Exact Learning of Multitrees and Almost-Trees Using Path Queries

Ramtin Afshar(✉) and Michael T. Goodrich

Department of Computer Science, University of California, Irvine, USA
{afsharr,goodrich}@uci.edu

Abstract. Given a directed graph, $G = (V, E)$, a ***path*** query, $\mathsf{path}(u, v)$, returns whether there is a directed path from u to v in G, for $u, v \in V$. Given only V, exactly learning all the edges in G using path queries is often impossible, since path queries cannot detect transitive edges. In this paper, we study the query complexity of exact learning for cases when learning G is possible using path queries. In particular, we provide efficient learning algorithms, as well as lower bounds, for multitrees and almost-trees, including butterfly networks.

Keywords: Graph reconstruction · Exact learning · Directed acyclic graphs

1 Introduction

The exact learning of a graph, which is also known as ***graph reconstruction***, is the process of learning how a graph is connected using a set of queries, each involving a subset of vertices of the graph, to an all-knowing oracle. In this paper, we focus on learning a directed acyclic graph (DAG) using path queries. In particular, for a DAG, $G = (V, E)$, we are given the vertex set, V, but the edge set, E, is unknown and learning it through a set of path queries is our goal. A ***path*** query, $\mathsf{path}(u, v)$, takes two vertices, u and v in V, and returns whether there is a directed path from u to v in G.

This work is motivated by applications in various disciplines of science, such as biology [34,37,47,48], computer science [11,13,18–20,22,31,39], economics [26,27], psychology [38], and sociology [24]. For instance, it can be useful for learning phylogenetic networks from path queries. Phylogenetic networks capture ancestry relationships between a group of objects of the same type. For example, in a digital phylogenetic network, an object may be a multimedia file (a video or an image) [13,18–20], a text document [35,44], or a computer virus [22,39]. In such a network, each node represents an object, and directed edges show how an object has been manipulated or edited from other objects [5]. In a digital phylogenetic network, objects are usually archived and we can issue path queries between a pair of objects (see, e.g., [18]).

The full version of this paper is available in [3].

A. Castañeda and F. Rodríguez-Henríquez (Eds.): LATIN 2022, LNCS 13568, pp. 293–311, 2022.
https://doi.org/10.1007/978-3-031-20624-5_18

Learning a phylogenetic network has several applications. For instance, learning a multimedia phylogeny can be helpful in different areas such as security, forensics, and copyright enforcement [18]. Afshar *et al.* [5] studied learning phylogenetic trees (rooted trees) using path queries, where each object is the result of a modification of a single parent. Our work extends this scenario to applications where objects can be formed by merging two or more objects into one, such as image components. In addition, our work also has applications in biological scenarios that involve hybridization processes in phylogenetic networks [10].

Another application of our work is to learn the directed acyclic graph (DAG) structure of a causal Bayesian network (CBN). It is well-known that observational data (collected from an undisturbed system) is not sufficient for exact learning of the structure, and therefore interventional data is often used, by forcing some independent variables to take some specific values through experiments. An interventional path query requires a small number of experiments, since, $\mathsf{path}(i, j)$, intervenes the only variable correspondent to i. Therefore, applying our learning methods (similar to the method by Bello and Honorio, see [11]) can avoid an exponential number of experiments [33], and it can improve the results of Bello and Honorio [11] for the types of DAGs that we study.

We measure the efficiency of our methods in terms of the number of vertices, $n = |V|$, using these two complexities:

- Query complexity, $Q(n)$: This is the total number of queries that we perform. This parameter comes from the learning theory [2,14,21,46] and complexity theory [12,51].
- Round complexity, $R(n)$: This is the number of rounds that we perform our queries. The queries performed in a round are in a batch and they may not depend on the answer of the queries in the same round (but they may depend on the queries issued in the previous rounds).

Related Work. The problem of exact learning of a graph using a set of queries has been extensively studied [1,4–7,25,29,30,32,36,41–43,50]. With regard to previous work on learning directed graphs using path queries, Wang and Honorio [50] present a sequential randomized algorithm that takes $Q(n) \in O(n \log^2 n)$ path queries in expectation to learn rooted trees of maximum degree, d. Their divide and conquer approach is based on the notion of an even-separator, an edge that divides the tree into two subtrees of size at least n/d. Afshar *et al.* [5] show that learning a degree-d rooted tree with n nodes requires $\Omega(nd + n \log n)$ path queries [5] and they provide a randomized parallel algorithm for the same problem using $Q(n) \in O(n \log n)$ queries in $R(n) \in O(\log n)$ rounds with high probability (w.h.p.)[1], which instead relies on finding a near-separator, an edge that separates the tree into two subtrees of size at least $n/(d + 2)$, through a "noisy" process that requires noisy estimation of the number of descendants of a node by sampling. Their method, however, relies on the fact the ancestor set

[1] We say that an event happens with high probability if it occurs with probability at least $1 - \frac{1}{n^c}$, for some constant $c \geq 1$.

of a vertex in a rooted tree forms a total order. In Sect. 4, we extend their work to learn a rooted spanning tree for a DAG.

Regarding the reconstruction of trees with a specific height, Jagadish and Anindya [29] present a sequential deterministic algorithm to learn undirected fixed-degree trees of height h using $Q(n) \in O(nh \log n)$ separator queries, where a separator query given three vertices a, b, and c, it returns "true" if and only if b is on the path from a to c. Janardhanan and Reyzin [30] study the problem of learning an almost-tree of height h (a directed rooted tree with an additional cross-edge), and they present a randomized sequential algorithm using $Q(n) \in O(n \log^3 n + nh)$ queries.

Our Contributions. In Sect. 3, we present our learning algorithms for multitrees—a DAG with at most one directed path for any two vertices. We begin, however, by first presenting a deterministic result for learning directed rooted trees using path queries, giving a sequential deterministic approach to learn fixed-degree trees of height h, with $O(nh)$ queries, which provides an improvement over results by Jagadish and Anindya [29]. We then show how to use a tree-learning method to design an efficient learning method for a multitree with a roots using $Q(n) \in O(an \log n)$ queries and $R(n) \in O(a \log n)$ rounds w.h.p. We finally show how to use our tree learning method to design an algorithm with $Q(n) \in O(n^{3/2} \cdot \log^2 n)$ queries to learn butterfly networks w.h.p.

In Sect. 4, we introduce a separator theorem for DAGs, which is useful in learning a spanning-tree of a rooted DAG. Next, we present a parallel algorithm to learn almost-trees of height h, using $O(n \log n + nh)$ path queries in $O(\log n)$ parallel rounds w.h.p. We also provide a lower bound of $\Omega(n \log n + nh)$ for the worst case query complexity of a deterministic algorithm or an expected query complexity of a randomized algorithm for learning fixed-degree almost-trees proving that our algorithm is optimal. Moreover, this asymptotically-optimal query complexity bound, improves the sequential query complexity for this problem, since the best known results by Janardhanan and Reyzin [30] achieved a query complexity of $O(n \log^3 n + nh)$ in expectation.

2 Preliminaries

For a DAG, $G = (V, E)$, we represent the in-degree and out-degree of vertex $v \in V$ with $d_i(v)$ and $d_o(v)$ respectively. Throughout this paper, we assume that an input graph has maximum degree, d, i.e., for every $v \in V$, $d_i(v) + d_o(v) \leq d$. A vertex, v, is a root of the DAG if $d_i(v) = 0$. A DAG may have several roots, but we call a DAG rooted if it has only one root. Note that in a rooted DAG with root r, there is at least one directed path from r to every $v \in V$.

Definition 1 ***(arborescence).*** *An arborescence is a rooted DAG with root r that has exactly one path from r to each vertex $v \in V$. It is also referred to as a spanning directed tree at root r of a directed graph.*

We next introduce multitree, which is a family of DAGs useful in distributed computing [16,28] that we study in Sect. 3.

Definition 2 ***(multitree).*** *A multitree is a DAG in which the subgraph reachable from any vertex induces a tree, that is, it is a DAG with at most one directed path for any pair of vertices.*

We next review the definition of a butterfly network, which is a multitree used in high speed distributed computing [17,23,40] for which we provide efficient learning method in Sect. 3.

Definition 3 ***(Butterfly network).*** *A butterfly network, also known as depth-k Fast Fourier Transform (FFT) graph is a DAG recursively defined as $F^k = (V, E)$ as follows:*

- *For $k = 0$: F^0 is a single vertex, i.e. $V = \{v\}$ and $E = \{\}$.*
- *Otherwise, suppose $F_A^{k-1} = (V_A, E_A)$ and $F_B^{k-1} = (V_B, E_B)$ each having m sources and m targets $(t_0, ..., t_{m-1}) \in V_A$ and $(t_m, ..., t_{2m-1}) \in V_B$. Let $V_C = (v_0, v_1, ..., v_{2m-1})$ be $2m$ additional vertices. We have $F^k = (V, E)$, where $V = V_A \cup V_B \cup V_C$ and $E = E_A \cup E_B \bigcup_{0 \le i \le m-1} (t_i, v_i) \cup (t_i, v_{i+m}) \cup (t_{i+m}, v_i) \cup (t_{i+m}, v_{i+m})$ (See Fig. 1 for illustration).*

Definition 4 ***(ancestory).*** *Given a directed acyclic graph, $G = (V, E)$, we say u is a **parent** of a vertex v (v is a **child** of u), if there exists a directed edge $(u, v) \in E$. The **ancestor** relationship is a transitive closure of the parent relationship, and **descendant** relationship is a transitive closure of child relationship. We denote the descendant (resp. ancestor) set of vertex v, with $D(v)$, (resp. $A(v)$). Also, let $C(v)$ denote children of v.*

Definition 5. *A **path query** in a directed graph, $G = (V, E)$, is a function that takes two vertices u and v, and returns 1, if there is a directed path from u to v, and returns 0 otherwise. We also let $count(s, X) = \Sigma_{x \in X} path(s, x)$.*

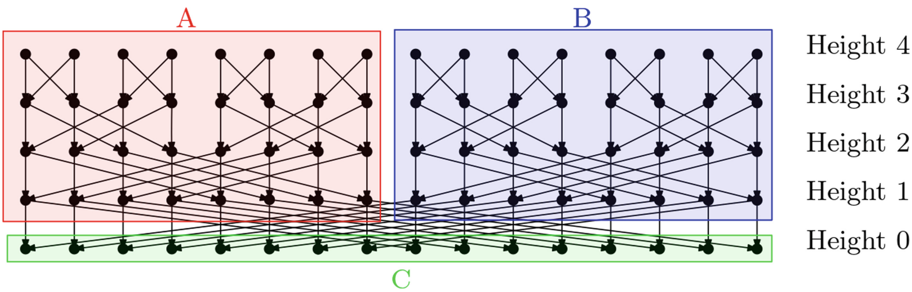

Fig. 1. An example of a butterfly network with height 4 (Depth 4), F^4, as a composition of two F^3 (A and B) and 2^4 additional vertices, C, in Height 0.

As Wang and Honorio observed [50], transitive edges in a directed graph are not learnable by path queries. Thus, it is not possible using path queries to learn all the edges for a number of directed graph types, including strongly connected graphs and DAGs that are not equal to their transitive reductions (i.e., graphs that have at least one transitive edge). Fortunately, transitive edges are not likely in phylogenetic networks due to their derivative nature, so, we focus on learning DAGs without transitive edges.

Definition 6. *In a directed graph, $G = (V, E)$, an edge $(u, v) \in E$ is called a **transitive edge** if there is a directed path from u to v of length greater than* 1.

Definition 7 ***(almost-tree).*** *An almost-tree is a rooted DAG resulting from the union of an arborescence and an additional cross edge. The **height** of an almost-tree is the length of its longest directed path.*

Note: some researchers define almost-trees to have a constant number of cross edges (see, e.g., [8,9]. But allowing more than one cross edge can cause transitive edges; hence, almost-trees with more than one cross edge are not all learnable using path queries, which is why we follow Janardhanan and Reyzin [30] to limit almost-trees to have one cross edge. We next introduce even-separator, which will be used in Sect. 4.

Definition 8 ***(even-separator).*** *Let $G = (V, E)$ be a rooted degree-d DAG. We say that vertex $v \in V$ is an even-separator if $\frac{|V|}{d} \leq count(v, V) \leq \frac{|V|(d-1)}{d}$.*

3 Learning Multitrees

In this section, we begin by presenting a deterministic algorithm to learn a rooted tree (a multitree with a single root) of height h, using $O(nh)$ path queries. This forms the building blocks for the main results of this section, which are an efficient algorithm to learn a multitree of arbitrary height with a number of roots and an efficient algorithm to learn a butterfly network.

Rooted Trees. Let $T = (V, E, r)$ be a directed tree rooted at r with maximum degree that is a constant, d, with vertices, V, and edges, E. At the beginning of any exactly learning algorithm, we only know V, and $n = |V|$, and our goal is to learn r, and E by issuing path queries.

To begin with, learning the root of the tree can be deterministically done using $O(n)$ path queries as suggested by Afshar *et al.* [5, Corollary 10]. Their approach is to first pick an arbitrary vertex v, (ii) learning its ancestor set and establishing a total order on them, and (iii) finally applying a maximum-finding algorithm [15,45,49] by simulating comparisons using path queries.

Next, we show how to learn the edges, E. Jagadish and Anindya [29] propose an algorithm to reconstruct fixed-degree trees of height h using $O(nh \log n)$ queries. Their approach is to find an edge-separator—an edge that splits the tree into two subtrees each having at least n/d vertices—and then to recursively build the two subtrees. In order to find such an edge, (i) they pick an arbitrary vertex, v, and learn an arbitrary neighbor of it such as, u, (ii) if (u, v) is not an edge-separator, they move to the neighboring edge that lies on the direction of maximum vertex set size. Hence, at each step after performing $O(n)$ queries, they get one step closer to the edge-separator. Therefore, they learn the edge-separator using $O(nh)$ queries, and they incur an extra $O(\log n)$ factor to build the tree recursively due to their edge-separator based recursive approach.

We show that finding an edge-separator for a deterministic algorithm is unnecessary, however. We instead propose a vertex-separator based learning algorithm. Our learn-short-tree(V, r) method takes as an input, the vertex set, V, and root vertex, r, and returns edges of the tree, E. Let $\{r_1, \ldots, r_d\}$ be a tentative set of children for vertex r initially set to *Null*, and for $1 \leq i \leq d$, let V_i represents the vertex set of the subtree rooted at r_i. For $1 \leq i \leq d$, we can find child r_i, by starting with an arbitrary vertex r_i, and looping over $v \in V$ to update r_i if for $v \neq r$, path$(v, r_i) = 1$. Since, in a rooted tree, an ancestor relationship for ancestor set of any vertex is a total order, r_i will be a child of root r. Once we learn r_i, its descendants are the set of nodes $v \in V$ such that path$(r_i, v) = 1$. We then remove V_i from the set of vertices of V to learn another child of r in the next iteration. It finally returns the union of edges (r, r_i) and edges returned by the recursive calls learn-short-tree(V_i, r_i), for $1 \leq i \leq d$. The full pseudo-code of function learn-short-tree(V, r) is provided in the full version of the paper [3].

The query complexity, $Q(n)$, for learning the tree is as following:

$$Q(n) = \Sigma_{i=1}^{d} Q(|V_i|) + O(n) \tag{1}$$

Since the height of the tree is reduced by at least 1 for each recursive call, $Q(n) \in O(nh)$. Hence, we have the following theorem.

Theorem 1. *One can deterministically learn a fixed-degree height-h directed rooted tree using $O(nh)$ path queries.*

This provides an improvement upon the results of Jagadish and Anindya [29] (see the full version of this paper [3]).

Multitrees of Arbitrary Height. We next provide a parallel algorithm to learn a multitree of arbitrary height with a number of roots. Remind that Wang and Honorio [50, Theorem 8] prove that learning a multitree with $\Omega(n)$ roots requires $\Omega(n^2)$ queries. Suppose that $G = (V, E)$ is a multitree with a roots. We show that we can learn G using $Q(n) \in O(an \log n)$ queries in $R(n) \in O(a \log n)$ parallel rounds w.h.p.

Let us first explain how to learn a root. Our learn-root method learns a root using $Q(n) \in O(n)$ queries in $R(n) \in O(1)$ rounds w.h.p. Note that in a multitree with more than one root, the ancestor set of an arbitrary vertex does not necessarily form a total order, so, we may not directly apply a parallel maximum finding algorithm on the ancestor set to learn a root.

Our learn-root method takes as input vertex set V, and returns a root of the DAG. It first learns in parallel, Y, the ancestor set of v (the nodes $u \in V$ such that $path(u, v) = 1$). While $|Y| > m$, where $m = C_1 * \sqrt{|V|}$ for some constant C_1 fixed in the analysis, it takes a sample, S, of expected size of m from Y uniformly at random. Then, it performs path queries for every pair $(a, b) \in S \times S$ in parallel to learn a partial order of S, that is, we say $a < b$ if and only if $path(a, b) = 1$. Hence, a root of the DAG should be an ancestor of a minimal element in S. Using this fact, we keep narrowing down Y until $|Y| \leq m$, when we can afford to generate a partial order of Y in Line 7, and return a minimal element of Y (see Algorithm 1).

Algorithm 1: Our algorithm to find a root in V

Function learn-root(V)**:**

1 $m = C_1 * \sqrt{|V|}$ Pick an arbitrary vertex $v \in V$ **for** *each* $u \in V$ **do in parallel**

2 Perform query $path(u, v)$ to find ancestor set Y

3 **while** $|Y| > m$ **do**

4 $S \leftarrow$ a random sample of expected size m from Y **for** $(a, b) \in S \times S$ **do in parallel**

Perform query $path(a, b)$

5 Pick a vertex $y \in S$ such that for all $a \in S$: $path(a, y) == 0$ **for** $a \in Y$ **do in parallel**

Perform query $path(a, y)$ to find ancestors of y, Y'

6 $Y \leftarrow Y'$

7 **for** $(a, b) \in Y \times Y$ **do in parallel**

Perform query $path(a, b)$

8 $y \leftarrow$ a vertex in Y such that for all $a \in Y$: $path(a, y) == 0$ **return** y

Before providing the anlaysis of our efficient learn-root method, let us present Lemma 1, which is an important lemma throughout our analysis, as it extends Afshar *et al.* [5, Lemma 14] to directed acyclic graphs.

Lemma 1. *Let $G = (V, E)$ be a DAG, and let Y be the set of vertices formed by the union of at most c directed (not necessarily disjoint) paths, where $c \leq |V|$ and $|Y| > m = C_1\sqrt{|V|}$. If we take a sample, S, of m elements from Y, then with probability $1 - \frac{1}{|V|^2}$, for each of these c paths such as P, every two consecutive nodes of S in the sorted order of P are within distance $O(|Y| \log |V| / \sqrt{|V|})$ from each other in P.*

Proof. Since we pick our sample S independently and uniformly at random, some nodes of Y may be picked more than once, and each vertex will be picked with probability $p = \frac{m}{|Y|} = \frac{C_1 \cdot \sqrt{|V|}}{|Y|}$. Let P be the set of vertices of an arbitrary path among these c paths. Divide P into consecutive sections of size, $s = \frac{|Y| \log |V|}{\sqrt{|V|}}$. The last section on P can have any length from 1 to $\frac{|Y| \log |V|}{\sqrt{|V|}}$. Let R be the set of vertices of an arbitrary section of path P (any section except the last one). We have that expected size of $|R \cap S|$, $E[|R \cap S|] = s \cdot p = C_1 \log |V|$. Since we pick our sample independently, using standard Chernoff bound for any constant

$C_1 > 8 \ln 2$, we have that $Pr[|R \cap S| = 0] < 1/|V|^4$. Using a union bound, with probability at least $1 - c/|V|^3$, our sample S will pick at least one node from all sections except the last section of all paths. Therefore, if $c \leq |V|$, with probability at least $1 - \frac{1}{|V|^2}$, the distance between any two consecutive nodes on a path in our sample is at most 2 s.

Lemma 2. *Let $G = (V, E)$ be a DAG, and suppose that roots have at most $c \in O(n^{1/2-\epsilon})$ for constant $0 < \epsilon < 1/2$ paths (not necessarily disjoint) in total to vertex v, then,* learn-root(V) *outputs a root with probability at least $1 - \frac{1}{|V|}$, with $Q(n) \in O(n)$ and $R(n) \in O(1)$.*

Proof. The correctness of the learn-root method relies on the fact that if Y is a set of ancestors of vertex v, then for vertex r, a root of the network, and for all $y \in Y$, we have: $path(y, r) = 0$. Using Lemma 1 and a union bound, after at most $1/\epsilon$ iterations of the **While** loop, with probability at least $1 - \frac{1/\epsilon}{|V|^2}$, the size of $|Y|$ will be $O(m)$. Hence, we will be able to find a root using the queries performed in Line 7. Note that this Las Vegas algorithm always returns a root correctly. We can simply derive a Monte Carlo algorithm by replacing the **while** loop with a **for** loop of two iterations.

Therefore, the query complexity of the algorithm is as follows w.h.p:

- We have $O(|V|)$ queries in 1 round to find ancestors of v.
- Then, we have $1/\epsilon$ iterations of the **while** loop, each having $O(m^2)+O(|Y|) \in O(|V|)$ queries in $1/\epsilon$ rounds.
- Finally, we have $O(m^2)$ in 1 round in Line 7.

Overall, this amounts to $Q(n) \in O(n)$, $R(n) \in O(1)$ w.h.p.

Since in a multitree with $a \in O(n^{1/2-\epsilon})$ roots (for $0 < \epsilon < 1/2$), each root has at most one path to a given vertex v, we have at most $a \in O(n^{1/2-\epsilon})$ directed paths in total from roots to an arbitrary vertex v. Therefore, we can apply Lemma 2 to learn a root w.h.p. Note that if $a \notin O(n^{1/2-\epsilon})$, as an alternative, we can learn a root w.h.p. using $O(n \log n)$ queries with $R(n) \in O(\log n)$ rounds by (i) picking an arbitrary vertex $v \in V$ and learning its ancestors, $A(v) \cap V$ in parallel (ii) replacing path queries with inverse-path queries (inverse-path$(u, v) =$ 1 if and only if v has a directed path to u), (ii) and applying the rooted tree learning method by Afshar *et al.* [5, Algorithm 2] to learn the tree with inverse direction to v. Note that any of the leaves of the inverse tree rooted at v is a root of the multitree.

Our multitree learning algorithm works by repetitively learning a root, r, from the set of candidate roots, R ($R = V$ at the beginning). Then, it learns a tree rooted at R by calling the rooted tree learning method by Afshar *et al.* [5, Algorithm 2]. Finally, it removes the set of vertices of the tree from R to perform another iteration of the algorithm so long as $|R| > 0$. We give the details of the algorithm below.

1. Let R be the set of candidate roots for the multitree initialized with V.
2. Let $r \leftarrow \textsf{learn-root}(R)$.
3. Issue queries in parallel, $path(r, v)$ for all $v \in V$ to learn descendants, $D(r)$.
4. Learn the tree rooted at r by calling $\textsf{learn-rooted-tree}(r, D(r))$.
5. Let $R = R \setminus D(r)$, and if $|R| > 0$ go to step 2..

Theorem 2 analyzes the complexity of our multitree learning algorithm.

Theorem 2. *One can learn a multitree with a roots using* $Q(n) \in O(an \log n)$ *path queries in* $R(n) \in O(a \log n)$ *parallel rounds w.h.p.*

Proof. The query complexity and the round complexity of our multitree learning method is dominated by the calls to the learn-rooted-tree by Afshar *et al.* [5, Algorithm 2] which takes $Q(n) \in O(n \log n)$ queries in $R(n) \in O(\log n)$ parallel rounds w.h.p. Hence, using a union bound and by adjusting the sampling constants for learn-rooted-tree by Afshar *et al.* [5, Algorithm 2] we can establish the high probability bounds.

Butterfly Networks. Next, we provide an algorithm to learn a butterfly network. Suppose that $F^h = (V, E)$ is a butterfly network with height h (i.e., a depth-h FFT graph, see definition 3). We show that we can learn F^h using $Q(n) \in O(2^{3h/2}h^2)$ path queries with high probability. Note that in a butterfly networks of height h, the number of nodes will be $n = 2^h \cdot (h + 1)$. Also, note that the graph has a symmetry property, that is, all leaves are reachable from the root, and all roots are reachable from the leaves if we reverse the directions of the edges, and that each node but the leaves has exactly two children, and each node but the roots have exactly two parents, and so on. Due to this symmetry property, we can apply learn-short-tree but with inverse path query ($\textsf{inverse-path}(u, v) = 1$ if and only if v has a directed path to u) to find the tree with inverse direction to a leaf.

Our algorithm first learns all the roots and all the leaves of the graph. We first perform a sequential search to find an arbitrary root of the network, r. Note that we can learn r by picking an arbitrary vertex x and looping over all the vertices and updating x to y if $\textsf{path}(y, x) = 1$. After learning its descendants, $D(r)$, we make a call to our learn-short-tree method to build the tree rooted at r, which enables us to learn all the leaves, L. Then, we pick an arbitrary leaf, $l \in L$, and after learning its ancestors, $A(l)$, we call the learn-short-tree method (with inverse path query) to learn the tree with inverse direction to l, which enables us to learn all the roots, R. We then take two sample subsets, S, and T, of expected size $O(2^{h/2}h)$ from R, and L respectively, and uniformly at random. We will show that the union of the edges of trees rooted at r for all $r \in S$ and the inverse trees rooted at l for all $l \in T$ includes all the edges of the network w.h.p. We give the details of our algorithm below.

1. Learn a root, r, using a sequential search.
2. Perform path queries to learn descendant set, $D(r)$, of r.

3. Call learn-short-tree$(r, D(r))$ method to learn the leaves of the network, L.
4. Let $l \in L$ be an arbitrary leaf in the network, then perform path queries to learn the ancestors of l, $A(l)$.
5. Call learn-short-tree$(l, A(l))$ with inverse path query definition to learn the roots of the network, R.
6. Pick a sample S of size $c \cdot 2^{h/2}h$ from R, and a sample T of size $c \cdot 2^{h/2}h$ from L uniformly at random for a constant $c > 0$.
7. Perform queries to learn descendant set, $D(s)$, for every $s \in S$, and to learn ancestor set $A(t)$, for every $t \in T$.
8. Call learn-short-tree$(s, D(s))$ to learn the tree rooted at s for all $s \in S$.
9. Call learn-short-tree$(t, A(t))$ using inverse reverse path query to learn the tree rooted at t for all $t \in T$.
10. Return the union of all the edges learned.

Theorem 3. *One can learn a butterfly network of height, h, using $Q(n) \in O(2^{3h/2}h^2)$ path queries with high probability.*

Proof. The query complexity of the algorithm is dominated by $O(2^{h/2}h)$ times the running time of our learn-short-tree method, which takes $O(2^h h)$ queries for each tree. Consider a directed edge from vertex x at height k to vertex y at height $k-1$ in the network. If $k \leq h/2$, then x has at least $2^{\lfloor h/2 \rfloor}$ ancestors in the root, that is, $|A(x) \cap R| \geq 2^{\lfloor h/2 \rfloor}$. Since our sample, S, has an expected size of $2^{h/2} \cdot ch$, the expected size of $|S \cap A(x) \cap R| \geq ch/2$. Using a standard Chernoff bound, the probability, $Pr[|S \cap A(x) \cap R| = 0] \leq e^{-ch/4}$. Hence, for large enough c, this probability is less than $1/2^{2h}$. Therefore, we will be able to learn edge (x, y) through a tree rooted at $s \in S$. Similarly, we can show that if $k > h/2$, then y has at least $2^{\lfloor h/2 \rfloor}$ descendants in the leaves, that is, $|D(y) \cap L| \geq 2^{\lfloor h/2 \rfloor}$. Since, our sample T, has an expected size of $2^{h/2} \cdot ch$, the expected size of $|T \cap D(y) \cap L| \geq ch/2$. Using a standard Chernoff bound, the probability, $Pr[|T \cap D(y) \cap L| = 0] \leq e^{-ch/4}$. Hence, for large enough c, this probability is less than $1/2^{2h}$. Therefore, we will be able to learn edge (x, y) through a tree inversely rooted at $t \in T$ in this case. A union bound establishes the high probability.

4 Parallel Learning of Almost-Trees

Let $G = (V, E)$ be an almost-tree of height h. We learn G with $Q(n) \in O(n \log n + nh)$ path queries in $R(n) \in O(\log n)$ rounds w.h.p. Note that we can learn the root of an almost-tree by Algorithm 1, and given that the root has at most 2 paths to any vertex, it will take $Q(n) \in O(n)$ queries and $R(n) \in O(1)$ w.h.p. by Lemma 2. We then learn a spanning rooted tree for it, and finally we learn the cross-edge. We will also prove that our algorithm is optimal by showing that any randomized algorithm needs an expected number of $\Omega(n \log n + nh)$.

Learning an Arborescence in a DAG. Our parallel algorithm learns an arborescence, a spanning directed rooted tree, of the graph with a divide and conquer approach based on our separator theorem, which is an extension of Afshar *et al.* [5, Lemma 5] for DAGs.

Theorem 4. *Every degree-d rooted DAG, $G = (V, E)$, has an even-separator (see Definition 8).*

Proof. We prove through a iterative process that there exists a vertex v such that $\frac{|V|}{d} \leq |D(v)| \leq \frac{|V|\cdot(d-1)}{d}$. Let r be the root of the DAG. We have that $|D(r)| = |V|$. Since r has at most d children and each $v \in V$ is a descendent of at least one of the children of r, r has a child x, such that $D(x) \geq |V|/d$. If $D(x) \leq \frac{|V|\cdot(d-1)}{d}$, x is an even-separator. Otherwise, since $d_o(x) \leq d-1$, x has a child, y, such that $|D(y)| \geq |V|/d$. If $|D(y)| \leq \frac{|V|\cdot(d-1)}{d}$, y is an even-separator. Otherwise, we can repeat this iterative procedure with a child of y having maximum number of descendants. Since, $|D(y)| < |D(x)|$, and a directed path in a DAG ends at vertices of out-degree 0 (with no descendants), this iterative procedure will return an even-separator at some point.

Next, we introduce Lemma 3 which shows that for fixed-degree rooted DAGs, if we pick a vertex v uniformly at random, there is an even separator in $A(v)$, ancestor set of v, with probability depending on d.

Lemma 3. *Let $G = (V, E)$ be a degree-d DAG with root r, and let v be a vertex chosen uniformly at random from v. Let Y be the ancestor set for v in V. Then, with probability at least $\frac{1}{d}$, there is an even-separator in Y.*

Proof. By Theorem 4, G has an even-separator, e. Since $|D(e)| \geq \frac{|V|}{d}$, with probability at least $\frac{1}{d}$, v will be one of the descendants of e.

Although a degree-d rooted DAG has an even-separator, checking if a vertex is an even-separator requires a lot of queries for exact calculation of the number of descendants. Thus, we use a more relaxed version of the separator, which we call ***near-separator***, for our divide and conquer algorithm.

Definition 9. *Let $G = (V, E)$ be a rooted degree-d DAG. We say that vertex $v \in V$ is a near-separator if $\frac{|V|}{d+2} \leq |D(v)| \leq \frac{|V|(d+1)}{d+2}$.*

Note that every even-separator is also a near-separator. We show if an even-separator exists among $A(v)$ for an arbitrary vertex v, then we can locate a near-separator among $A(v)$ w.h.p. Incidentally, Afshar *et al.* [5] used a similar divide and conquer approach to learn directed rooted trees, but their approach relied on the fact that there is exactly one path from root to every vertex of the tree. We will show how to meet the challenge of having multiple paths to a vertex from the root in learning an arborescence for a rooted DAG.

Our learn-spanning-tree method takes as input vertex set, V, of a DAG rooted at r, and returns the edges, E, of an arborescence of it. In particular, it enters a repeating **while** loop to learn a near-separator by (i) picking a random vertex $v \in V$, (ii) learning its ancestors, $Y = A(v) \cap V$, (iii) and checking if Y has a near-separator, w, by calling learn-separator method, which we describe next. Once learn-separator returns a vertex, w, we split V into $V_1 = D(w) \cap V$ and $V_2 = V \setminus V_1$ given that $path(w, z) = 1$ if and only if $z \in V_1$. If $\frac{|V|}{d} \le |V_1| \le \frac{|V|(d-1)}{d}$, we verify w is a near-separator. If w is a near separator, then it calls learn-parent method, to learn a parent, u, for w. Finally, it makes two recursive calls to learn a spanning tree rooted at w for vertex set V_1, and a spanning tree rooted at r with vertex set V_2 (see full version of the paper [3] for a full pseudo-code of the algorithm). Note that our learn-parent(v, V) method is similar to our learn-root(V) method except that it passes closest nodes to v to the next iteration rather than the farthest nodes (please refer to the full version of the paper [3] for details).

Next, we show how to adapt an algorithm to learn a near-separator for DAGs by extending the work of Afshar *et al.*[5, Algorithm 3]. Our learn-separator method takes as input vertex v, its ancestors, Y, vertex set V of a DAG rooted at r, and returns w.h.p. a near-separator among vertices of Y provided that there is an even-separator in Y. If $|Y|$ is too large ($|Y| > |V|/K$), then it enters **Phase 1**. The goal of this phase is to remove the nodes that are unlikely to be a separator in order to pass a smaller set of candidate separator to **Phase 2**. It chooses a random sample, S, of expected size $m = C_1\sqrt{|V|}$, where $C_1 > 0$ is a fixed constant. It adds $\{v, r\}$ to the sample S. It then estimates $|D(s) \cap V|$ for each $s \in S$, using a random sample, X_s, of size $K = O(\log |V|)$ from V by issuing path queries. If all of the estimates, $count(s, X_s)$, are smaller than $\frac{K}{d+1}$, we return $Null$, as we argue that in this case the nodes in Y do not have enough descendants to act as a separator. Similarly, If all of the estimates, are greater than $\frac{Kd}{d+1}$, we return $Null$, as we show that in this case the nodes in Y have too many descendants to act as a separator. If one of these estimates for a vertex s lies in the range of $[\frac{K}{d+1}, \frac{Kd}{d+1}]$, we return it as a near-separator. Otherwise, we filter the set of Y by removing the nodes that are unlikely to be a separator through a call to filter-separator method, which we present next. Then, we enter **Phase 2**, where for every $s \in Y$, we take a random sample X_s of expected size of $O(log|V|)$ from V to estimate $|D(s) \cap V|$. If one of these estimates for a vertex s lies in the range of $[\frac{K}{d+1}, \frac{Kd}{d+1}]$, we return it as a near-separator. We will show later that the output is a near-separator w.h.p (please refer to the full version of the paper [3] for a pseudo-code description of learn-separator method).

Next, let us explain our filter-separator method, whose purpose is to remove some of the vertices in Y that are unlikely to be a separator to shrink the size of Y. We first establish a partial order on elements of S by issuing path queries in parallel. Since there are at most $c = 2$ directed paths from root to vertex v, for path $1 \le i \le c$, let $l_i \in S$ be the oldest node on path i having $count(l_i, X_{l_i}) < \frac{K}{d+1}$ (resp. $g_i \in S$ be the youngest node on path i having $count(g_i, X_{g_i}) > \frac{Kd}{d+1}$). We then perform queries to remove ancestors of g_i, and descendants of l_i from Y. We will prove later that this filter reduces $|Y|$ considerably without filtering an even-separator. We will give the details of this method in Algorithm 2.

Lemma 4 shows that our filter-separator method efficiently in parallel eliminates the nodes that are unlikely to act as a separator.

Lemma 4. *Let $G = (V, E)$ be a DAG rooted at r, with at most c directed (not necessarily disjoint) paths from r to vertex v, and let $Y = A(v) \cap V$, and let*

Algorithm 2: Filter out the vertices unlikely to be a separator

Function filter-separator(S, Y, V)**:**

1 **for** *each* $\{a, b\} \in S$ **do in parallel**

2 perform query $path(a, b)$

3 Let $P_1, P_2, \ldots, P_c$ be the c paths from r to v. For $1 \leq i \leq c$: let $l_i \in (S \cap P_i)$ such that $count(l_i, X_{l_i}) < \frac{K}{d+1}$, and there exists no $b \in (S \cap A(l_i))$ where $count(b, X_b) < \frac{K}{d+1}$. For $1 \leq i \leq c$: let $g_i \in (S \cap P_i)$ such that $count(g_i, X_{g_i}) > \frac{K \cdot d}{d+1}$, and there exists no $b \in (S \cap D(g_i))$ where $count(b, X_b) > \frac{K \cdot d}{d+1}$. **for** $1 \leq i \leq c$ *and* $v \in V$ **do in parallel**

4 perform query $path(v, g_i)$ to find $(A(g_i) \cap V)$. Remove $(A(g_i) \cap V)$ from Y. perform query $path(l_i, v)$ to find $(D(l_i) \cap V)$. Remove $(D(l_i) \cap V)$ from Y.

5 **return** Y

S be a random sample of expected size m that includes v, and r as well. The call to filter-separator(S, Y, V) *in our* learn-separator *method returns a set of size $O(c \cdot |Y| \log |V| / \sqrt{|V|})$, and If Y has an even-separator, the returned set includes an even-separator with probability at least $1 - \frac{|S|+1}{|V|^2}$.*

Proof. The proof idea is to first employ Lemma 1 to show that with very high probability the size of the returned set is at most $c \cdot O(|Y| \log |V| / \sqrt{|V|})$. Then, it follows by arguing that if e is an even-separator it is unlikely for e to be an ancestor of g_i or a descendant of d_i in Lines 4, 4 of filter-separator method. Please refer to the full version of the paper [3] for details.

Lemma 5 establishes the fact that our learn-separator finds w.h.p. a near-separator among ancestors $A(v) \cap V$, if there is an even-separator in $A(v) \cap V$.

Lemma 5. *Let $G = (V, E)$ be a DAG rooted at r, with at most c directed (not necessarily disjoint) paths from r to vertex v, and let $Y = A(v) \cap V$. If Y has an even-separator, then our* learn-separator *method returns a near-separator w.h.p.*

Proof. See full version of the paper [3].

Lemma 6. *Let $G = (V, E)$ be a DAG rooted at r, with at most c directed (not necessarily disjoint) paths from r to vertex v. Then, our* learn-separator(v, Y, V, r) *method, takes $Q(n) \in O(c|V|)$ queries in $R(n) \in O(1)$ rounds.*

Proof.
- In **phase 1**, it takes $O(mK) \in o(|V|)$ queries in 1 round to estimate the number of descendants for sample S.
- The call to filter-separator in **phase 1** takes m^2 queries in one round to derive a partial order for S, and since there are at most c paths from r to v, it takes $O(c \cdot |V|)$ in one round to remove nodes from Y.
- In **Phase 2**, it takes $O(|Y|K) \in O(|V|)$ queries in 1 round to estimate the number of descendants for all nodes of Y.

Algorithm 3: lean a cross-edge for an almost tree

Function learn-cross-edge(V, E):
1 **for** $v \in V$ **do**
2 **for** $c \in C(v)$ **do**
3 **for** $t \in (D(V) \setminus D(c))$ **do in parallel**
4 Perform query $path(c, t)$
5 Let c be the only node and let t be the node with maximum height having $path(c, t) = 1$ **for** $s \in D(c)$ **do in parallel**
6 Perform query $path(s, t)$
7 Let s be the node with minimum height having $path(s, t) = 1$. **return** (s, t)

Theorem 5. *Suppose $G = (V, E)$ is a rooted DAG with $|V| = n$, and maximum constant degree, d, with at most constant, c directed (not necessarily disjoint) paths from root, r, to each vertex. Our* learn-spanning-tree *algorithm learns an arborescence of G using $Q(n) \in O(n \log n)$ and $R(n) \in O(\log n)$ w.h.p.*

Proof. See full version of the paper [3].

Learning a Cross-Edge. Next, we will show that a cross-edge can be learnt using $O(nh)$ queries in just 2 parallel rounds for an almost-tree of height h. Our learn-cross-edge algorithm takes as input vertices V and edges E of an arborescence of a almost-tree, and returns the cross-edge from the source vertex, s, to the destination vertex, t. In this algorithm, we refer to $D(v)$ for a vertex v as the set of descendants of v according to E (the only edges learned by the arborescence). We will show later that there exists a vertex, c, whose parent is vertex, v, such that the cross-edge has to be from a source vertex $s \in D(c)$ to a destination vertex $t \in (D(v) \setminus D(c))$. In particular, this algorithm first learns t and c with $O(nh)$ queries in 1 parallel round. Note that $t \in (D(v) \setminus D(c))$ is a node with maximum height having $path(c, t) = 1$. Once it learns t and c, then it learns source s, where $s \in D(c)$ is the node with minimum height satisfying $path(s, t) = 1$, using $O(n)$ queries in 1 round. We give the details in Algorithm 3.

The following lemma shows that Algorithm 3 correctly learns the cross-edge using $O(nh)$ queries in just 2 rounds.

Lemma 7. *Given an arborescence with vertex set V, and edge set, E, of an almost-tree, Algorithm 3 learns the cross-edge using $O(nh)$ queries in 2 rounds.*

Proof. Suppose that the cross-edge is from a vertex s to to a vertex t. Let v be the least common ancestor of s and t in the arborescence, and let c be a child of v on the path from v to s. Since $t \in (D(v) \setminus D(c))$, we have that $path(c, t) = 1$ in Line 4. Note that since there is only one cross-edge, there will be exactly one node such as c satisfying $path(c, t) = 1$. Note that in Line 4 we can also learn t, which is the node with maximum height satisfying $path(c, t) = 1$. Finally, we just do a parallel search in the descendant set of c to learn s in Line 6.

We charge each $path(c, t)$ query in Line 4 to the vertex v. Since each vertex has at most d children the number of queries associated with vertex v will be at most $O(|D(v)| \cdot d)$. Hence, using a double counting argument and the fact that each vertex is a descendant of $O(h)$ vertices, the sum of the queries performed Line 4 will be, $\Sigma_{v \in V} O(|D(v)| \cdot d) = O(nh)$. Finally, we need $O(n)$ queries 1 round to learn s in Line 6.

Theorem 6. *Given vertices, V, of an almost-tree, we can learn root, r, and the edges, E, using $Q(n) \in O(n \log n + nh)$ path queries, and $R(n) \in O(\log n)$ w.h.p.*

Proof. Note that in almost-trees there are at most $c = 2$ paths from root r to each vertex. Therefore, by Lemma 2, we can learn root of the graph using $O(n)$ queries in $O(1)$ rounds with probability at least $1 - \frac{1}{|V|}$. Then, by Theorem 5, we can learn a spanning tree of the graph using $O(n \log n)$ queries in $O(\log n)$ rounds with probability at least $1 - \frac{1}{|V|}$. Finally, by Lemma 7 we can deterministically learn a cross-edge using $O(nh)$ queries in just 2 rounds.

Lower Bound. The following lower bound improves the one by Janardhanan and Reyzin [30] and proves that our algorithm to learn almost-trees in optimal.

Theorem 7. *Let G be a a degree-d almost-tree of height h with n vertices. Learning G takes $\Omega(n \log n + nh)$ queries. This lower bound holds for both worst case of a deterministic algorithm and for an expected cost of a randomized algorithm.*

Proof. We use the same graph as the one used by Janardhanan and Reyzin [30], but we improve their bound using an information-theoretic argument. Consider a caterpillar graph with height h, and a complete d-ary tree with $\Omega(n)$ leaves attached to the last level of it. If there is a cross-edge from one of the leaves of the caterpillar to one of the leaves of the d-ary tree, it takes $\Omega(nh)$ queries involving a leaf of the caterpillar and a leaf of the d-ary tree. Suppose that a querier, Bob, knows the internal nodes of the d-ary, and he wants to know that for each leaf l of the d-ary, what is the parent of l in the d-ary tree. If there are m leaves for the caterpillar, the number of possible d-ary trees will be at least $\frac{m!}{(d!)^{m/d}}$. Therofore, using an information-theoretic lower bound, we need $\Omega\left(\log\left(\frac{m!}{(d!)^{m/d}}\right)\right)$ bit of information to be able to learn the parent of the leaves of d-ary tree. Since the queries involving a leaf of the caterpillar and a leaf of the d-ary tree do not provide any information about how the d-ary tree is built, it takes $\Omega(n \log n)$ queries to learn the d-ary tree.

References

1. Abrahamsen, M., Bodwin, G., Rotenberg, E., Stöckel, M.: Graph reconstruction with a betweenness oracle. In: Ollinger, N., Vollmer, H. (eds.) 33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016, 17–20 February 2016, Orléans, France. LIPIcs, vol. 47, pp. 5:1–5:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.STACS.2016.5
2. Afshani, P., Agrawal, M., Doerr, B., Doerr, C., Larsen, K.G., Mehlhorn, K.: The query complexity of finding a hidden permutation. In: Brodnik, A., López-Ortiz, A., Raman, V., Viola, A. (eds.) Space-Efficient Data Structures, Streams, and Algorithms. LNCS, vol. 8066, pp. 1–11. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40273-9_1

3. Afshar, R., Goodrich, M.T.: Exact learning of multitrees and almost-trees using path queries. 10.48550/ARXIV.2208.04216, https://arxiv.org/abs/2208.04216
4. Afshar, R., Goodrich, M.T., Matias, P., Osegueda, M.C.: Reconstructing binary trees in parallel. In: Scheideler, C., Spear, M. (eds.) SPAA 2020: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 15–17 July 2020, pp. 491–492. ACM (2020). https://doi.org/10.1145/3350755.3400229
5. Afshar, R., Goodrich, M.T., Matias, P., Osegueda, M.C.: Reconstructing biological and digital phylogenetic trees in parallel. In: Grandoni, F., Herman, G., Sanders, P. (eds.) 28th Annual European Symposium on Algorithms, ESA 2020, 7–9 September 2020, Pisa, Italy (Virtual Conference). LIPIcs, vol. 173, pp. 3:1–3:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.ESA.2020.3
6. Afshar, R., Goodrich, M.T., Matias, P., Osegueda, M.C.: Parallel network mapping algorithms. In: Agrawal, K., Azar, Y. (eds.) SPAA 2021: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6–8 July 2021, pp. 410–413. ACM (2021). https://doi.org/10.1145/3409964.3461822
7. Afshar, R., Goodrich, M.T., Matias, P., Osegueda, M.C.: Mapping networks via parallel kth-hop traceroute queries. In: Berenbrink, P., Monmege, B. (eds.) 39th International Symposium on Theoretical Aspects of Computer Science, STACS 2022, 15–18 March 2022, Marseille, France (Virtual Conference). LIPIcs, vol. 219, pp. 4:1–4:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.STACS.2022.4
8. Akutsu, T.: A polynomial time algorithm for finding a largest common subgraph of almost trees of bounded degree. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **76**(9), 1488–1493 (1993)
9. Bannister, M.J., Eppstein, D., Simons, J.A.: Fixed parameter tractability of crossing minimization of almost-trees. In: Wismath, S., Wolff, A. (eds.) GD 2013. LNCS, vol. 8242, pp. 340–351. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03841-4_30
10. Barton, N.H.: The role of hybridization in evolution. Mol. Ecol. **10**(3), 551–568 (2001)
11. Bello, K., Honorio, J.: Computationally and statistically efficient learning of causal Bayes nets using path queries. In: Bengio, S., Wallach, H.M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018(December), pp. 3–8, 2018. Montréal, Canada, pp. 10954–10964 (2018). https://proceedings.neurips.cc/paper/2018/hash/a0b45d1bb84fe1bedbb8449764c4d5d5-Abstract.html
12. Bernasconi, A., Damm, C., Shparlinski, I.E.: Circuit and decision tree complexity of some number theoretic problems. Inf. Comput. **168**(2), 113–124 (2001). https://doi.org/10.1006/inco.2000.3017
13. Bestagini, P., Tagliasacchi, M., Tubaro, S.: Image phylogeny tree reconstruction based on region selection. In: 2016 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2016, Shanghai, China, 20–25 March 2016, pp. 2059–2063. IEEE (2016). https://doi.org/10.1109/ICASSP.2016.7472039
14. Choi, S., Kim, J.H.: Optimal query complexity bounds for finding graphs. Artif. Intell. **174**(9–10), 551–569 (2010). https://doi.org/10.1016/j.artint.2010.02.003

15. Cole, R., Vishkin, U.: Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In: Hartmanis, J. (ed.) Proceedings of the 18th Annual ACM Symposium on Theory of Computing, 28–30 May 1986, Berkeley, California, USA, pp. 206–219. ACM (1986). https://doi.org/10.1145/12130.12151
16. Colombo, C., Lepage, F., Kopp, R., Gnaedinger, E.: Two SDN multi-tree approaches for constrained seamless multicast. In: Pop, F., Negru, C., González-Vélez, H., Rak, J. (eds.) 2018 IEEE International Conference on Computational Science and Engineering, CSE 2018, Bucharest, Romania, 29–31 October 2018, pp. 77–84. IEEE Computer Society (2018). https://doi.org/10.1109/CSE.2018.00017
17. Comellas, F., Fiol, M.A., Gimbert, J., Mitjana, M.: The spectra of wrapped butterfly digraphs. Networks **42**(1), 15–19 (2003). https://doi.org/10.1002/net.10085
18. Dias, Z., Goldenstein, S., Rocha, A.: Exploring heuristic and optimum branching algorithms for image phylogeny. J. Vis. Commun. Image Represent. **24**(7), 1124–1134 (2013). https://doi.org/10.1016/j.jvcir.2013.07.011
19. Dias, Z., Goldenstein, S., Rocha, A.: Large-scale image phylogeny: tracing image ancestral relationships. IEEE Multim. **20**(3), 58–70 (2013). https://doi.org/10.1109/MMUL.2013.17
20. Dias, Z., Rocha, A., Goldenstein, S.: Image phylogeny by minimal spanning trees. IEEE Trans. Inf. Forensics Secur. **7**(2), 774–788 (2012). https://doi.org/10.1109/TIFS.2011.2169959
21. Dobzinski, S., Vondrák, J.: From query complexity to computational complexity. In: Karloff, H.J., Pitassi, T. (eds.) Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, 19–22 May 2012, pp. 1107–1116. ACM (2012). https://doi.org/10.1145/2213977.2214076
22. Goldberg, L.A., Goldberg, P.W., Phillips, C.A., Sorkin, G.B.: Constructing computer virus phylogenies. J. Algorithms **26**(1), 188–208 (1998). https://doi.org/10.1006/jagm.1997.0897
23. Goodrich, M.T., Jacob, R., Sitchinava, N.: Atomic power in forks: a super-logarithmic lower bound for implementing butterfly networks in the nonatomic binary fork-join model. In: Marx, D. (ed.) Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, 10–13 January 2021, pp. 2141–2153. SIAM (2021). https://doi.org/10.1137/1.9781611976465.128
24. Heckerman, D., Meek, C., Cooper, G.: A Bayesian approach to causal discovery. In: Holmes, D.E., Jain, L.C. (eds.) Innovations in Machine Learning, vol. 194, pp. 1–28. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-33486-6_1
25. Hein, J.J.: An optimal algorithm to reconstruct trees from additive distance data. Bull. Math. Biol. **51**(5), 597–603 (1989)
26. Hünermund, P., Bareinboim, E.: Causal inference and data fusion in econometrics. arXiv preprint arXiv:1912.09104 (2019)
27. Imbens, G.W.: Potential outcome and directed acyclic graph approaches to causality: relevance for empirical practice in economics. J. Econ. Lit. **58**(4), 1129–79 (2020)
28. Itai, A., Rodeh, M.: The multi-tree approach to reliability in distributed networks. Inf. Comput. **79**(1), 43–59 (1988). https://doi.org/10.1016/0890-5401(88)90016-8
29. Jagadish, M., Sen, A.: Learning a bounded-degree tree using separator queries. In: Jain, S., Munos, R., Stephan, F., Zeugmann, T. (eds.) ALT 2013. LNCS (LNAI), vol. 8139, pp. 188–202. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40935-6_14

30. Janardhanan, M.V., Reyzin, L.: On learning a hidden directed graph with path queries. CoRR abs/2002.11541 (2020). https://arxiv.org/abs/2002.11541
31. Ji, J.H., Park, S.H., Woo, G., Cho, H.G.: Generating pylogenetic tree of homogeneous source code in a plagiarism detection system. Int. J. Control Autom. Syst. **6**(6), 809–817 (2008)
32. King, V., Zhang, L., Zhou, Y.: On the complexity of distance-based evolutionary tree reconstruction. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 12–14 January 2003, Baltimore, Maryland, USA, pp. 444–453. ACM/SIAM (2003). http://dl.acm.org/citation.cfm?id=644108.644179
33. Kocaoglu, M., Shanmugam, K., Bareinboim, E.: Experimental design for learning causal graphs with latent variables. In: Guyon, I., von Luxburg, U., Bengio, S., Wallach, H.M., Fergus, R., Vishwanathan, S.V.N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017(December), pp. 4–9, 2017. Long Beach, CA, USA, pp. 7018–7028 (2017), https://proceedings.neurips.cc/paper/2017/hash/291d43c696d8c3704cdbe0a72ade5f6c-Abstract.html
34. Lagani, V., Triantafillou, S., Ball, G., Tegnér, J., Tsamardinos, I.: Probabilistic computational causal discovery for systems biology. In: Geris, L., Gomez-Cabrero, D. (eds.) Uncertainty in Biology. SMTEB, vol. 17, pp. 33–73. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-21296-8_3
35. Marmerola, G.D., Oikawa, M.A., Dias, Z., Goldenstein, S., Rocha, A.: On the reconstruction of text phylogeny trees: evaluation and analysis of textual relationships. PLoS ONE **11**(12), e0167822 (2016)
36. Mathieu, C., Zhou, H.: A simple algorithm for graph reconstruction. In: Mutzel, P., Pagh, R., Herman, G. (eds.) 29th Annual European Symposium on Algorithms, ESA 2021, 6–8 September 2021, Lisbon, Portugal (Virtual Conference). LIPIcs, vol. 204, pp. 68:1–68:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ESA.2021.68
37. Meinshausen, N., Hauser, A., Mooij, J.M., Peters, J., Versteeg, P., Bühlmann, P.: Methods for causal inference from gene perturbation experiments and validation. Proc. Natl. Acad. Sci. **113**(27), 7361–7368 (2016)
38. Moffa, G.: Using directed acyclic graphs in epidemiological research in psychosis: an analysis of the role of bullying in psychosis. Schizophr. Bull. **43**(6), 1273–1279 (2017)
39. Pfeffer, A., et al.: Malware analysis and attribution using genetic information. In: 2012 7th International Conference on Malicious and Unwanted Software, pp. 39–45. IEEE (2012)
40. Ranade, A.G.: Optimal speedup for backtrack search on a butterfly network. In: Leighton, T. (ed.) Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '91, Hilton Head, South Carolina, USA, 21–24 July 1991, pp. 40–48. ACM (1991). https://doi.org/10.1145/113379.113383
41. Reyzin, L., Srivastava, N.: On the longest path algorithm for reconstructing trees from distance matrices. Inf. Process. Lett. **101**(3), 98–100 (2007). https://doi.org/10.1016/j.ipl.2006.08.013
42. Rong, G., Li, W., Yang, Y., Wang, J.: Reconstruction and verification of chordal graphs with a distance oracle. Theor. Comput. Sci. **859**, 48–56 (2021). https://doi.org/10.1016/j.tcs.2021.01.006
43. Rong, G., Yang, Y., Li, W., Wang, J.: A divide-and-conquer approach for reconstruction of $\{c_{\geq 5}\}$-free graphs via betweenness queries. Theor. Comput. Sci. **917**, 1–11 (2022). https://doi.org/10.1016/j.tcs.2022.03.008

44. Shen, B., Forstall, C.W., de Rezende Rocha, A., Scheirer, W.J.: Practical text phylogeny for real-world settings. IEEE Access **6**, 41002–41012 (2018). https://doi.org/10.1109/ACCESS.2018.2856865
45. Shiloach, Y., Vishkin, U.: Finding the maximum, merging and sorting in a parallel computation model. In: Brauer, W., et al. (eds.) CONPAR 1981. LNCS, vol. 111, pp. 314–327. Springer, Heidelberg (1981). https://doi.org/10.1007/BFb0105127
46. Tardos, G.: Query complexity, or why is it difficult to seperate NP ^{a} cap co np^{a} from p^{a} by random oracles a? Comb. **9**(4), 385–392 (1989). https://doi.org/10.1007/BF02125350
47. Tennant, P.W., et al.: Use of directed acyclic graphs (DAGS) to identify confounders in applied health research: review and recommendations. Int. J. Epidemiol. **50**(2), 620–632 (2021)
48. Triantafillou, S., Lagani, V., Heinze-Deml, C., Schmidt, A., Tegner, J., Tsamardinos, I.: Predicting causal relationships from biological data: applying automated causal discovery on mass cytometry data of human immune cells. Sci. Rep. **7**(1), 1–11 (2017)
49. Valiant, L.G.: Parallelism in comparison problems. SIAM J. Comput. **4**(3), 348–355 (1975). https://doi.org/10.1137/0204030
50. Wang, Z., Honorio, J.: Reconstructing a bounded-degree directed tree using path queries. In: 57th Annual Allerton Conference on Communication, Control, and Computing, Allerton 2019, Monticello, IL, USA, 24–27 September 2019, pp. 506–513. IEEE (2019). https://doi.org/10.1109/ALLERTON.2019.8919924
51. Yao, A.C.: Decision tree complexity and betti numbers. J. Comput. Syst. Sci. **55**(1), 36–43 (1997). https://doi.org/10.1006/jcss.1997.1495