thus it takes $O(\log(n))$ label queries to learn any node's identifier. If the node has not been encountered before, its $|X|$ transitions must be explored, as in Proposition 7.3.2. This gives us a learning algorithm using $O(|X|n\log(n))$ label queries. The labeling uses at most $3k|X|+1$ different labels.

If the automaton is c-concentrating for some $c > 1$, then the teacher selects a set of at most c nodes such that every node can reach at least one of them and constructs a forest of at most c inward directed disjoint spanning trees, and proceeds as above. This increases the number of unique identifiers for the roots from 1 to c. □

An open question is whether an arbitrary finite automaton with n states can be helpfully labeled with $O(1)$ labels in such a way that it can be learned using $O(|X|n\log n)$ label queries.

7.3.2 Labels Randomly Chosen

In this section we turn from labels carefully chosen by the teacher to an independent uniform random choice of labels for states from a label alphabet L. With nonzero probability the labeling may be completely uninformative, so results in this scenario incorporate a confidence parameter $\delta > 0$ that is an input to the learner. The goal of the learner is to learn an automaton that is output equivalent to the hidden automaton M with probability at least $(1 - \delta)$, where this probability is taken over the labelings of M. Results on random labelings can be used in the careful labeling scenario: the teacher generates a number of random labelings until one is found that has the desired properties.

We first review the learning scenario considered by Freund *et al.* [44]. There is a finite automaton over the input alphabet $X = \{0, 1\}$ and output alphabet $\{+, -\}$, where the transition function and start state of the automaton are arbitrary, but the output symbol for each state is chosen independently and uniformly from $\{+, -\}$. The learner moves from state to state in the target automaton according to a random walk (the next input symbol is chosen independently and uniformly from $\{0, 1\}$) and, after learning what the next input symbol will be, attempts to predict the output ($+$ or $-$) of the next state. After the prediction, the learner is told the correct output and the process repeats with the next input symbol in the random walk. If the learner's prediction was incorrect, this counts as a **prediction mistake**. In the first scenario they consider, the learner may reset the machine to the initial state by predicting "?" instead of "$+$" or "$-$"; this counts as a **default mistake**. In this model, the learner is completely passive, dependent upon the random walk process to disclose useful information about the behavior of the underlying automaton. For this setting they prove the following.

Theorem 7.3.5 (Freund *et al.* [44])**.** *There exists a learning algorithm that takes n*

and δ as input, runs in time polynomial in n and $1/\delta$ and with probability at least $(1-\delta)$ makes no prediction mistakes and an expected $O((n^5/\delta^2)\log(n/\delta))$ default mistakes.

The main idea is to use the d-signature tree of a state as the identifier for the state, where $d \geq 2\log(n^2/\delta)$. For this setting, there are at least n^4/δ^2 strings in a signature tree of depth d. The following theorem of Trakhtenbrot and Barzdin' [84] establishes that signature trees of this depth are sufficient.

Theorem 7.3.6 (Trakhtenbrot and Barzdin' [84])**.** *For any natural number d and for any finite automaton with n states and randomly chosen outputs from Y, the probability that for some pair of distinguishable states the shortest distinguishing string is of length greater than d is less than*

$$n^2(1/|Y|)^{d/2}.$$

We may apply these ideas to prove the following.

Theorem 7.3.7. *For any positive integer s, any finite automaton with n states, over the input alphabet X and output alphabet Y, with its states randomly labeled with labels from a label alphabet L with $|L| = |X|^s$ can be learned using*

$$O\left(|X|\frac{n^{1+4/s}}{\delta^{2/s}}\right)$$

label queries, with probability at least $(1-\delta)$ (with respect to the choice of labeling.)

Proof. Assume that the learning algorithm is given n, a bound on the number of states of the hidden automaton, and the confidence parameter $\delta > 0$. It calculates a bound $d = d(n,\delta)$ (described below) and proceeds as follows, starting with the empty input string. To explore the input string w, the learning algorithm calculates the d signature tree (in the labeled automaton) of the state reached by w by making label queries on wz for all input strings z of length at most d. This requires $O(|X|^d)$ queries. If this signature tree has not been encountered before, then the algorithm explores the transitions wa for all $a \in X$. Assuming that the labeling is "good", that is, that all distinguishable pairs of states have a distinguishing string in the labeled automaton of length at most d, then this correctly learns the output behavior of the hidden automaton using $O(|X|^{d+1}n)$ label queries.

To apply Theorem 7.3.6, we assume that the hidden automaton M is an arbitrary finite automaton with output with at most n states, input alphabet X and output alphabet Y. The labels randomly chosen from L then play the role of the random outputs in Theorem 7.3.6. There is a somewhat subtle issue: states distinguishable in M by their outputs may not be distinguishable in the labeled automaton by their labels alone. Fortunately, Freund *et al.* [44] have shown us how to address this point. In the first case, if two states of M are distinguishable by their outputs in M by a string of length at most d, then their d signature trees (in the labeled automaton) will differ. Otherwise, if the shortest distinguishing string for the two states (using just outputs) is of length at least $d+1$, then generalizing the argument for Theorem 2 in [44] from $|Y| = 2$ to arbitrary $|Y|$, the probability that this pair of states is not distinguished by the random labeling by a string of length at most d is bounded above by $(1/|Y|)^{(d+1)/2}$. Summing over all pairs of states gives the required bound.

Thus, choosing

$$d \geq \frac{2}{\log |L|} \log\left(\frac{n^2}{\delta}\right),$$

suffices to ensure that the labeling is "good" with probability at least $(1 - \delta)$. If we use more labels, the signature trees need not be so deep and the algorithm does not need to make as many queries to determine them. In particular, if $|L| = |X|^s$, then the bound of $O(|X|^{d+1}n)$ on the number of label queries used by the algorithm becomes

$$O\left(|X|\frac{n^{1+4/s}}{\delta^{2/s}}\right),$$

completing the proof. □

Corollary 7.3.8. *Any finite automaton with n states can be learned using $O(|X|n^{1+\epsilon})$ label queries with probability at least $1/2$, when it is randomly labeled with $|L| = f(|X|, \epsilon)$ labels.*

Proof. With $\delta = 1/2$ a choice of $|L| \geq |X|^{4/\epsilon}$ suffices. □

We remark that this implies that there exists a careful labeling with $O(|X|^4)$ labels that achieves learnability with $O(|X|n^2)$ label queries, substantially improving on the size of the label set used in Theorem 7.3.3. An open question is whether a random labeling with $O(1)$ labels enables efficient learning of an arbitrary n state automaton with $O(n \log n)$ queries with high probability.

7.4 Unfolding Finite Automata

We now consider giving more power to the teacher. Because many automata have the same output behavior, we ask what happens if a teacher can change the underlying machine (without changing its output behavior) before placing labels on it. In Sections 7.3.1 and 7.3.2, the teacher had to label a fixed machine. Now we will examine what happens when a teacher can unfold an automaton before putting labels on it. That is, given M, the teacher chooses another automaton M' with the same output behavior as M and labels the states of M' for the learner.

7.4.1 Unfolding and then Labeling

We first remark that unfolding an automaton M from n to $O(n \log n)$ states allows a careful labeling with just 2 labels to encode a description of the machine.

Proposition 7.4.1. *Any finite automaton with n states can be unfolded to have $N = O(|X|n\log(n) + n\log(|Y|))$ states and carefully labeled with 2 labels, in such a way that it can be learned using N label queries.*

Proof. The total number of automata with output having n states, input alphabet X and output alphabet Y is at most

$$n^{|X|n+1}|Y|^n.$$

Thus, $N = O(|X|n\log(n) + n\log(|Y|))$ bits suffice to specify any one of these machines.

The teacher chooses $a \in X$ and unfolds the target automaton M as follows. The strings a^i for $i = 0, 1, \ldots, N-1$ each send the learner to a newly created state, which act (with respect to transitions on other input symbols and output behavior) just like their counterparts in the original machine. The remaining states are unchanged. The unfolded automaton is output equivalent to M. The teacher then specifies M by labeling these N new states with the bits of the specification of M. The learner simply asks a sequence of N queries on strings of the form a^i to receive the encoding of the hidden machine. □

This method does not work if we restrict the unfolding to $O(|X|n)$ states, but we show that this much unfolding is sufficient to reduce the in-degree of the automaton to $O(|X|)$.

Lemma 7.4.2. *Let M be an arbitrary automaton of n states. There is an automaton*

M' with the same output behavior as M, with at most $(|X|+1)n$ states whose in-degree is bounded by $2|X|+1$.

Proof. Given M, we repeat the following process until it terminates. While there is some state q with in-degree greater than $2|X|+1$, split q into two copies, dividing the incoming edges as evenly as possible between the two copies, and duplicating all $|X|$ outgoing edges for the second copy of q.

It is clear that each step of this process preserves the output behavior of M. To see that it terminates, for each node q let $f(q)$ be the maximum of 0 and $d_{in}(q)-(|X|+1)$, where $d_{in}(q)$ is the in-degree of q. Consider the potential function Φ that is the sum of $f(q)$ for all nodes q in the transition graph. Φ is initially at most $|X|n-(|X|+1)$, and each step reduces it by at least $1=(|X|+1)-|X|$. Thus, the process terminates after no more than $|X|n$ steps producing an output-equivalent automaton M' with no more than $(|X|+1)n$ states and in-degree at most $2|X|+1$. □

In particular, an automaton with a sink state of high in-degree will be unfolded by this process to have multiple copies of the sink state. Using this idea for degree reduction, the teacher may use linear unfolding and helpful labeling to enable a strongly connected automaton to be learned with $O(n \log n)$ label queries.

Corollary 7.4.3. *For any strongly connected automaton M of n states, there is an unfolding M' of M with at most $(|X|+1)n$ states and a careful labeling of M' using $O(|X|^2)$ labels that allows the behavior of M to be learned using $O(|X|^2 n \log n)$ label queries.*

Proof. Given a strongly connected automaton M with n states, the teacher uses the method of Lemma 7.4.2 to produce an output equivalent machine M' with at most $(|X|+1)n$ states and in-degree bounded by $2|X|+1$. This unfolding may not preserve the property of being strongly connected, but there is at least one state q

that has at most $(|X|+1)$ copies in the unfolded machine M'. Because M is strongly connected, every state of M' must be able to reach at least one of the copies of q, so M' is $(|X|+1)$-concentrating. Applying the method of Theorem 7.3.4, the teacher can use $3(2|X|+1)|X|+(|X|+1)$ labels to label M' so that it can be learned with $O(|X|^2 n \log n)$ label queries. □

We now consider uniform random labelings of the states when the teacher is allowed to choose the unfolding of the machine.

Theorem 7.4.4. *Any automaton with n states can be unfolded to have $O(n \log(n/\delta))$ states and randomly labeled with 2 labels, such that with probability at least $(1-\delta)$, it can be learned using $O(|X|n(\log(n/\delta))^2)$ queries.*

Proof. Given n and δ, let $t = \lceil \log(n^2/\delta) \rceil$. The teacher chooses $a \in X$ and unfolds the target machine M to construct the machine M' as follows. M' has nt states (q, i) where q is a state of M and $0 \leq i \leq (t-1)$. The start state is $(q_0, 0)$, where q_0 is the start state of M. The output symbol for (q, i) is $\gamma(q, a^i)$, where γ is the output function of M. For $0 < i < (t-1)$, the a transition from (q, i) is to $(q, (i+1))$. The a transition from $(q, t-1)$ is to $(q', 0)$, where $q' = \tau(q, a^t)$ and τ is the transition function of M. For all other input symbols b with $b \neq a$, the b transition from (q, i) is to $(q', 0)$, where $q' = \tau(q, a^i b)$.

To see that M' is an unfolding of M, that is, M' is output equivalent to M, we show that each state (q, i) of M' is output equivalent to state $\tau(q, a^i)$ of M. By construction, these two states have the same output. If $i < (t-1)$ then the a transition from (q, i) is to $(q, i+1)$, which has the same output symbol as $\tau(q, a^{i+1})$. The a transition from $(q, t-1)$ is to $(q', 0)$, where $q' = \tau(q, a^t)$, which has the same output symbol as $\tau(\tau(q, a^{t-1}), a)$. If $b \neq a$ is an input symbol, then the b transition from (q, i) is to $(q', 0)$ where $q' = \tau(q, a^i b)$, which has the same output symbol as $\tau(\tau(q, a^i), b)$.

Suppose M' is randomly labeled with two labels. For each state q of M, define its label identifier in M' to be the sequence of labels of (q, i) for $i = 0, 1, \ldots, (t-1)$. For two distinct states q_1 and q_2 of M, the probability that their label identifiers in M' are equal is $(1/2)^t$, which is at most δ/n^2. Thus, the probability that there exist two distinct states q_1 and q_2 with the same label identifier in M' is at most δ.

Given n and δ, the learning algorithm takes advantage of the known unfolding strategy to construct states (j, i) for $0 \leq j \leq n-1$ and $0 \leq i \leq (t-1)$ with a transitions from (j, i) to $(j, i+1)$ for $i < (t-1)$. It starts with the empty input string and uses the following exploration strategy. Given an input string w that is known to arrive at some $(q, 0)$ in M', the learning algorithm makes label queries on wa^i for $i = 0, 1, \ldots, (t-1)$ to determine the label identifier of q in M'. If this label identifier has not been seen before, the learner uses the next unused $(j, 0)$ to represent q and records the outputs and labels for the states (j, i) for $i = 0, 1, \ldots, (t-1)$. It must also explore all unknown transitions from the states (j, i). If distinct states of M receive distinct label identifiers in M', the learner learns a finite automaton output equivalent to M using $O(|X|nt^2)$ label queries. □

7.5 Automata with Random Structure

We may also ask whether randomly labeled finite automata are hard to learn "on average". We consider automata with randomly chosen transition functions and random labels. The model of random structure that we consider is as follows. Let the states be q_i for $i = 0, 1, \ldots, (n-1)$, where q_0 is the start state. For each state q_i and input symbol $a \in X$, choose j uniformly at random from $0, 1, \ldots, (n-1)$ and let $\tau(q_i, a) = q_j$.

Theorem 7.5.1. *A finite automaton with n states, a random transition function and a random labeling can be learned using $O(n \log(n))$ label queries, with high probability.*

The probability is over the choice of transition function and labeling.

Proof. This was first proved by Korshunov in [68]; here we give a simpler proof. Korshunov showed that the signature trees only need to be of depth asymptotically equal to $\log_{|X|}(\log_{|L|}(n))$ for the nodes to have unique signatures with high probability. We use a method similar to signature trees, but simpler to analyze. Instead of comparing signature trees for two states to tell whether or not they are distinct, we compare the labels along at most four sets of transitions, which we call **signature paths** – like a signature tree consisting only of four paths.

Lemmas 7.5.2 and 7.5.3 show that given X and n there are at most four signature paths, each of length $3\log(n)$, such that for a random finite automaton of n states with input alphabet X and for any pair s_1 and s_2 of different states, the probability is $O\left(\frac{\log^6(n)}{n^3}\right)$ that s_1 and s_2 are distinguishable but not distinguished by any of the strings in the four signature paths. By the union bound, the probability that there exist two distinguishable states that are not distinguished by at least one of the strings in the four signature paths is at most

$$\binom{n}{2}\left(O\left(\frac{\log^6(n)}{n^3}\right)\right) = o(1).$$

Hence, by running at most four signature paths, each of length $3\log(n)$, per newly reached state, we get unique labels on the states. Then for each of the n states, we can find their $|X|$ transitions, and learn the machine, as in Proposition 7.3.2. $\square$

We now turn to the two lemmas used in the proof of Theorem 7.5.1. We first consider the case $|X| > 2$. If $a, b, c \in X$ and ℓ is a nonnegative integer, let $D_\ell(a, b, c)$ denote the set of all strings a^i, b^i, and c^i such that $0 \leq i \leq \ell$.

Lemma 7.5.2. *Let s_1 and s_2 be two different states in a random automaton with $|X| > 2$. Let $a, b, c \in X$ and $\ell = 3\log(n)$. The probability that s_1 and s_2 are* *distinguishable, but not by any string in* $D_\ell(a,b,c)$ *is* $O\left(\frac{\log^6(n)}{n^3}\right)$.

Proof. We analyze the three (attempted) paths from two states s_1 and s_2, which we will call $\pi^1_{s_1}, \pi^2_{s_1}, \pi^3_{s_1}$ and $\pi^1_{s_2}, \pi^2_{s_2}, \pi^3_{s_2}$, respectively. Each path will have length $3\log(n)$. We define each of the π_i as a set of nodes reached by its respective set of transitions.

We first look at the probability that the following event does not happen: that both $|\pi^1_{s_1}| > 3\log(n)$ and $|\pi^1_{s_2}| > 3\log(n)$, and that $\pi^1_{s_1} \cap \pi^1_{s_2} = \emptyset$, that is the probability that both of these strings succeed in reaching $3\log(n)$ different states, and that they share no states in common. We call the event that two sets of states π_1 and π_2 have no states in common, and both have size at least l, $S(\pi_1, \pi_2, l)$ (success) and the failure event $F(\pi_1, \pi_2, l) = \overline{S(\pi_1, \pi_2, l)}$. So,

$$\begin{aligned} P(F(\pi^1_{s_1}, \pi^1_{s_2}, 3\log(n))) &\leq \sum_{i=1}^{3\log(n)} \left(\frac{i + |\pi^1_{s_1}|}{n}\right) + \sum_{i=1}^{3\log(n)} \left(\frac{i + |\pi^1_{s_2}|}{n}\right) \\ &\leq 2\sum_{i=1}^{3\log(n)} \left(\frac{i + 3\log(n)}{n}\right) \\ &= O\left(\frac{\log^2(n)}{n}\right). \end{aligned} \tag{7.1}$$

Now we look at the probability that $F(\pi^2_{s_1}, \pi^2_{s_2}, 3\log(n))$ given that we failed on the first paths, or $F(\pi^1_{s_1}, \pi^1_{s_2}, 3\log(n))$, with $l = 3\log(n)$,

$$\begin{aligned} P\left(F(\pi^2_{s_1}, \pi^2_{s_2}, l) | F(\pi^1_{s_1}, \pi^1_{s_2}, l)\right) &\leq \sum_{i=1}^{3\log(n)} \left(\frac{i + |\pi^2_{s_1}| + |\pi^1_{s_1}| + |\pi^1_{s_2}|}{n}\right) \\ &\quad + \sum_{i=1}^{3\log(n)} \left(\frac{i + |\pi^2_{s_2}| + |\pi^1_{s_1}| + |\pi^1_{s_2}|}{n}\right) \\ &\leq 2\sum_{i=1}^{3\log(n)} \left(\frac{i + 9\log(n)}{n}\right) \\ &= O\left(\frac{\log^2(n)}{n}\right). \end{aligned}$$

So the probability of failing on both of the first two paths is

$$\begin{aligned} P\left(F(\pi^2_{s_1},\pi^2_{s_2},l),F(\pi^1_{s_1},\pi^1_{s_2},l)\right) &\leq P\left(F(\pi^2_{s_1},\pi^2_{s_2},l)|F(\pi^1_{s_1},\pi^1_{s_2},l)\right) \\ &\quad P(F(\pi^1_{s_1},\pi^1_{s_2},3\log(n))) \\ &= O\left(\frac{\log^2(n)}{n}\right)O\left(\frac{\log^2(n)}{n}\right) \\ &= O\left(\frac{\log^4(n)}{n^2}\right). \end{aligned} \tag{7.2}$$

Now, we will compute the probability that $F(\pi^3_{s_1},\pi^3_{s_2},3\log(n))$ given failures on the previous two pairs of states. Let $l = 3\log(n)$,

$$\begin{aligned} P\left(F(\pi^3_{s_1},\pi^3_{s_2},l)|F(\pi^1_{s_1},\pi^1_{s_2},l),F(\pi^2_{s_1},\pi^2_{s_2},l)\right) &\leq 2\sum_{i=1}^{3\log(n)}\left(\frac{i+25\log(n)}{n}\right) \\ &= O\left(\frac{\log^2(n)}{n}\right). \end{aligned}$$

Last, we compute the probability none of these pairs of paths made it to $l = 3\log(n)$, or $P(\text{failure}) = P\left(F(\pi^1_{s_1},\pi^1_{s_2},l),F(\pi^2_{s_1},\pi^2_{s_2},l),F(\pi^3_{s_1},\pi^3_{s_2},l)\right)$

$$\begin{aligned} P(\text{failure}) &= P(F(\pi^1_{s_1},\pi^1_{s_2},l))\cdot P\left(F(\pi^2_{s_1},\pi^2_{s_2},l)|F(\pi^1_{s_1},\pi^1_{s_2},l)\right)\cdot \\ &\quad P\left(F(\pi^3_{s_1},\pi^3_{s_2},l)|F(\pi^1_{s_1},\pi^1_{s_2},l),F(\pi^2_{s_1},\pi^2_{s_2},1)\right) \\ &= O\left(\frac{\log^2(n)}{n}\right)O\left(\frac{\log^2(n)}{n}\right)O\left(\frac{\log^2(n)}{n}\right) \\ &= O\left(\frac{\log^6(n)}{n^3}\right). \end{aligned} \tag{7.3}$$

Thus, given two distinct states with corresponding nonoverlapping signature paths of length $3\log(n)$, the probability that all of the randomly chosen labels along the paths will be the same is $2^{3\lg(n)} = \frac{1}{n^3} = O\left(\frac{\log^6(n)}{n^3}\right)$, which is the probability that no string in $D_\ell(a,b,c)$ distinguishes s_1 from s_2. $\square$

When $|X| = 2$, we do not have enough alphabet symbols to construct three completely independent paths as in the proof of Lemma 7.5.2, but four paths suffice. If $a, b \in X$ and ℓ is a nonnegative integer, let $D_\ell(a, b)$ denote the set of all strings a^i, b^i, ab^i and ba^i such that $0 \leq i \leq \ell$.

Lemma 7.5.3. *Let s_1 and s_2 be two different states in a random automaton with $|X| = 2$. Let $a, b \in X$ and $\ell = 3\log(n)$. The probability that s_1 and s_2 are distinguishable, but not by any string in $D_\ell(a, b)$ is $O\left(\frac{\log^6(n)}{n^3}\right)$.*

Proof. We do a case analysis that uses reasoning similar to that of Lemma 7.5.2. If s_1 and s_2 are assigned different labels, then they are distinguished by the empty string, so assume that they are assigned the same label. If we consider $\tau(s_1, a)$ and $\tau(s_2, a)$, there are four cases, as follows.

1. Neither of $\tau(s_1, a)$ and $\tau(s_2, a)$ is in the set $\{s_1, s_2\}$, and $\tau(s_1, a) \neq \tau(s_2, a)$. In this case, we use the argument from Lemma 7.5.2 that shows that the probability that the paths a^i, ab^i and b^i fail to produce a distinguishing string for s_1 and s_2 is bounded by $O(\log^6(n)/n^3)$ from Equation 7.3.

2. Exactly one of $\tau(s_1, a)$ and $\tau(s_2, a)$ is in the set $\{s_1, s_2\}$. This happens with probability $O(1/n)$, and in this case an argument similar to Equation 7.2 shows that the probability that the paths a^i and b^i do not produce a distinguishing string for s_1 and s_2 is bounded by $O(\log^4(n)/n^2)$, for a total failure probability of $O(\log^4(n)/n^3)$ for this case.

3. Both of $\tau(s_1, a)$ and $\tau(s_2, a)$ are in the set $\{s_1, s_2\}$. This happens with probability $O(1/n^2)$, and in this case an argument similar to Equation 7.1 that the probability that the path b^i does not produce a distinguishing string for s_1 and s_2 is bounded by $O(log^2(n)/n)$, for a total failure probability of $O(\log^2(n)/n^3)$ for this case.

4. Neither of $\tau(s_1, a)$ and $\tau(s_2, a)$ is in the set $\{s_1, s_2\}$, but $\tau(s_1, a) = \tau(s_2, a)$. This happens with probability $O(1/n)$, and we proceed to analyze four parallel subcases for $\tau(s_1, b)$ and $\tau(s_2, b)$.

   (a) We have $\tau(s_1, b) \neq \tau(s_2, b)$ and neither of them is in the set $\{s_1, s_2\}$. We can show that the probability that the paths b^i and ba^i do not produce a distinguishing string for s_1 and s_2 is bounded by $O(\log^4(n)/n^2)$ (via Equation 7.2), for a failure probability of $O(\log^4(n)/n^3)$ in this subcase, because the probability of case (4) is $O(1/n)$.

   (b) Exactly one of $\tau(s_1, b)$ and $\tau(s_2, b)$ is in the set $\{s_1, s_2\}$. In this subcase, we can show (via Equation 7.1) that the probability that the path b^i fails to produce a distinguishing string for s_1 and s_2 is bounded by $O(\log^2(n)/n)$, for a total failure probability in this subcase of $O(\log^2(n)/n^3)$, because the probability of case (4) is $O(1/n)$ and the probability that one of $\tau(s_1, b)$ and $\tau(s_2, b)$ is in $\{s_1, s_2\}$ is $O(1/n)$.

   (c) Both of $\tau(s_1, b)$ and $\tau(s_2, b)$ are in $\{s_1, s_2\}$. The probability of this happening is $O(1/n^2)$, for a total probability of this subcase of $O(1/n^3)$, because the probability of case (4) is $O(1/n)$.

   (d) We have $\tau(s_1, b) = \tau(s_2, b)$. Then because we are in case (4), $\tau(s_1, a) = \tau(s_2, a)$ and the labels assigned s_1 and s_2 are equal, so the states s_1 and s_2 are equivalent and therefore indistinguishable.

□

Bibliography

[1] Aho, A. V., Garey, M. R., and Ullman, J. D. The transitive reduction of a directed graph. *SIAM J. Comput. 1* (1972), 131–137.

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

[3] Alon, N., and Asodi, V. Learning a hidden subgraph. *SIAM J. Discrete Math. 18*, 4 (2005), 697–712.

[4] Alon, N., Awerbuch, B., Azar, Y., Buchbinder, N., and Naor, J. The online set cover problem. In *Proceedings of the 35th annual ACM symposium on Theory of computing* (2003), pp. 100–105.

[5] Alon, N., Awerbuch, B., Azar, Y., Buchbinder, N., and Naor, J. A general approach to online network optimization problems. *ACM Transactions on Algorithms 2*, 4 (2006), 640–660.

[6] Alon, N., Beigel, R., Kasif, S., Rudich, S., and Sudakov, B. Learning a hidden matching. *SIAM J. Comput. 33*, 2 (2004), 487–501.

[7] Alon, N., Yuster, R., and Zwick, U. Color-coding. *J. ACM 42*, 4 (1995), 844–856.

[8] Angluin, D. A note on the number of queries needed to identify regular languages. *Information and Control 51*, 1 (1981), 76–87.

[9] Angluin, D. Learning regular sets from queries and counterexamples. *Inf. Comput. 75*, 2 (1987), 87–106.

[10] Angluin, D. Queries and concept learning. *Machine Learning 2*, 4 (1987), 319–342.

[11] Angluin, D., Aspnes, J., Chen, J., Eisenstat, D., and Reyzin, L. Learning acyclic probabilistic circuits using test paths. In *21st Annual Conference on Learning Theory* (2008), pp. 169–180.

[12] Angluin, D., Aspnes, J., Chen, J., and Reyzin, L. Learning large-alphabet and analog circuits with value injection queries. In *20th Annual Conference on Learning Theory* (2007), pp. 51–65.

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

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

[15] Angluin, D., Aspnes, J., and Reyzin, L. Optimally learning social networks with activations and suppression. To appear, Theor. Comput. Sci. 2009.

[16] Angluin, D., Aspnes, J., and Reyzin, L. Optimally learning social networks with activations and suppressions. In *19th International Conference on Algorithmic Learning Theory* (2008), pp. 272–286.

[17] Angluin, D., Becerra-Bonache, L., Dediu, A. H., and Reyzin, L. Learning finite automata using label queries. To appear, 20th International Conference on Algorithmic Learning Theory.

[18] Angluin, D., and Chen, J. Learning a hidden graph using $O(\log(n))$ queries per edge. In *17th Annual Conference on Learning Theory* (2004), pp. 210–223.

[19] Angluin, D., and Chen, J. Learning a hidden hypergraph. *Journal of Machine Learning Research 7* (2006), 2215–2236.

[20] Angluin, D., and Chen, J. Learning a hidden graph using $O(\log n)$ queries per edge. *J. Comput. Syst. Sci. 74*, 4 (2008), 546–556.

[21] Angluin, D., Frazier, M., and Pitt, L. Learning conjunctions of Horn clauses. *Machine Learning 9* (1992), 147–164.

[22] Angluin, D., Hellerstein, L., and Karpinski, M. Learning read-once formulas with queries. *J. ACM 40* (1993), 185–210.

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

[24] Angluin, D., and Laird, P. Learning from noisy examples. *Mach. Learn. 2*, 4 (1988), 343–370.

[25] Becerra-Bonache, L., Dediu, A. H., and Tîrnăucă, C. Learning DFA from correction and equivalence queries. In *ICGI* (2006), pp. 281–292.

[26] Beerliova, Z., Eberhard, F., Erlebach, T., Hall, A., Hoffmann, M., Mihalák, M., and Ram, L. S. Network discovery and verification. In *31st International Workshop on Graph-Theoretic Concepts in Computer Science* (2005), pp. 127–138.

[27] Beigel, R., Alon, N., Kasif, S., Apaydin, M. S., and Fortnow, L. An optimal procedure for gap closing in whole genome shotgun sequencing. In *RECOMB* (2001), pp. 22–30.

[28] Booth, K. S., and Lueker, G. S. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. Syst. Sci. 13*, 3 (1976), 335–379.

[29] Bouvel, M., Grebinski, V., and Kucherov, G. Combinatorial search on graphs motivated by bioinformatics applications: A brief survey. In *31st International Workshop on Graph-Theoretic Concepts in Computer Science* (2005), pp. 16–27.

[30] Brodal, G. S., Fagerberg, R., Pedersen, C. N. S., and Ostlin, A. The complexity of constructing evolutionary trees using experiments. In *28th International Colloquium on Automata, Languages, and Programming*. Springer, 2001, pp. 140–151.

[31] Bshouty, N. H. Exact earning boolean functions via the monotone theory. *Inf. Comput. 123*, 1 (1995), 146–153.

[32] Buchbinder, N. *Designing Competitive Online Algorithms Via A Primal-Dual Approach*. PhD thesis, Technion – Israel Institute of Technology, Haifa, Israel, 2008.

[33] Chapelle, O., Schölkopf, B., and Zien, A., Eds. *Semi-Supervised Learning*. MIT Press, Cambridge, MA, 2006.

[34] Choi, S.-S., and Kim, J. H. Optimal query complexity bounds for finding graphs. In *Proceedings of the 40th annual ACM symposium on Theory of computing* (New York, NY, USA, 2008), ACM, pp. 749–758.

[35] Cohn, D. A., Atlas, L. E., and Ladner, R. E. Improving generalization with active learning. *Machine Learning 15*, 2 (1994), 201–221.

[36] Cohn, D. A., Ghahramani, Z., and Jordan, M. I. Active learning with statistical models. *J. Artif. Intell. Res. (JAIR) 4* (1996), 129–145.

[37] Culberson, J. C., and Rudnicki, P. A fast algorithm for constructing trees from distance matrices. *Inf. Process. Lett. 30*, 4 (1989), 215–220.

[38] de Bruijn, N. G. *Asymptotic Methods in Analysis.* Dover, 1981.

[39] Domaratzki, M., Kisman, D., and Shallit, J. On the number of distinct languages accepted by finite automata with n states. *Journal of Automata, Languages and Combinatorics 7*, 4 (2002).

[40] Downey, R. G., and Fellows, M. R. *Parameterized Complexity.* Springer-Verlag, 1999.

[41] Erdös, P., and Rényi, A. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci 5* (1960), 17–61.

[42] Feige, U. A threshold of $\ln(n)$ for approximating set cover. *J. ACM 45*, 4 (1998), 634–652.

[43] Freivalds, R. Probabilistic machines can use less running time. In *IFIP Congress* (1977), pp. 839–842.

[44] Freund, Y., Kearns, M. J., Ron, D., Rubinfeld, R., Schapire, R. E., and Sellie, L. Efficient learning of typical finite automata from random walks. *Information and Computation 138*, 1 (1997), 23–48.

[45] Freund, Y., Seung, H. S., Shamir, E., and Tishby, N. Selective sampling using the query by committee algorithm. *Machine Learning 28*, 2-3 (1997), 133–168.

[46] Gold, E. M. Language identification in the limit. *Information and Control 10*, 5 (1967), 447–474.

[47] Goldenberg, J., Libai, B., and Muller, E. 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).

[48] Goldreich, O., Goldwasser, S., and Ron, D. Property testing and its connection to learning and approximation. *J. ACM 45*, 4 (1998), 653–750.

[49] Grebinski, V., and Kucherov, G. Reconstructing a hamiltonian cycle by querying the graph: Application to DNA physical mapping. *Discrete Applied Mathematics 88*, 1-3 (1998), 147–165.

[50] Grebinski, V., and Kucherov, G. Optimal reconstruction of graphs under the additive model. *Algorithmica 28*, 1 (2000), 104–124.

[51] Gupta, A., Krishnaswamy, R., and Ravi, R. Online and stochastic survivable network design. In *Proceedings of the 41st annual ACM symposium on Theory of computing* (2009), pp. 685–694.

[52] Hein, J. J. An optimal algorithm to reconstruct trees from additive distance data. *Bulletin of Mathematical Biology 51*, 5 (1989), 597–603.

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

[54] JACKSON, J. C. An efficient membership-query algorithm for learning DNF with respect to the uniform distribution. *J. Comput. Syst. Sci. 55*, 3 (1997), 414–440.

[55] JACKSON, J. C., KLIVANS, A. R., AND SERVEDIO, R. A. Learnability beyond AC0. In *Proceedings of the 34th annual ACM symposium on Theory of computing* (New York, NY, USA, 2002), ACM Press, pp. 776–784.

[56] JEFFREYS, H., AND JEFFREYS, B. *Methods of Mathematical Physics*, 3rd. ed. Cambridge University Press, Cambridge, England, 1988.

[57] KAELBLING, L. P., LITTMAN, M. L., AND MOORE, A. P. Reinforcement learning: A survey. *J. Artif. Intell. Res. (JAIR) 4* (1996), 237–285.

[58] KANNAN, S. K., LAWLER, E. L., AND WARNOW, T. J. Determining the evolutionary tree using experiments. *J. Algorithms 21*, 1 (1996), 26–50.

[59] KEARNS, M. Efficient noise-tolerant learning from statistical queries. *J. ACM 45*, 6 (1998), 983–1006.

[60] KEARNS, M., AND VALIANT, L. Cryptographic limitations on learning boolean formulae and finite automata. *J. ACM 41*, 1 (1994), 67–95.

[61] KEARNS, M. J., SCHAPIRE, R. E., AND SELLIE, L. M. Toward efficient agnostic learning. *Mach. Learn. 17*, 2-3 (1994), 115–141.

[62] KEMPE, D., KLEINBERG, J., 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* (New York, NY, USA, 2003), ACM, pp. 137–146.

[63] Kempe, D., Kleinberg, J. M., and Tardos, É. Influential nodes in a diffusion model for social networks. In *32nd International Colloquium on Automata, Languages and Programming* (2005), pp. 1127–1138.

[64] Kharitonov, M. Cryptographic hardness of distribution-specific learning. In *Proceedings of the 25th annual ACM symposium on Theory of computing* (New York, NY, USA, 1993), ACM Press, pp. 372–381.

[65] King, V., Zhang, L., and Zhou, Y. On the complexity of distance-based evolutionary tree reconstruction. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 2003), Society for Industrial and Applied Mathematics, pp. 444–453.

[66] Korach, E., and Stern, M. The clustering matroid and the optimal clustering tree. *Mathematical Programming 98*, 1-3 (2003), 345–414.

[67] Korach, E., and Stern, M. The complete optimal stars-clustering-tree problem. *Discrete Applied Mathematics 156*, 4 (2008), 444–450.

[68] Korshunov, A. The degree of distinguishability of automata. *Diskret. Analiz. 10*, 36 (1967), 39–59.

[69] Lee, D., and Yannakakis, M. Testing finite-state machines: State identification and verification. *IEEE Trans. Computers 43*, 3 (1994), 306–320.

[70] Lewis, D. D., and Catlett, J. Heterogenous uncertainty sampling for supervised learning. In *11th International Conference on Machine Learning* (1994), pp. 148–156.

[71] Lewis, D. D., and Gale, W. A. A sequential algorithm for training text classifiers. In *17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval* (1994), pp. 3–12.

[72] Lingas, A., Olsson, H., and Ostlin, A. Efficient merging, construction, and maintenance of evolutionary trees. In *26th International Colloquium on Automata, Languages and Programming* (London, UK, 1999), Springer-Verlag, pp. 544–553.

[73] Linial, N., Mansour, Y., and Nisan, N. Constant depth circuits, Fourier transform, and learnability. *Journal of the ACM 40*, 3 (1993), 607–620.

[74] Littlestone, N. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Mach. Learn. 2*, 4 (1987), 285–318.

[75] Littlestone, N., and Warmuth, M. K. The weighted majority algorithm. In *30th Annual Symposium on Foundations of Computer Science* (1989), pp. 256–261.

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

[77] Niedermeier, R. *Invitation to Fixed-Parameter Algorithms.* Oxford University Press, 2006.

[78] Reyzin, L., and Srivastava, N. Learning and verifying graphs using queries with a focus on edge counting. In *18th International Conference on Algorithmic Learning Theory* (2007), pp. 285–297.

[79] Reyzin, L., and Srivastava, N. On the longest path algorithm for reconstructing trees from distance matrices. *Inf. Process. Lett. 101*, 3 (2007), 98–100.

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

[81] Settles, B. Active learning literature survey. Tech. Rep. UW-CS-TR-1648, University of Wisconsin - Madison, Computer Sciences Department, January 2009.

[82] Seung, H. S., Opper, M., and Sompolinsky, H. Query by committee. In *5th Annual Conference on Learning Theory* (1992), pp. 287–294.

[83] Slavík, P. A tight analysis of the greedy algorithm for set cover. In *Proceedings of the 28th annual ACM symposium on Theory of computing* (New York, NY, USA, 1996), ACM, pp. 435–441.

[84] Trakhtenbrot, B. A., and Barzdin', Y. M. *Finite Automata: Behavior and Synthesis.* North Holland, Amsterdam, 1973.

[85] Valiant, L. G. A theory of the learnable. *Commun. ACM 27* (1984), 1134–1142.