UNIVERSITY OF CALIFORNIA,
IRVINE

Exact Learning of Graphs Using Queries

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Ramtin Afshar

2023

DEDICATION

To my parents Elham and Ali, to my wife Parinaz, and to my sister Melika, for their unconditional support throughout my journey.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to start by thanking my advisor, Michael Goodrich, for conceding me this journey and for his continuous support. I also want to thank him for the independence to pursue the research directions that interested me the most despite all the ups and downs. During periods of uncertainty, he led by example showing me how to do things better and faster helping me to have little progress each and every single day. I thank David Eppstein for his helpful feedback on my projects during theory seminars and for serving in my defense and advancement committees. I thank Sandy Irani for serving in my defense and advancement committees. I also thank Milena Mihail and Carter T. Butts for serving in my advancement committee. I am thankful to Vijay Vazirani, without him my journey probably would have been a different one, as he reviewed my Ph.D. application and he connected me with Michael Goodrich. I also want to thank Michael Dillencourt with whom I had the pleasure of being a TA most of the times, for his understanding about my research commitments.

I want to thank all my co-authors for their collaboration during my doctoral studies. In particular, I thank Ami Amir, Michael Dillencourt, Michael Goodrich, Pedro Matias, Martha Osegueda, and Evrim Ozel. I am thankful that I received funding support from Donald Bren School of Information and Computer Science. A part of this dissertation is also supported by National Science Foundation, under grant 1815073. Chapter 3 of this dissertation is a reprint of the material as it appears in [8], used with permission from Springer, and the co-author listed in this publication is Michael Goodrich.

Finally, I want to thank my family and friends for being always there for me and motivating me throughout this journey. I want to thank my parents, starting from my mother, Elham, who dedicated her youth to raise me and motivated me to reach my ideal self, and my father, Ali, who had faith in me, supported me unconditionally. I also thank my sister, Melika, for her cheerfulness and her sense of humor who turned around my bad days. I want to thank my lovely wife, Parinaz, whom I met in her senior year of her bachelor at UC Irvine. She always pushed me to the next level and supported me when I was mentally broken. Having her in my life undoubtedly made me a better person in every single way. I also thank my parents-in-law, Farhnaz and Kamran, who welcomed me in their family. I want to thank my best friend, Saman, who was always like an older brother for me and I'm grateful that I found him at UC Irvine.

VITA

Ramtin Afshar

EDUCATION

Doctor of Philosophy in Computer Science	**2023**
University of California, Irvine	*Irvine, USA*
Master of Science in Computer Science	**2022**
University of California, Irvine	*Irvine, USA*
Bachelor of Science in Computer Engineering, Software	**2014**
Sharif University of Technology	*Tehran, Iran*

EXPERIENCE

Graduate Research Assistant	**2018–2023**
University of California, Irvine	*Irvine, California*
Teaching Assistant	**2018–2023**
University of California, Irvine	*Irvine, California*
Software Engineering Intern	**2022**
Google	*Mountain View, California*
Data Analyst Intern	**2021**
Samsung	*San Jose, California*

REFEREED CONFERENCE PUBLICATIONS

Noisy Sorting Without Searching: Data Oblivious Sorting with Comparison Errors **2023**
Ramtin Afshar, Michael Dillencourt, Michael Goodirhc, Evrim Ozel

In reivew for Symposium on Experimental Algorithms (SEA)

Exact Learning of Multitrees and Almost-Trees Using Path Queries **2022**
Ramtin Afshar, Michael T. Goodrich

Latin American Theoretical Informatics Symposium

Efficient Exact Learning Algorithms for Road Networks and Other Graphs with Bounded Clustering Degrees **2022**
Ramtin Afshar, Michael T. Goodrich, Evrim Ozel

International Symposium on Experimental Algorithms (SEA)

Mapping Networks via Parallel kth-Hop Traceroute Queries **2022**
Ramtin Afshar, Michael T. Goodrich, Pedro Matias, Martha C. Osegueda

International Symposium on Theoretical Aspects of Computer Science (STACS)

Parallel Network Mapping Algorithms **2021**
Ramtin Afshar, Michael T. Goodrich, Pedro Matias, Martha C. Osegueda

Symposium on Parallelism in Algorithms and Architectures (SPAA)

Reconstructing Biological and Digital Phylogenetic Trees in Parallel **2020**
Ramtin Afshar, Michael T. Goodrich, Pedro Matias, Martha C. Osegueda

European Symposium on Algorithms (ESA)

Reconstructing Binary Trees in Parallel **2020**
Ramtin Afshar, Michael T. Goodrich, Pedro Matias, Martha C. Osegueda

Symposium on Parallelism in Algorithms and Architectures (SPAA)

Adaptive Exact Learning in a Mixed-Up World: Dealing with Periodicity, Errors and Jumbled-Index Queries in String Reconstruction **2020**
Ramtin Afshar, Amihood Amir, Michael T. Goodrich, Pedro Matias

International Symposium on String Processing and Information Retrieval (SPIRE)

ABSTRACT OF THE DISSERTATION

Exact Learning of Graphs Using Queries

By

Ramtin Afshar

Doctor of Philosophy in Computer Science

University of California, Irvine, 2023

Distinguished Professor Michael T. Goodrich, Chair

Given the vertices, V, of an unknown graph $G = (V, E)$, exact learning refers to the process of reconstructing the edges, E, to learn its structure using an all-knowing black box oracle. This is motivated by a broad range of applications from network discovery, where the goal is to infer the topology of a communication network from queries, to computational biology and digital phylogeny, where we wish to learn how a set of objects have been evolved through a set of experiments. In this dissertation, we study various instances of exact learning of graphs using different query settings. For instance, we present an optimal algorithm to learn digital phylogenetic trees (directed rooted trees) using path queries, where a path query given two vertex, u and v, it returns true if and only if there is a directed path from u to v. We also provide efficient algorithms to learn other directed graphs such as multitrees, butterfly networks, and almost-trees from path queries. In addition, we study efficient learning algorithms for network mapping using kth-hop queries, where a kth-hop query given vertex u and v and integer k, it returns the kth vertex on a shortest path from u to v.

Chapter 1

Introduction

The exact learning of a graph, also known as **graph reconstruction**, refers to the process of reconstructing an unknown ground-truth graph data structure, through an all-knowing oracle, which answers certain types of queries involving a subset of vertices of the graph. This process can be abstracted in terms of two parties, a *querier*, Bob, who issues queries of a certain type with the goal of learning a hidden ground-truth graph, G, and, a *responder*, Alice, who must correctly answer queries regarding the structure G.

We distinguish three settings for exact learning of graphs, namely, **non-adaptive**, **adaptive**, and an intermediate setting, **parallel** [42]. In non-adaptive, the querier, has to issue all the queries at once upfront. In adaptive setting, the querier may choose queries depending on the answers to the prior queries. Finally, in the parallel setting, the querier issues queries in batches where the queries in the same batch should not depend on the answer of the other queries in the same batch, but they may rely on the queries issued in the previous batches. Since the provided query responses may eliminate the search space, that is, it may remove the need to ask some specific queries, it is reasonable to think the more parallel a solution is, the more likely it demands more queries.

In this dissertation, we mostly focus on studying parallel exact learning algorithms. We measure the efficiency of our methods in terms of the number of vertices of the graph, n, using two complexities, i) query complexity, $Q(n)$, which is the total number of queries that we perform, and it comes from the learning theory [5, 32, 47, 7, 95] and complexity theory [26, 107], and ii) round complexity, $R(n)$, which is the number of rounds that we perform our queries. In particular, we provide efficient algorithms for exact learning of various graph classes including directed graphs such as rooted trees, multi-trees, almost-trees, and undirected connected graphs with various query models which we describe next.

Note that while the focus of this dissertation is to study exact learning methods for learning a hidden ground truth graph, there are other frameworks for learning as well. For instance, as proposed by Valiant [102], probably approximate correct (PAC) learning is a framework with the goal of learning with high probability an approximately correct hidden concept (e.g. graphs in this dissertation). In addition, throughout this dissertation we assume all the query responses are correct. Note that in the presence of an independent noise probability $p < 1/2$ for query responses, that is, when response of each query is correct independently with probability $1-p$ and incorrect otherwise, all of our results still hold with high probability by repeating each query for $O(\log n)$ times in parallel and taking the majority vote as the response [30].

1.1 Learning Rooted Trees

We present an optimal parallel algorithm to learn an unknown directed rooted tree $T = (V, E, r)$, with vertex set V, and root vertex, $v \in V$, and edges E oriented away from r using path queries. A path query given two vertices u and v, it returns true if and only if there is a directed path from u to v. The querier knows V, but learning r and E is the objective of the algorithm. Motivated by applications in biological and digital phylogeny, we assume that the tree has a maximum degree of some constant, d.

Results.

We show that a fixed-degree rooted tree with n vertex can be learnt using path queries with $R(n) \in O(\log n)$ and $Q(n) \in O(n \log n)$, with high probability, w.h.p. [1] We also prove that our algorithm is optimal in terms of query complexity and round complexity by providing a matching lower bound of $\Omega(dn + n \log n)$ for any randomized or deterministic algorithm. In addition, our parallel algorithm outperforms the best-known sequential complexity of $Q(n) \in O(n \log^2 n)$ for this problem by Wang and Honorio [103]. We also suggest some real-world applications by accompanying an experimental analysis of our algorithm.

1.2 Learning Multitrees and Almost-trees

Given the vertices V of a directed acyclic graph (DAG), $G = (V, E)$, we aim at learning E using path queries. However, not every directed acyclic graphs can be learnt using path queries, for instance, transitive edges in a DAG cannot be distinguished. For this purpose, we study families of DAGs that do not have transitive edges. We devise exact learning algorithms for multitrees—a Dag with at most one directed path between any pair of vertices. We also study how to efficiently learn an almost-tree, where we define an almost-tree as a union of a directed rooted tree with an additional cross edge.

Results.

We first present a deterministic result for learning a directed rooted tree using path queries, giving a sequential deterministic approach to learn fixed-degree trees of height h, with $O(nh)$ path queries, which forms a building block for some of the algorithms in Chapter 3. We then

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

employ a tree-learning method to design a learning method for a multitree with a roots using $Q(n) \in O(an \log n)$ path queries and $R(n) \in O(a \log n)$ rounds w.h.p. Additionally, we use our tree learning to devise a method with $Q(n) \in O(n^{3/2} \cdot \log^2 n)$ path queries to learn butterfly networks w.h.p. We present a parallel algorithm to learn almost-trees of height h, with $Q(n) \in O(n \log n + nh)$ and $R(n) \in O(\log n)$ w.h.p. We prove that our our almost-tree learning algorithm is optimal by providing 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. In addition, our asymptotically-optimal query complexity bound improves the best-known sequential query complexity for this problem due to Janardhanan and Reyzin [69] who achieved $Q(n) \in O(n \log^3 n + nh)$ in expectation.

1.3 Learning Undirected Graphs

For a source node u and a target node v, the kth-hop query returns kth vertex on a shortest path from u to v. Indeed, kth-hop query forms the inner loop of how the `traceroute` command works, as `traceroute` issues kth-hop query for $k = 1, 2, \cdots, \delta(u, v)$, where $\delta(u, v)$ is the number of edges on a shortest path from u to v. Suppose we are given access to a subset $U \subseteq V$ of vertices of a connected, undirected, unweighted graph $G = (V, E)$, in the network mapping problem, we wish to learn the induced shortest path graph $H = (U, \tilde{E})$, such that there is an edge $(u, v) \in \tilde{E}$ if and only if kth-hop(k, u, v) return vertices of a shortest path from u to v that does not include any other vertex in U except v, that is, no kth-hop(k, u, v) query would return a vertex $w \in U$ aside from vertex v. Therefore, the objective of the network mapping problem is to learn a weighted, connected, undirected, graph H, where for two vertices u and v the edge (u, v) in H has weight $\delta(u, v)$

Results.

We begin by introducing a new parallel implementation of a well-known graph clustering technique of Thorup and Zwick [98] with round complexity of $O(1)$, while their original implementation implies an expected round complexity of $O(\log n)$. In doing so, we introduce a parameter that allows us to trade off parallel time and cluster size. We will use this new construction to learn a graph-theoretic Voronoi diagram in our network mapping algorithm. We then provide the first non-trivial algorithmic results for the network mapping problem. We characterize the query complexity and our round complexity using the size of the vantage point, $n = |U|$, and some interesting parameters that capture the sampling coverage provided by the set U. For instance, let Δ be the maximum degree of the graph, H, and we introduce a distance coverage parameter, $\delta_{\max}$, which is the maximum weight for an edge in H, and a nearby-vertices parameter, μ, which is an upper bound on the number of vertices within a distance of $2\delta_{\max}$ of any given vertex $v \in U$. We show that these parameters are required to avoid trivial quadratic solutions. For instance, under reasonable assumptions regarding these parameters, we present the first constant-round network-mapping algorithm with query complexity better than the trivial brute-force algorithm. We also introduce a greedy approach for network mapping that is based on parallel greedy approximate set cover, which allows us to achieve a near-quasilinear query complexity.

Chapter 2

Learning Rooted Trees

2.1 Introduction

Phylogenetic trees demonstrate how a group of objects have been evolved from one another. Phylogenetic trees most commonly refer to the biological phylogenetic trees, however, recently, there has been a growing interest to study the evolutionary relationships in digital phylogenetic trees [54, 83, 70, 77, 92, 27, 46, 45, 44]. Within a digital phylogenetic tree, each vertex is a data object, such as a multimedia object (e.g. images or videos) [27, 46, 45, 44], a text document [77, 92], a source-code [70], or a computer virus [54, 83], and the edges show how these objects are evolved using edits, data compression, or data corruption. (See fig. 2.1.)

In this chapter, we provide efficient algorithms for exact learning of digital phylogenetic trees using queries involving nodes of the tree. In digital phylogenetic trees, we can perform queries involving any nodes in the tree, including internal nodes, as these represent digital artifacts, which are often archived. In particular, we focus on *path queries*, where one is given two nodes, v and w, and the response is "true" if and only if v is an ancestor of w.

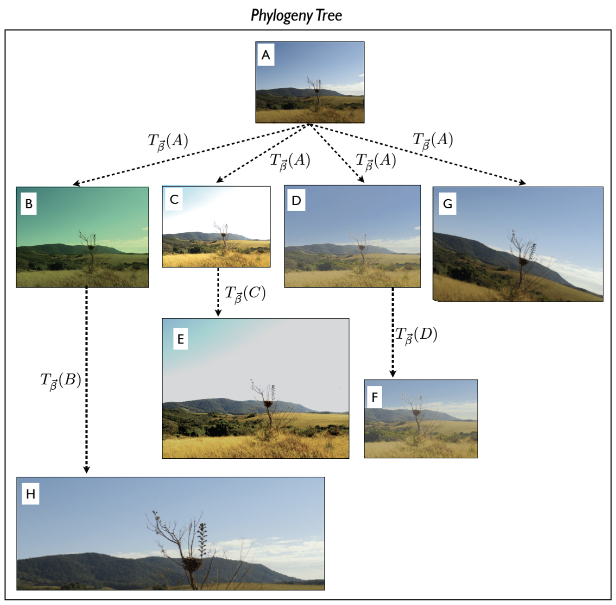

Figure 2.1: A digital phylogenetic tree of images, from Dias *et al.* [44] showing the evolutionary relationship between a set of near-duplicate images.

Reconstructing these phylogenetic trees helps us to better understand the evolutionary process of the digital artifacts. Specifically, learning the structure of digital phylogenetic trees has applications in several areas such as security, forensics, and copyright enforcement [54, 83, 70, 77, 92, 27, 46, 45, 44]. For instance, learning a phylogeny of original and near-duplicate documents can help forensic experts as they may achieve better results if they analyze the original document rather than near-duplicate documents [44]. On the other hand, these experts may want to focus more on individuals who distribute the contents closer to the root of tree since they are more likely to be the one who created the original content, therefore, learning the tree can help them to identify the nodes close to the root.

While previous work focused on learning trees sequentially, we provide an efficient parallel exact learning method to learn trees. We measure the performance of our exact learning methods in terms of the number of vertices of the tree, n, using two complexities:

- Query complexity, $Q(n)$: This is the total number of queries that we perform. This parameter comes from learning theory [5, 32, 47, 95] and complexity theory [106, 26].

- 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).

Roughly speaking, $R(n)$ corresponds to the span of a parallel exact learning algorithm, while $Q(n)$ corresponds to its work. In this chapter, we are interested in studying complexities for $R(n)$ and $Q(n)$ with respect to digital phylogenetic trees with fixed maximum degree, d.

2.1.1 Related Work

Kannan *et al.* [71] study the problem of learning a connected, undirected, and unweighted fixed-degree graph using distance queries, where the query returns the distance between two given vertices of the graph, and they provide a randomized algorithm for learning a graph of n vertices using $\tilde{O}(n^{3/2})$ distance queries.[1] Abrahamsen *et al.* [3] study the same problem with a weaker query type, called betweenness, where the query given three vertices u, v, w returns whether v lies on a shortest path from u to w, and they provide an algorithm to exactly learn the graph using $\tilde{O}(n^{3/2})$ betweenness queries.

We are not aware of previous parallel work for learning digital phylogenetic trees using a similar query model to ours. With respect to prior work on sequential tree learning, Culberson and Rudnicki [39] provide an algorithm to learn a weighted undirected tree with n vertices using additive distance queries, where each query given two vertices returns the sum of the weights of edges on the path between these two vertices. They show that their algorithm takes $O(n^2)$ queries to learn the tree in general, but for a tree of maximum degree, d, they provide an analysis showing that their algorithm takes $O(dn \log_d n)$ additive queries. Reyzin and Srivastava [87] show that Culberson-Rudnicki algorithm indeed uses $O(n^{3/2} \cdot \sqrt{d})$

[1]The $\tilde{O}(\cdot)$ notation hides poly-logarithmic factors.

queries for learning a tree of maximum degree, d, and they provide tight examples. Hein [61] study the problem of learning a biological phylogenetic tree using additive distance query where the query returns the distance between two species and they give an algorithm using $O(dn \log_d n)$ additive queries.

With respect to digital phylogenetic tree reconstruction, there are a number of sequential algorithms with $O(n^2)$ query complexities, including the use of what we are calling path queries, where the queries are also individually expensive, e.g., see [54, 83, 70, 77, 92, 27, 46, 45, 44]. Jagadish and Sen [68] consider reconstructing undirected unweighted degree-d trees, giving a deterministic algorithm that requires $O(dn^{1.5} \log n)$ separator queries, which answer if a vertex lies on the path between two vertices. They also give a randomized algorithm using an expected $O(d^2 n \log^2 n)$ number of separator queries, and they give an $\Omega(dn)$ lower bound for any deterministic algorithm. Wang and Honorio [103] consider the problem of reconstructing bounded-degree rooted trees, giving a randomized algorithm that uses expected $O(dn \log^2 n)$ path queries. They also prove that any randomized algorithm requires $\Omega(n \log n)$ path queries.

Our Contributions. In this chapter, we show that an n-node fixed-degree digital phylogenetic tree can be reconstructed from *path queries*, which ask whether a given node, u, is an ancestor of a given node, w, with $R(n)$ that is $O(\log n)$ and $Q(n)$ that is $O(n \log n)$, w.h.p. We also provide an $\Omega(dn + n \log n)$ lower bound for the query complexity of any randomized or deterministic algorithm suggesting that our algorithm is optimal in terms of query complexity and round complexity. Further, this asymptotically-optimal $Q(n)$ bound actually improves the sequential complexity for this problem, as the previous best bound for $Q(n)$, due to Wang and Honorio [103], had a $Q(n)$ bound of $O(n \log^2 n)$ for reconstructing fixed-degree rooted trees using path queries. Of course, our method also applies to biological phylogenetic trees that support path queries. These results are accompanied by an experimental analysis of our algorithm at the end of this chapter showing the real-world applications.

2.2 Preliminaries

A digital phylogenetic tree can be represented using using a directed rooted tree, $T = (V, E, r)$, with a vertex set V, root $r \in V$, and the edge set E oriented away from the root. The *degree* of a vertex in such a tree is the sum of its in-degree and out-degree, and the degree of a tree, T, is the maximum degree of all vertices in T. In this chapter, we assume that the trees have a maximum degree of constant, d.

Next, we review a few commonly used terms in this chapter.

Definition 2.1. *(ancestry) Assume $T = (V, E, r)$ is a rooted tree, we call u a* child *of v (and v* parent *of u) if there exists a directed edge (v, u) in E. The* descendant *relation is the transitive closure of the child relation, and the* ancestor *relation is the transitive closure of the parent relation. We call a node leaf if its out-degree is* 0*. Let $D(v)$ be the set of descendants of a vertex v.*

Definition 2.2. *(path query) Given two nodes, u and v in a rooted tree T, a* path query *returns* 1 *if there is a directed path from u to v, and otherwise returns* 0*. Also, for $v \in V$ and $U \subseteq V$, we represent the number of descendants of v in U with $count(v, U) = |D(v) \cap U| = \sum_{u \in U} path(v, u)$.*

2.3 Learning Rooted Trees using Path Queries

Let $T = (V, E, r)$ be a rooted digital phylogenetic tree with fixed degree, d. In this section, we show how a querier can reconstruct T by issuing $Q(n) \in O(n \log n)$ path queries in $R(n) \in O(\log n)$ rounds, w.h.p., where $n = |V|$. We provide a lower bound to prove that our algorithm is optimal in terms of query complexity and round complexity. At the outset, the only thing we assume the querier knows is n and V, that is, the vertex set for T, and that the names of the nodes in V are unique, i.e., we may assume, w.l.o.g., that $V = \{1, 2, \ldots, n\}$. The querier doesn't know E or r—learning these is his goal.

2.3.1 Algorithms

We start by learning r, which we show can be done via any maximum-finding algorithm in Valiant's parallel model [100], which only counts parallel steps involving comparisons. The challenge, of course, is that the ancestor relationship in T is, in general, not a total order, as required by a maximum-finding algorithm. This does not actually pose a problem, however.

Lemma 2.1. *Suppose A be a parallel maximum-finding algorithm in Valiant's model, with $O(f(n))$ span and $O(g(n))$ work. We can use A to find the root, r in a rooted tree $T = (V, E, r)$, using $R(n) \in O(f(n))$ rounds and $Q(n) \in O(g(n) + n)$ total queries.*

Proof. We pick an arbitrary vertex $v \in T$. In the first round, we perform queries $path(u, v)$ in parallel for every other vertex $u \in V$ to find S, the ancestor set for v. If $S = \emptyset$, then v is the root. Otherwise, we know all the vertices in a path from root to the parent of v, albeit unsorted. Still, note that for S the ancestor relation is a total order; hence, we can simulate A with path queries to resolve the comparisons made by A. We have just a single round and $O(n)$ queries more than what it takes for A to find the maximum. Thus, we can find the root in $O(f(n))$ rounds and $O(g(n) + n)$ queries. □

Thus, by well-known maximum-finding algorithms, e.g., see [34, 93, 100]:

corollary 2.1. *We can find r of a rooted tree $T = (V, E, r)$ deterministically in $O(\log \log n)$ rounds and $O(n)$ queries.*

Determining the rest of the structure of T is more challenging, however. At a high level, our approach to solving this challenge is to use a separator-based divide-and-conquer strategy.

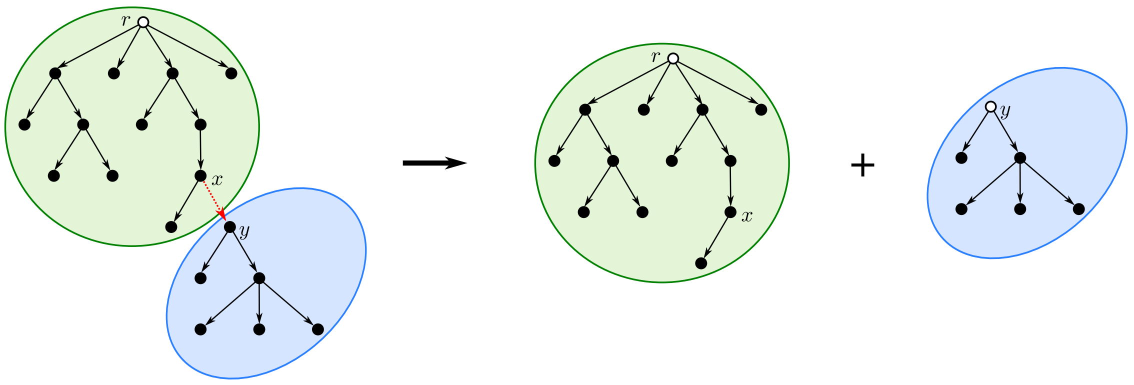

Figure 2.2: This figure shows the divide and conquer approach using an edge-separator, (x, y) for rooted trees. Note that the root of T'' is r, while y becomes root of T'.

We next study a separator for degree-d rooted trees.

Definition 2.3. *In a tree $T = (V, E, r)$ of maximum degree d, we call an edge $e = (x, y) \in E$ an* even-edge-separator *if removing e from T partitions it into two rooted trees of with at most $|V| \cdot (d-1)/d$ vertices (see fig. 2.2).*

Lemma 2.2. *Every rooted tree of maximum degree d has an even-edge-separator.*

Proof. This follows from a result by Valiant [101, Lemma 2]. □

If we can find an even-edge-separator, then we can cut the tree in two by removing that edge and recurse on the two remaining subtrees in parallel (see fig. 2.2), but this requires an exact calculation of the number of descendants of a node which takes too many queries. We instead find a "near" edge-separator in T, divide T using this edge, and recurse on the two remaining subtrees in parallel. The difficulty, of course, is that the querier has no knowledge of the edges of T; hence, the very first step, finding a "near" edge-separator, is a bottleneck computation. Fortunately, as we show in lemma 2.3, if v is a randomly-chosen vertex, then, with probability depending on d, the path from root r to v includes an edge-separator.

Lemma 2.3. *Let $T = (V, E, r)$ be a rooted tree of degree d and let v be a vertex chosen uniformly at random from V. Then, with probability at least $\frac{1}{d}$, an even-edge-separator is one of the edges on the path from r to v.*

Proof. By lemma 2.2, T has an even-edge-separator. Let $e = (x, y)$ be an even-edge-separator for $T = (V, E, r)$ and let $T' = (V', E', y)$ be the subtree rooted at y when we remove e. Then, every path from r to each $v \in V'$ must contain e. By definition 2.3, T' has at least $|V|/d$ vertices. Therefore, if we choose v uniformly at random from V, then with probability $\frac{|V'|}{|V|} \geq \frac{1}{d}$, the path from r to v contains e. $\square$

Definition 2.4. *(splitting-edge) In a degree-d directed rooted tree, an edge $(parent(s), s)$ is a splitting-edge if $\frac{|V|}{d+2} \leq |D(s)| \leq \frac{|V|(d+1)}{d+2}$, where $D(s)$ is the set of descendants of s.*

Note that a degree-d rooted tree T always has a splitting-edge, as every even-edge-separator is also a splitting-edge and by lemma 2.2, it always has an even-edge-separator—a fact we use in our tree learning algorithm, which we describe next. This recursive algorithm (given in pseudo-code in Algorithm 1), assumes the existence of a randomized method, find-splitting-edge, which returns a splitting-edge in T, with probability $\Omega(1/d)$, and otherwise returns *Null*. Our reconstruction algorithm is therefore a randomized recursive algorithm that takes as input a set of vertices, V, with a (known) root vertex $r \in V$, and returns the edge set, E, for V. At a high level, our algorithm is to repeatedly call the method, find-splitting-edge, until it returns a splitting-edge, at which point we divide the set of vertices using this edge and recurse on the two resulting subtrees.

In more detail, during each iteration of a repeating **while** loop, we choose a vertex $v \in V$ uniformly at random. Then, we find the vertices on the path from r to v and store them in a set, Y, using the fact that a vertex, z, is on the path from r to v if and only if $path(z, v) = 1$. Then, we attempt to find a splitting-edge using the function find-splitting-edge (shown in pseudo-code in Algorithm 2). If find-splitting-edge is unsuccessful, we give up on vertex v, and restart the **while** loop with a new choice for v. Otherwise, find-splitting-edge succeeded and we cut the tree at the returned splitting-edge, (u, w). All vertices, $z \in V$, where $path(w, z) = 1$ belong to the subtree rooted at w, thus belonging to V_1, whereas the remaining vertices belong to V_2 and the partition containing both u and rooted at r. Thus,

Algorithm 1: Reconstruct a rooted tree with path queries

```
Function reconstruct-rooted-tree($V, r$):
    $E \leftarrow \emptyset$
    if $|V| \leq g$ then // $g$ is a chosen constant
        return edges found by a quadratic brute-force algorithm
    while true do
        Pick a vertex $v \in V$ uniformly at random
        for $z \in V$ do in parallel
            Perform query $path(z, v)$
        Let $Y$ be the vertex set of the path from $r$ to $v$
        splitting-edge $\leftarrow$ find-splitting-edge($v, Y, V$)
        if splitting-edge $\neq$ Null then
            $(u, w) \leftarrow$ splitting-edge
            $E \leftarrow E \cup \{(u, w)\}$
            for $z \in V$ do in parallel
                Perform query $path(w, z)$
            split $V$ into $V_1, V_2$ at $(u, w)$ using query results
            parallel do
                $E \leftarrow E \cup$ reconstruct-rooted-tree($V_1, w$)
                $E \leftarrow E \cup$ reconstruct-rooted-tree($V_2, r$)
            return $E$
```

after cutting the tree we recursively reconstruct-rooted-tree on V_1 and V_2.

The main idea for our efficient tree reconstruction algorithm lies in our find-splitting-edge method (see Algorithm 2), which we describe next. This method takes as input the vertex v, the vertex set Y, (comprising the vertices on the path from r to v), and the vertex set V. As we show, with probability depending on d, the output of this method is a splitting-edge; otherwise, the output is *Null*. Our algorithm performs a type of "noisy" search in Y to either locate a likely splitting-edge or return *Null* as an indication of failure.

Our find-splitting-edge algorithm consists of two phases. We enter **Phase 1** if the size of path Y is too big, i.e., $|Y| > |V|/K = \frac{|Y|}{C_2 \log |V|}$, where C_2 is a predetermined constant and $K = C_2 \log |V|$. The purpose of this phase is either to pass a shorter path including an even-edge-separator to the second phase or to find a splitting-edge in this iteration. The search on the set Y is noisy, because it involves random sampling. In particular, we take a

Algorithm 2: Finding a splitting-edge from vertex set, Y, on the path from vertex v to the root r

```
Function find-splitting-edge($v, Y, V$):
    splitting-edge ← $Null$
    $m = C_1\sqrt{|V|}$, $K = C_2 \log |V|$
    Phase 1:
        if $|Y| > |V|/K$ then
            $S$ ← subset of $m$ random elements from $Y$
            $S \leftarrow S \cup \{v, r\}$
            for each $s \in S$ do in parallel
                $X_s$ ← subset of $K$ random elements from $V$
                Perform queries to find $count(s, X_s)$
            if $\forall s \in S : count(s, X_s) < \frac{K}{d+1}$ then return $Null$
            if $\forall s \in S : count(s, X_s) > \frac{Kd}{d+1}$ then return $Null$
            if $\exists s \in S : \frac{K}{d+1} \leq count(s, X_s) \leq \frac{Kd}{d+1}$ then
                return verify-splitting-edge($s, V$)
            for each $\{a, b\} \in S$ do in parallel
                perform query $path(a, b)$
            Find $w, z$ such that they are two consecutive nodes in the sorted order of
             $S$ such that $count(w, X_w) > \frac{Kd}{d+1}$ and $count(z, X_z) < \frac{K}{d+1}$
            $Y$ ← nodes from $Y$ in the path from $w$ to $z$
        Phase 2:
            if $|Y| > |V|/K$ then return $Null$
            for each $s \in Y$ do in parallel
                $X_s$ ← subset of $K$ random elements from $V$
                Perform queries to find $count(s, X_s)$
            if $\exists s \in Y$ s.t. $\frac{K}{d+1} \leq count(s, X_s) \leq \frac{Kd}{d+1}$ then
                return verify-splitting-edge($s, V$)
    return $Null$
```

random sample S of size $m = C_1\sqrt{|V|}$ from path Y (where C_1 is a predetermined constant). We include r and v, the two endpoints of the path Y, to S. Then, we estimate the number of descendants of s, $D(s)$, for each $s \in S$. To estimate this number for each $s \in S$, we take a random sample X_s of K elements from V and we perform queries to find $count(s, X_s)$, the number of descendants of s in X_s. Here, we use $m \cdot K \in O(\sqrt{|V|}\log|V|)$ queries in a single round. Then, if all the estimates were less than $K/(d+1)$, we return *Null* as an indication of failure (we guess that all the nodes on the path Y have too few descendants to be a separator). Similarly, if all the estimates were greater than $\frac{Kd}{d+1}$, we return *Null* (we guess that all the nodes on the path Y have too many descendants to be a separator). If there exists a node s such that $\frac{K}{d+1} \leq count(s, X_s) \leq \frac{Kd}{d+1}$, we check if s is a splitting-edge by counting its descendants using a function, verify-splitting-edge. This function takes vertex s and the full vertex set V to return edge (find-parent$(s, V), s)$ if $\frac{|V|}{d+2} \leq count(s, V) \leq |V| \cdot \frac{d+1}{d+2}$ and return *Null* otherwise.

If none of these three cases happens, we perform queries to sort elements of S using a trivial quadratic work parallel sort which takes $O(m^2) \in O(|V|)$ queries in a single round. We know that two consecutive nodes w and z exist on the sorted order of S, where $count(w, X_w) > \frac{Kd}{d+1}$ and $count(z, X_z) < \frac{K}{d+1}$. We find all the nodes on Y starting at w and ending at z, and use this as our new path Y.

In **Phase 2**, we expect a path of size under $|Y|/K$, we will later prove this is true with high probability. Otherwise, we just return *Null*. In this phase, we estimate the number of descendants much like we did in the previous phase, except the only difference is that we estimate the number of descendants for all the nodes on our new path Y. If there exists a node $s \in Y$ such that $\frac{K}{d+1} \leq count(s, X_s) \leq \frac{Kd}{d+1}$, we verify if it is a splitting-edge, as described earlier.

Finally, let us describe how we find the parent of a node s in V. We first find, Y, the set of ancestors of v in V in parallel using $|V|$ queries. Let $x \succ y$ describe the total order of nodes in path Y, where for any $x, y \in Y : x \succ y$ if and only if $path(x, y) = 1$. The parent of s is the lowest vertex on the path. Then, the key idea is that if $|Y| \in O(\sqrt{|V|})$, we can sort them using $O(|V|)$ queries. If the path is greater than this amount, we instead use S, a sample of size $O(\sqrt{|V|})$ from the path. Next, we sort the sample to obtain $x_1 < \ldots < x_m$ for S and then find all of the nodes in Y which are less than the smallest sample x_1. Finally, we replace Y with these descendants of x_1 and repeat the whole procedure again. We later prove that with high probability after two iterations of this sampling, the size of the path is $O(\sqrt{|V|})$, allowing us to sort all nodes in Y to return the minimum (see Function find-parent).

Function find-parent(s, V)

```
1  find $Y$, ancestor set of $s$ from $V$ using $|V|$ queries in parallel
2  $m = C_1\sqrt{|V|}$
3  for $i \leftarrow 1$ to $2$ & $|Y| > m$ do
4      $S \leftarrow$ random subset of $Y$ with $m$ elements
5      sort $S$ as $x_1 < ... < x_m$ using $O(m^2)$ queries in parallel
6      find $Y' = \{u \in Y \mid u \leq x_1\}$ using $O(|Y|)$ queries in parallel, replace $Y$ with $Y'$
7  if $|Y| \leq m$ then
8      sort $Y$ using $O(m^2)$ queries in parallel
9      return (minimum of this path)
10 return Null
```

2.3.2 Analysis

The correctness of the algorithm follows from the fact that our method first learns the root, r, of T and then learns the parent of each other node, v in T.

Theorem 2.1. *Given a set, V, of nodes of a rooted tree, T, such as a biological or digital phylogenetic tree, with degree bounded by a fixed constant, d, we can construct T using path queries with round complexity, $R(n)$, that is $O(\log n)$ and query complexity, $Q(n)$, that is $O(n \log n)$, with high probability.*

Our proof of theorem 2.1 begins with lemma 2.4.

Lemma 2.4. *In a rooted tree, $T = (V, E, r)$, let Y be a (directed) path, where $|Y| > m = C_1\sqrt{|V|}$. If we take a sample, S, of m elements from Y, then with probability $1 - \frac{1}{|V|}$, every two consecutive nodes of S in the sorted order of S are within distance $O\left(\frac{|Y| \log |V|}{\sqrt{|V|}}\right)$ from each other in Y.*

Proof. Note that some nodes of Y may be picked more than once as we pick S in parallel. Divide the path Y into $\frac{\sqrt{|V|}}{\log |V|}$ equal size sections (the difference between the size of any two sections is at most 1). For each $1 \leq i \leq \frac{\sqrt{|V|}}{\log |V|}$, let A_i be the subset of S lying in the i^{th} section of Y. (See fig. 2.3.) It is clear that each node $s \in S$ ends up in section i with probability

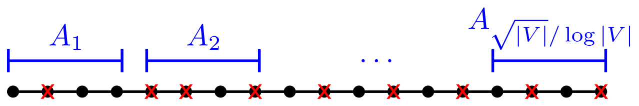

Figure 2.3: Illustration of how scattered sample S is on path Y. The i^{th} blue interval represents the i^{th} section of Y, the black dots correspond to the nodes on the path Y, and red crossed marks represent elements of S.

$\frac{\log|V|}{\sqrt{|V|}}$, and therefore, for each $1 \le i \le \frac{\sqrt{|V|}}{\log|V|}$, $E[|A_i|] = C_1 \log|V|$. Thus, using standard a Chernoff bound, $\Pr[|A_i| = 0] < \frac{1}{|V|^2}$ for any constant $C_1 > 6\ln 10$. Using a union bound, all the sections are non-empty with probability at least $1 - \frac{1}{|V|}$. Hence, the distance between any two consecutive nodes of S from each other in Y is at most $\frac{2|Y|\log|V|}{\sqrt{|V|}}$. □

Lemma 2.4 allows us to analyze the find-parent method, as follows.

Lemma 2.5. *The find-parent(s, V) method outputs the parent of s with probability at least $1 - 2/|V|$, with $Q(n) \in O(n)$ and $R(n) \in O(1)$.*

Proof. The find-parent method succeeds if, after the **for** loop, the size of the set of remaining ancestors of s, Y, is $|Y| \le m$, so it is enough to show that this occurs with probability at least $1-2/|V|$. By lemma 2.4, the size of Y at the end of the first iteration is $|Y| \in O(m \log|V|)$, with probability at least $1 - 1/|V|$. Similarly, a second iteration, if required, further reduces the size of Y into $|Y| \in o(m)$, with probability at least $1 - 1/|V|$. Thus, by a union bound, the probability of success is at least $1 - 2/|V|$.

The query complexity can be broken down as follows, where $m \in O(\sqrt{|V|})$:

1. $O(|V|)$ queries in 1 round to determine the ancestor set, Y, of s.

2. $O(m^2) + O(|Y|) \in O(|V|)$ queries in 2 rounds for each of the (at most) 2 iterations performed in find-parent, whose purpose is to discard non-parent ancestors of s in Y.

3. $O(m^2) \in O(|V|)$ queries to find, in 1 round, the minimum among the remaining ancestors of Y (at most m w.h.p.). If $|Y| > m$, then no further queries are issued.

In total, the above amounts to $Q(n) \in O(n)$ and $R(n) \in O(1)$. □

We next analyze the find-splitting-edge method. Let us first use an intricate Chernoff-bound analysis to prove the following useful probability bounds in lemma 2.6.

Lemma 2.6. *There exists a constant $C_2 > 0$, as used $K = C_2 \log |V|$in line 3 of Algorithm 2, such that if we take a sample X of size K from V, the following probability bounds always hold:*

$$\begin{cases} \Pr\left(count(s,X) \geq \frac{K}{d+1}\right) \geq 1 - \frac{1}{|V|^2} & \textit{if } count(s,V) \geq \frac{|V|}{d}, \\ \Pr\left(count(s,X) \leq K\frac{d}{d+1}\right) \geq 1 - \frac{1}{|V|^2} & \textit{if } count(s,V) \leq |V|\frac{d-1}{d}, \end{cases} \tag{2.1}$$

$$\begin{cases} \Pr\left(count(s,X) < \frac{K}{d+1}\right) \geq 1 - \frac{1}{|V|^2} & \textit{if } count(s,V) < \frac{|V|}{d+2}, \\ \Pr\left(count(s,X) > K\frac{d}{d+1}\right) \geq 1 - \frac{1}{|V|^2} & \textit{if } count(s,V) > |V|\frac{d+1}{d+2} \end{cases} \tag{2.2}$$

Proof. Recall that $count(s,X)\frac{|V|}{K}$ is an estimation of $|D(s)| = count(s,V)$, the number of descendants of vertex s in algorithm 2. Let Z be sum of K independent binary random variables with expected value $E[Z]$. Using a Chernoff bound, we know that $\Pr\left[\left|Z - E[Z]\right| \geq \epsilon E[Z]\right] \leq 2e^{\frac{-1}{3}\epsilon^2 E[Z]}$:

In this case, our random variable is $Z = count(s,X)$ and $E[Z] = |D(s)|\frac{K}{|V|}$. By reformulating a Chernoff bound, we have

$$\Pr\left[\left|Z - |D(s)|\frac{K}{|V|}\right| \geq \epsilon|D(s)|\frac{K}{|V|}\right] \leq 2e^{\frac{-1}{3}\epsilon^2|D(s)|\frac{K}{|V|}} \tag{2.3}$$

Now, we find the value of C_2 used in line 3 of algorithm 2 to compute K, the size of the sample. We do this for each of the 4 cases distinguished in equations 2.1, 2.2:

Case 1: We want to prove that if $|D(s)| \geq \frac{|V|}{d}$,

then $\Pr\left[\mathit{count}(s,X)\frac{|V|}{K} \geq \frac{|V|}{(d+1)}\right] \geq 1 - \frac{1}{|V|^2}$:

Suppose $|D(s)| \geq \frac{|V|}{d}$; we prove that $\Pr\left[\mathit{count}(s,X)\frac{|V|}{K} < \frac{|V|}{d+1}\right] < \frac{1}{|V|^2}$.

If we set $\epsilon = \frac{1}{d+1}$, we show that

$$\Pr\left[\mathit{count}(s,X)\frac{|V|}{K} < \frac{|V|}{d+1}\right] \leq \Pr\left[\left|\mathit{count}(s,X) - |D(s)|\frac{K}{|V|}\right| \geq \epsilon|D(s)|\frac{K}{|V|}\right] \tag{2.4}$$

In order to prove this, given the facts that $\epsilon = \frac{1}{d+1}$, and $|D(s)| \geq \frac{|V|}{d}$, we show that, for any $\mathit{count}(s,X)\frac{|V|}{K}$ such that the inequality $\mathit{count}(s,X)\frac{|V|}{K} < \frac{|V|}{d+1}$ holds, then the inequality $\left[\left|\mathit{count}(s,X) - |D(s)|\frac{K}{|V|}\right| \geq \epsilon|D(s)|\frac{K}{|V|}\right]$ also holds.

$$\begin{aligned}
&\frac{d}{d+1}|D(s)|\frac{K}{|V|} \geq \frac{K}{d+1} > \mathit{count}(s,X) \\
\Longrightarrow &\left[\left(1 - \frac{1}{d+1}\right)\left(|D(s)|\frac{K}{|V|}\right) \geq \mathit{count}(s,X)\right] \\
\Longrightarrow &\left[\left(|D(s)|\frac{K}{|V|} - \mathit{count}(s,X)\right) \geq \frac{1}{d+1}|D(s)|\frac{K}{|V|}\right] \\
\Longrightarrow &\left[\left|\mathit{count}(s,X) - |D(s)|\frac{K}{|V|}\right| \geq \epsilon|D(s)|\frac{K}{|V|}\right] \qquad \left(\epsilon = \frac{1}{d+1}\right)
\end{aligned}$$

Thus, inequality 2.4 is true. Combining inequalities 2.3 and 2.4, we have that:

$$\Pr\left[\mathit{count}(s,X)\frac{|V|}{K} < \frac{|V|}{d+1}\right] \leq 2e^{\frac{-1}{3}\epsilon^2|D(s)|\frac{K}{|V|}}$$

Now, we find the value of C_2, such that for $K = C_2 \log|V|$:

$$2e^{\frac{-1}{3}\epsilon^2|D(s)|\frac{K}{|V|}} < \frac{1}{|V|^2}$$

Taking the logarithm of both sides and given that $|D(s)| \geq \frac{|V|}{d}$ and $\epsilon = \frac{1}{d+1}$, we have that:

$$\frac{1}{3}\epsilon^2 |D(s)| \frac{K}{|V|} \geq \frac{1}{3}\left(\frac{1}{d+1}\right)^2 \frac{|V|}{|d|}\frac{K}{|V|} = \frac{1}{3}\left(\frac{1}{d+1}\right)^2 \frac{K}{d} > 2\ln\left(2|V|\right)$$
$$\iff K > 6d(d+1)^2 \ln\left(2|V|\right)$$
$$\xLeftrightarrow{K=C_2 \log|V|} C_2 > \frac{6d(d+1)^2 \ln\left(2|V|\right)}{\log|V|}$$

Thus, C_2 is not more than a constant.

Case 2: We want to prove that if $|D(s)| \leq \frac{|V|(d-1)}{d}$,

then $\Pr\left[count(s,X)\frac{|V|}{K} \leq \frac{|V|d}{(d+1)}\right] \geq 1 - \frac{1}{|V|^2}$:

Suppose $|D(s)| \leq \frac{|V|(d-1)}{d}$; we prove that $\Pr\left[count(s,X)\frac{|V|}{K} > \frac{|V|d}{(d+1)}\right] < \frac{1}{|V^2}$.

Reminding that $Z = count(s,X)$, if we set $\epsilon = \frac{K}{d(d+1)E[Z]}$, we show:

$$\Pr\left[count(s,X)\frac{|V|}{K} > \frac{|V|\,d}{(d+1)}\right] \leq \Pr\left[\left|Z - |D(s)|\frac{K}{|V|}\right| \geq \epsilon|D(s)|\frac{K}{|V|}\right] \tag{2.5}$$

In order to prove this, given the facts that $\epsilon = \frac{K}{d(d+1)E[Z]}$, and $|D(s)| \leq \frac{|V|(d-1)}{d}$, we show that, for any $count(s,X)\frac{|V|}{K}$ such that the inequality $Z = count(s,X) > \frac{Kd}{d+1}$ holds, then the inequality $\left[\left|Z - |D(s)|\frac{K}{|V|}\right| \geq \epsilon|D(s)|\frac{K}{|V|}\right]$ also holds.

Given that $Z > \frac{Kd}{d+1}$ and that $|D(s)| \leq \frac{|V|(d-1)}{d}$, we have:

$$\left|Z - |D(s)|\frac{K}{|V|}\right| = \left(Z - |D(s)|\frac{K}{|V|}\right)$$
$$> \frac{Kd}{d+1} - \frac{|V|\,(d-1)}{d}\frac{K}{|V|} = \frac{K}{d\,(d+1)}$$

Therefore, using the facts that $\epsilon = \frac{K}{d(d+1)E[Z]}$ and that $E[Z] = |D(s)|\frac{K}{|V|}$, we can say:

$$\left|Z - |D(s)|\frac{K}{|V|}\right| \geq \epsilon|D(s)|\frac{K}{|V|}$$

Thus, inequality 2.5 is true. Combining inequalities 2.3 and 2.5, we have:

$$\Pr\left[count(s,X)\frac{|V|}{K} > \frac{|V|d}{(d+1)}\right] \leq 2e^{\frac{-1}{3}\epsilon^2|D(s)|\frac{K}{|V|}}$$

Now, we find the value of C_2, such that for $K = C_2 \log|V|$:

$$2e^{\frac{-1}{3}\epsilon^2|D(s)|\frac{K}{|V|}} < \frac{1}{|V|^2}$$

Taking the logarithm of both sides and given that $\epsilon E[Z] = \frac{K}{d(d+1)}$ and $E[Z] = |D(s)|\frac{K}{|V|} \leq \frac{K(d-1)}{d}$, we can obtain $\epsilon \geq \frac{1}{(d-1)(d+1)}$, therefore:

$$\begin{aligned}
&\frac{1}{3}\epsilon^2|D(s)|\frac{K}{|V|} \geq \frac{1}{3}\frac{1}{(d-1)(d+1)}\frac{|K|}{d(d+1)} > 2\ln(2|V|)\\
&\Longleftrightarrow K > 6d(d-1)(d+1)^2\ln(2|V|)\\
&\xLeftrightarrow{K=C_2\log|V|} C_2 > \frac{6d(d-1)(d+1)^2\ln(2|V|)}{\log|V|}
\end{aligned}$$

Thus, C_2 is not more than a constant.

Case 3: We want to show that if $|D(s)| < \frac{|V|}{d+2}$,

then $\Pr\left[count(s,X)\frac{|V|}{K} < \frac{|V|}{(d+1)}\right] \geq 1 - \frac{1}{|V|^2}$:

Suppose $|D(s)| < \frac{|V|}{d+2}$; we prove that $\Pr\left[count(s,X)\frac{|V|}{K} \geq \frac{|V|}{d+1}\right] < \frac{1}{|V|^2}$.

Reminding that $Z = count(s,X)$, if we set $\epsilon = \frac{K}{(d+1)(d+2)E[Z]}$, we show:

$$\Pr\left[count(s,X)\frac{|V|}{K} \geq \frac{|V|}{d+1}\right] \leq \Pr\left[\left|Z - D(s)\frac{K}{|V|}\right| \geq \epsilon D(s)\frac{K}{|V|}\right] \tag{2.6}$$

In order to prove this, given the facts that $\epsilon = \frac{K}{(d+1)(d+2)E[Z]}$, and $|D(s)| < \frac{|V|}{d+2}$, we show that, for any $count(s,X)\frac{|V|}{K}$ such that the inequality $Z = count(s,X) \geq \frac{K}{d+1}$ holds, then the inequality $\left[\left|Z - |D(s)|\frac{K}{|V|}\right| \geq \epsilon|D(s)|\frac{K}{|V|}\right]$ also holds.

Given that $Z \geq \frac{K}{d+1}$ and that $|D(s)| < \frac{|V|}{d+2}$, we can say:

$$\left|Z - |D(s)|\frac{K}{|V|}\right| = \left(Z - |D(s)|\frac{K}{|V|}\right)$$
$$> \frac{K}{d+1} - \frac{|V|}{d+2}\frac{K}{|V|} = \frac{K}{(d+1)\,(d+2)}$$

Therefore, using the facts that $\epsilon = \frac{K}{(d+1)(d+2)E[Z]}$ and that $E[Z] = |D(s)|\frac{K}{|V|}$, we have:

$$\left|Z - |D(s)|\frac{K}{|V|}\right| \geq \epsilon|D(s)|\frac{K}{|V|}$$

Thus, inequality 2.6 is true. Combining inequalities 2.3 and 2.6, we can say:

$$\Pr\left[count(s, X)\frac{|V|}{K} \geq \frac{|V|}{d+1}\right] \leq 2e^{\frac{-1}{3}\epsilon^2|D(s)|\frac{K}{|V|}}$$

Now, we find the value of C_2, such that for $K = C_2 \log |V|$:

$$2e^{\frac{-1}{3}\epsilon^2|D(s)|\frac{K}{|V|}} < \frac{1}{|V|^2}$$

Taking the logarithm of both sides and given that

$\epsilon E[Z] = \frac{K}{(d+1)(d+2)}$ and $E[Z] = |D(s)|\frac{K}{|V|} < \frac{K}{d+2}$, we can obtain $\epsilon > \frac{1}{(d+1)}$, therefore:

$$\frac{1}{3}\epsilon^2|D(s)|\frac{K}{|V|} \geq \frac{1}{3}\frac{1}{(d+1)}\frac{|K|}{(d+2)(d+1)} > 2\ln(2|V|)$$
$$\iff K > 6(d+2)(d+1)^2 \ln(2|V|)$$
$$\xLeftrightarrow{K=C_2 \log|V|} C_2 > \frac{6(d+2)(d+1)^2 \ln(2|V|)}{\log|V|}$$

Thus, C_2 is not more than a constant.

Case 4: We want to prove that if $|D(s)| > \frac{|V|(d+1)}{d+2}$,

then $\Pr\left[count(s,X)\frac{|V|}{K} > \frac{|V|d}{(d+1)}\right] \geq 1 - \frac{1}{|V|^2}$:

Suppose $|D(s)| > \frac{|V|(d+1)}{d+2}$; we prove that $\Pr\left[count(s,X)\frac{|V|}{K} \leq \frac{|V|d}{(d+1)}\right] \geq 1 - \frac{1}{|V|^2}$.

Reminding that $Z = count(s,X)$, if we set $\epsilon = \frac{K}{(d+1)(d+2)E[Z]}$, we show:

$$\Pr\left[count(s,X)\frac{|V|}{K} \leq \frac{|V|d}{(d+1)}\right] \leq \Pr\left[\left|X - |D(s)|\frac{K}{|V|}\right| \geq \epsilon|D(s)|\frac{K}{|V|}\right] \tag{2.7}$$

In order to prove this, given the facts that $\epsilon = \frac{K}{(d+1)(d+2)E[Z]}$, and $D(s) > \frac{|V|(d+1)}{d+2}$, we show that, for any $count(s,X)\frac{|V|}{K}$ such that the inequality $Z = count(s,X) \leq \frac{Kd}{d+1}$ holds, then the inequality $\left[\left|Z - |D(s)|\frac{K}{|V|}\right| \geq \epsilon|D(s)|\frac{K}{|V|}\right]$ also holds.

Given that $Z \leq \frac{Kd}{d+1}$ and that $|D(s)| > \frac{|V|(d+1)}{d+2}$, we can say:

$$\begin{aligned}&\left|Z - |D(s)|\frac{K}{|V|}\right| = \left(|D(s)|\frac{K}{|V|} - Z\right)\\ &> \frac{|V|(d+1)}{d+2}\frac{K}{|V|} - \frac{Kd}{d+1} = \frac{K}{(d+1)(d+2)}\end{aligned}$$

Therefore, using the facts that $\epsilon = \frac{K}{(d+1)(d+2)E[Z]}$ and that $E[Z] = |D(s)|\frac{K}{|V|}$, we have:

$$\left|Z - |D(s)|\frac{K}{|V|}\right| \geq \epsilon|D(s)|\frac{K}{|V|}$$

Thus, inequality 2.7 is true. Combining inequalities 2.3 and 2.7, we can see:

$$\Pr\left[count(s,X)\frac{|V|}{K} \leq \frac{|V|d}{(d+1)}\right] \leq 2e^{\frac{-1}{3}\epsilon^2|D(s)|\frac{K}{|V|}}$$

Now, we find the value of C_2, such that for $K = C_2\log|V|$:

$$2e^{\frac{-1}{3}\epsilon^2|D(s)|\frac{K}{|V|}} < \frac{1}{|V|^2}$$

Taking the logarithm of both sides and given that

$\epsilon E[Z] = \frac{K}{(d+1)(d+2)}$ and $E[Z] = |D(s)|\frac{K}{|V|} \leq K$, we can obtain $\epsilon \geq \frac{1}{(d+1)(d+2)}$, therefore:

$$\frac{1}{3}\epsilon^2|D(s)|\frac{K}{|V|} \geq \frac{1}{3}\frac{1}{(d+1)(d+2)}\frac{|K|}{(d+2)(d+1)} > 2\ln(2|V|)$$
$$\iff K > 6(d+2)^2(d+1)^2\ln(2|V|)$$
$$\xLeftrightarrow{K=C_2\log|V|} C_2 > \frac{6(d+2)^2(d+1)^2\ln(2|V|)}{\log|V|}$$

Thus, C_2 is not more than a constant.

Therefore, it's enough to choose C_2 as the maximum of these 4 constants at the beginning of the algorithm. □

Lemma 2.7. *Any call to* find-splitting-edge *returns true with probability* $\frac{1}{2d}$*; hence Algorithm 1 calls* find-splitting-edge $O(d)$ *times in expectation.*

Proof. By lemma 2.3, we know that if we pick a vertex v, uniformly at random, then with probability $\frac{1}{d}$, an even-edge-separator lies on the path from r to v. We show that if there is such an even-edge-separator (Definition 2.3) on that path, find-splitting-edge(v, Y, V) returns a splitting-edge (Definition 2.4) with probability at least $\frac{1}{2}$, and otherwise returns *Null*.

It is clear that we either return a splitting-edge or *Null* when passing through verify-splitting-edge. We break the probability of returning *Null* according to the phases. We call a vertex v **ineligible** if $count(v, V) < \frac{|V|}{d+2}$ or $count(v, V) > \frac{|V|(d+1)}{d+2}$ ($(parent(v), v)$ is not a splitting-edge). On the other hand, we call vertex v **candidate** if after estimating the number of its descendants: $\frac{K}{d+1} \leq count(v, X) \leq \frac{Kd}{d+1}$. Let (a, b) be an even-edge-separator on path Y.

Phase 1:

- lines 10,11: By definition 2.3, $\frac{|V|}{d} \leq count(b, V) \leq \frac{|V|(d-1)}{d}$. We add $\{r, v\}$ to S in line 6 of the algorithm (the two endpoints of path Y). Notice that $\frac{|V|}{d} \leq count(b, V) \leq count(r, V)$ and that $count(v, V) \leq count(b, V) \leq \frac{|V|(d-1)}{d}$. So, by eq. (2.1), with probability at least $1 - \frac{2}{|V|^2}$, $count(r, X_r) \geq \frac{K}{d+1}$ and $count(v, X_v) \leq \frac{Kd}{d+1}$, and consequently, we don't return *Null* in lines 10,11 of the algorithm.

- line 12: eq. (2.2) shows that an ineligible node is not a candidate with probability $1 - \frac{1}{|V|^2}$. Thus, by a union bound, none of our candidates is ineligible in line 12 with probability at least $1 - \frac{|S|}{|V|^2}$. Moreover, if there exists a candidate s in S, the algorithm outputs a splitting-edge $(parent(s), s)$ with probability at least $1 - \frac{2}{|V|}$, by lemma 2.5.

- lines 15,16: Let us partition S into S_l and S_r (see fig. 2.4), as follows:

$$S_l = \{s \in S \mid count(s, V) > count(b, V)\}, \qquad S_r = \{s \in S \mid count(s, V) < count(b, V)\}$$

Then, by definition of b:

$$\forall s \in S_l : \quad count(s, V) > |V|/d, \qquad \forall s \in S_r : \quad count(s, V) < |V|(d-1)/d.$$

and thus, by eq. (2.1) and a union bound, we have with probability at least $1 - \frac{|S|}{|V|^2}$:

$$\forall s \in S_l : \; count(s, X_s) \geq K/(d+1), \qquad \forall s \in S_r : \; count(s, X_s) \leq Kd/(d+1).$$

Finally, since S does not contain any candidate nodes (otherwise we would have picked them in line 12), the above inequalities imply that:

$$\forall s \in S_l : \; count(s, X_s) > Kd/(d+1), \qquad \forall s \in S_r : \; count(s, X_s) < K/(d+1).$$

Therefore, $w \in S_l$ and $z \in S_r$, which implies that the subpath from w to z in Y must include vertex b. This means that with probability $1 - O(\frac{1}{|V|})$, we either find a splitting-edge in this phase or pass b to the next phase.

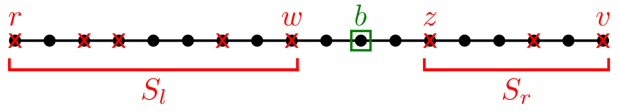

Figure 2.4: Illustration of the path reduction in Phase 1 of find-splitting-edge. At the end of this phase, the path Y is trimmed down into the subpath consisting of the nodes between w and z, which contains b w.h.p.

Now, consider **Phase 2:**

- line 17: Here, if $|Y| > |V|/K$, then we have passed through Phase 1. Using lemma 2.4, we know that since S was a sample of size $C_1\sqrt{|V|}$ from Y, with probability $1 - \frac{1}{|V|}$ the distance between any two consecutive nodes of S in Y was $O(|Y|\frac{\log |V|}{\sqrt{|V|}}) \in O(\sqrt{|V|}\log |V|)$. Thus, the size of path Y after passing through line 15 is at most $O(\sqrt{|V|}\log |V|)$. Thus, the probability of returning *Null* in line 17 is at most $\frac{2}{|V|}$.
- line 21: By eq. (2.2), with probability at least $1 - \frac{|Y|}{|V|^2}$, no ineligible node is between candidate set. Besides, for a candidate node s, the algorithm outputs a splitting-edge $(parent(s), s)$ with probability at least $1 - \frac{2}{|V|}$, by lemma 2.5.
- line 22: The probability that we return *Null* here is equal to the probability that our candidate set in line 21 is empty. By eq. (2.1), with probability at least $1 - \frac{2}{|V|^2}$, b is between candidates at line 21 and candidate set is non-empty. Thus, the total probability of failing to return a splitting-edge in this phase is at most $O(\frac{1}{|V|})$.

Therefore, for $|V|$ greater than the chosen constant g, the probability of returning *Null* in the existence of an even-edge-separator is at most $O(\frac{1}{|V|}) \le 1/2$. Thus, the probability of returning a splitting-edge in any call to find-splitting-edge is at least $\frac{1}{2d}$. □

Lemma 2.8. *The subroutine find-splitting-edge(v, Y, V) has query complexity, $Q(n)$, that is $O(|V|)$, and round complexity, $R(n)$, that is $O(1)$.*

Proof. The queries done by find-splitting-edge(v, Y, V), in the worst case, can be broken down as follows, where $m = O(\sqrt{|V|})$:

Phase 1: A total of $O(|V|)$ queries in $O(1)$ rounds, consisting of:

- $O(mK) \in O(\sqrt{|V|}\log|V|)$ queries in one round for estimating the number of descendants for the m samples.
- $O(m^2) \in O(|V|)$ queries in one round for sorting the m samples.
- $O(|Y|) \in O(|V|)$ queries in a round to find the subpath of Y that is the input for Phase 2.
- $O(|V|)$ queries in $O(1)$ rounds for determining the parent of s (see lemma 2.5).

Phase 2: If we enter this phase, it spends $O(|V|)$ queries in $O(1)$ rounds:

- $|Y| \cdot K \in O(|V|)$ queries in one round to evaluate count(s, X_s) for each $s \in Y$.
- $O(|V|)$ queries in one round to find the number of descendants of node s.
- $O(|V|)$ queries in $O(1)$ rounds to determine the parent of s (see lemma 2.5).

Overall, the above break down amounts to $Q(n) \in O(n)$ and $R(n) \in O(1)$. □

Now, recall theorem 2.1: *Given a set, V, of nodes of a rooted tree, T, such as a biological or digital phylogenetic tree, with degree bounded by a fixed constant, d, we can construct T using path queries with round complexity, $R(n)$, that is $O(\log n)$ and query complexity, $Q(n)$, that is $O(n \log n)$, with high probability.*

Proof. The expected query complexity $Q(n)$ of Algorithm 1 is dominated by the two recursive calls $\left(Q\left(\frac{n}{d+2}\right) \text{ and } Q\left(\frac{n(d+1)}{d+2}\right)\right)$ and the calls to find-splitting-edge. By lemma 2.7, we call find-splitting-edge an expected $O(d)$ times, incurring a cost of $O(dn) \in O(n)$ path queries in $O(d) \in O(1)$ rounds (see lemma 2.8). Thus, $Q(n)$ and $R(n)$ are:

$$Q(n) = Q\left(\frac{n}{d+2}\right) + Q\left(\frac{n(d+1)}{d+2}\right) + O(n),$$

$$R(n) = \max\left(R\left(\frac{n}{d+2}\right), R\left(\frac{n(d+1)}{d+2}\right)\right) + O(1)$$

which shows it needs $Q(n) \in O(n \log n)$ and $R(n) \in O(\log n)$ in expectation. To prove the high probability results, note that the main algorithm is a divide-and-conquer algorithm with two recursive calls per call; hence, it can be modeled with a recursion tree that is a binary tree, B, with height $h = O(\log_{\frac{d+2}{d+1}} n) = O(\log n)$. For any root-to-leaf path in B, the time taken can modeled as a sum of independent random variables, $X = X_1 + X_2 + \cdots + X_h$, where each X_i is the number of calls to find-splitting-edge (each of which uses $O(|V|)$ queries in $O(1)$ rounds) required before it returns true, which is a geometric random variable with parameter $p = \frac{1}{2d}$. Thus, by a Chernoff bound for sums of independent geometric random variables (e.g., see [56, 80]), the probability that X is more than $O(d \log_{\frac{d+2}{d+1}} n)$ is at most $1/n^{c+1}$, for any given constant $c \geq 1$. The theorem follows, then, by a union bound for the n root-to-leaf paths in B. $\square$

2.3.3 Lower Bound

We establish the following simple lower bound, which extends and corrects lower-bound proofs of Wang and Honorio [103].

Theorem 2.2. *Learning an n-node, degree-d tree requires $\Omega(dn + n\log n)$ path queries. This lower bound holds for the worst case of a deterministic algorithm and for the expected value of a randomized algorithm.*

Proof. Consider an n-node, degree-d tree, T, as shown in fig. 2.5, which consists of a root,

r, with d children, each of which is the root of a chain, T_i, of at least one node rooted at a child of r. Since a querier, Bob, can determine the root, r, in $O(n)$ queries anyway, let us assume for the sake of a lower bound that r is known; hence, no additional information is gained by path queries involving the root. Let us denote the vertices in chain T_i as V_i. In order to reconstruct T, Bob must determine the nodes in each V_i and must also determine their order in T_i. For a given path query, $path(u, v)$, say this query is *internal* if $u, v \in V_i$, for some $i \in [1, d]$, and this query is *external* otherwise. Note that even if Bob knows the full structure of T except for a given node, v, he must perform at least $d - 1$ external queries in the worst case, for a deterministic algorithm, or $\Omega(d)$ external queries in expectation, for a randomized algorithm, in order to determine the chain, T_i, to which v belongs. Furthermore, the result of an (internal or external) query, $path(u, v)$, provides no additional information for a vertex w distinct from u and v regarding the set, V_i, to which w belongs. Thus, Bob must perform $\Omega(d)$ external queries for each vertex $v \neq r$, i.e., he must perform $\Omega(dn)$ external queries in total. Moreover, note that the results of external queries involving a vertex, v, provide no information regarding the location of v in its chain, T_i. Even if Bob knows all the vertices that comprise each V_i, he must determine the ordering of these vertices in the chain, T_i, in order to reconstruct T. That is, Bob must *sort* the vertices in V_i using a comparison-based algorithm, where each comparison is an internal query involving two

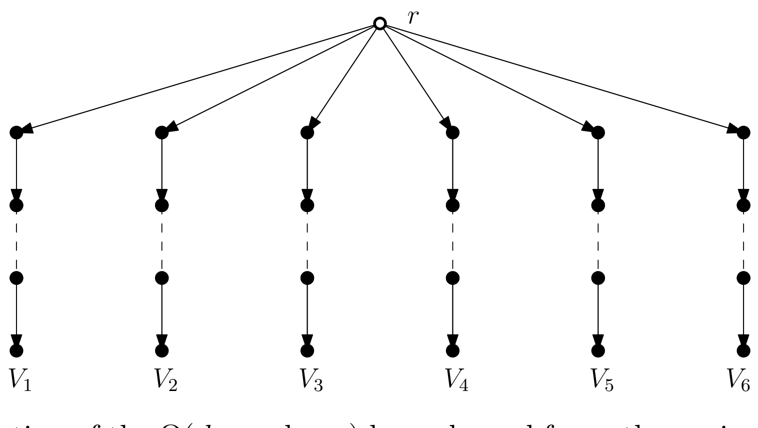

Figure 2.5: Illustration of the $\Omega(dn + n \log n)$ lower bound for path queries in directed rooted trees (shown for $d = 6$).

vertices, $u, v \in V_i$. By well-known sorting lower bounds (which also hold in expectation for randomized algorithms), e.g., see [56, 37], determining the order of the vertices in each T_i requires $\Omega(|V_i| \log |V_i|)$, as one of the chain can be as great as $n - d$ vertices, then he needs $\Omega(n \log n)$ internal queries. □

corollary 2.2. *Algorithm 1 is optimal for bounded-degree trees when asking $\theta(n)$ queries per round.*

The query complexity of Algorithm 1 matches the lower bound provided by theorem 2.2 when d is constant. Besides, we need $\Omega(d + \log n)$ rounds if we have $\theta(n)$ processors; hence, the round complexity of Algorithm 1 is also optimal.

2.4 Experiments

To assess the practical performance of our method for learning (biological and digital) phylogenetic trees from path queries, we performed experiments using both synthetic and real data to compare our algorithm with the algorithm by Wang and Honorio [103], which is the best known reconstruction algorithm for phylogenetic trees from path queries. [2] Our experimental results provide evidence that Algorithm 1 provides significant parallel speedup, while simultaneously improving the total number of queries.

Synthetic Data.

To perform our extensive experimental analysis on synthetic data, we designed a generator of a random degree-d trees using the fact that Prüfer sequences [85] provide a bijection between trees of n vertices and sequences of length $n - 2$ on labels 1 to n. That is, a labeled

[2]The complete source code for our experiments, including the implementation of our algorithm and the algorithms we compared against, is available at github.com/UC-Irvine-Theory/ParallelTreeReconstruction .

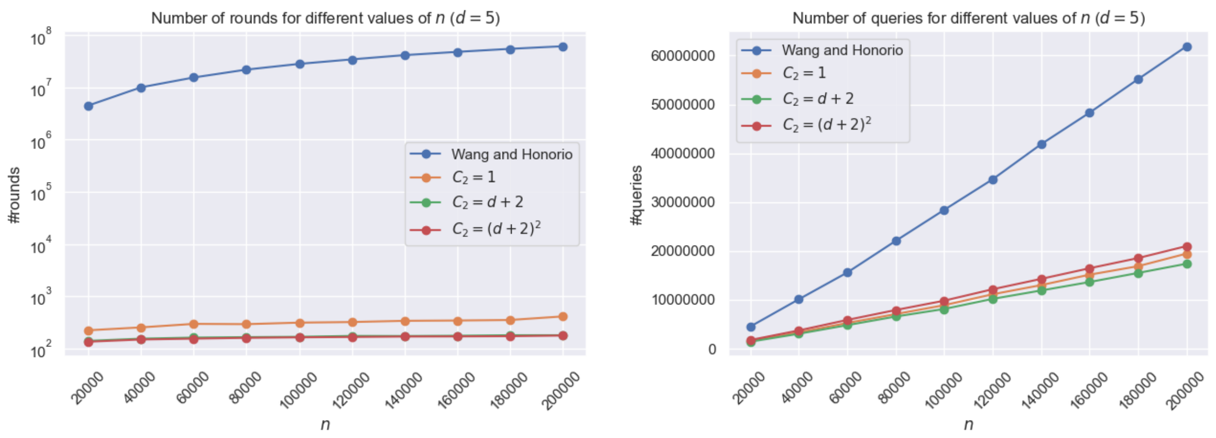

Figure 2.6: Comparing Our Algorithm's number of rounds (left) and total queries (right) with Wang and Honorio's [103], for fixed $d = 5$ and varying n.

tree $T = (V, E)$ with $|V| = n$ is associated with a unique Prüfer sequence $x_1, x_2, \cdots, x_{n-2}$, such that for all $1 \leq i \leq n-2$, $x_i \in V$ and a node of degree k in the tree appears exactly $k-1$ times in the sequence. Therefore, we compiled a data set of trees with various number of vertices, n, and maximum degree, d by recovering the corresponding tree from a random Prüfer sequence generated while simultaneously maintaining that each label appears at most $d-1$ times in the sequence and at least 1 label appears exactly $d-1$ times.

Since our parallel reconstruction algorithm using path queries is parameterized by a constant, C_2, we ran our algorithm using different values for C_2. The constant C_2 controls sample size from V used to estimate the number of descendants of a node. Furthermore, to reduce noise from randomization, each data-point will be averaged for 3 runs on 10 randomly generated trees. In fig. 2.6, we compare our algorithm's rounds and total number of queries with the one by Wang and Honorio [103], for fixed degree trees $d = 5$ and varying tree-sizes. These results provide empirical evidence that our algorithm provides a noticeable speedup in parallel round complexity while also outperforming the algorithm by Wang and Honorio [103] in total number of queries.

In fig. 2.7, we compare Algorithm 1 with the one by Wang and Honorio [103] for fixed size and varying values of d. Again, this supports our theoretical findings that our algorithm

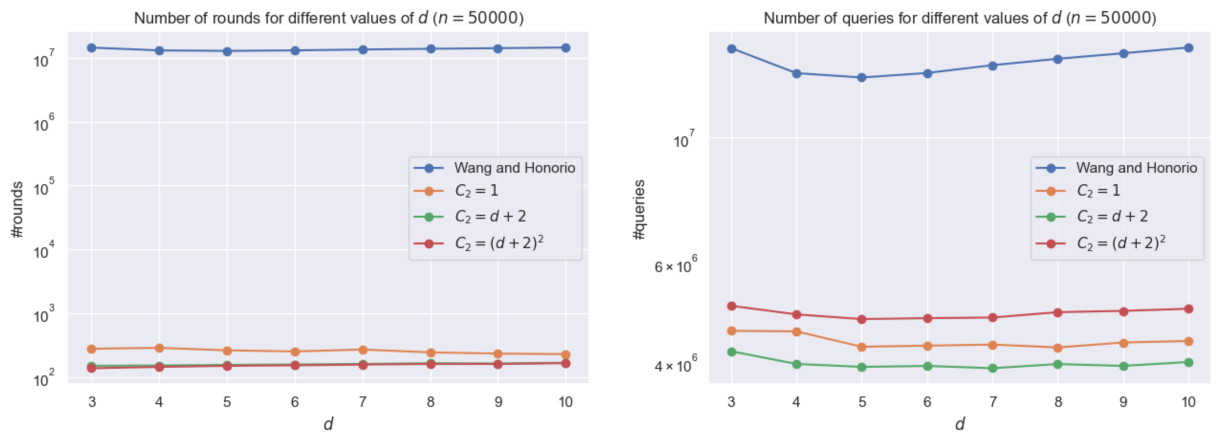

Figure 2.7: Comparing our algorithm's number of rounds (left) and total queries (right) with Wang and Honorio's [103], for $n = 50000$ and varying values for d.

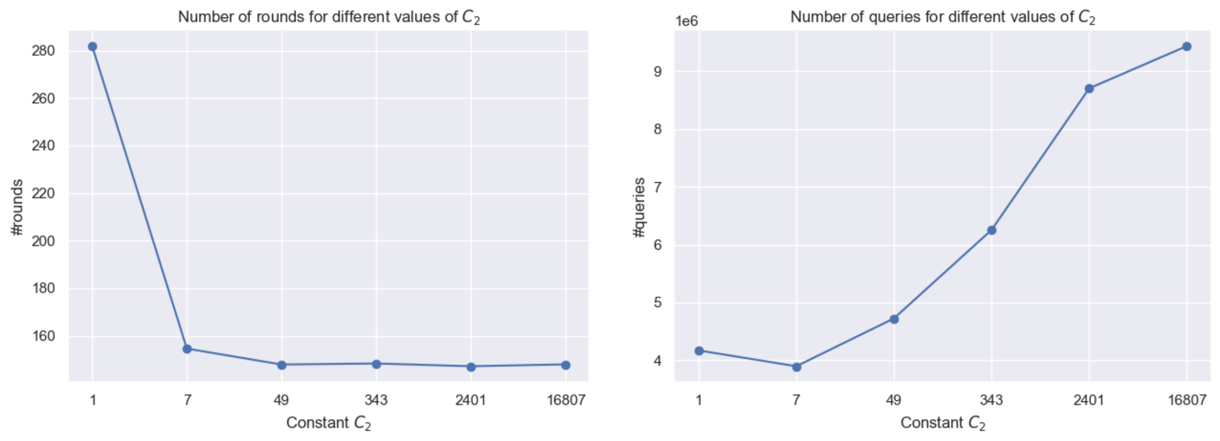

Figure 2.8: Change in the total number of rounds (left) and total number of queries (right) when running our algorithm for varying values of C_2 ($n = 50000$, $d = 5$).

achieves both a significant parallel speedup and a simultaneous improvement in the number of total queries.

In fig. 2.8, we study the behavior of Algorithm 1 under different values of C_2, so as to experimentally find the best value for C_2. While our high probability analysis requires $C_2 \approx (d+2)^4$, fig. 2.8 suggests that we do not need that high probability reassurance in practice, and we can use smaller sample to reduce the total number of queries.

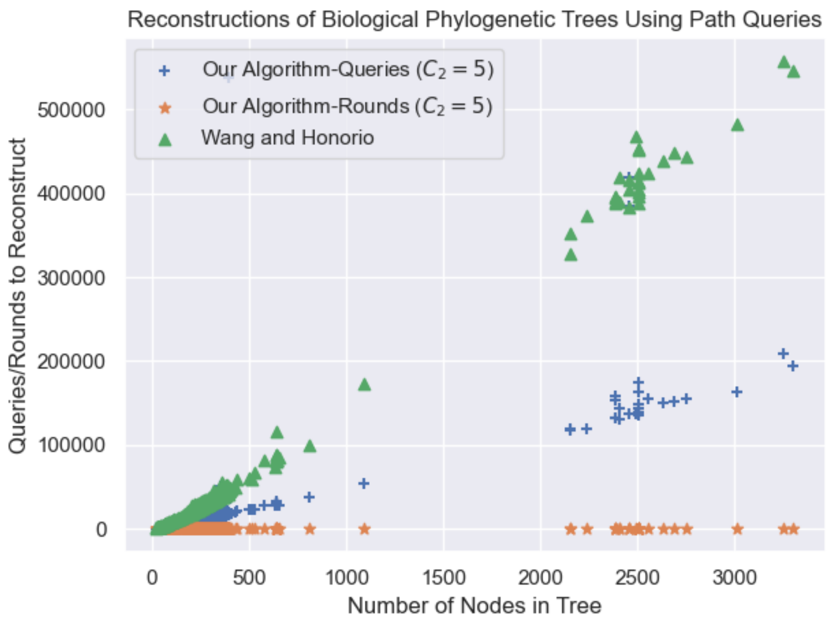

Figure 2.9: A scatter plot comparing the number of queries and rounds of our algorithm and with the one by Wang and Honorio [103] for real-world trees from TreeBase [84]. Since our algorithm is parallel, we include round complexity to serve as a comparison for the sequential complexity.

Real Data.

Our experiments on real-world biological phylogenetic trees also confirm the superiority of our algorithm in terms of performance as compared to the one by Wang and Honorio [103]. We used a dataset of trees from the phylogenetic library TreeBase [84], which includes more than 100000 taxa. Figure 2.9 summarizes our experimental results, where each data point corresponds to an average performance of 3 runs on the same tree. Our algorithm is superior in both queries and rounds for all the values of C_2 we tried: $C_2 \in \{1, d+2, (d+2)^2\}$. The best performance corresponds to $C_2 = d + 2 = 5$, which is the one illustrated in fig. 2.9.

2.5 Conclusion

We have provided an optimal parallel algorithm for learning digital phylogenetic trees using path queries. Our methods assume that the tree has a maximum degree of some constant d, which is a reasonable assumption for these trees. Additionally, we provided a lower bound for this problem, where we showed that there is no non-trivial learning algorithm for learning directed rooted trees without bounding the maximum degree of the tree. We also compile a set of experiments to compare our algorithm and the best known prior work, and we showed that our algorithm not only provides a significant parallel speedup, but it also uses fewer number of queries in total.

Chapter 3

Learning Multitrees and Almost-trees

3.1 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 chapter, 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.

The results of this chapter are motivated by applications in various disciplines of science, such as biology [99, 79, 75, 97], computer science [24, 44, 27, 45, 46, 54, 70, 82], economics [66, 65], psychology [81], and sociology [60]. 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) [44, 27, 45, 46], a text document[77, 92], or a computer virus [54, 82]. In such a network, each node represents an object, and directed edges show how an object has been manipulated or edited from other objects [10]. In a digital phylogenetic network, objects are usually archived and we can issue path queries between a pair of objects (see, e.g., [44]).

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 [44]. Afshar *et al.* [10] studied learning phylogenetic trees (rooted trees) using path queries, where each object is the result of a modification of a single parent, as presented in Chapter 2. 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 [21].

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, `(i,j)`, intervenes the only variable correspondent to i. Therefore, applying our learning methods (similar to the method by Bello and Honorio, see [24]), can avoid an exponential number of experiments [73], and it can improve the results of Bello and Honorio [24] 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 [5, 32, 47, 95] and complexity theory [26, 107].

- 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).

3.1.1 Related Work

The problem of exact learning of a graph using a set of queries has been extensively studied [10, 9, 88, 78, 11, 12, 90, 69, 3, 103, 68, 61, 72, 87]. With regard to previous work on learning directed graphs using path queries, Wang and Honorio [103] 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. As explained in Chapter 2, learning a degree-d rooted tree with n nodes requires $\Omega(nd+n\log n)$ path queries [10] and we 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. That method, however, relies on the fact the ancestor set of a vertex in a rooted tree forms a total order. In section 3.4, we extend that work to learn a rooted spanning tree for a DAG.

Regarding the reconstruction of trees with a specific height, Jagadish and Anindya [68] 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

[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 $c \geq 1$.

a, b, and c, it returns "true" if and only if b is on the path from a to c. Janardhanan and Reyzin [69] 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.

A more general form of this problem is studied in terms of sorting the partially ordered sets (or posets) [28, 31, 43, 52]. Faigle and Turán [52] study the problem of sorting posets and they provide an algorithm with query complexity of $O(wn \log n)$ queries, where n is the number of elements and w is the width of the poset. Daskalakis *et al.* [43] give an algorithm with optimal query complexity of $O(n \log n + nw)$ and a matching lower bound, where each query given a pair of elements, it returns whether the two are not comparable or which element is greater than the other. In this problem, the width, w, is defined as the maximum cardinality antichain of the poset, where an antichain is a subset of mutually incomparable elements. Note that the width of the tree can be very large even when the maximum degree of the corresponding DAG is small, for instance, a binary tree can have a width of $O(n)$ with a maximum degree of 3. Therefore, our results provide an improvement upon those DAGs with fixed maximum degree and large width.

3.1.2 Our Contributions

In Section 3.3, 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 [68]. 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 Section 3.4, 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 [69] achieved a query complexity of $O(n \log^3 n + nh)$ in expectation.

3.2 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 3.1. (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 a multitree, which is a family of DAGs useful in distributed computing [35, 67] that we study in Section 3.3.

Definition 3.2. (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 [86, 36, 55] for which we provide efficient learning method in Section 3.3.

Definition 3.3. (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 Figure 3.1 for illustration).*

Definition 3.4. (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

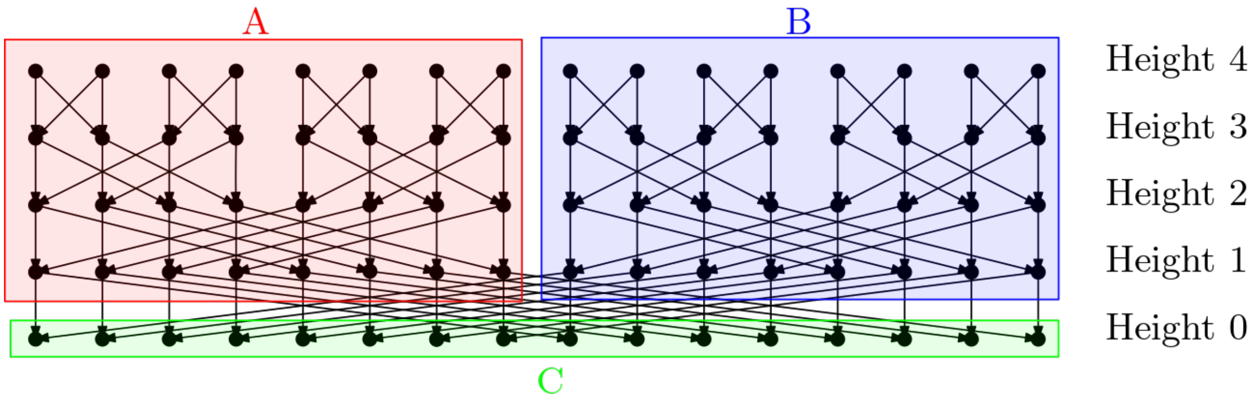

Figure 3.1: An example of a butterfly network with height 4 (Depth 4), F^4, as a composition of two F^3 (A and B) and 2^4 additional vertices, C, in Height 0.

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 3.5. *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 $count(s, X) = \Sigma_{x \in X} path(s, x)$.*

As Wang and Honorio observed [103], 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 3.6. *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 3.7. (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., [14, 19]). 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 [69] to limit almost-trees to have one cross edge. We next introduce even-separator, which will be used in Section 3.4.

Definition 3.8. (even-separator) *Let $G = (V, E)$ be a rooted degree-d DAG. We say that vertex $v \in V$ is an even-separator if $\frac{|V|}{d} \leq count(v, V) \leq \frac{|V|(d-1)}{d}$.*

3.3 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.

3.3.1 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 in Chapter 2. Recall that our approach is to 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 [34, 93, 100] by simulating comparisons using path queries.

Next, we show how to learn the edges, E. Jagadish and Anindya [68] 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 $1 \leq i \leq d$ (see Algorithm 3).

Algorithm 3: Our algorithm to learn trees of height-h

```
1  Function learn-short-tree($V, r$):
2      $E \leftarrow \emptyset$, $V \leftarrow V \setminus \{r\}$
3      for $i \leftarrow 1$ to $d$ do
4          $r_i \leftarrow Null, V_i \leftarrow \emptyset$
5      for $i \leftarrow 1$ to $d$ do
6          if $|V| \geq 1$ then
7              Let $r_i$ be an arbitrary vertex in $V$
8              for $v \in V$ do
9                  if path$(v, r_i) = 1$ then  $r_i \leftarrow v$
10             for $v \in V$ do
11                 if path$(r_i, v) = 1$ then  $V_i \leftarrow V_i \cup \{v\}$
12             $V \leftarrow V \setminus V_i$
13             $E \leftarrow E \cup (r, r_i)$
14             $E \leftarrow E \cup$ small-height-tree-reconstruction$(V_i, r_i)$
15     return $E$
```

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

$$Q(n) = \Sigma_{i=1}^{d} Q(|V_i|) + O(n) \tag{3.1}$$

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

This, in turn, implies the following theorem (3.2) for rooted trees of arbitrary height by employing our method learn-short-tree in an algorithm by Jagadish and Andyia [68], which was introduced for learning undirected trees of large height using separator queries.

Theorem 3.2. *One can deterministically learn a fixed-degree directed rooted tree of arbitrary height using $O(n^{3/2}\sqrt{\log n})$ path queries.*

Proof. Jagadish and Anindya [68, Section 5.2] provided an algorithm to learn undirected trees of arbitrary height with separator queries through the following subroutine: Given a tree T and an arbitrary node set as root r, return a subgraph T' such that for any path such as P, from r to a leaf in T, T' contains at least $n - h$ vertices of P. Once they find T', they use a an algorithm to learn trees of short height for the missing parts on each path. They control h by a controlling parameter, l, where $h = n/l$. Besides, all the queries to find T' are in the form ancestor queries which can be simply simulated by $O(1)$ path queries. Further, we can replace their short height tree learning algorithm with our learn-short-tree algorithm. Their algorithm takes $O(nl \log n)$ to learn T', and $O(nh \log n) \in O(\frac{n^2}{l} \log n)$ queries to learn the missing parts on the paths through their short depth tree learning method. We learn T' using $O(nl \log n)$ path queries since all of their separator quries are in the form of $sep(r, x, y)$ where r is the root, by simulating it with $\mathsf{path}(x, y)$. Since our learn-short-tree method takes

$O(nh) \in O(n^2/l)$ queries, if we set $l = \sqrt{n/\log n}$, this amounts to a method using a total number of $Q(n) \in O(n^{3/2}\sqrt{\log n})$ queries to learn trees of arbitrary height. □

We now show how to adapt a path-querying algorithm to derive an algorithm for learning an undirected fixed-degree tree using separator queries. This will establish improvements upon the results of Jagadish and Anindya [68].

Definition 3.9. *(separator query) On an undirected tree $T = (V, E)$, a separator query is a function, $sep : V \times V \times V \to \{0, 1\}$, such that $sep(a, b, c) = 1$ if removing vertex b disconnects vertex a from vertex c, and $sep(a, b, c) = 0$ otherwise.*

Our separator querying method (learn-undirected-tree) is based on a simple simulation of a path-query algorithm (learn-rooted-tree), and an observation that we can implement path queries using separator queries. Given an undirected tree $T = (V, E')$, we transform it into a rooted directed tree $T = (V, E, r)$ by arbitrarily choosing a vertex, r, as the root of the tree. Then, we orient the edges in E away from r. Given this view, for each path query in our tree-reconstruction algorithm, we note that $path(u, v) = 1$ if and only if $sep(r, u, v) = 1$ (see Figure 3.2). Finally, we report the edges returned in learn-rooted-tree(V, r) with direction removed.

Algorithm 4: Learn an undirected rooted tree with separator queries

```
1 Function learn-undirected-tree($V$):
2     pick a vertex $r$ arbitrarily from $V$ and set it as root.
3     We define the path query $path(u, v)$ according to $sep(r, u, v)$: if $sep(r, u, v) = 1$,
       then $path(u, v) = 1$; otherwise, $path(u, v) = 0$.
4     $E \leftarrow$ learn-rooted-tree$(V, r)$
5     $E' \leftarrow$ edges of $E$ with direction removed
6     return $E'$
```

Theorem 3.3. *Let $T = (V, E)$ be a fixed-degree undirected tree. If T has height h, we can deterministically learn T with $O(nh)$ separator queries, and if it has an arbitrary height, we can learn it with $O(n^{3/2}\sqrt{\log n})$ queries.*

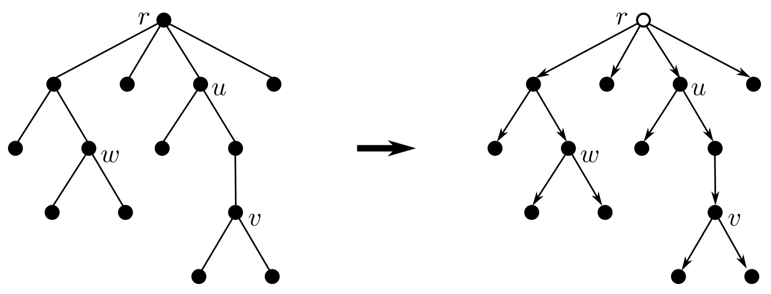

Figure 3.2: The reduction of separator queries (left) to path queries (right). We have that (i) $sep(r, u, v) = 1 \iff path(u, v) = 1$ and (ii) $sep(r, u, w) = 0 \iff path(u, w) = 0$.

Proof. This follows directly from our results in Subsection 3.3.1, which establish the query query complexity of learn-rooted-tree, the subroutine used in Algorithm 4 that dominates the query complexity. □

3.3.2 Multitrees of Arbitrary Height

We next provide a parallel algorithm to learn a multitree of arbitrary height with a number of roots. Recall that Wang and Honorio [103, 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 11, and return a minimal element of Y (see Algorithm 5).

Algorithm 5: Our algorithm to find a root in V

```
   Function learn-root($V$):
1      $m = C_1 * \sqrt{|V|}$
2      Pick an arbitrary vertex $v \in V$
3      for each $u \in V$ do in parallel
4          Perform query $path(u, v)$ to find ancestor set $Y$
5      while $|Y| > m$ do
6          $S \leftarrow$ a random sample of expected size $m$ from $Y$
7          for $(a, b) \in S \times S$ do in parallel
               Perform query $path(a, b)$
8          Pick a vertex $y \in S$ such that for all $a \in S$: $path(a, y) == 0$
9          for $a \in Y$ do in parallel
               Perform query $path(a, y)$ to find ancestors of $y$, $Y'$
10         $Y \leftarrow Y'$
11     for $(a, b) \in Y \times Y$ do in parallel
           Perform query $path(a, b)$
12     $y \leftarrow$ a vertex in $Y$ such that for all $a \in Y$: $path(a, y) == 0$
13     return $y$
```

Before providing the anlaysis of our efficient learn-root method, let us present Lemma 3.1, which is an important lemma throughout our analysis, as it extends Lemma 2.4 to directed acyclic graphs.

Lemma 3.1. *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 $2s$. □

Lemma 3.2. *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 3.1 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 11. 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 $O(m^2)$ queries performed in 1 round in Line 11.

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

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 3.2 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, Algorithm 1, 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, Algorithm 1. 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 $r \leftarrow$ 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 learn-rooted-tree$(r, D(r))$.
5. Let $R = R \setminus D(r)$, and if $|R| > 0$ go to step 2.

Theorem 3.4 analyzes the complexity of our multitree learning algorithm.

Theorem 3.4. *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 Algorithm 1, 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 Algorithm 1 we can establish the high probability bounds. □

3.3.3 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 3.3). 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 (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 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 3.5. *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 $k \le h/2$, then x has at least $2^{\lfloor h/2 \rfloor}$ ancestors in the root, that is, $|A(x) \cap R| \ge 2^{\lfloor h/2 \rfloor}$. Since our sample, S, has an expected size of $2^{h/2} \cdot ch$, the expected size of $|S \cap A(x) \cap R| \ge ch/2$. Using a standard Chernoff bound, the probability, $Pr[|S \cap A(x) \cap R| = 0] \le e^{-ch/4}$. 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, $|D(y) \cap L| \ge 2^{\lfloor h/2 \rfloor}$. Since, our sample T, has an expected size of $2^{h/2} \cdot ch$, the expected size of $|T \cap D(y) \cap L| \ge ch/2$. Using a standard Chernoff bound, the probability, $Pr[|T \cap D(y) \cap L| = 0] \le e^{-ch/4}$. 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. □

3.4 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 5, 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 3.2. 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 $\Omega(n \log n + nh)$.

3.4.1 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 Lemma 2.2 for DAGs.

Theorem 3.6. *Every degree-d rooted DAG, $G = (V, E)$, has an even-separator (see Definition 3.8).*

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. $\square$

Next, we introduce Lemma 3.3 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 3.3. *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 3.6, 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.

$\square$

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 3.10. *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, we used a similar divide and conquer approach to learn directed rooted trees in Chapter 2, but our 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 $\frac{|V|}{d} \leq |V_1| \leq \frac{|V|(d-1)}{d}$, 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 Algorithm 6). 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.

Algorithm 6: learn a spanning tree in a DAG

```
Function learn-spanning-tree($V, r$):
1    $E \leftarrow \emptyset$
2    if $|V| \leq g$ then // $g$ is a chosen constant
3        return edges found by a quadratic brute-force algorithm
4    while true do
5        Pick a vertex $v \in V$ uniformly at random
6        for $z \in V$ do in parallel
             Perform query $path(z, v)$ to find $Y = A(v) \cap V$
7        w $\leftarrow$ learn-separator($v, Y, V, r$)
8        for $z \in V$ do in parallel
             Perform query $path(w, z)$
9        split $V$ into $V_1, V_2$ using query results
10       if $w \neq Null$ and $\frac{|V|}{d+2} \leq |V_1| \leq \frac{|V|(d+1)}{d+2}$ then
11           $u \leftarrow$ learn-parent($w, V$)
12           $E \leftarrow E \cup \{(u, w)\}$
13           parallel do
14               $E \leftarrow E \cup$ learn-spanning-tree($V_1, w$)
15               $E \leftarrow E \cup$ learn-spanning-tree($V_2, r$)
16           return $E$
```

Next, we show how to adapt an algorithm to learn a near-separator for DAGs by extending Algorithm 2. 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 (see Algorithm 7).

Algorithm 7: For a vertex v, find a separator among $Y = A(v) \cap V$

```
   Function learn-separator($v, Y, V, r$):
1      $m = C_1\sqrt{|V|}$, $K = C_2 \log |V|$
       Phase 1:
2          if $|Y| > |V|/K$ then
3              $S \leftarrow$ subset of $m$ random elements from $Y$
4              $S \leftarrow S \cup \{v, r\}$
5              for each $s \in S$ do in parallel
6                  $X_s \leftarrow$ subset of $K$ random elements from $V$
7                  Perform queries to find $count(s, X_s)$
8              if $\forall s \in S : count(s, X_s) < \frac{K}{d+1}$ then return Null
9              if $\forall s \in S : count(s, X_s) > \frac{Kd}{d+1}$ then return Null
10             if $\exists s \in S : \frac{K}{d+1} \leq count(s, X_s) \leq \frac{Kd}{d+1}$ then return $s$
11             $Y \leftarrow$ filter-separator$(S, Y, V)$
       Phase 2:
12         for each $s \in Y$ do in parallel
13             $X_s \leftarrow$ subset of $K$ random elements from $V$
14             Perform queries to find $count(s, X_s)$
15         if $\exists s \in Y$ s.t. $\frac{K}{d+1} \leq count(s, X_s) \leq \frac{Kd}{d+1}$ then return $s$
16 return Null
```

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 $1 \leq i \leq c$, 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 8.

Algorithm 8: 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 $v$.
4      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 $count(b, X_b) < \frac{K}{d+1}$.
5      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 $count(b, X_b) > \frac{K \cdot d}{d+1}$.
6      for $1 \leq i \leq c$ and $v \in V$ do in parallel
7          perform query $path(v, g_i)$ to find $(A(g_i) \cap V)$.
8          Remove $(A(g_i) \cap V)$ from $Y$.
9          perform query $path(l_i, v)$ to find $(D(l_i) \cap V)$.
10         Remove $(D(l_i) \cap V)$ from $Y$.
11 return $Y$
```

Lemma 3.4 shows that our filter-separator method efficiently in parallel eliminates the nodes that are unlikely to act as a separator.

Lemma 3.4. *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 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. We first prove that the size of the set returned by our filter-separator method is $O(c \cdot |Y| \log |V| / \sqrt{|V|})$. We run Line 11 of our learn-separator method only if we do not return in Lines 8, 9, and 10; hence, for every vertex $s \in S$, we should have that $count(s, X_s) > \frac{Kd}{d+1}$ or $count(s, X_s) < \frac{K}{d+1}$ and there should be nodes $\{x, y\} \subseteq S$ such that $count(x, X_x) > \frac{Kd}{d+1}$ and $count(y, X_y) < \frac{K}{d+1}$.

Consider an arbitrary path, P_i, among these c paths from r to v. We argue that filter-separator returns at most $O(|Y| \log |V| / \sqrt{|V|})$ vertices of P_i. By Lemma 3.1, with probability at least $1 - \frac{1}{|V|^2}$, the distance between any two consecutive vertices of $P_i \cap S$ is at most

$O(|Y|\log|V|/\sqrt{|V|})$, and if $|P_i| > 4|Y|\log|V|/\sqrt{|V|}$, then $|P_i \cap S| \geq 4$. Recall that $l_i \in (P_i \cap S)$ was the oldest node having $count(l_i, X_{l_i}) < \frac{K}{d+1}$ (there is no node $b \in (S \cap A(l_i))$ having $count(b, X_b) < \frac{K}{d+1}$). Also, recall that $g_i \in (P_i \cap S)$ was the youngest node having $count(g_i, X_{g_i}) > \frac{Kd}{d+1}$ (there is no node $b \in (S \cap D(g_i))$ having $count(b, X_b) > \frac{Kd}{d+1}$). Since we remove ancestors of g_i, and descendants of l_i from P_i, our filter-separator algorithm returns at most vertices between two consecutive vertices of $P_i \cap S$, having a size of $O(|Y|\log|V|/\sqrt{|V|})$. Since, we have at most c paths, therefore the size of the set returned by this algorithm is at most $O(c \cdot |Y|\log|V|/\sqrt{|V|})$.

Let e be an even-separator in Y. Next, we prove that the returned set includes e. As we showed inn Chapter 2, there exists a constant $C_2 > 0$, as used in Line 1 of our learn-separator algorithm such that the following probability bound hold:

$$\begin{cases} \Pr\left(count(s, X_s) \geq \frac{K}{d+1}\right) \geq 1 - \frac{1}{|V|^2} & \text{if } count(s, V) \geq \frac{|V|}{d}, \\ \Pr\left(count(s, X_s) \leq K\frac{d}{d+1}\right) \geq 1 - \frac{1}{|V|^2} & \text{if } count(s, V) \leq |V|\frac{d-1}{d}, \end{cases} \tag{3.2}$$

$$\begin{cases} \Pr\left(count(s, X_s) < \frac{K}{d+1}\right) \geq 1 - \frac{1}{|V|^2} & \text{if } count(s, V) < \frac{|V|}{d+2}, \\ \Pr\left(count(s, X_s) > K\frac{d}{d+1}\right) \geq 1 - \frac{1}{|V|^2} & \text{if } count(s, V) > |V|\frac{d+1}{d+2} \end{cases} \tag{3.3}$$

Let $s \in S$ be an arbitrary ancestor of e. Hence, $count(s, V) \geq count(e, V) \geq \frac{|V|}{d}$. By Equation 3.2, with probability at least $1 - \frac{1}{|V|^2}$, $count(s, X_s) \geq \frac{K}{d+1}$. Hence, s cannot be equal with l_i, for $1 \leq i \leq c$, and therefore, we do not remove descendants of s. Similarly, for an arbitrary descendant, $s \in S$, of e, $count(s, V) \leq count(e, V) \leq \frac{|V|(d-1)}{d}$. By Equation 3.2, with probability at least $1 - \frac{1}{|V|^2}$, $count(s, X_s) \leq \frac{Kd}{d+1}$. Hence, s cannot be equal with g_i, for $1 \leq i \leq c$, and therefore, we do not remove ancestors of s. Therefore, using a union bound, with probability at least $1 - \frac{|S|}{|V|^2}$, we do not remove e from Y.

All together, using a union bound with probability at least $1 - \frac{|S|+1}{|V|^2}$, the call to our filter-separator in Line 11 of our learn-separator algorithm returns a set of size $O(c \cdot |Y| \log |V| / \sqrt{|V|})$ without filtering an even-separator. □

Lemma 3.5 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 3.5. *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. We will show that the probability of returning *Null* or returning a vertex that is not a near-separator is at most $\frac{1}{|V|}$. Let $e \in Y$ be an even-separator. We evaluate this probability according to different lines of the algorithm.

- Lines 8, 9: Since e is an even-separator, we have $\frac{|V|}{d} \leq \mathit{count}(e, V) \leq \frac{|V|(d-1)}{d}$. On the other hand, as $r \in A(v)$ we have that $\mathit{count}(r, V) \geq \mathit{count}(e, V) \geq \frac{|V|}{d}$. Also, since $v \in D(e)$, we can establish that $\mathit{count}(v, V) \leq \mathit{count}(e, V) \leq \frac{|V|(d-1)}{d}$. Hence, by Equation 3.2 and a union bound, with probability at least $1 - \frac{2}{|V|^2}$, we have that $\mathit{count}(r, X_r) \geq \frac{K}{d+1}$, and $\mathit{count}(v, X_v) \leq \frac{Kd}{d+1}$, and therefore we do not return *Null* in Lines 8, 9.

- Line 10: Consider a vertex $s \in S$ with $\mathit{count}(s, V) < \frac{|V|}{d+2}$. By Equation 3.3, with probability at least $1 - \frac{1}{|V|^2}$, $\mathit{count}(s, X_s) < \frac{K}{d+1}$, and therefore, s is not picked in Line 10 as a near-separator. Similarly, for vertex $s \in S$ with $\mathit{count}(s, V) > \frac{|V|(d+1)}{d+2}$, by Equation 3.3, with probability at least $1 - \frac{1}{|V|^2}$, $\mathit{count}(s, X_s) > \frac{Kd}{d+1}$, indicating that we do not pick s as a near-separator Line 10. Hence, using a union bound with probability at least $1 - \frac{|S|}{|V|^2}$, the returned vertex in Line 10 is a near-separator.

- Line 15: Recall that in Lemma 3.4, we proved that with probability at least $1-\frac{|S|+1}{|V|^2}$, our call to filter-separator method in Line 11 of our learn-separator algorithm returns a set including an even-separator, e, and the set size of $O\left(c \cdot |Y| \frac{\log |V|}{\sqrt{|V|}}\right) \in O(\sqrt{|V|}\log |V|)$. Using an argument similar to the one for Line 10, we can show that with probability at least $1 - \frac{O(\sqrt{|V|}\log |V|)}{|V|^2}$, if we return a vertex in Line 15, it will be near-separator.

- Line 16: Note that the set returned by filter-separator method includes e. By Equation 3.2, with probability at least $1 - \frac{2}{|V|^2}$, $\frac{K}{d+1} \le count(e, X_e) \le \frac{Kd}{d+1}$ in Line 15, and therefore, we do not get to run Line 16.

Therefore, using a union bound, we can show that if Y has an even separator, with probability at least $1 - \frac{2+|S|+|S|+1+O(\sqrt{|V|}\log |V|)+2}{|V|^2} \ge 1 - \frac{1}{|V|}$, our learn-separator method returns a near-separator.

□

Lemma 3.6. *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.

□

Theorem 3.7. *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. By Lemma 3.6, each call to learn-separator method takes at most $O(c|V|) \in O(|V|)$ queries in $O(1)$ rounds, and using Lemma 3.3 and Lemma 3.5, it returns a near-separator with probability at least $\frac{1}{d} \cdot \left(1 - \frac{1}{|V|}\right)$. Then, it learns an edge through a call to learn-parent method using $O(|V|)$ queries in $O(1)$ rounds with probability at least $1 - 1/|V|$. Hence, for $|V| \geq 4d$, using a union bound, with probability at least $\frac{1}{2d}$, we have that our near-separator splits the DAG with vertices V into two DAGs of size at least $\frac{|V|}{d+2}$. Hence, we have

$$Q(n) = Q\left(\frac{n}{d+2}\right) + Q\left(\frac{n \cdot (d+1)}{d+2}\right) + O(n)$$

$$R(n) = R\left(\frac{n}{d+2}\right) + R\left(\frac{n \cdot (d+1)}{d+2}\right) + O(1)$$

Therefore, we have $Q(n) \in O(n \log n)$ and $R(n) \in O(\log n)$ in expectation. We can also use a Chernoff bound for sum of the independent geometric random variables (see [56, 80]) to prove the bounds with high probability similar to the work done in Theorem 2.1. □

3.4.2 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).

Algorithm 9: 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
       $path(c, t) = 1$
6     for $s \in D(c)$ do in parallel
7         Perform query $path(s, t)$
8     Let $s$ be the node with minimum height having $path(s, t) = 1$.
9     return $(s, t)$
```

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 9.

The following lemma shows that Algorithm 9 correctly learns the cross-edge using $O(nh)$ queries in just 2 rounds.

Lemma 3.7. *Given an arborescence with vertex set V, and edge set, E, of an almost-tree, Algorithm 9 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 7.

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 7.

□

Theorem 3.8. *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 3.2, 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 3.7, 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 3.7 we can deterministically learn a cross-edge using $O(nh)$ queries in just 2 rounds. □

3.4.3 Lower bound

The following lower bound improves the one by Janardhanan and Reyzin [69] and proves that our algorithm to learn almost-trees in optimal.

Theorem 3.9. *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 [69], but we improve their bound using an information-theoretic argument. Let G be an almost-tree of

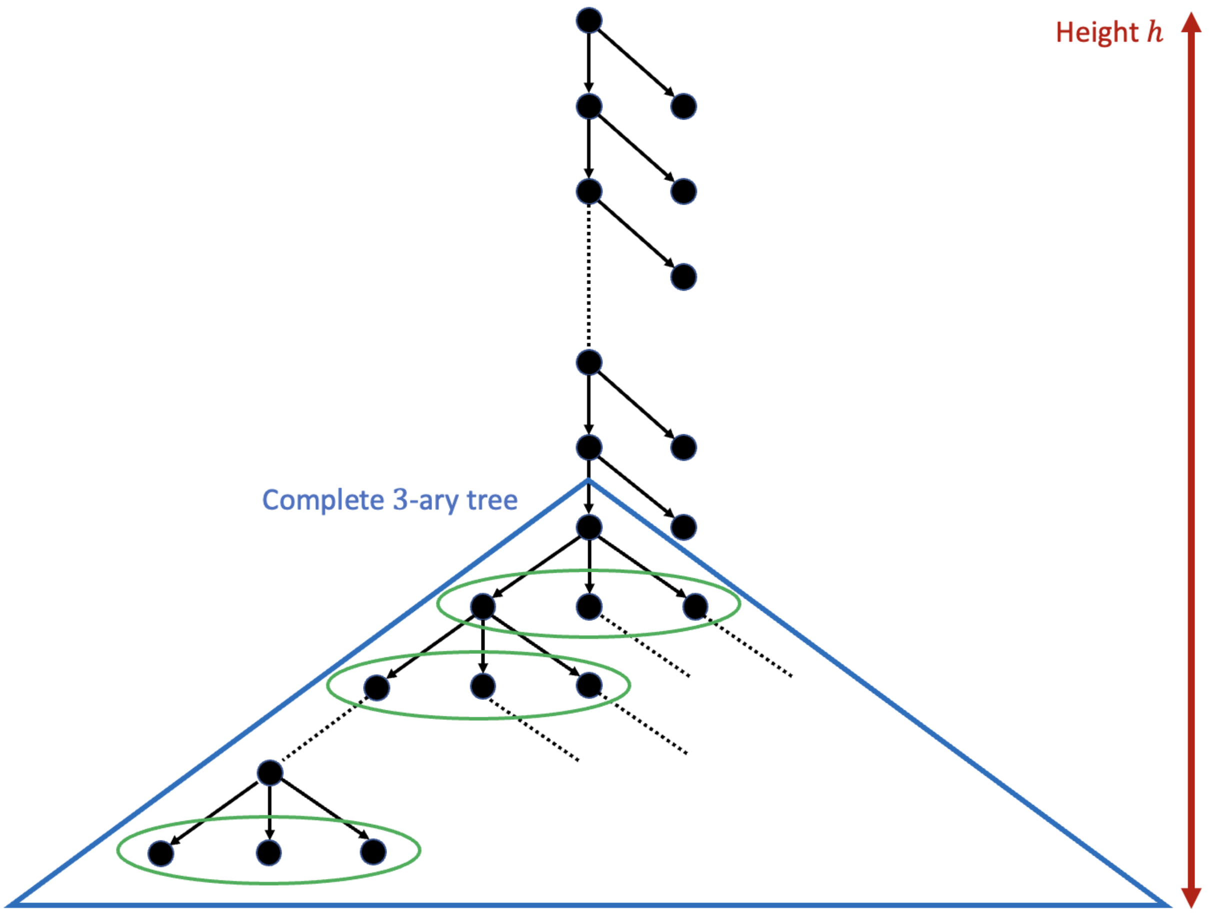

Figure 3.3: An example of a complete 3-ary tree attached to the last level of a caterpillar graph of height $\Theta(h)$.

height h consisting of a caterpillar graph with height $\Theta(h)$, 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 d-ary tree, the number of possible d-ary trees will be at least $\frac{m!}{(d!)^{m/d}}$. Therefore, 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. □

3.5 Conclusion

In this chapter, we extended the results of Chapter 2 to other directed acyclic graphs. In particular, we provided efficient algorithms for learning multitrees, butterfly networks, and almost-trees using path queries. While our almost-tree learning algorithm provides an optimal solution for almost-trees with only one additional cross-edge, one research direction for future work is to study efficient algorithms for learning an almost-tree with constant number of cross-edges. On the other hand, some of the functionalities that we provided in this chapter may be used for learning other almost-trees. For instance, our learn-spanning-tree can efficiently learn a spanning tree for an almost-tree as long as there are at most c directed path from root to any vertex.

Chapter 4

Learning Connected Graphs

4.1 Introduction

Network mapping involves inferring the topology of a communication network, such as the Internet, from queries, e.g., see Figure 4.1 and [50, 109]. A prominent technique for network mapping is *active probing* using the Unix `traceroute` command to perform queries that reveal routing-path information, e.g., see [50, 63, 109].

We formulate the *network mapping* problem as follows. Suppose we are given access to a subset, $U \subseteq V$, of the vertices of a connected, undirected, unweighted graph, $G = (V, E)$, so that the *distance*, $\delta(u, v)$, between two vertices, u and v, in G is defined as the number of edges on a shortest path joining u and v in G. The n vertices in U are known, but the set of edges, E, is unknown. The subset U represents *vantage point* nodes from which we may issue the following type of queries:

- $\mathsf{kth\text{-}hop}(k, u, v)$: return the vertex, w, that is the kth vertex on a shortest path from u to v in G. If $k \geq \delta(u, v)$, then return v.

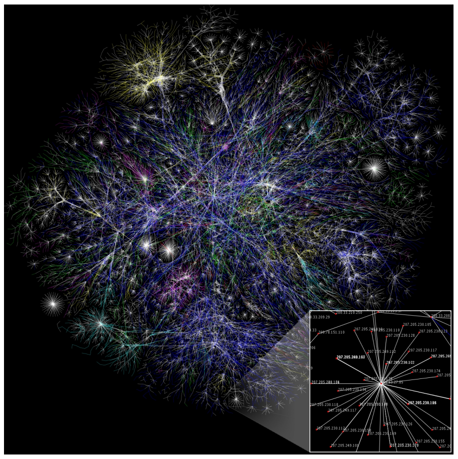

Figure 4.1: Partial map of the Internet circa 2005. Image by The Opte Project, unchanged and licensed under the Creative Commons Attribution 2.5 Generic license.

Note that for $u, v \in U$, kth-hop(k, u, v) returns vertices in a single shortest path from u to v. Shortest paths in G are not necessarily unique, however. So, for example, if $\delta(u, v) = \delta(u, w) + \delta(w, v)$, it is not necessarily the case that kth-hop$(\delta(u, w), u, v) = w$. In the network mapping problem, we are interested in using kth-hop queries to learn the edges of the *induced shortest-path graph*, $H = (U, \tilde{E})$, such that there is an edge $(u, v) \in \tilde{E}$, for $u, v \in U$, if and only if no kth-hop(k, u, v) query would return a vertex $w \in U$ other than v, that is, kth-hop(k, u, v) would return vertices of a shortest path from u to v that does not include any other vertex in U. Thus, H is a weighted, connected, undirected graph such that each edge (u, v) in H has weight $\delta(u, v)$.

Our motivation for focusing on kth-hop queries is that they form the "inner loop" of how `traceroute` works by default. In particular, by default `traceroute` works by sending a series of packets in a network from a source, u, to a destination, v, with the packets having increasing time-to-live (TTL) values, up to an upper bound for the diameter, diam(G), of G, which `traceroute` typically sets to 30 or 64 by default depending on the underlying operating system. The TTL field in a packet is decremented with each hop it traverses and when it reaches 1, then that node sends an ICMP message to the source address (with message including the node's address), e.g., see [2, 1]. Thus, the `traceroute` tool can be viewed as first performing a kth-hop$(1, u, v)$ query, then a kth-hop$(2, u, v)$ query, and so on, until getting a response from the vertex v. In fact, one can use options with the `traceroute` command to issue a `kth-hop` query directly, e.g., to find the 5th hop from a node to `example.com`, one could use the command, "`traceroute -m 5 -M 5 example.com`".

Our formulation of the network mapping problem abstracts away certain system issues. In particular, we are implicitly assuming that messages in G are routed along shortest paths, which is a widely used setting assumed by the prior work [4, 53, 40]. An important system issue that we do not abstract away, however, is that only vertices in $U \subseteq V$ may issue queries. Indeed, there is some interesting prior work regarding the sampling biases introduced by only being able to issue queries from a subset, U, of the set of vertices, V, in G. For example, Achlioptas, Clauset, Kempe, and Moore [4] show that `traceroute` sampling[1] finds power-law degree distributions in both Δ-regular and Poisson-distributed random graphs, even though these underlying graphs do not themselves have power-law degree distributions, which is a statistical finding in experiments by Lakhina, Byers, Crovella, and Xie [76]. Maciej, Markopoulou, and Patrick [74] study ways to correct for this bias when samping large graphs. Further, Zhang, Kolaczyk, and Spencer [110] and Flaxman and Vera [53] study ways to correct for this bias for estimating degree distributions. Interestingly, Barrat, Alvarez-Hamelin, Dall'Asta, Vázquez, and Vespignani [20] provide an analysis that power laws still exist in the Internet graph in spite of the `traceroute` sampling bias, which these authors show is related to betweenness (see also [40]).

In spite of this interesting prior work concerning the sampling biases inherent in performing `traceroute` queries only from the nodes in the subset, U, we are not familiar with any prior work on efficient algorithms for solving the network mapping problem. We focus on two complexity measures for a network mapping algorithm, $\mathcal{A}$, in terms of $n = |U|$:

- $Q(n)$: the *query complexity* of $\mathcal{A}$. This is the total number of `kth-hop` queries issued.

[1] *Traceroute sampling* samples the network graph as the union of paths that packets traverse in performing `traceroute` queries from a subset of the nodes in a network.

This complexity measure comes from learning theory (e.g., see [6, 33, 48, 94]) and complexity theory (where it is also known as "decision-tree complexity," e.g., see [108, 25]).

- $R(n)$: the *round complexity* of $\mathcal{A}$. This is the number of rounds of querying performed by $\mathcal{A}$, where the queries issued in a round are given in a batch such that any query issued in a round may not depend on the response to any other query in that round (but each query may depend on results of queries from previous rounds).

4.1.1 Prior Related Work

As mentioned above, we are not aware of prior algorithmic work on network mapping. If we analyze the algorithm used in existing mapping systems that use active probing, this amounts to a brute-force quadratic algorithm implemented by cooperating nodes of the network, which perform a `traceroute` to every other known node in the network, e.g., see [50, 49, 64]. Viewed combinatorially, this algorithm has query complexity, $Q(n)$, that is $O(\text{diam}(G) \cdot n^2)$, and round complexity, $R(n)$, that is $O(\text{diam}(G))$, for kth-hop queries.

The network mapping problem is related to *graph reconstruction*, e.g., see [71, 78, 3, 22, 91, 15, 16, 17, 23, 18, 29, 57, 58, 59, 87, 105, 61, 72, 41, 96, 9, 104, 33, 10, 89, 13, 8]. In this problem, one is given a connected unweighted graph, $G = (V, E)$, for which V is known and goal is to discover E through queries, such as:

- distance(u, v): return the distance, $\delta(u, v)$, between u to v in G.
- shortest-path(u, v): return the vertices (in order) in a shortest path from u to v in G.

There is also work on other types of queries, including vertex-betweenness queries [3]; queries returning whether a given subset of vertices induce a given edge [23, 16, 15, 29, 17, 18];

queries returning the number of edges induced by a given subset of vertices [58, 57, 59, 33]; queries returning *all* shortest paths from a given node to all other nodes [22, 91]; queries returning the distance between two leaves in a phylogenetic tree [9, 10, 105, 61, 72, 87]; and queries returning whether a given vertex is an ancestor of another given vertex in a rooted tree [104, 9, 10].

There are a number of important differences between the network mapping problem and graph reconstruction, however. Most significantly, the graph reconstruction problem assumes queries can be performed for any vertices in V, whereas in the network mapping problem we may only issue kth-hop queries for nodes in the subset $U \subseteq V$. In addition, even if we restrict the network mapping problem to the case where $U = V$, previous work on graph reconstruction has not considered kth-hop queries, which, as we mentioned above, form the "inner-loop" for how `traceroute` works and are distinct from distance and shortest-path queries. For example, it doesn't seem possible to simulate a kth-hop query with fewer than $\Theta(n)$ distance queries, while a distance query can be simulated with $O(\log \operatorname{diam}(G))$ kth-hop queries via binary search. Also, although it is trivial to simulate a kth-hop query with a single shortest-path query, it takes $\Theta(\operatorname{diam}(G))$ kth-hop queries to simulate a single shortest-path query. Thus, kth-hop queries are strictly weaker than shortest-path queries while being better at capturing the true message complexity of the `traceroute` command.

Another difference between the network mapping problem and graph reconstruction is that previous work on graph reconstruction has mostly focused on how to sequentially reconstruct the graph, G, whereas the network mapping problem is inherently parallel, due to the motivation from mapping real-world networks, where each node is a computer. In terms of previous work on graph reconstruction in parallel, Mathieu and Zhou [78] recently provided a simple algorithm to reconstruct a connected, unweighted graph G, using an expected number of $\tilde{O}(N^{5/3})$ distance queries in 2 rounds.[2] They also show that their algorithm takes an

[2]The notation $\tilde{O}(f(N))$ is equivalent with $O(f(N) \cdot polylog(f(N)))$.

expected number of $\tilde{O}(N)$ distance queries to reconstruct a random Δ-regular graphs.

The most relevant prior work on graph reconstruction, however, is by Kannan, Mathieu, and Zhou [71], who show how to reconstruct a connected, unweighted graph, G, using an expected number of $O(\Delta^3 N^{3/2} \log^2 N \log\log N)$ distance queries, or an expected number of $N^{1+O(\tau(N))}$ shortest-path queries, where $N = |V|$ and $\tau(N) = \sqrt{(\log\log N + \log \Delta)/\log N}$, which is $o(1)$ when Δ, the maximum degree of G, is $N^{o(1)}$. They also show that verifying a given set of edges can be done using $O(N^{1+O(\tau(N))})$ expected distance queries.

4.1.2 Our Results

A preliminary announcement of some of this chapter, using distance queries for graph reconstruction, where queries can be performed for any vertices in V, was presented in [11].

In Section 4.2, we introduce a new technique that may be of independent interest, where we provide a new parallel implementation of a well-known graph clustering technique of Thorup and Zwick [98] with round complexity of $O(1)$, while their original implementation implies an expected round complexity of $O(\log n)$. In doing so, we introduce a parameter that allows us to trade off parallel time and cluster size. Moreover, we show that our complexity bounds hold with high probability,[3] whereas Thorup and Zwick proved their complexity bounds only in expectation. In Section 4.3, we will use this new construction to compute a graph-theoretic Voronoi diagram in our network mapping algorithm. On the other hand, our graph clustering technique can be applied to other problems, such as that studied by Honiden, Houle, and Sommer [62] for balancing graph-theoretic Voronoi diagrams, to reduce the number of centers to $O(s)$ from $O(s \log n)$.

In Section 4.3, we provide the first non-trivial algorithmic results for the network mapping

[3]We say an event holds *with high probability* (w.h.p.) if it occurs with probability at least $1 - 1/n^c$, for some constant $c \geq 1$.

problem. Our query complexities and round complexities are characterized in terms of $n = |U|$ and some interesting parameters that capture the sampling coverage provided by the set U. For example, in addition to characterizing complexities in terms of Δ, the maximum degree of the graph, H, we introduce a distance coverage parameter, $\delta_{\max}$, which is the maximum weight for an edge in H, and a nearby-vertices parameter, μ, which is an upper bound on the number of vertices within a distance of $2\delta_{\max}$ of any given vertex $v \in U$. As we show, these parameters are required for the sake of efficiency, for we show that without these parameters the network mapping problem has a quadratic query-complexity lower bound. For example, under reasonable assumptions regarding these parameters, we are the first to give a constant-round network-mapping algorithm with query complexity better than the trivial brute-force algorithm.

In Section 4.4, we introduce a greedy approach for network mapping that is based on parallel greedy approximate set cover, which allows us to achieve a near-quasilinear query complexity (when Δ is $n^{o(1)}$). As with a related sequential greedy graph reconstruction result of Kannan, Mathieu, and Zhou [71], our query and round complexity bounds are parameterized in terms of the best sequential query complexity for verifying the edges of a graph using distance queries (without knowing the exact value of this query complexity). Further, for small values of the parameters, $\delta_{\max}$ and Δ, our greedy approach uses a near-quasilinear number of kth-hop queries, which are strictly weaker than the shortest-path queries used by Kannan, Mathieu, and Zhou. We summarize our results in Table 4.1.

Table 4.1: Our w.h.p. bounds for the network mapping problem, where ϵ denotes a fixed constant, $0 < \epsilon < 1/2$, $n = |U|$ and Δ, $\delta_{\max}$, μ, and $\tau(\cdot)$ are as defined above.

$R(n)$	$Q(n)$
$O(1)$	$O(\delta_{\max}\, \mu\, n^{3/2+\epsilon})$
$O(\log n \cdot \log \operatorname{diam}(G))$	$O(\mu\, n^{3/2} \log^{3/2} n \cdot \log \operatorname{diam}(G))$
if $U \subset V$: $O(\Delta n)$	$\operatorname{diam}(G) \cdot n^{1+O(\tau(n))}$
if $U = V$: $O(\Delta n \log n)$	$n^{1+O(\tau(n))}$

4.2 Parallel Graph Clustering

Thorup and Zwick [98] introduced a graph clustering technique in presenting a stretch[4] 3 network routing scheme. We begin by describing our parallel graph clustering algorithm, which may be of independent interest, as it provides a parameterized parallel extension of the one by Thorup and Zwick [98]. Also, whereas Thorup and Zwick establish their bounds in expectation, we establish ours with high probability. In Section 4.3, we apply our parallel graph clustering algorithm in creating a graph-theoretic Voronoi diagram for our network mapping algorithm.

We begin with some review from Thorup and Zwick [98]. Let $G = (V, E)$ be a connected, undirected n-vertex graph, and let $\delta(u, v)$ denote the distance between vertices u and v in G. In this section, we allow G to be weighted, where $\delta(u, v)$ is the sum of weights on a shortest path (lowest weight path) from u to v, but in our algorithms for parallel network mapping, we assume G is unweighted, in which case $\delta(u, v)$ is the number of edges on a shortest path from u to v. For a subset $A \subseteq V$, let $\delta(A, v) = \min_{a \in A} \delta(a, v)$, and, for vertices $w, v \in V$, let $C_A(w)$ be the *cluster* of w and $B_A(v)$ be the *bunch* of v with respect to A, defined as follows:

$$C_A(w) = \{v \in V \mid \delta(w, v) < \delta(A, v)\} \quad \text{and} \quad B_A(v) = \{w \in V \mid \delta(w, v) < \delta(A, v)\}.$$

Note that if $w \in A$, then $C_A(w) = \emptyset$. Also, observe that bunches and clusters are "inverses" of each other, in that $v \in C_A(w)$ if and only if $w \in B_A(v)$. In addition, notice that clusters and bunches can only shrink as we add vertices to A; that is, if $A' \subseteq A$, then $C_A(w) \subseteq C_{A'}(w)$ and $B_A(v) \subseteq B_{A'}(v)$, for all v and w in V.

Now, let $\beta \in [4,\, n)$, be a "parallelism" parameter and let $s \in [4 \ln n,\, n)$ be a "size" parameter. Define a subset, $A \subseteq V$, to be a set of (β, s)-*balanced centers* if $|C_A(w)| \leq \beta n/s$, for all $w \in V$.

[4]*Routing Stretch* is the worst ratio between the length of a path on which a message is routed and the length of the shortest path in the network from the source to the destination.

Algorithm 10: parallel-centers(V, s, β):

```
1 $A \leftarrow \emptyset$, $W \leftarrow V$
2 while $|W| > 0$ do
3     $A' \leftarrow \mathsf{Sample}(W, s)$    // a random sample of expected size $s$ (or $W$ if $s \geq |W|$)
4     $A \leftarrow A \cup A'$
5     for $w \in W$ do in parallel
6         $C_A(w) \leftarrow \{v \in V : \delta(w, v) < \delta(A, v)\}$
7     $W \leftarrow \{w \in W : |C_A(w)| > \beta n/s\}$
8 return $A$
```

Thorup and Zwick [98] give a sequential algorithm for finding a set of $(4, s)$-balanced centers of expected size $O(s \log n)$. In Algorithm 10, we give a parallel algorithm for finding a set of (β, s)-balanced centers of size $O(s \log_\beta n)$ in $O(\log_\beta n)$ rounds w.h.p. Thus, the parameter β allows one to trade off parallel time and cluster size.

Our algorithm (Algorithm 10) takes a graph $G = (V, E)$ as input and initializes A, the eventual output of the algorithm, to be empty, and W, the set of nodes $v \in V$ where $|C_A(v)| > \beta n/s$, to be V. Then, we iteratively add $\mathsf{Sample}(W, s)$ to A, and replace W with vertices $w \in W$ such that $|C_A(w)| > \beta n/s$, in parallel, where the function, $\mathsf{Sample}(W, s)$, returns W if $|W| \leq s$ and, otherwise, returns a set of elements from W such that each element in W is selected independently at random with probability $s/|W|$. We continue in this way until $W = \emptyset$.

Since the size of a cluster, $|C_A(w)|$, does not increase as we add more vertices to A, the set A returned by our algorithm is a set of (β, s)-balanced centers. Also, the Sample function returns a sample of size at most $2s$ with probability at least $1 - e^{-s/3}$, which holds with high probability across all iterations when $s \geq 4 \ln n$, by a standard Chernoff bound, e.g., see [80, p. 69]. Incidentally, Thorup and Zwick use the same Sample function, but don't bound its maximum size as we do. This high-probability upper bound for the sample size is not sufficient to achieve a high-probability bound, however, for the entire parallel graph clustering algorithm.

To that end, we define a parameter, α, as follows:

$$\alpha = \begin{cases} 2 & \text{if } \beta \leq ((4/3)e)^4 \\ (4/3)e\beta^{1/2} & \text{otherwise} \end{cases}$$

where $e \approx 2.71828$ is Euler's number. This definition of α is made so that we may achieve high probability bounds for a range of β values.

Let W_i denote the set W at the beginning of iteration i, let A'_i denote the set A' that was added in iteration i, and let A_i denote the set A in this iteration, including the set, A'_i, i.e., $A_i = A_{i-1} \cup A'_i$, for $i = 1, 2, \ldots$, where $A_0 = \emptyset$. Say that iteration i is "bad" if the following inequality holds:

$$\sum_{w \in W_i} |C_{A'_i}(w)| > \frac{\alpha n |W_i|}{s},$$

and that otherwise it is "good". Note that, since W_i is a given for iteration i, whether iteration i is good or bad depends only on A'_i.

Lemma 4.1 (Thorup-Zwick [98], Lemma 3.2)**.** *Let $W \subseteq V$, let $1 \leq s \leq n$, and let $A' \leftarrow$ Sample(W, s). Then, for every $v \in V$, $E[\,|B_{A'}(v) \cap W|\,] \leq |W|/s$.*

This implies the following:

$$E\left[\sum_{w \in W_i} |C_{A'_i}(w)|\right] = E\left[\sum_{v \in V} |B_{A'_i}(v) \cap W_i|\right] \leq \frac{n|W_i|}{s}.$$

Therefore, by Markov's inequality, an iteration is bad with probability at most $1/\alpha$.

Let W_{i+1} denote the set of vertices, W, whose clusters have size at least $\beta n/s$ at the end of a good iteration i. As $W_{i+1} \subseteq W_i$, and $C_{A_i}(w) \subseteq C_{A'_i}(w)$, for all $w \in V$, in a good iteration we have:

$$\frac{\beta n|W_{i+1}|}{s} \leq \sum_{w \in W_i} |C_{A_i}(w)| \leq \sum_{w \in W_i} |C_{A'_i}(w)| \leq \frac{\alpha n |W_i|}{s};$$

hence, $|W_{i+1}| \leq (\alpha/\beta)|W_i|$ in a good iteration. Thus, the number of good iterations of our algorithm is $L = O(\log_{(\beta/\alpha)} n) = O(\log_\beta n)$, for either choice of α. We wish to show that the number of bad iterations is $O(L)$ w.h.p.

Since W_i is a given for iteration i, whether iteration i is good or bad depends only on A'_i; therefore, an iteration is good independent of whether any other iteration is good or bad, so, for the sake of analysis, consider a set of $c_0 L$ iterations (i.e., padding out with "dummy" iterations if necessary) where $c_0 \geq 4$ is a constant chosen below and each iteration is bad independently with probability $1/\alpha$. Let X denote the number of bad iterations in this set. So $E[X] = c_0 L/\alpha$; hence, the probability that over 3/4 of our iterations are bad can be bounded as $p = \Pr(X > (3/4)c_0 L) = \Pr(X > (3/4)\alpha \cdot E[X])$. Thus, at least L of our iterations are good with probability at least $1 - p$.

Case 1: $\alpha = 2$. In this case, β is $O(1)$; hence, L is $\Theta(\log_\beta n) = \Theta(\log n)$, since $\beta \geq 4$. Futher, $\Pr(X > (3/4)\alpha \cdot E[X]) = \Pr(X > (3/2) \cdot E[X])$, and, by a standard Chernoff bound,[5] e.g., see [80, p. 69],

$$\Pr(X > (3/2) \cdot E[X]) \leq e^{-E[X]/12} = e^{-c_0 L/24}.$$

Thus, choosing c_0 so that $c_0 L/24 \geq 2 \ln n$, we will have more than $(3/4)c_0 L$ bad iterations with probability at most $1/n^2$.

Case 2: $\alpha = (4/3)e\beta^{1/2}$. In this case, $(\alpha/\beta) \leq \beta^{-1/4}$; hence, L is $O(\log_\beta n)$. Further, we have that $\Pr(X > (3/4)c_0 L) = \Pr(X > (3/4)\alpha \cdot E[X]) = \Pr(X > e\beta^{1/2} \cdot E[X])$, and, by a

[5]$\Pr(X \geq (1+\delta) \cdot E[X]) \leq e^{-E[X] \cdot \delta^2/3}$, for $0 < \delta \leq 1$.

non-standard Chernoff bound,[6] e.g., see [80, p. 70],

$$\Pr(X > e\beta^{1/2} \cdot E[X]) \leq \left(\frac{e}{e\beta^{1/2}}\right)^{(3/4)c_0L} = \beta^{-(3/8)c_0L}.$$

Thus, by choosing c_0 so that $(3/8)c_0L \geq 2\log_\beta n$, we will have more than $(3/4)c_0L$ bad iterations with probability at most $1/n^2$.

Therefore, we have the following.

Lemma 4.2. *The number of good and bad iterations in Algorithm 10 is $O(\log_\beta n)$ w.h.p.*

This gives us the following:

Theorem 4.1. *Given an undirected, connected graph, $G = (V, E)$, we can find a set, A, of (β, s)-balanced centers of size $O(s\log_\beta n)$ in $O(\log_\beta n)$ parallel rounds w.h.p.*

For example, if $\beta = 4$, then A is constructed to have size $O(s\log n)$ in $O(\log n)$ rounds; if $\beta = n^\epsilon$, for constant $0 < \epsilon < 1/2$, then A is constructed to have size $O(s)$ in $O(1)$ rounds.

4.3 Our Fast Parallel Network Mapping Algorithms

In this section, we provide our fast parallel network mapping algorithms for a connected, undirected, unweighted network, $G = (V, E)$, given a subset $U \subseteq V$ from which we may perform kth-hop queries. We denote the size of U by n and the size of V by N. Let H be the graph induced by the shortest paths in G between pairs of vertices in U. That is, H has vertex set U and there is an edge (u, v) in H, for $u, v \in U$, if the shortest path between u and v in G determined by kth-hop queries contains no other vertex in U besides u and v. The weight of each edge (u, v) in H is the distance, $\delta(u, v)$, between u and v in G. The

[6] $\Pr(X \geq (1+\delta) \cdot E[X]) \leq (e/(1+\delta))^{(1+\delta)\cdot E[X]}$.

goal of network mapping is to determine the edges of H (which can then be used to easily determine the vertices in G in a shortest path corresponding to each edge (u, v) in H in a single round of $\delta(u, v)$ $\mathsf{kth\text{-}hop}$ queries). We assume we know the value of $\delta_{\max}$, which is the weight of a maximum-weight edge in H. For example, if $U = V$, then $\delta_{\max} = 1$. In the worst case, $\delta_{\max}$ is equal to the diameter of G, but in real-world network mapping applications, $\delta_{\max}$ is likely to be a constant.

We perform all the queries needed in our parallel network mapping algorithms in a subroutine, $\mathsf{Distances}(v, W)$, which determines the distance, $\delta(v, w)$, for a given $v \in U$ and every other $w \in W \subseteq U$. We describe two possible implementations for $\mathsf{Distances}(v, W)$, which we choose between depending on our desired goals. In our first implementation, we perform a simple binary search using $\mathsf{kth\text{-}hop}(k, v, w)$ queries to determine $\delta(v, w)$, for each w. This requires $O(\log \operatorname{diam}(G))$ rounds and a total of $O(|W| \log \operatorname{diam}(G))$ $\mathsf{kth\text{-}hop}$ queries, and this implementation doesn't require any assumptions about W. Note that we assume that we know $\operatorname{diam}(G)$, and if this is not the case, we can instead perform a doubling binary search with the same query complexity. In our second implementation, we perform $\delta_{\max}$ $\mathsf{kth\text{-}hop}(k, w, v)$ queries in parallel, for each w, for $k = 1$ to $\delta_{\max}$. This implementation requires a single round of $O(\delta_{\max} |W|)$ $\mathsf{kth\text{-}hop}$ queries, and it requires that $v \in W$, and the nodes in W induce a connected subgraph of H that contains the shortest path in H from each w in W to v, and that we are only interested in finding the edges of this subgraph. This set of queries finds all the edges of a breadth-first search (BFS) tree, B_v, rooted at v, in the induced graph, H, since a shortest path from w to v is also a shortest path from v to w, and a subpath of any shortest path is a shortest path for its endpoints. Thus, in this second implementation, we can determine $\delta(v, w)$, for each $w \in U$, from B_v, by summing the weights of the edges from v to w in B_v (which doesn't require any additional queries). This gives us the following lemma.

Lemma 4.3. *$\mathsf{Distances}(v, W)$ can be implemented in $O(\log \operatorname{diam}(G))$ rounds using a total of*

$O(|W| \log \operatorname{diam}(G))$ kth-hop queries. Alternatively, if $v \in W$, and the subgraph of H induced by W is connected and we are interested only in finding the edges of this subgraph, then Distances(v, W) can be implemented in 1 round with $O(\delta_{\max} |W|)$ kth-hop queries.

Let the cluster of vertex w with respect to centers A be $C_A(w) = \{v \in U \mid \delta(w, v) < \delta(A, v)\}$. The key idea of our parallel network mapping algorithm is to first find a set, A, of (β, s)-balanced centers, using our parallel algorithm from the previous section, and then use this set of centers to compute a graph-theoretic Voronoi diagram [51, 62] for G, from which we may efficiently then perform a brute-force querying step for each Voronoi region. This approach is similar in spirit to the one by Kannan, Mathieu, Zhou [71, Section 2], with some key important differences: i) the restriction of our queries to the vantage point $U \subseteq V$ and the parameters capturing sampling coverage of set U, ii) the usage of kth-hop queries, and iii) our parallel graph clustering that allows us to trade off between round complexity and query complexity.

The initial center-finding step builds a set, A, of size $O(s \log_\beta n)$ such that each vertex in U has a cluster with respect to A of size at most $\beta n/s$. One of the challenges in implementing this algorithm efficiently in parallel using kth-hop queries is that we need to determine cluster sizes for all vertices in U in each iteration, which would take too many queries to compute exactly. So, rather than compute such sizes exactly, we instead build a global random set, R, which we use to approximate the size of each cluster. We give the details in Algorithm 11.

Lemma 4.4. *Our estimated-parallel-centers algorithm constructs a set, A, of $(3\beta, s)$-balanced centers of size $O(s \log_\beta n)$. Suppose Distances(r, U) executes in $R(n)$ rounds and $Q(n)$ kth-hop queries. Then estimated-parallel-centers algorithm executes in $O(R(n) \log_\beta n)$ rounds and $O(Q(n)(s/\beta) \log n \log_\beta n)$ kth-hop queries, w.h.p.*

Proof. Recall that the estimated-parallel-centers algorithm uses a global random sample set, R, for estimating cluster sizes, where R is a random subset of U of size $T = c_1(s/\beta) \log n$.

Algorithm 11: Our parallel querying algorithm, estimated-parallel-centers(U, s, β), for finding a set of (β, s)-balanced centers A.

```
1  $A \leftarrow \emptyset$, $W \leftarrow U$
2  $T \leftarrow c_1(s/\beta)\log n$  // $c_1$ is a constant set in the analysis
3  while $|W| > 0$ do
4      $A' \leftarrow$ Sample$(W, s)$
5      $A \leftarrow A \cup A'$
6      $R \leftarrow$ a random subset with $v \in U$ chosen independently with probability $T/n$
7      for each $r \in R$ do in parallel
8          Distances$(r, U)$
9      for $w \in W$ do in parallel
10         $S(w) \leftarrow \{v \in R : \delta(w, v) < \delta(A, v)\}$    // $S(w) = C_A(w) \cap R$
11     $W \leftarrow \{w \in W : |S(w)| > 2\beta T/s\}$    // that is, $|S(w)|(n/T) > 2\beta n/s$
12 return $A$
```

Recall that, for each vertex $w \in W$, we defined $S(w)$ such that $S(w) = R \cap C_A(w)$. Thus, $E[|S(w)|] = |C_A(w)|(T/n)$. We are interested in showing that w.h.p. this sample of $C_A(w)$ is giving neither an over-estimate nor an under-estimate for the size of $C_A(w)$, which we define respectively as follows:

- *Over-estimate event*: $|C_A(w)| \leq \beta n/s$, but $|S(w)| > 2\beta T/s$. In this case, we would be including w in W even though its cluster size is sufficiently small.
- *Under-estimate event*: $|C_A(w)| > 3\beta n/s$, but $|S(w)| \leq 2\beta T/s$. In this case, we would be excluding w from W even though its cluster size is big.

Let us consider each of these types of events in turn.

Over-estimate event. We wish to bound the probability that $|C_A(w)| \leq \beta n/s$ but $|S(w)| > 2\beta T/s$, where $T = c_1(s/\beta)\log n$. Let X denote the sum of $|C_A(w)|$ indicator random variables for counting the members of $C_A(w) \cap R$, i.e., where each variable is 1 independently with probability T/n. Thus, $E[X] = E[|S(w)|] = |C_A(w)|(T/n)$. So

$$\Pr(|S(w)| > 2\beta T/s) = \Pr(X > 2\beta T/s) = \Pr\left(X > \frac{2\beta n}{s|C_A(w)|} \cdot E[X]\right)$$
$$= \Pr(X > (1+\delta) \cdot E[X]),$$

where

$$\delta = \left(\frac{2\beta n}{s|C_A(w)|} - 1\right) \; > \; 1.$$

In addition,

$$\delta \cdot E[X] = \left(\frac{2\beta n}{s|C_A(w)|} - 1\right) \cdot \frac{|C_A(w)|T}{n} = \frac{2\beta T}{s} - \frac{|C_A(w)|T}{n}$$
$$\geq \frac{2\beta T}{s} - \frac{\beta T}{s} = \frac{\beta T}{s} = c_1 \log n.$$

Thus, by a standard Chernoff bound,[7] and the fact that $\delta > 1$,

$$\Pr(X \geq (1+\delta) \cdot E[X]) \leq e^{-\delta^2 \cdot E[X]/(2+\delta)} \leq e^{-\delta \cdot E[X]/3} \leq e^{-(c_1 \log n)/3} \leq \frac{1}{n^3},$$

for $c_1 \geq 9 \ln 2 \approx 6.24$.

Under-estimate event. We wish to bound the probability that $|C_A(w)| > 3\beta n/s$ but $|S(w)| \leq 2\beta T/s$, where $T = c_1(s/\beta)\log n$. Let X denote the sum of $|C_A(w)|$ indicator random variables for counting the members of $C_A(w) \cap R$, i.e., where each variable is 1 independently with probability T/n. Thus, $E[X] = E[|S(w)|] = |C_A(w)|(T/n) > 3c_1 \log n$.

[7] $\Pr(X \geq (1+\delta) \cdot E[X]) \leq e^{-\delta^2 \cdot E[X]/(2+\delta)}$, for $\delta > 0$, e.g., see `https://en.wikipedia.org/wiki/Chernoff_bound`.

So

$$\Pr(|S(w)| \le 2\beta T/s) = \Pr(X \le 2\beta T/s)$$
$$= \Pr\left(X \le \frac{2\beta n}{s|C_A(w)|} \cdot E[X]\right) \le \Pr(X \le (2/3) \cdot E[X]).$$

Thus, by a standard Chernoff bound,[8] e.g., see [80, p. 71],

$$\Pr(|S(w)| \le 2\beta T/s) \le e^{-(3c_1 \log n)/18} \le \frac{1}{n^3},$$

when $c_1 \ge 18 \ln 2 \approx 12.48$.

Of course, R is the same random sampling set for all our samples, $S(w)$, for $w \in W$. Nevertheless, by a union bound, the above analysis shows that R causes an over-estimate event or an under-estimate event, for some $S(w)$, with probability at most $1/n^2$.

By the bound on over-estimate events, we have shown that w.h.p. every cluster with size over $\beta n/s$ is included in W in any given iteration of our estimated-parallel-centers algorithm. In addition, by the bound on under-estimate events, we have shown that w.h.p. every vertex, w, that we exclude from W has a cluster size of at most $3\beta n/s$. Thus, using essentially the same analysis as we gave for the proofs of theorem 4.1 and lemma 4.2, and noting that each iteration of our estimated-parallel-centers algorithm has round complexity $O(R(n))$ and query complexity $O(Q(n)(s/\beta)\log n)$, where $R(n)$ and $Q(n)$ are the respective round and query complexities for the Distances algorithm, we establish the lemma.

□

[8] $\Pr(X \le (1-\delta) \cdot E[X]) \le e^{-\delta^2 \cdot E[X]/2}$, for $0 < \delta < 1$.

Now that we have defined and analyzed the function estimated-parallel-centers(U, s, β), let us next turn to our parallel algorithm for mapping a connected, undirected graph, $G = (V, E)$, given a subset, $U \subseteq V$, from which we can perform kth-hop queries. This algorithm takes as input the vertex set U, and outputs, $\tilde{E}$, the set of edges of the induced graph, H, defined by the vertex set U and the shortest paths in G returned by kth-hop queries.

Let $A \subseteq U$ be a set of *centers*, which in our network mapping algorithm will come from a call to our estimated-parallel-centers(U, s, β) algorithm, but a graph-theoretic Voronoi diagram can be defined for any weighted graph and any set of centers. Given a center, $a \in A$, define the *Voronoi cell*, $\mathrm{Vor}_A(a)$, for a in H as $\mathrm{Vor}_A(a) = \{v \in U : \delta(a, v) \leq \delta(A \backslash \{a\}, v)\}$. The *graph-theoretic Voronoi diagram* for A in U consists of the union of Voronoi cells, $\mathrm{Vor}_A(a)$, for each center, $a \in A$. We say that an edge $(v, w) \in \tilde{E}$ is an *interior* edge if $v, w \in \mathrm{Vor}_A(a)$, for some center $a \in A$, and it is a *boundary* edge if $v \in \mathrm{Vor}_A(a)$ and $w \in \mathrm{Vor}_A(b)$, where $a \neq b$. If we were to perform a set of kth-hop queries for every pair of vertices in a Voronoi cell, then we are guaranteed to discover every internal edge in $\mathrm{Vor}_A(a)$, but we will miss boundary edges going between two Voronoi cells. Thus, we need to "branch out" a little bit from the vertices of $\mathrm{Vor}_A(a)$ in order to discover all the boundary edges. To facilitate this, for any center, $a \in A$, let $N_{2\delta_{\max}}(a)$ be the set of "nearby" vertices in H, that is, vertices that are within a distance of $2\delta_{\max}$ of a. Formally,

$$N_{2\delta_{\max}}(a) = \{v \in U : \delta(a, v) \leq 2\delta_{\max}\}.$$

We assume we know μ, the maximum size of $N_{2\delta_{\max}}(a)$, for any $a \in A$. Of course, $\mu < n$. The following lemma shows that it is sufficient to consider these nearby neighbors, for each center $a \in A$, in order to cover all the edges in H, including interior edges and boundary edges. (See also Figure 4.2.)

We give the details of our network mapping algorithm in Algorithm 12. Through a call to

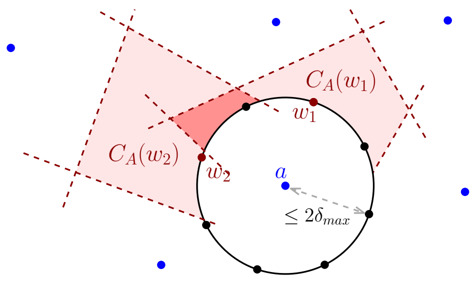

Figure 4.2: This figure represents a partial structure of our Voronoi Diagram. Blue vertices represent centers from A. The circle centered at $a \in A$ represents the vertices of distance at most $2\delta_{\max}$ from a. We use clusters of nearby vertices of a to discover boundary edges. For simplicity, we draw only two clusters for two arbitrary nodes $w_1, w_2 \in N_{2\delta_{\max}}(a)$.

estimated-parallel-centers(U, s, β), we find a set of $(O(\beta), s)$-balanced centers, A. Next, we build a BFS tree from each vertex $a \in A$ to be able to identify nodes in $N_{2\delta_{\max}}(a)$. Then, our mapping algorithm, map, constructs graph-theoretic Voronoi diagram for the centers in A, and then "branches out" from each center $a \in A$ by considering the nodes in $N_{2\delta_{\max}}(a)$ and the clusters defined by nodes in $N_{2\delta_{\max}}(a)$. Finally, after having done this Voronoi decomposition, our algorithm performs exhaustive searches in each cluster in parallel. This part of the algorithm uses a method, Exhaustive-Query(W), which finds all the edges of H between vertices in W by calling Distances(v, W), for each $v \in W$.

Algorithm 12: Parallel network mapping using kth-hop queries

```
1  Function map($U$):
2      $A \leftarrow$ estimated-parallel-centers$(U, s, \beta)$
3      for each $a \in A$ do in parallel
4          Distances$(a, U)$        // gives us $N_{2\delta_{\max}}(a)$ as well
5      for each $a \in A$ do in parallel
6          $E_a \leftarrow$ Exhaustive-Query$(N_{2\delta_{\max}}(a))$
7          for $b \in N_{2\delta_{\max}}(a)$ do in parallel
8              Distances$(b, U)$
9              $C_A(b) \leftarrow \{v \in U : \delta(b, v) < \delta(A, v)\}$
10             $E_{a,b} \leftarrow$ Exhaustive-Query$(C_A(b))$
11     return $\bigcup_{a \in A} \left( E_a \cup \bigcup_{b \in N_{2\delta_{\max}}(a)} E_{a,b} \right)$
```

The following lemmas establish the correctness and performance complexities for our network mapping algorithm.

Lemma 4.5. *Let (u, v) be an edge in H. Then there exists a center, $a \in A$, such that u and*

v are both in $N_{2\delta_{\max}}(a)$ or both in $C_A(b)$, for some $b \in N_{2\delta_{\max}}(a)$.

Proof. Let (u, v) be an edge in H, and note that, by definition, $\delta(u, v) \leq \delta_{\max}$. Assume, without loss of generality, that $\delta(A, u) \leq \delta(A, v)$. Also, let a be a vertex in A such that $\delta(a, u) = \delta(A, u)$. If $\delta(a, u) \leq \delta_{\max}$, then both u and v are in $N_{2\delta_{\max}}(a)$, by the triangle inequality. So, suppose $\delta(a, u) > \delta_{\max}$. Let b be a vertex in U on a shortest path from a to u such that $\delta_{\max} < \delta(a, b) \leq 2\delta_{\max}$. Note that b must exist, since no edge in H has weight more than $\delta_{\max}$ (it is possible that $b = u$). Further, b is in $N_{2\delta_{\max}}(a)$ and $\delta(a, u) = \delta(a, b) + \delta(b, u)$. Also, $\delta(b, u) < \delta(a, u) = \delta(A, u)$; hence, u is in $C_A(b)$.

By the triangle inequality, and the above observations,

$$\begin{aligned}
\delta(b, v) &\leq \delta(b, u) + \delta(u, v) \\
&\leq \delta(b, u) + \delta_{\max} \\
&= \delta(a, u) - \delta(a, b) + \delta_{\max} \\
&< \delta(a, u) - \delta_{\max} + \delta_{\max} \\
&= \delta(a, u) \\
&= \delta(A, u).
\end{aligned}$$

Therefore, $\delta(b, v) < \delta(A, v)$; hence, v is also in $C_A(b)$. □

Lemma 4.6. *If Distances(v, W) executes in $R(|W|)$ rounds using $Q(|W|)$ kth-hop queries, then Algorithm 12 uses $O(Q(n)(s/\beta) \log n \log_\beta n + \mu(Q(n) + (\beta n/s)Q(\beta n/s))s \log_\beta n)$ queries in $O(R(n) \log_\beta n)$ rounds, w.h.p., where $\mu = \max_{a \in A} |N_{2\delta_{\max}}(a)|$.*

Proof. By lemma 4.4, estimated-parallel-centers(U, s, β) executes in $O(R(n) \log_\beta n)$ rounds and $O(Q(n)(s/\beta) \log n \log_\beta n)$ kth-hop queries, and returns a set of $(3\beta, s)$-balanced centers of size $O(s \log_\beta n)$, w.h.p. The parallel Distances calls in line 4 thus executes in $O(R(n))$

rounds using $O(Q(n)s\log_\beta n)$ kth-hop queries, w.h.p., and the calls to Exhaustive-Query in line 6 execute in $O(R(\mu))$ rounds using a total of $O(\mu Q(\mu)s\log_\beta n)$ kth-hop queries, but these bounds are dominated by the Distances calls in line 8, all of which execute in $O(R(n))$ rounds using $O(\mu Q(n)s\log_\beta n)$ kth-hop queries, w.h.p. Finally, the calls to Exhaustive-Query in line 10 execute in $O(R(\beta n/s))$ rounds using $O(\mu(\beta n/s)Q(\beta n/s)s\log_\beta n)$ kth-hop queries, w.h.p. □

Plugging in our derived bounds for Distances, we get the following theorem.

Theorem 4.2. *Given a connected graph, $G = (V, E)$, and subset, $U \subseteq V$, one can map the n-vertex induced shortest-path graph, H, with respect to G and U in $O(1)$ rounds using $O(\delta_{\max}\,\mu\, n^{3/2+\epsilon})$ kth-hop queries, for constant $0 < \epsilon < 1/2$, w.h.p. Alternatively, one can map H in $O(\log n \cdot \log \operatorname{diam}(G))$ rounds using $O(\mu\, n^{3/2}\log^{3/2} n \cdot \log \operatorname{diam}(G))$ queries, w.h.p.*

Proof. For the first result, set $\beta = n^\epsilon$ and $s = n^{1/2+\epsilon}$, and let us use the implementation of Distances that executes in 1 round using $O(\delta_{\max}\, n)$ kth-hop queries from lemma 4.3. For the second result, set $\beta = 4$ and $s = (n/\log n)^{1/2}$ and let us use the implementation of Distances that executes in $O(\log \operatorname{diam}(G))$ rounds using $O(n \log \operatorname{diam}(G))$ kth-hop queries from lemma 4.3. The bounds follow by lemma 4.6. □

For example, depending on the values of $\delta_{\max}$ and μ, the above theorem establishes an improvement over the brute-force querying algorithm for solving the parallel network mapping problem in $O(1)$ rounds. The following theorem shows that bounding the parameters $\delta_{\max}$ and μ is needed in order to do better than a quadratic number of kth-hop queries.

Theorem 4.3. *There is an infinite family of n-vertex graphs, G, such that mapping the induced shortest-path graph, H, for a set, U, of $O(n)$ vertices requires $\Omega(n^2)$ kth-hop queries, when μ is $\Theta(n)$ and $\delta_{\max}$ is $\Theta(\log n)$, even if G has maximum degree 3.*

Proof. Let G be the graph of a complete binary tree with n nodes and let U be the set of leaves of G. Thus, the distance in H between any node, v, in the left subtree of G, and a node, w, in the right subtree of G, is $2 \log n$. Thus, $\delta_{\max} = 2 \log n$ and μ is $\Theta(n)$. Now, let G' be G plus a single path in G of length $(2 \log n) - 1$ between two vertices, v and w, in U such that v is in the left subtree of G and w is in the right subtree of G. Thus, there is an edge with weight $(2 \log n) - 1$ joining v and w in the induced shortest-path graph, H', for G', and otherwise H' has the same edge set as H. But there are $\Omega(n^2)$ such possible pairs and the only way to discover the edge (v, w) in H is to perform a kth-hop(k, v, w) query. Any other type of kth-hop query cannot distinguish between H and H'. $\square$

4.4 A Greedy Network Mapping Algorithm

Kannan, Methieu, and Zhou [71] introduce a proof technique that sequentially uses a verification algorithm for unweighted graphs as an oracle for issuing shortest-path queries in a greedy graph reconstruction algorithm. In this section, we show how to adapt this proof technique to a parallel setting and apply it to map the weighted graph, H. For example, our algorithm uses kth-hop queries and provides parallelism according to a parameter, $1 \leq p < n$.

Our greedy algorithm is based on performing steps of the classic greedy set cover algorithm in parallel batches. Recall that in this problem, one is given a collection of sets, $S_1, S_2, \ldots, S_m$, whose union is the universe $\mathcal{U}$, and the goal is to find a smallest sub-collection of sets whose union is $\mathcal{U}$, that is, a sub-collection that covers $\mathcal{U}$. The greedy algorithm repeatedly chooses the set covering the maximum number of uncovered items in $\mathcal{U}$, and this results in a number of sets that is at most an $O(\log n)$ factor more than optimum [38].

Let $f(n, \Delta)$ be the query complexity of the best sequential algorithm for the problem of *graph verification* for any connected unweighted graph of n vertices and maximum degree

Δ via distance queries, that is, for determining whether an unknown graph, $G = (V, E)$, is equal to a given graph, $\hat{G} = (V, \hat{E})$. For example, Kannan, Methieu, and Zhou [71] show that $f(n, \Delta)$ is $n^{1+O(\tau(n))}$, where $\tau(n) = \sqrt{(\log \log n + \log \Delta)/\log n}$. The function, $f(n, \Delta)$, is used only in our analysis, where we show that, given a parallelism parameter, $1 \le p < n$, our parallel network mapping algorithm can be tuned to have the desired query complexity, $Q(n)$, and round complexity, $R(n)$.

The distance queries in an unweighted graph verification algorithm perform two functions—confirming that edges in $\hat{E}$ are actually in E and confirming that edges not in $\hat{E}$ are also not in E, that is, confirming every $(u, v) \notin \hat{E}$ is not in E. To that latter end, let $\hat{\delta}_h(u, v)$ denote the *hop-count* (number of edges) distance between u and v based on the edges in $\hat{E}$, and let $\hat{E}^c$ denote the set of non-edges in $\hat{G}$, that is, the set of pairs, (u, v), such that $u \neq v$ and $(u, v) \notin \hat{E}$. Similarly, let E^c denote the set of non-edges in G. For any set of tentative edges, $\hat{E}$, define the following set for each pair of vertices, $(u, v) \in \hat{E}^c$: $S_{u,v}(\hat{E}) = \Big\{(x, y) \in \hat{E}^c \mid \hat{\delta}_h(u, x) + \hat{\delta}_h(y, v) + 1 < \hat{\delta}_h(u, v)\Big\}$.

Kannan, Methieu, and Zhou [71] prove the following two lemmas.

Lemma 4.7. *Suppose $\hat{E} \subseteq E$. For any $(u, v) \in \hat{E}^c$, if $\delta_h(u, v) = \hat{\delta}_h(u, v)$, where $\delta_h(u, v)$ denotes the hop-count distance between u and v in G, then $S_{u,v}(\hat{E}) \subseteq E^c$, that is, each pair in $S_{u,v}(\hat{E})$ is a non-edge in G.*

Lemma 4.8. *If a set of distance queries, T, verifies that every non-edge of $\hat{G}$ is a non-edge of G, then $\cup_{(u,v)\in T} S_{u,v}(\hat{E}) = \hat{E}^c$.*

We present our parallel greedy algorithm for mapping $H = (U, \tilde{E})$ in $G = (V, E)$, for when $U \subset V$, that is, we incrementally build our tentative edge set $\hat{E} \subseteq \tilde{E}$:

1. We initialize a set of tentative edges, $\hat{E}$, to a spanning tree of H by calling kth-hop(k, v, u) from every vertex, $v \in U$, to an arbitrarily chosen vertex, $u \in U$, for

$k = 1, \ldots, \mathrm{diam}(G)$, in parallel. We initialize a set of confirmed non-edges of H, $F \leftarrow \emptyset$. Note that we always maintain that $F \subseteq \hat{E}^c$. This requires 1 round of $O(\mathrm{diam}(G)n)$ kth-hop queries.

2. We compute all the $S_{u,v}(\hat{E})$ sets, for pairs $(u, v) \in \hat{E}^c$, which requires no queries.

3. We perform p steps of the greedy set-cover algorithm applied to the sets, $S_{u,v}(\hat{E})\backslash F$, with the goal of covering the remaining pairs, in $\hat{E}^c\backslash F$, in a greedy fashion, which also requires no queries. Let $\{(u_1, v_1), (u_2, v_2), \ldots, (u_p, v_p)\}$ denote the vertex pairs for the $S_{u,v}(\hat{E})$ sets chosen by these greedy steps.

4. We perform kth-hop(k, u_i, v_i) queries, for $k = 1, \ldots, \mathrm{diam}(G)$, in parallel, to determine the actual hop-count distance, $\delta_h(u_i, v_i)$, between u_i and v_i in H, for each $i = 1, 2, \ldots, p$ in parallel. This step requires $O(1)$ rounds of $O(p \cdot \mathrm{diam}(G))$ kth-hop queries in total.

5. For each i such that $\delta_h(u_i, v_i) = \hat{\delta}_h(u_i, v_i)$, we add all the pairs in $S_{u_i,v_i}(\hat{E})$ to F. If $F = \hat{E}^c$, then we are done, by lemma 4.8.

6. Otherwise, if $F \neq \hat{E}^c$ and $\delta_h(u_i, v_i) = \hat{\delta}_h(u_i, v_i)$, for all $i = 1, 2, \ldots, p$, then we repeat the above process, performing another p steps of greedy set cover, looping back to Step 3.

7. If, on the other hand, $F \neq \hat{E}^c$ and $\delta_h(u_i, v_i) < \hat{\delta}_h(u_i, v_i)$, for some i, then there must be at least one edge on a shortest path from u_i to v_i that is in $\tilde{E}$ and not yet in $\hat{E}$. In this case, we add all such edges (which were discovered when we performed the diam(G) kth-hop(k, u_i, v_i) queries) to $\hat{E}$, and repeat the above greedy searching for this updated set, $\hat{E}$, of candidate edges, returning to Step 2.

For the case when $U = V$, we modify our algorithm to be the following (note that in this case, hop-count distance and graph distance are the same):

1. We initialize a set of tentative edges, $\hat{E}$, to a spanning tree of H by calling kth-hop$(1, v, u)$ from every vertex, $v \in U$, to an arbitrarily chosen vertex, $u \in U$. We initialize a set of confirmed non-edges, $F \leftarrow \emptyset$. This requires 1 round of $O(n)$ kth-hop queries.

2. We compute all the $S_{u,v}(\hat{E})$ sets, for pairs $(u, v) \in \hat{E}^c$, which requires no queries.

3. We perform p steps of the greedy set-cover algorithm applied to the sets, $S_{u,v}(\hat{E})\backslash F$, with the goal of covering the remaining pairs, in $\hat{E}^c\backslash F$, in a greedy fashion, which also requires no queries. Let $\{(u_1, v_1), (u_2, v_2), \ldots, (u_p, v_p)\}$ denote the vertex pairs for the $S_{u,v}(\hat{E})$ sets chosen by these greedy steps.

4. We perform a binary search using kth-hop queries to determine the actual distance, $\delta(u_i, v_i)$, between u_i and v_i in H, for each $i = 1, 2, \ldots, p$ in parallel. This step requires $O(\log n)$ rounds of $O(p \log n)$ kth-hop queries in total.

5. For each i such that $\delta(u_i, v_i) = \hat{\delta}(u_i, v_i)$, we add all the pairs in $S_{u_i,v_i}(\hat{E})$ to F. If $F = \hat{E}^c$, then we are done, by lemma 4.8.

6. Otherwise, if $F \neq \hat{E}^c$ and $\delta(u_i, v_i) = \hat{\delta}(u_i, v_i)$, for all $i = 1, 2, \ldots, p$, then we repeat the above process, performing another p steps of greedy set cover, repeating a loop returning to Step 3.

7. If, on the other hand, $F \neq \hat{E}^c$ and $\delta(u_i, v_i) < \hat{\delta}(u_i, v_i)$, for some i, then there must be at least one edge on a shortest path from u_i to v_i that is in E and not yet in $\hat{E}$. In this case, we perform a binary search, described below, to find at least one such an edge, add all such edges to $\hat{E}$, and repeat the above greedy searching for this updated set, $\hat{E}$, of candidate edges, returning to Step 2. This step requires $O(\log n)$ rounds of at most $O(p \log n)$ kth-hop queries in total.

Before we give our analysis, let us describe the details for the binary search to find an undiscovered edge when $\delta(u_i, v_i) < \hat{\delta}(u_i, v_i)$, for some i. We begin with a query, kth-hop(k, u_i, v_i), where $k = \lfloor \delta(u_i, v_i)/2 \rfloor$, and let w denote the returned vertex. So, $\delta(u_i, w) = k$ and $\delta(w, v_i) = \delta(u_i, v_i) - k$. Since $\delta(u_i, v_i) < \hat{\delta}(u_i, v_i)$, we know that $\delta(u_i, w) < \hat{\delta}(u_i, w)$ or $\delta(w, v_i) < \hat{\delta}(w, v_i)$. Thus, we recursively search for one of these until we discover a new edge not in $\hat{E}$, which must exist, since $\delta(u_i, v_i) < \hat{\delta}(u_i, v_i)$.

This gives us the following result.

Theorem 4.4. *Let $f(n, \Delta)$ be the query complexity of an optimal sequential algorithm for graph verification for any unweighted connected graph with n vertices and maximum degree Δ using distance queries. Then, for $1 \leq p < n$, our parallel network mapping algorithm has kth-hop query complexity, $Q(n) \in O((\Delta np + f(n, \Delta) \log n)\text{diam}(G))$ and round complexity, $R(n) \in O((\Delta n + (f(n, \Delta)/p) \log n))$, if $U \subset V$, or $Q(n) \in O((\Delta np + f(n, \Delta) \log n) \log n)$ and round complexity, $R(n) \in O((\Delta n + (f(n, \Delta)/p) \log n) \log n)$, if $U = V$.*

Proof. Building a spanning tree of H is a one-time expense taking $O(n \cdot \text{diam}(G))$ kth-hop queries and a round complexity of $O(1)$ (step 1), for the case when $U \subset V$, or $O(n)$ queries with a round complexity of $O(1)$ (step 1), for the case when $U = V$. Each iteration of our greedy algorithm takes $O(p \cdot \text{diam}(G))$ kth-hop queries with a round complexity of $O(1)$ (step 4), for the case when $U \subset V$, or $O(p \log n)$ kth-hop queries with a round complexity of $O(\log n)$ (step 4 and step 7), for the case when $U = V$.

In the case when $\delta_h(u_i, v_i) < \hat{\delta}_h(u_i, v_i)$, for some $i \in [1, p]$, we discover at least one new edge—let us charge the queries for this iteration to this edge. Thus, the total number of kth-hop queries due to this case is $O(\Delta np \cdot \text{diam}(G))$, with $O(\Delta n)$ rounds, for the $U \subset V$ case, or $O(\Delta np \log n)$, with $O(\Delta n \log n)$ rounds, for the $U = V$ case. So, let us consider the case when $\delta_h(u_i, v_i) = \hat{\delta}_h(u_i, v_i)$, for all $i \in [1, p]$, which we call a "completely-greedy" iteration. We will provide an upper bound for the number of such iterations. Recall that in step 3, for the case when $U \subset V$ (similarly in step 3, for the case when $U = V$) we performed p steps of greedy set-cover algorithm applied to the sets, $S_{u,v}(\hat{E})\backslash F$, with the goal of covering the remaining pairs, in $\hat{E}^c\backslash F$ without additional queries. Let F_i denote the set of (x, y) pairs covered by the i-th step of this greedy set-cover algorithm, for $i = 1, 2, \ldots, p$. Thus,

$$|F_1| \geq |F_2| \geq \cdots \geq |F_p|,$$

and at the moment we chose the subset F_i it was the largest subset covering the uncovered pairs in $\mathcal{U}_i = \hat{E}^c\backslash(\bigcup_{j=1}^{i-1} F_j \cup F)$. The optimal sequential graph verification algorithm performs $f(n, \Delta)$ distance queries and confirms all the pairs in $\hat{E}^c$. Thus, in particular, this optimal algorithm must perform queries that cover $\mathcal{U}_i$ as a part of its $f(n, \Delta)$ queries; hence, because F_i is the subset for a distance query that covers the largest number of pairs in $\mathcal{U}_i$, and the average number of pairs in $\mathcal{U}_i$ covered by any distance query of the optimal algorithm is at least $|\mathcal{U}_i|/f(n, \Delta)$, we have that

$$|F_i| \geq \frac{|\mathcal{U}_i|}{f(n, \Delta)}.$$

Thus, in any iteration of our algorithm, since we perform p greedy steps, the size of the remaining pairs in $\hat{E}^c\backslash F$ is reduced by a multiplicative factor of

$$\left(1 - \frac{1}{f(n, \Delta)}\right)^p \leq e^{-p/f(n,\Delta)}.$$

Therefore, since $\hat{E}^c \leq n(n-1)$ and by the end of our algorithm we cover every pair in $\hat{E}^c$, the total number of completely-greedy iterations, g, can be bounded above by the smallest value of g such that

$$e^{-(p/f(n,\Delta))g} < n^{-2};$$

hence, the total number of completely-greedy iterations, g, is at most $O((f(n, \Delta)/p) \log n)$.

Note that the set $\hat{E}^c$ is potentially growing during our algorithm, with completely-greedy iterations possibly interspersed with iterations that discover new edges in $\tilde{E}$. Nevertheless, the above analysis still holds, because (1) the function, $f(n,\Delta)$ is a uniform bound for any connected graph with n nodes and maximum degree Δ, and (2) each time we (greedily) confirm that $\hat{\delta}(u,v) = \delta(u,v)$ for a set, $S_{u,v}(\hat{E})$, all the pairs in $S_{u,v}(\hat{E})$ are, in fact, non-edges in $\tilde{E}^c$. The claimed complexity bounds follow then, since each completely-greedy iteration requires $O(p \cdot \mathrm{diam}(G))$ kth-hop queries with round complexity $O(1)$, for the $U \subset V$ case, or $O(p \log n)$ kth-hop queries with round complexity $O(\log n)$, for the $U = V$ case. □

Thus, setting p to be $n^{O(\tau(n))}$ gives us the following.

corollary 4.1. *One can solve the network mapping problem with query complexity, $Q(n)$, that is $\mathrm{diam}(G) \cdot n^{1+O(\tau(n))}$ and round complexity, $R(n)$, that is $O(\Delta n)$, if $U \subset V$, or with $Q(n)$ that is $n^{1+O(\tau(n))}$ and round complexity, $R(n)$, that is $O(\Delta n \log n)$, if $U = V$.*

This query complexity is within an $n^{o(1)}$ factor of optimal when Δ is $n^{o(1)}$, by the following simple lower bound.

Theorem 4.5. *Solving the network mapping problem for an n-vertex graph, G, with maximum degree, Δ, requires $\Omega(\Delta n)$ kth-hop queries, even if H has only n edges.*

Proof. Let H be a caterpillar (i.e., a tree where every leaf is at distance 1 from a vertex on a central path), such that every internal node has degree Δ. Choose any pair, u and v, of sibling leaves and connect them with an edge. The only way to discover the edge, (u,v), is to perform a kth-hop(k,u,v) query, for $k \geq 1$. Thus, in expectation, any graph reconstruction algorithm must perform a query for over half of the pairs of siblings in H, that is, at least $\Omega((n/\Delta)\Delta^2) = \Omega(\Delta n)$ queries, in order to discover all the edges of H. □

4.5 Conclusion

We have given efficient algorithms for solving the network mapping problem in parallel. Such algorithms show the effectiveness of kth-hop queries, even though they are weaker than shortest-path queries. Our methods assume knowledge of $\delta_{\max}$ and μ, but this assumption can be relaxed at the expense of increasing the round complexity by an $O(\log n)$ factor, while keeping the query complexity unchanged, by using our algorithm as a blackbox to perform a doubling search for the values of these parameters. Our methods also assume kth-hop(k, u, v) remains same in the algorithm, which is a reasonable assumption in static routing. In our network mapping formulation, we abstracted away some system issues such that when the TTL field of a packet reaches 1, the node sends an ICMP message to the source address; however, in the real Internet, some nodes may have their ICMP responses switched off. Therefore, a direction to extend this work would be to design algorithms addressing such system issues.

We have also given new, parallel implementations for graph clustering, which provide trade-offs between the number of center vertices and the sizes of clusters. Even for sequential algorithms, this result may prove useful for applications where minimizing the number of center points is a primary optimization goal. For instance, one can apply our construction to the problems studied by Honiden *et al.* [62] for balancing graph-theoretic Voronoi diagrams to shave a $O(\log n)$ factor of the number of centers. It seems likely, therefore, that this result will have other applications as well.

Bibliography

[1] TRACEROUTE for Linux. `http://traceroute.sourceforge.net/`. Accessed: April 6, 2020.

[2] *TRACEROUTE(8) Traceroute For Linux*, October 2006. `http://man7.org/linux/man-pages/man8/traceroute.8.html`. Accessed: April 6, 2020.

[3] M. Abrahamsen, G. Bodwin, E. Rotenberg, and M. Stöckel. Graph reconstruction with a betweenness oracle. In N. Ollinger and H. Vollmer, editors, *33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016, February 17-20, 2016, Orléans, France*, volume 47 of *LIPIcs*, pages 5:1–5:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[4] D. Achlioptas, A. Clauset, D. Kempe, and C. Moore. On the bias of traceroute sampling: or, power-law degree distributions in regular graphs. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 694–703, 2005.

[5] P. Afshani, M. Agrawal, B. Doerr, C. Doerr, K. G. Larsen, and K. Mehlhorn. The query complexity of finding a hidden permutation. In A. Brodnik, A. López-Ortiz, V. Raman, and A. Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, volume 8066 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2013.

[6] P. Afshani, M. Agrawal, B. Doerr, C. Doerr, K. G. Larsen, and K. Mehlhorn. The query complexity of finding a hidden permutation. In A. Brodnik, A. López-Ortiz, V. Raman, and A. Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, pages 1–11, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[7] R. Afshar, A. Amir, M. T. Goodrich, and P. Matias. Adaptive exact learning in a mixed-up world: Dealing with periodicity, errors and jumbled-index queries in string reconstruction. In C. Boucher and S. V. Thankachan, editors, *String Processing and Information Retrieval - 27th International Symposium, SPIRE 2020, Orlando, FL, USA, October 13-15, 2020, Proceedings*, volume 12303 of *Lecture Notes in Computer Science*, pages 155–174. Springer, 2020.

[8] R. Afshar and M. T. Goodrich. Exact learning of multitrees and almost-trees using path queries. In A. Castañeda and F. Rodríguez-Henríquez, editors, *LATIN 2022: Theoretical Informatics - 15th Latin American Symposium, Guanajuato, Mexico, November 7-11, 2022, Proceedings*, volume 13568 of *Lecture Notes in Computer Science*, pages 293–311. Springer, 2022.

[9] R. Afshar, M. T. Goodrich, P. Matias, and M. C. Osegueda. Reconstructing binary trees in parallel. In C. Scheideler and M. Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 491–492. ACM, 2020.

[10] R. Afshar, M. T. Goodrich, P. Matias, and M. C. Osegueda. Reconstructing biological and digital phylogenetic trees in parallel. In F. Grandoni, G. Herman, and P. Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPIcs*, pages 3:1–3:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[11] R. Afshar, M. T. Goodrich, P. Matias, and M. C. Osegueda. Parallel network mapping algorithms. In K. Agrawal and Y. Azar, editors, *SPAA '21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021*, pages 410–413. ACM, 2021.

[12] R. Afshar, M. T. Goodrich, P. Matias, and M. C. Osegueda. Mapping networks via parallel kth-hop traceroute queries. In P. Berenbrink and B. Monmege, editors, *39th International Symposium on Theoretical Aspects of Computer Science, STACS 2022, March 15-18, 2022, Marseille, France (Virtual Conference)*, volume 219 of *LIPIcs*, pages 4:1–4:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[13] R. Afshar, M. T. Goodrich, and E. Ozel. Efficient exact learning algorithms for road networks and other graphs with bounded clustering degrees. In C. Schulz and B. Uçar, editors, *20th International Symposium on Experimental Algorithms, SEA 2022, July 25-27, 2022, Heidelberg, Germany*, volume 233 of *LIPIcs*, pages 9:1–9:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[14] T. Akutsu. A polynomial time algorithm for finding a largest common subgraph of almost trees of bounded degree. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 76(9):1488–1493, 1993.

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

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

[17] D. Angluin and J. Chen. Learning a hidden hypergraph. *J. Mach. Learn. Res.*, 7:2215–2236, 2006.

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

[19] M. J. Bannister, D. Eppstein, and J. A. Simons. Fixed parameter tractability of crossing minimization of almost-trees. In *International Symposium on Graph Drawing*, pages 340–351. Springer, 2013.

[20] A. Barrat, I. Alvarez-Hamelin, L. Dall'Asta, A. Vázquez, and A. Vespignani. Sampling of networks with traceroute-like probes. *Complexus*, 3(1-3):83–96, 2006.

[21] N. H. Barton. The role of hybridization in evolution. *Molecular ecology*, 10(3):551–568, 2001.

[22] Z. Beerliova, F. Eberhard, T. Erlebach, A. Hall, M. Hoffmann, M. Mihalák, and L. S. Ram. Network discovery and verification. *IEEE Journal on Selected Areas in Communications*, 24(12):2168–2181, 2006.

[23] R. Beigel, N. Alon, S. Kasif, M. S. Apaydin, and L. Fortnow. An optimal procedure for gap closing in whole genome shotgun sequencing. In T. Lengauer, editor, *Proceedings of the Fifth Annual International Conference on Computational Biology, RECOMB 2001, Montréal, Québec, Canada, April 22-25, 2001*, pages 22–30. ACM, 2001.

[24] K. Bello and J. Honorio. Computationally and statistically efficient learning of causal bayes nets using path queries. In S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 10954–10964, 2018.

[25] A. Bernasconi, C. Damm, and I. Shparlinski. Circuit and decision tree complexity of some number theoretic problems. *Information and Computation*, 168(2):113 – 124, 2001.

[26] A. Bernasconi, C. Damm, and I. E. Shparlinski. Circuit and decision tree complexity of some number theoretic problems. *Inf. Comput.*, 168(2):113–124, 2001.

[27] P. Bestagini, M. Tagliasacchi, and S. Tubaro. Image phylogeny tree reconstruction based on region selection. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2016, Shanghai, China, March 20-25, 2016*, pages 2059–2063. IEEE, 2016.

[28] A. Biswas, V. Jayapaul, and V. Raman. Improved bounds for poset sorting in the forbidden-comparison regime. In D. R. Gaur and N. S. Narayanaswamy, editors, *Algorithms and Discrete Applied Mathematics - Third International Conference, CALDAM 2017, Sancoale, Goa, India, February 16-18, 2017, Proceedings*, volume 10156 of *Lecture Notes in Computer Science*, pages 50–59. Springer, 2017.

[29] M. Bouvel, V. Grebinski, and G. Kucherov. Combinatorial search on graphs motivated by bioinformatics applications: A brief survey. In D. Kratsch, editor, *Graph-Theoretic*

Concepts in Computer Science, 31st International Workshop, WG 2005, Metz, France, June 23-25, 2005, Revised Selected Papers, volume 3787 of *Lecture Notes in Computer Science*, pages 16–27. Springer, 2005.

[30] I. Caragiannis, A. D. Procaccia, and N. Shah. When do noisy votes reveal the truth? *ACM Trans. Economics and Comput.*, 4(3):15:1–15:30, 2016.

[31] J. Cardinal and S. Fiorini. On generalized comparison-based sorting problems. In A. Brodnik, A. López-Ortiz, V. Raman, and A. Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, volume 8066 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2013.

[32] S. Choi and J. H. Kim. Optimal query complexity bounds for finding graphs. *Artif. Intell.*, 174(9-10):551–569, 2010.

[33] S.-S. Choi and J. H. Kim. Optimal query complexity bounds for finding graphs. *Artificial Intelligence*, 174(9):551 – 569, 2010.

[34] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In J. Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 206–219. ACM, 1986.

[35] C. Colombo, F. Lepage, R. Kopp, and E. Gnaedinger. Two SDN multi-tree approaches for constrained seamless multicast. In F. Pop, C. Negru, H. González-Vélez, and J. Rak, editors, *2018 IEEE International Conference on Computational Science and Engineering, CSE 2018, Bucharest, Romania, October 29-31, 2018*, pages 77–84. IEEE Computer Society, 2018.

[36] F. Comellas, M. A. Fiol, J. Gimbert, and M. Mitjana. The spectra of wrapped butterfly digraphs. *Networks*, 42(1):15–19, 2003.

[37] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

[38] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

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

[40] L. Dall'Asta, J. I. Alvarez-Hamelin, A. Barrat, A. Vázquez, and A. Vespignani. Exploring networks with traceroute-like probes: Theory and simulations. *Theor. Comput. Sci.*, 355(1):6–24, 2006.

[41] L. Dall'Asta, I. Alvarez-Hamelin, A. Barrat, A. Vázquez, and A. Vespignani. Exploring networks with traceroute-like probes: Theory and simulations. *Theoretical Computer Science*, 355(1):6–24, 2006.

[42] P. Damaschke. Adaptive versus nonadaptive attribute-efficient learning. In J. S. Vitter, editor, *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 590–596. ACM, 1998.

[43] C. Daskalakis, R. M. Karp, E. Mossel, S. J. Riesenfeld, and E. Verbin. Sorting and selection in posets. In C. Mathieu, editor, *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 392–401. SIAM, 2009.

[44] Z. Dias, S. Goldenstein, and A. Rocha. Exploring heuristic and optimum branching algorithms for image phylogeny. *J. Vis. Commun. Image Represent.*, 24(7):1124–1134, 2013.

[45] Z. Dias, S. Goldenstein, and A. Rocha. Large-scale image phylogeny: Tracing image ancestral relationships. *IEEE Multim.*, 20(3):58–70, 2013.

[46] Z. Dias, A. Rocha, and S. Goldenstein. Image phylogeny by minimal spanning trees. *IEEE Trans. Inf. Forensics Secur.*, 7(2):774–788, 2012.

[47] S. Dobzinski and J. Vondrák. From query complexity to computational complexity. In H. J. Karloff and T. Pitassi, editors, *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 1107–1116. ACM, 2012.

[48] S. Dobzinski and J. Vondrak. From query complexity to computational complexity. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 1107–1116, New York, NY, USA, 2012. ACM.

[49] B. Donnet. Internet topology discovery. In E. Biersack, C. Callegari, and M. Matijasevic, editors, *Data Traffic Monitoring and Analysis - From Measurement, Classification, and Anomaly Detection to Quality of Experience*, volume 7754 of *LNCS*, pages 44–81. Springer, 2013.

[50] B. Donnet and T. Friedman. Internet topology discovery: A survey. *IEEE Communications Surveys and Tutorials*, 9(1-4):56–69, 2007.

[51] M. Erwig. The graph Voronoi diagram with applications. *Networks*, 36(3):156–163, 2000.

[52] U. Faigle and G. Turán. Sorting and recognition problems for ordered sets. In K. Mehlhorn, editor, *STACS 85, 2nd Symposium of Theoretical Aspects of Computer Science, Saarbrücken, Germany, January 3-5, 1985, Proceedings*, volume 182 of *Lecture Notes in Computer Science*, pages 109–118. Springer, 1985.

[53] A. D. Flaxman and J. Vera. Bias reduction in traceroute sampling - towards a more accurate map of the internet. In A. Bonato and F. R. K. Chung, editors, *Algorithms and Models for the Web-Graph, 5th International Workshop, WAW 2007, San Diego, CA, USA, December 11-12, 2007, Proceedings*, volume 4863 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2007.

[54] L. A. Goldberg, P. W. Goldberg, C. A. Phillips, and G. B. Sorkin. Constructing computer virus phylogenies. *J. Algorithms*, 26(1):188–208, 1998.

[55] M. T. Goodrich, R. Jacob, and N. Sitchinava. Atomic power in forks: A super-logarithmic lower bound for implementing butterfly networks in the nonatomic binary fork-join model. In D. Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2141–2153. SIAM, 2021.

[56] M. T. Goodrich and R. Tamassia. *Algorithm Design and Applications*. Wiley, New York, NY, 2011.

[57] V. Grebinski. On the power of additive combinatorial search model. In W. Hsu and M. Kao, editors, *Computing and Combinatorics, 4th Annual International Conference, COCOON '98, Taipei, Taiwan, R.o.C., August 12-14, 1998, Proceedings*, volume 1449 of *Lecture Notes in Computer Science*, pages 194–203. Springer, 1998.

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

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

[60] D. Heckerman, C. Meek, and G. Cooper. A bayesian approach to causal discovery. In *Innovations in Machine Learning*, pages 1–28. Springer, 2006.

[61] J. J. Hein. An optimal algorithm to reconstruct trees from additive distance data. *Bulletin of mathematical biology*, 51(5):597–603, 1989.

[62] S. Honiden, M. E. Houle, and C. Sommer. Balancing graph Voronoi diagrams. In *Sixth International Symposium on Voronoi Diagrams*, pages 183–191, 2009.

[63] B. Huffaker, D. Plummer, D. Moore, and K. Claffy. Topology discovery by active probing. In *IEEE Symposium on Applications and the Internet (SAINT)*, pages 90–96, 2002.

[64] B. Huffaker, D. Plummer, D. Moore, and K. Claffy. Topology discovery by active probing. In *Proceedings 2002 Symposium on Applications and the Internet (SAINT) Workshops*, pages 90–96. IEEE, 2002.

[65] P. Hünermund and E. Bareinboim. Causal inference and data fusion in econometrics. *arXiv preprint arXiv:1912.09104*, 2019.

[66] G. W. Imbens. Potential outcome and directed acyclic graph approaches to causality: Relevance for empirical practice in economics. *Journal of Economic Literature*, 58(4):1129–79, 2020.

[67] A. Itai and M. Rodeh. The multi-tree approach to reliability in distributed networks. *Inf. Comput.*, 79(1):43–59, 1988.

[68] M. Jagadish and A. Sen. Learning a bounded-degree tree using separator queries. In S. Jain, R. Munos, F. Stephan, and T. Zeugmann, editors, *Algorithmic Learning Theory - 24th International Conference, ALT 2013, Singapore, October 6-9, 2013. Proceedings*, volume 8139 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2013.

[69] M. V. Janardhanan and L. Reyzin. On learning a hidden directed graph with path queries. *CoRR*, abs/2002.11541, 2020.

[70] J.-H. Ji, S.-H. Park, G. Woo, and H.-G. Cho. Generating pylogenetic tree of homogeneous source code in a plagiarism detection system. *International Journal of Control, Automation, and Systems*, 6(6):809–817, 2008.

[71] S. Kannan, C. Mathieu, and H. Zhou. Graph reconstruction and verification. *ACM Trans. Algorithms*, 14(4):40:1–40:30, 2018.

[72] V. King, L. Zhang, and Y. Zhou. On the complexity of distance-based evolutionary tree reconstruction. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*, pages 444–453. ACM/SIAM, 2003.

[73] M. Kocaoglu, K. Shanmugam, and E. Bareinboim. Experimental design for learning causal graphs with latent variables. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 7018–7028, 2017.

[74] M. Kurant, A. Markopoulou, and P. Thiran. Towards unbiased BFS sampling. *IEEE Journal on Selected Areas in Communications*, 29(9):1799–1809, 2011.

[75] V. Lagani, S. Triantafillou, G. Ball, J. Tegnér, and I. Tsamardinos. Probabilistic computational causal discovery for systems biology. *Uncertainty in biology*, pages 33–73, 2016.

[76] A. Lakhina, J. Byers, M. Crovella, and P. Xie. Sampling biases in IP topology measurements. In *IEEE INFOCOM*, volume 1, pages 332–341, 2003.

[77] G. D. Marmerola, M. A. Oikawa, Z. Dias, S. Goldenstein, and A. Rocha. On the reconstruction of text phylogeny trees: evaluation and analysis of textual relationships. *PloS one*, 11(12):e0167822, 2016.

[78] C. Mathieu and H. Zhou. A simple algorithm for graph reconstruction. In P. Mutzel, R. Pagh, and G. Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 68:1–68:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[79] N. Meinshausen, A. Hauser, J. M. Mooij, J. Peters, P. Versteeg, and P. Bühlmann. Methods for causal inference from gene perturbation experiments and validation. *Proceedings of the National Academy of Sciences*, 113(27):7361–7368, 2016.

[80] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, 2005.

[81] G. Moffa, G. Catone, J. Kuipers, E. Kuipers, D. Freeman, S. Marwaha, B. R. Lennox, M. R. Broome, and P. Bebbington. Using directed acyclic graphs in epidemiological research in psychosis: an analysis of the role of bullying in psychosis. *Schizophrenia bulletin*, 43(6):1273–1279, 2017.

[82] A. Pfeffer, C. Call, J. Chamberlain, L. Kellogg, J. Ouellette, T. Patten, G. Zacharias, A. Lakhotia, S. Golconda, J. Bay, et al. Malware analysis and attribution using genetic information. In *2012 7th International Conference on Malicious and Unwanted Software*, pages 39–45. IEEE, 2012.

[83] A. Pfeffer, C. Call, J. Chamberlain, L. Kellogg, J. Ouellette, T. Patten, G. Zacharias, A. Lakhotia, S. Golconda, J. Bay, R. Hall, and D. Scofield. Malware analysis and attribution using genetic information. In *7th International Conference on Malicious and Unwanted Software, MALWARE 2012, Fajardo, PR, USA, October 16-18, 2012*, pages 39–45. IEEE Computer Society, 2012.

[84] W. Piel, L. Chan, M. Dominus, J. Ruan, R. Vos, and V. Tannen. Treebase v. 2: A database of phylogenetic knowledge. e-biosphere, 2009.

[85] H. Prüfer. Neuer beweis eines satzes über permutationen. *Arch. Math. Phys*, 27(1918):742–744, 1918.

[86] A. G. Ranade. Optimal speedup for backtrack search on a butterfly network. In T. Leighton, editor, *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '91, Hilton Head, South Carolina, USA, July 21-24, 1991*, pages 40–48. ACM, 1991.

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

[88] G. Rong, W. Li, Y. Yang, and J. Wang. Reconstruction and verification of chordal graphs with a distance oracle. *Theor. Comput. Sci.*, 859:48–56, 2021.

[89] G. Rong, W. Li, Y. Yang, and J. Wang. Reconstruction and verification of chordal graphs with a distance oracle. *Theoretical Computer Science*, 859:48–56, 2021.

[90] G. Rong, Y. Yang, W. Li, and J. Wang. A divide-and-conquer approach for reconstruction of $\{c_{\geq 5}\}$-free graphs via betweenness queries. *Theor. Comput. Sci.*, 917:1–11, 2022.

[91] S. Sen and V. N. Muralidhara. The covert set-cover problem with application to network discovery. In M. S. Rahman and S. Fujita, editors, *WALCOM: Algorithms and Computation, 4th International Workshop, WALCOM 2010, Dhaka, Bangladesh, February 10-12, 2010. Proceedings*, volume 5942 of *Lecture Notes in Computer Science*, pages 228–239. Springer, 2010.

[92] B. Shen, C. W. Forstall, A. de Rezende Rocha, and W. J. Scheirer. Practical text phylogeny for real-world settings. *IEEE Access*, 6:41002–41012, 2018.

[93] Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. In W. Händler, editor, *CONPAR 81: Conference on Analysing Problem Classes and Programming for Parallel Computing, Nürnberg, Germany, June 10-12, 1981, Proceedings*, volume 111 of *Lecture Notes in Computer Science*, pages 314–327. Springer, 1981.

[94] G. Tardos. Query complexity, or why is it difficult to separate $NP^A \cap coNP^A$ from P^A by random oracles A? *Combinatorica*, 9(4):385–392, Dec 1989.

[95] G. Tardos. Query complexity, or why is it difficult to seperate NP $^{\mathrm{a}}$ cap co np$^{\mathrm{a}}$ from p$^{\mathrm{a}}$ by random oracles a? *Comb.*, 9(4):385–392, 1989.

[96] F. Tarissan, M. Latapy, and C. Prieur. Efficient measurement of complex networks using link queries. *CoRR*, abs/0904.3222, 2009.

[97] P. W. Tennant, E. J. Murray, K. F. Arnold, L. Berrie, M. P. Fox, S. C. Gadd, W. J. Harrison, C. Keeble, L. R. Ranker, J. Textor, et al. Use of directed acyclic graphs (dags) to identify confounders in applied health research: review and recommendations. *International journal of epidemiology*, 50(2):620–632, 2021.

[98] M. Thorup and U. Zwick. Compact routing schemes. In A. L. Rosenberg, editor, *13th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–10, 2001.

[99] S. Triantafillou, V. Lagani, C. Heinze-Deml, A. Schmidt, J. Tegner, and I. Tsamardinos. Predicting causal relationships from biological data: Applying automated causal discovery on mass cytometry data of human immune cells. *Scientific reports*, 7(1):1–11, 2017.

[100] L. G. Valiant. Parallelism in comparison problems. *SIAM J. Comput.*, 4(3):348–355, 1975.

[101] L. G. Valiant. Universality considerations in VLSI circuits. *IEEE Trans. Computers*, 30(2):135–140, 1981.

[102] L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, 1984.

[103] Z. Wang and J. Honorio. Reconstructing a bounded-degree directed tree using path queries. In *57th Annual Allerton Conference on Communication, Control, and Computing, Allerton 2019, Monticello, IL, USA, September 24-27, 2019*, pages 506–513. IEEE, 2019.

[104] Z. Wang and J. Honorio. Reconstructing a bounded-degree directed tree using path queries. In *57th IEEE Allerton Conference on Communication, Control, and Computing*, 2019. See also `arxiv.org/abs/1606.05183`.

[105] M. S. Waterman, T. F. Smith, M. Singh, and W. A. Beyer. Additive evolutionary trees. *Journal of theoretical Biology*, 64(2):199–213, 1977.

[106] A. C. Yao. Decision tree complexity and betti numbers. In F. T. Leighton and M. T. Goodrich, editors, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23-25 May 1994, Montréal, Québec, Canada*, pages 615–624. ACM, 1994.

[107] A. C. Yao. Decision tree complexity and betti numbers. *J. Comput. Syst. Sci.*, 55(1):36–43, 1997.

[108] A. C.-C. Yao. Decision tree complexity and Betti numbers. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 615–624, New York, NY, USA, 1994. ACM.

[109] X. Zhang and C. Phillips. A survey on selective routing topology inference through active probing. *IEEE Communications Surveys Tutorials*, 14(4):1129–1141, 2012.

[110] Y. Zhang, E. D. Kolaczyk, and B. D. Spencer. Estimating network degree distributions under sampling: An inverse problem, with applications to monitoring social media networks. *The Annals of Applied Statistics*, 9(1):166 – 199, 2015.