Learning Graphs via Queries
690 Report

Lev Reyzin

July 11, 2007

Abstract

In this report[1] , we explore various aspects of query learning. We focus on learning hidden structures given various queries. In Chapter 1, we consider learning evolutionary trees given distance queries. In Chapter 2 we focus on learning and verifying general graph structures with various queries. In Chapter 3 we are interested in learning circuits with value-injection queries.

Chapter 1 is based on a paper coauthored with Nikhil Srivastava, entitled "On the Longest Path Algorithm for Reconstructing Trees from Distance Matrices." This paper appears in Information Processing Letters, 2007 [35].

Chapter 2 is based on a paper coauthored with Nikhil Srivastava, entitled "Learning and Verifying Graphs using Queries with a Focus on Edge Counting." This paper has been submitted to the Symposium on Algorithmic Learning Theory, 2007 [34].

Chapter 3 is based on a paper coauthored with Dana Angluin, James Aspnes, and Jiang Chen, entitled "Learning Large-Alphabet and Analog Circuits with Value Injection Queries." This paper appears in the Conference on Learning Theory, 2007 [5].

[1]I would especially like to thank Dana Angluin for being such a great advisor for this 690 project (and in general). I would also like to thank Nikhil Srivastava for the close collaborations on two papers, as well as my other co-authors - Jim Aspnes and Jiang Chen

Contents

Chapter 1

Evolutionary Tree Reconstruction

Chapter Summary

Culberson and Rudnicki [16] gave an algorithm that reconstructs a degree d restricted tree from its distance matrix. According to their analysis, it runs in time $O(dn \log_d n)$ for topological trees. However, this turns out to be false; in this chapter, we show that the algorithm takes $\Theta(n^{3/2}\sqrt{d})$ time in the topological case, giving tight examples.

1.1 Introduction and Background

In [16], Culberson and Rudnicki consider the problem of reconstructing a degree d restricted weighted tree given its distance matrix D. If the tree has n vertices, D is an $n \times n$ matrix where each entry $d_{ij} \in \mathbb{R}_{\geq 0}$ gives the 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 looked at only two columns of D.

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 queries to D. To be precise, z is in the subtree rooted at w iff $d_{zu} - d_{zv} = d_{wu} - d_{wv}$.

3. No further queries to D 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 queries to D are concerned.

The algorithm presented in [16] is equivalent to what is described above, with minor simplifications. We will call this algorithm LongestPath.

In their analysis, Culberson and Rudnicki refer to the lookups to D in step (1) as 'hub computations' and establish that the running time is dominated by them. They claim that for topological trees (where all edge weights are 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.*

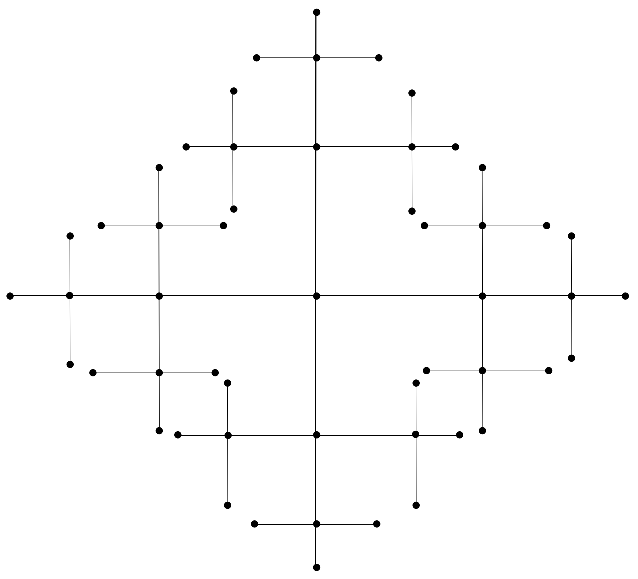

Figure 1.1: Tree that minimizes the longest path in every subtree, from [16].

So according to [16], the worst case for degree four is the kind of tree shown in Figure 1.1.

Yet, this proof is incorrect. In section 1.2, we construct a topological tree on which the algorithm takes $\Omega(n^{3/2}\sqrt{d})$ time, contradicting the claim in [16]. In section 1.3 we present a revised analysis of LongestPath, showing that $\Theta(n^{3/2}\sqrt{d})$ is tight for the topological case.

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 [30]. A quite different algorithm called DeepestPoint[1], discovered by Hein [23], reconstructs both general and topological trees in $O(dn \log_d n)$ time. Further, variants of this problem have been studied in the experiment model for constructing evolu-

[1]In constructing DeepestPoint[23], Hein mentions in passing that an algorithm recursively placing longest paths would run in $\Omega(n^{\frac{3}{2}})$

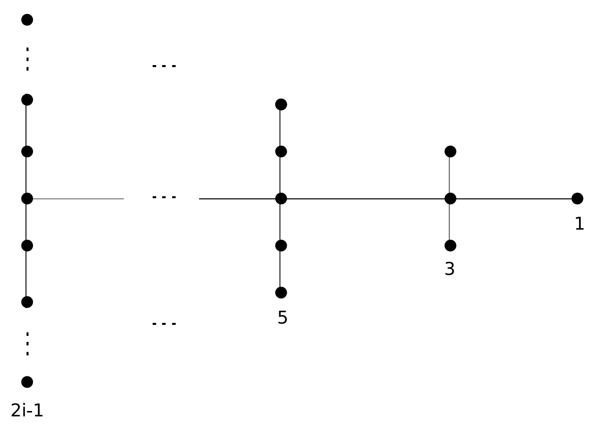

Figure 1.2: A counterexample to the $O(dn \log_d n)$ analysis of [16], the tree G_i.

tionary trees by Kannan et al. [27], Brodal et al. [14], and Lingas et al. [31]. Notwithstanding, LongestPath is one of the first algorithms claimed to run in sub-quadratic time for tree reconstruction and also one of the simplest. This paper presents a correct analysis of it.

1.2 A Counterexample

While the tree in Figure 1.1 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 remaining vertices in a single subtree, as shown in Figure 1.2. 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}.

For a tree of size n, finding a longest path lying vertically along the spine takes $\Omega(n)$ queries to D 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:

$$\begin{aligned} T(n) &\geq n + (n-\sqrt{n}) + (n - \sqrt{n} - \sqrt{n-\sqrt{n}}) \ldots \\ &\geq n + (n-\sqrt{n}) + (n-\sqrt{n}-\sqrt{n}) \ldots \\ &= n + (n-\sqrt{n}) + (n-2\sqrt{n}) \ldots (n - \sqrt{n}\cdot\sqrt{n}) \\ &= \sqrt{n}\cdot n - \sqrt{n}\cdot\frac{\sqrt{n}(\sqrt{n}+1)}{2} \end{aligned}$$

to get $T(n) = \Omega(n^{3/2})$.

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), LongestPath 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 $T(n) = \Omega(n^{3/2}\sqrt{d})$, showing that the $O(dn\log_d n)$ analysis of [16] is incorrect.

1.3 Analysis of the Topological 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 queries to D 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 LongestPath by $n\times$(number of phases).

The correct analysis of LongestPath relies on following observation:

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

Proof. Suppose the longest path π chosen by LongestPath 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 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 LongestPath terminates. $\square$

Theorem 3. LongestPath *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 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. $\square$

Chapter 2

Learning and Verifying Graphs via Queries

Chapter Summary

In this chapter, we consider the problem of learning and verifying hidden graphs and their properties given query access to the graphs. We analyze various queries (edge detection, edge counting, shortest path), but we focus mainly on edge counting queries. We give an algorithm for learning graph partitions using $O(n \log n)$ edge counting queries. We introduce a problem that has not been considered, verifying graphs with edge counting queries, and give a randomized algorithm with error ϵ for graph verification using $O(\log(1/\epsilon))$ edge counting queries. We examine the current state of the art and add some original results for edge detection and shortest path queries to give a more complete picture of the relative power of these queries to learn various graph classes. Finally, we relate our work to Freivalds' 'fingerprinting technique' – a probabilistic method for verifying that two matrices are equal by multiplying them by random vectors.

2.1 Introduction

Graph learning appears in many different contexts. 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.

In a different setting [7], testing which pairs of chemicals react in a solution is modeled by 'edge detection' queries. Here, vertices correspond to chemicals, edges designate chemical reactions, and a set of chemicals 'reacts' iff it induces an edge. Applications of this model extend to bioinformatics, where learning a hidden matching [4] turns out to be useful in DNA sequencing. With each setup we have different tools and target concepts to learn.

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 detection 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, 4, 7, 8, 21].*

- **Edge counting query (EC)**: Return the number of edges in the subgraph induced by $S \subseteq V$. *This has extensive uses in bioinformatics and was studied in [13, 22].*

- **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 [23, 30, 35].*

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? 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 we know not to have underlying cycles, for example evolutionary trees. Finally, we consider the problem of learning the **partition** of a graph into connected components. 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 paper 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.

2.2 Previous Work

In one of the earliest works in graph discovery, Hein [23] 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. [30] provide a matching lower bound by showing that solving this problem requires solving multiple partition problems whose difficulty they then analyze.

Angluin and Chen [7] 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 [8] generalizes these results to hypergraphs using different techniques.

The work of Angluin and Chen is preceded by a few papers [3, 4, 21] that tackle learning restricted families of graphs, such as stars, cliques, and matchings. Alon et al. [4] 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 [21] study reconstructing Hamiltonian paths with **ED** queries. It turns out that many of these results are subsumed by those of [7] if we ignore constant factors.

Grebinski and Kucherov [22] 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 [13], 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. [12] 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 graph-theoretic distance 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 [20].

2.3 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 paper, 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 4. $\Omega(n^2)$ **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.

$\square$

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 5. $O(nk)$ **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.
- 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 6. $\Omega(n^2)$ **ED** *queries are needed to learn the* ***partition*** *of a hidden graph on n vertices.*

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 7. $\Omega(n)$ **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 [17]:

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

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.

$\square$

Theorem 8. *$O(n \lg n)$* **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 $c_j \in \mathcal{C}$[1]. This implies that there are edges between c_j and v; we will call c_j a *live* component.

  Repeat on $\mathcal{C} \setminus \{c_j\}$ 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

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

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:

- $\Delta_i = 1$: 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 $k - 1$.[2] 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.

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

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.

□

Proposition 9. $O(|E| \lg n)$ **EC** *queries are sufficient to learn a hidden* ***graph*** *on* n *vertices.*

Proof. The algorithm of Angluin and Chen ([7]) achieves this since $\mathbf{ED} \leq \mathbf{EC}$, 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 8. Split $V \setminus \{v\}$ into halves V_1, V_2 and query $\mathbf{EC}(V_1 + v), \mathbf{EC}(V_i + v)$. 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 10. $\Omega(n^2)$ **SP** *queries are needed to learn a hidden **tree**.*

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

Suppose a 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 **E** queries, which we know by [4] takes $\Omega(n^2)$ queries.

$\square$

Query	**partition**	**graph**	**tree**
ED	$\Theta(n^2)$	$\Theta(\lvert E\rvert \lg n), \Theta(n^2)$[7]	$\Theta(n \lg n)$
EC	$O(n \lg n)$ $\Omega(n)$	$O(\lvert E\rvert \lg n), O(\frac{n^2}{\lg n}), O(dn)$[7, 22] $\Omega(dn), \Omega(\frac{n^2}{\lg n})$[22]	$\Theta(n)$
SP	$\Theta(nk)$	$\Theta(n^2)$	$\Theta(n^2), \Theta(dn \lg n)$ [23, 30]

Table 2.1: Summary of results. n denotes the number of vertices, $|E|$ the number of edges, d the degree restriction, and k the number of components.

Table 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.

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

Proof. Consider the problem of verifying a clique, when the hidden graph is a clique with some edge (u, v) removed, and the verifier knows this. $\mathbf{SP}(u', v') = 2$ 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 12. *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.

$\square$

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 13. *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 14. *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 13. Since this result has 1-sided error, we can easily boost the $\frac{1}{4}$ probability to any constant, and Corollary 15 follows immediately.

Corollary 15. *Any graph can be verified by a randomized algorithm with error ϵ using $O(\log(\frac{1}{\epsilon}))$* **EC** *queries.*

2.4.1 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 [19]. 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 uniformally at random, if $A \neq B$ we have

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

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' iff $v^T Av = v^T Bv$. By Theorem 13 if $A \neq B$ then $Pr[v^T Av \neq v^T Bv] \geq \frac{1}{4}$.

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 $\mathbb{P}[v^T Av \neq v^T Bv] \geq 1/4$ (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 $v \in \{0,1\}^n$, $v^T Av = v^T Bv$. 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

$$v^T(A + A^T)v = v^T Av + v^T A^T v = v^T Av + (v^T Av)^T = 2v^T Av$$

for all v, so that $v^T Av = v^T Bv$ for all v.

Hence, we know that standard fingerprinting techniques do not imply Theorem 13. Furthermore, the proof to Theorem 13 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 16. *Let A and B be $n \times n$ symmetric matrices over a field such that $A \neq B$,[3] then for v chosen uniformally at random from $v \in \{0,1\}^n$, $Pr[v^T Av \neq v^T Bv] \geq \frac{1}{4}$.*

Proof. Let $C = A - B \neq 0$, and note that $v^T Av \neq v^T Bv \iff v^T Cv \neq 0$. 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 $\texttt{wt}(u, v) = C(u, v)$. We proceed as in the proof of Lemma 14. Fix $v_1 \dots v_n$ so that $\texttt{wt}(v_{n-1}, v_n) \neq 0$, and let H' be as before. Define:

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

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

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

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

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

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

as desired.

□

2.5 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.

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 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 2.2) 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.

Chapter 3

Learning Large-Alphabet Circuits

Chapter Summary

In this chapter, we consider the problem of learning an acyclic discrete circuit with n wires, fan-in bounded by k and alphabet size s using value injection queries. For the class of transitively reduced circuits, we develop the Distinguishing Paths Algorithm, that learns such a circuit using $(ns)^{O(k)}$ value injection queries and time polynomial in the number of queries. We note a generalization of the algorithm to the class of circuits with shortcut width bounded by b that uses $(ns)^{O(k+b)}$ value injection queries.

3.1 Introduction

We consider learning large-alphabet acyclic circuits in the value injection model introduced in [6]. 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 [6] 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 includes 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, which motivates our interest in larger alphabets.

Akutsu et al. [2] and Ideker, Thorsson, and Karp [24] 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 [28] 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 [11], even with respect to the uniform distribution [29]. Furthermore, positive learnability results exist only for fairly limited classes, including propositional Horn formulas [9], general read once Boolean formulas [10], and decision trees [15], and those for specific distributions, including **AC0** circuits [32], DNF formulas [25], and **AC0** circuits with a limited number of majority gates [26].[1]

[1]Algorithms in both [32] and [26] for learning **AC0** circuits and their variants run in quasi-polynomial time.

Thus, Angluin et al. [6] 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 paper. 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 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 [18, 33].

3.2 Preliminaries

3.2.1 Circuits

We give a general definition of acyclic circuits whose wires carry values from a set X. For each nonnegative integer k, a **gate function** of arity k is a function from X^k to X. 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.

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 paper we consider only acyclic circuits. 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]. 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. An acyclic directed graph G is **transitively reduced** if it contains no shortcut edges. A circuit is transitively reduced if its graph is transitively reduced.

3.2.2 Experiments on circuits

Let C be a circuit. An **experiment** e is a function mapping each wire of C to $X \cup \{*\}$, where $*$ is not an element of X. 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 $\sigma \in X$, 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 $X \cup \{*\}$ 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, denoted $w_i(e)$, of each wire in C under the experiment e as follows. If $e(w_i) = x$ and $x \neq *$, then $w_i(e) = x$. 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, this uniquely defines $w_i(e) \in X$ 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 X. If $C(e) = C'(e)$ for every experiment e, then we say that C and C' are **behaviorally equivalent**.

3.2.3 The learning problem

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 X 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 equivalent to C^*. The goal is behavioral equivalence and the learning algorithm should run in time polynomial in n.

3.3 Learning Large-Alphabet Circuits with Distinguishing Paths

In this section we consider the problem of learning a discrete circuit when the alphabet X of possible values is of size $n^{O(1)}$. We prove the following theorem.

Theorem 17. *The Distinguishing Paths Algorithm learns the class of transitively reduced discrete circuits with n wires, alphabet size s, and fan-in bound k, using $O(n^{2k+1}s^{2k+2})$ value injection queries and time polynomial in the number of queries.*

3.3.1 Preliminaries

Our goal is to learn transitively reduced circuits of n nodes, with an alphabet Σ of size s and constant fan-in k. For each wire, we will keep a table, T which we call a distinguishing table, of size s^2 that keeps track of all possible pairs of values the wire can take.

We say that values σ_1 and σ_2 are distinguishable for wire w if there exists an experiment e such that $C^*(e|_w = \sigma_1) \neq C^*(e|_w = \sigma_2)$. We define **side wires** for a path from a wire w to the root to be a set of wires disjoint from the the path that are inputs to wires beyond w along the path to the root. We define a **distinguishing path** for wire w and distinguishable values σ_1 and σ_2 to be a path of wires from w to the output and settings of side wires of the path such that if s_1 and s_2 are experiments with the path free and side wires set in the path from w to the root such that

1. $\forall s_1, s_2\ C^*(s_{1|w=\sigma}) = C^*(s_{2|w=\sigma})$
2. $\exists \sigma_1, \sigma_2\ C^*(s_{|w=\sigma_1}) \neq C^*(s_{|w=\sigma_2})$

An entry of 0 in position (i, j) in table T_w means that a distinguishing path experiment has not (yet) been found to differentiate wire w having value i versus value j. An entry of 1 in position (i, j) corresponds to an experiment having been found that differentiates w having values i versus j. Together with this entry, the corresponding distinguishing path and a bit for whether this entry has been processed are recorded.

3.3.2 The Algorithm

We begin with the output wire having all 1 entries in its distinguishing table since the output wire detects differences between any two output values. The corresponding distinguishing paths to the 1 entries are empty sets and they are initialized as unprocessed.

Algorithm 1 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 exists a $T_w$ with an unprocessed 1 entry do
  Let $\pi$ be the corresponding distinguishing path.
  Run Extend Known Paths on $\pi$.
  Add all new edges $(v, w)$ to $G$.
  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.
Construct a circuit $C$ with $G$ and the tables $T_w$
Output $C$ and halt.
```

Intuitively, the algorithm works in the following way. At each step, we look at our set of distinguishing tables. For each distinguishing table that has unprocessed entries, we try to find more inputs to the gate. These new inputs are used to make new distinguishing paths that are left to be processed. We continue this until we have no more unprocessed entries. We then use the information to reconstruct the learned circuit.

We now describe the procedure **Extend Known Paths**. This procedure has two parts. First it finds new inputs to the current distinguishing path.

Given a distinguishing path π for wire w we consider the set of all experiments that agree with π and for the remaining wires, for all possible choices of up to k wires, try all possible settings of the chosen wires. We say a set of k wires is determining with respect to π if we can determine the value of the output given π and the settings of the k wires. The procedure for finding new inputs finds the largest set of determining wires with respect to π. We call this $V(\pi)$.

The second part of this procedure is to extend π by $V(\pi)$. For each wire $v \in V(\pi)$, this procedure searches for all pairs $\sigma_1, \sigma_2 \in \Sigma$ such that for some two experiments e_1 where $v = \sigma_1$ and e_2 where $v = \sigma_2$ such that e_1 and e_2 fix the remaining wires in $V(\pi)$ and π the same. Since v was in the determining set of wires for π, we know at least such two values must exist. This corresponds to a new distinguishing path for v, with π extended to the settings of the wires in $V(\pi) - v$ and v left free, that differentiates σ_1 from σ_2. This adds a 1 entry and a corresponding distinguishing path in $T_v(\sigma_1, \sigma_2)$.

Constructing the circuit. Now we show how to reconstruct the behavior of a circuit given its complete set of distinguishing tables. Given the distinguishing tables and graph G, we reconstruct circuit C to be behaviorally equivalent to C^*. G is a subgraph of the graph of C^* that has edges for all inputs the distinguishing paths algorithm has found.

We will construct gate tables for wires in C. They will keep different combinations of the wires' input values and their corresponding output. For the values that are not distinguishable (that have 0's in the distinguishing tables), we record values up to equivalence. If σ_1 and σ_2 are not distinguishable for w, we will place them in the same equivalence class.

We then process wires in any order, one at a time. For each wire w, for each value σ per equivalence class, we record the outputs of $C^*(e_\pi | w = \sigma)$ of all distinguishing paths in T_w. For each path π we record the output of C^* for experiment e_π and inputs set to the fixed values. We do this for all settings of inputs to w and this completes w's gate table.

3.3.3 Correctness and Running Time

We wish to show that the following conditions hold:

1. Correctness

   If $T(\sigma_1, \sigma_2) = 1$ then there is a distinguishing path s such that $C^*(s_{w=\sigma_1}) \neq C^*(s_{w=\sigma_2})$ in the table.

2. Completeness

   There do not exist T_w, σ_1, σ_2 such that $T_w(\sigma_1, \sigma_2) = 0$ but there exists an experiment s such that $C^*(s_{w=\sigma_1}) \neq C^*(s_{w=\sigma_2})$

Lemma 18. *At any point in the run of the algorithm, every* 1 *entry placed in distinguishing table* T_w *has a corresponding distinguishing path* π*.*

Proof. We can prove this by induction on the number of 1 entries after initialization. The base case is trivially true. Since every new 1 entry is found by extending a distinguishing path by *Extend Known Paths*, we need to show is that extending a distinguishing path in this manner gives a new distinguishing path.

More formally, let π' be a path produced for wire v and values σ_1 and σ_2 by extending the distinguishing path π for wire w, then we wish to show π' is a distinguishing path for wire v and values σ_1 and σ_2. Since v is an input to w, then π' is a path of wires from v to the output of C^*. The new side wires are fixed such that $C^*(s_{v=\sigma_1}) \neq C^*(s_{v=\sigma_2})$. It is not hard to check that this is a new distinguishing path.

□

We now consider the completeness condition. A distinguishing table is **complete** if for every pair of values σ_1 and σ_2 that are distinguishable for w, $T_w(\sigma_1, \sigma_2) = 1$.

Lemma 19. *When the algorithm terminates, every table T_w is complete.*

Proof. To make this argument, we need to use the fact that if σ_1 and σ_2 are distinguishable for wire w then there exists a distinguishing path for wire w and values σ_1 and σ_2. This is not hard to see by an inductive argument on the depth of wire w.

We look at the wire, w, closest to the output wire that has an incomplete distinguishing table, *i.e.* it is missing a 1 entry for two distinguishable values σ_1 and σ_2. We wish to argue that if the distinguishing tables above w are complete, then distinguishing paths algorithm will have found a distinguishing path for w, σ_1, and σ_2 – contradicting the assumption.

Knowing that a distinguishing path π exists for w, σ_1, and σ_2 we look at the suffix of π, without w and side-wires to w's parents. This gives us another distinguishing path, which would be extended to w by the algorithm. This contradicts the assumption that there are incomplete distinguishing tables.

□

3.3.4 Running Time

We now show that the Distinguishing Paths Algorithm runs in time polynomial in the number of wires and the size of the alphabet.

We derive a loose upper bound. At each iteration of the algorithm for filling distinguishing tables, for it not to terminate, at least one new 1 needs to be added to a distinguishing table, so the number of total iterations cannot exceed the number of spaces in the distinguishing tables, which is ns^2. In each iteration, for each existing 1 value in a distinguishing table, at most $\binom{n}{k} = O(n^k)$ choices of inputs are made and $O(s^k)$ experiments are performed (at most all possible alphabet values for the assumed inputs). So for each 1 value at most $O(n^k s^k)$ queries are asked. Finally, there are again at most ns^2 1 values, making $O(n^{k+1}s^{k+2})$ queries per iteration. In ns^2 iterations, we make at most $O(n^{k+2}s^{k+4})$ queries. To build the gate tables, for each of n wires, we try all s^2 distinguishing table experiments for all s values of the wire, which takes at most s^3 time. We then run the same experiments for each input setting to the wire, which takes $s^k s^2$. This takes a total of $\mathrm{O}(n(s^3 + s^{k+2}))$.

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 gene regulatory networks by strategic gene disruptions and gene overexpressions. In *SODA '98: Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 1998), Society for Industrial and Applied Mathematics, pp. 695–702.

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

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

[5] Angluin, D., Aspnes, J., Chen, J., and Reyzin, L. Learning large-alphabet and analog circuits with value injection queries. In *COLT* (2007), pp. 51–65.

[6] Angluin, D., Aspnes, J., Chen, J., and Wu, Y. Learning a circuit by injecting values. In *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2006), ACM Press, pp. 584–593.

[7] Angluin, D., and Chen, J. Learning a hidden graph using O(log n) queries per edge. In *COLT* (2004), pp. 210–223.

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

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

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

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

[12] Beerliova, Z., Eberhard, F., Erlebach, T., Hall, A., Hoffmann, M., Mihalák, M., and Ram, L. S. Network discovery and verification. In *WG* (2005), pp. 127–138.

[13] Bouvel, M., Grebinski, V., and Kucherov, G. Combinatorial search on graphs motivated by bioinformatics applications: A brief survey. In *WG* (2005), pp. 16–27.

[14] Brodal, G. S., Fagerberg, R., Pedersen, C. N. S., and Ostlin, A. The complexity of constructing evolutionary trees using experiments. In *Proc. 28th International Colloquium on Automata, Languages, and Programming*, vol. 2076 of *Lecture Notes in Computer Science*. 2001, pp. 140–151.

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

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

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

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

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

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

[21] 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.

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

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

[24] 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.

[25] 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.

[26] Jackson, J. C., Klivans, A. R., and Servedio, R. A. Learnability beyond AC0. In *STOC '02: Proceedings of the thirty-fourth annual ACM symposium on Theory of computing* (New York, NY, USA, 2002), ACM Press, pp. 776–784.

[27] Kannan, S. K., Lawler, E. L., and Warnow, T. J. Determining the evolutionary tree using experiments. *J. Algorithms 21*, 1 (1996), 26–50.

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

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

[30] 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.

[31] Lingas, A., Olsson, H., and Ostlin, A. Efficient merging, construction, and maintenance of evolutionary trees. In *ICAL '99: Proceedings of the 26th International Colloquium on Automata, Languages and Programming* (London, UK, 1999), Springer-Verlag, pp. 544–553.

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

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

[34] Reyzin, L., and Srivastava, N. Learning and verifying graphs using queries with a focus on edge counting. May 2007.

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