Optimal Algorithms for Multiway Search on Partial Orders

Shangqi Lu
Chinese University of Hong Kong
Hong Kong, China
sqlu@cse.cuhk.edu.hk

Wim Martens
University of Bayreuth
Bayreuth, Germany
wim.martens@uni-bayreuth.de

Matthias Niewerth
University of Bayreuth
Bayreuth, Germany
matthias.niewerth@uni-bayreuth.de

Yufei Tao
Chinese University of Hong Kong
Hong Kong, China
taoyf@cse.cuhk.edu.hk

ABSTRACT

We study *partial order multiway search* (POMS), which is a game between an algorithm $\mathcal{A}$ and an oracle, played on a directed acyclic graph $\mathcal{G}$ known to both parties. First, the oracle picks a vertex t in $\mathcal{G}$ called the *target*. Then, $\mathcal{A}$ needs to figure out which vertex is t by probing reachability. Specifically, in each *probe*, $\mathcal{A}$ selects a set Q of vertices in $\mathcal{G}$ whose size is bounded by a (pre-agreed) limit; the oracle reveals, for each vertex $q \in Q$, whether q can reach the target in $\mathcal{G}$. The objective of $\mathcal{A}$ is to minimize the number of probes. This problem finds use in crowdsourcing, distributed file systems, software testing, etc.

We describe an algorithm to solve POMS in $O(\log_{1+k} n + \frac{d}{k} \log_{1+d} n)$ probes, where n is the number of vertices in $\mathcal{G}$, k is the maximum permissible $|Q|$, and d is the largest out-degree of the vertices in $\mathcal{G}$. We further establish the algorithm's asymptotic optimality by proving a matching lower bound.

We also introduce a variant of POMS in the *external memory* (EM) computation model, which is the key to a *black-box approach* for converting a class of pointer-machine structures to their I/O-efficient counterparts. In the EM version of POMS, $\mathcal{A}$ is allowed to pre-compute a (disk-based) structure on $\mathcal{G}$ and is then required to clear its memory. The oracle (as before) picks a target t. $\mathcal{A}$ still needs to find t by issuing probes, except that the set Q in each probe must be read from the disk. The objective of $\mathcal{A}$ is now to minimize the number of I/Os. We present a structure that uses $O(n/B)$ space and guarantees discovering the target in $O(\log_B n + \frac{d}{B} \log_{1+d} n)$ I/Os where B is the block size, and n and d are as defined earlier. We establish the structure's asymptotic optimality by proving that any structure demands $\Omega(\log_B n + \frac{d}{B} \log_{1+d} n)$ I/Os to find the target in the worst case regardless of the space consumption.

CCS CONCEPTS

• **Theory of computation → Graph algorithms analysis**; **Data structures design and analysis**;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PODS '22, June 12–17, 2022, Philadelphia, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9260-0/22/06…$15.00
https://doi.org/10.1145/3517804.3524150

KEYWORDS

Partial Order; Graph Algorithms; Data Structures; Lower Bounds

ACM Reference Format:
Shangqi Lu, Wim Martens, Matthias Niewerth, and Yufei Tao. 2022. Optimal Algorithms for Multiway Search on Partial Orders. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, 13 pages. https://doi.org/10.1145/3517804.3524150

1 INTRODUCTION

Binary search admits the following interpretation from a graph's perspective. We have a directed path π of n vertices where an "oracle" has chosen a target vertex t. In each round, the search algorithm picks a vertex q on π; then the oracle reveals whether q can reach t. Similarly, the B-tree exemplifies the multiway version of the above process. In each round, the search algorithm picks a set Q of $B \geq 1$ vertices from π; then the oracle reveals which of those vertices can reach t. In both cases, the algorithm aims to discover t with the fewest rounds.

This paper studies *partial order multiway search* (POMS), which generalizes the aforementioned problems to arbitrary partial orders. The next subsection will define two versions of POMS. The first one is the classical formulation that finds use in a variety of database applications (Section 1.2) and has received considerable attention (see Section 1.3). The second is a new formulation, which bears significance in designing I/O-efficient data structures (Section 1.2).

1.1 Problem Definitions

(Classical) POMS. The problem can be cast as a game between an *oracle* and an *algorithm* $\mathcal{A}$, both of which are given a *single-rooted* DAG $\mathcal{G}$, i.e., $\mathcal{G}$ has a unique *root* (a vertex with in-degree 0). The game starts by having the oracle pick a *target* vertex t from $\mathcal{G}$. Then, $\mathcal{A}$ needs to find out which vertex is t by issuing (reachability) *probes*. Specifically, in each probe:

- $\mathcal{A}$ chooses a set Q of vertices with $|Q| \leq k$, where k is a problem parameter;
- the oracle then reveals, for each vertex $q \in Q$, whether q can reach t in $\mathcal{G}$.

Clearly, $\mathcal{A}$ can always discover t with $\lceil n/k \rceil$ probes where n is the number of vertices in $\mathcal{G}$. The challenge is to prove a better bound on the number of probes that holds regardless of t.

POMS in external memory. In the *external memory* (EM) model, a machine is equipped with (i) a *disk*, which is an unbounded sequence

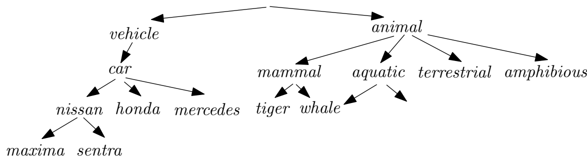

Figure 1: POMS in image classification with crowdsourcing

of words divided into *blocks* of $B \geq 2$ words, and (ii) *memory*, which is a sequence of M words. The *space* of a structure is the number of blocks occupied. An *input/output (I/O) operation* reads a block of data from the disk to memory[1]. M is assumed to be larger than B by a sufficiently large constant factor.[2]

We now introduce an EM version of POMS (referred to as EM POMS henceforth). Algorithm $\mathcal{A}$ is permitted to preprocess G into a structure whose information is stored in the disk, after which $\mathcal{A}$ clears the memory. To start a game, the oracle (as before) chooses a target t from $\mathcal{G}$. To carry out a probe, $\mathcal{A}$ needs to read a set Q of vertices *from the disk* into memory. The oracle then reveals the reachability (to t) for all the vertices in Q. $\mathcal{A}$ then clears up memory to finish the probe. Naively, $\mathcal{A}$ can store all the n vertices in $\lceil n/B \rceil$ blocks and discover t with $\lceil n/B \rceil$ I/Os. The challenge is reduce the number of I/Os without increasing the space asymptotically.

Remark. The assumption that $\mathscr{G}$ is single-rooted loses no generality. If $\mathscr{G}$ is not single-rooted, we can add a dummy vertex to $\mathscr{G}$ that serves as the new root and has an out-going edge to every root in the original $\mathscr{G}$.

We will analyze the cost of an algorithm using (i) the number n of vertices in $\mathcal{G}$, (ii) the maximum out-degree d of the vertices in $\mathcal{G}$, and (iii) the problem parameter k (in the classical version) or the model parameter B (in the EM version). As will be clear later, the performance of our algorithm does not depend on the number of edges in $\mathcal{G}$ and (for EM) the memory size M.

1.2 Motivation

Database relevance of (classical) POMS. The problem was first introduced to the database area in SIGMOD'19 [38]. A major application described in [38] is *image classification with crowdsourcing*, where the objective is to assign an appropriate label from a concept ontology to an image. As illustrated in Figure 1, an ontology is a DAG where each vertex is associated with a concept; furthermore, as we move down in the ontology, the concepts encountered are increasingly specialized. The application manifests the power of a crowdsourcing system where human beings are summoned to assist problem solving by answering (simple) questions with monetary rewards. Every question has the form "*is this an* **x**?" where **x** is a concept. Receiving a negative (resp., positive) answer to the question "*is this a* **vehicle**?", for instance, an algorithm can eliminate all the concepts that are (resp., are not) reachable from the vertex **vehicle**. The target t here is the concept eventually returned (e.g., **sentra**). As a crucial observation, although a human being is not aware of t, s/he can still answer questions based on straightforward reasoning and, thereby, play the role of oracle. As an example, when presented a car picture of the model sentra, a person will answer "yes" to "*is this a* **vehicle**?", no matter if s/he is aware of the concept **sentra** in the ontology. A crowdsourcing algorithm often asks $k > 1$ questions at a time to reduce interaction rounds.

As pointed out in [36], POMS also arises in distributed file systems. Suppose that server A maintains a backup of its file system (usually a tree but can also be a DAG, e.g., in Unix) in a remote server B. Periodically, the two servers need to synchronize their copies, which requires identifying the folders whose content has changed since the last synchronization. If a folder has an identical checksum at the two servers, (with high probability) it and its subfolders have incurred no changes. Based on this property, a POMS algorithm can find a modified folder with small communication between the two servers.

The reader may refer to [6, 36, 38] for more POMS applications in software testing, relational databases, and workflow management.

Significance of POMS in EM. Making a "conventional" data structure (i.e., designed for memory-resident data) I/O-efficient is nontrivial because one must take into account the effects of reading/writing in *blocks*. Ideally, we would like to have a generic reduction to convert an arbitrary internal-memory structure to an EM version with excellent performance. Designing such reductions is still a major challenge today.

We observe that a solution to EM POMS offers an interesting reduction that works on a class of *region-based* structures satisfying the following properties:

- The structure is a single-rooted DAG $\mathcal{G}$ where each vertex has out-degree at most d (the in-degree can be arbitrary).
- Each vertex u stores a *region* reg_u, which is a subset of a search space $\mathbb{Q}$ and can be described in $O(1)$ words.
- All the leaves (i.e., vertices with out-degree 0) have disjoint regions whose union is $\mathbb{Q}$.
- For each vertex u, reg_u is the region union of all the leaves reachable from u.
- A *query* chooses an element $q \in \mathbb{Q}$ and returns the (only) leaf whose region covers q. For any vertex u, whether q falls in reg_u can be decided in constant time.

For example, the binary search tree is a region-based structure where the region of each node is an interval of the form $[x, y)$ where x and y are real values. So are the quad-tree and the kd-tree where a node's region is a multidimensional rectangle. We will discuss more sophisticated region-based structures in Section 1.4.

In a region-based structure, a query can be modeled as an instance of POMS. Let t be the leaf whose region contains the query element q; we will treat t as the target selected by the oracle. Given the reg_u of a vertex u, we can play the oracle's role by deciding whether u can reach t in $O(1)$ time: the answer is yes if and only if $q \in \text{reg}_u$. An algorithm $\mathcal{A}$ solving EM POMS implies an EM version of the structure $\mathcal{G}$ as follows. In preprocessing, if $\mathcal{A}$ packs a set S of vertices in a disk block, we store S, as well as the regions of the vertices therein, in $O(|S|/B) = O(1)$ disk blocks. In answering a query, if $\mathcal{A}$ reads the block on S, we read the corresponding $O(1)$ blocks to acquire all the information needed to resolve reachability

[1]In general, the EM model also allows *write I/Os*, each of which overwrites a disk block using B words in memory. However, we do not need to be concerned with such I/Os in this paper.

[2]The strictest EM model [2] requires an algorithm to work even if $M \geq 2B$. However, as shown in [25], any algorithm designed for $M = \mu B$ for a constant $\mu > 2$ can be adapted to work under $M = 2B$ with only a constant blowup in the number of I/O operations.

problem	ref.	cost	remark
POMS	[5]	$O(d \log n)$	$\mathcal{G}$ is a tree and $k = 1$
POMS	[21, 23, 31]	$O(d \log_{1+d} n)$	$\mathcal{G}$ is a tree and $k = 1$
POMS	[38]	$O((\log n)(\log_{1+k} n) + \frac{d}{k} \log_{1+d} n)$	any DAG $\mathcal{G}$ and any k
POMS	**this paper**	$O(\log_{1+k} n + \frac{d}{k} \log_{1+d} n)$	any DAG $\mathcal{G}$ and any k
POMS	[5]	$\Omega(d \log_{1+d} n)$	$k = 1$
POMS	[38]	$\Omega(\frac{d}{k} \log_{1+d} n)$	any k
POMS	**this paper**	$\Omega(\log_{1+k} n + \frac{d}{k} \log_{1+d} n)$	any k
EM POMS	**this paper**	$O(\log_B n + \frac{d}{B} \log_{d+1} n)$	space consumption $O(n/B)$
EM POMS	**this paper**	$\Omega(\log_B n + \frac{d}{B} \log_{d+1} n)$	holds regardless of how much space is used

Table 1: Summary of the previous and new results

(to t) for the vertices in S. This enables us to simulate the execution of $\mathcal{A}$ with only a constant blowup in the I/O cost.

1.3 Previous Results and Related Work

To appreciate the POMS literature, it is important to distinguish between the *instance-oriented* and *class-oriented* categories which have drastically different objectives. We will start with the former category before discussing the latter.

Instance-oriented POMS. Consider any algorithm $\mathcal{A}$ for POMS. Given an input $\mathcal{G}$, define $cost_k(\mathcal{A}, \mathcal{G}, t)$ as the cost of $\mathcal{A}$ on $\mathcal{G}$ when the target is t. We can measure the instance-oriented quality of $\mathcal{A}$ by $maxcost^{\mathrm{I}}_k(\mathcal{A}, \mathcal{G}) = \max_t cost_k(\mathcal{A}, \mathcal{G}, t)$, namely, the largest cost on the worst t in $\mathcal{G}$. Because (i) the execution of $\mathcal{A}$ (given $\mathcal{G}$ and t) is merely a sequence of probes and (ii) only a finite number of execution sequences exist (one for each t), there are only a finite number of possible $\mathcal{A}$. Thus, the problem of finding an optimal algorithm $\mathcal{A}^*$ (with the lowest $maxcost^{\mathrm{I}}_k(\mathcal{A}^*, \mathcal{G})$) is decidable. The challenge is to understand how much we can reduce the computation time.

The problem is best understood in the special case where $\mathcal{G}$ is a (rooted) tree and $k = 1$. In that case, Ben-Asher et al. [6] were the first to show that an $\mathcal{A}^*$ can be found in polynomial time. Their work motivated a line of research looking for faster solutions [18, 20, 26, 31–33, 36]. The problem turns out to be solvable in $O(n)$ time. This was first stated by Mozes et al. [33]; later, Dereniowski [20] pointed out the problem's equivalence to another problem known as *edge ranking*, which had already been settled earlier in $O(n)$ time by Lam and Yue [32]. In contrast, the problem of computing an $\mathcal{A}^*$ on a DAG $\mathcal{G}$ is NP-hard [9] (even if $k = 1$). Arkin et al. [4] showed that, for any DAG $\mathcal{G}$ and $k = 1$, one can obtain in polynomial time an $\mathcal{A}$ whose $maxcost^{\mathrm{I}}_k(\mathcal{A}, \mathcal{G})$ is higher than $maxcost^{\mathrm{I}}_k(\mathcal{A}^*, \mathcal{G})$ by a factor of $O(\log n)$.[3] We are aware of no results for $k > 1$ even when $\mathcal{G}$ is a tree.

For $k = 1$, instance-oriented POMS has also been studied under other variants [1, 9–16, 19, 22, 27, 29, 30], which differ in whether (i) the cost of a probe depends on the vertex supplied, (ii) the goal is to minimize the cost of the worst t or the average cost over a distribution of t, and (iii) the answer of the oracle can be noisy.

[3] A better approximation ratio $O(\log n / \log \log n)$ was claimed in [20] but unfortunately is not correct, as has been confirmed by our personal communication with the author of [20].

Class-oriented POMS. Unlike the previous category that focuses on *computability*, the class-oriented category is *graph theoretic* in nature. Given a set $\mathscr{C}$ of single-rooted DAGs, the quality of $\mathcal{A}$ is measured by $maxcost^{\mathrm{C}}_k(\mathcal{A}, \mathscr{C}) = \max_{\mathcal{G} \in \mathscr{C}} maxcost^{\mathrm{I}}_k(\mathcal{A}, \mathcal{G})$, namely, the largest cost on the worst $\mathcal{G}$ in $\mathscr{C}$. Define

$$minmaxcost_k(\mathscr{C}) \quad = \quad \min_{\mathcal{A}} maxcost^{\mathrm{C}}_k(\mathcal{A}, \mathscr{C}) \qquad (1)$$

that is, the lowest upper bound that an algorithm can possibly place on its cost regardless of (i) the input $\mathcal{G} \in \mathscr{C}$ and (ii) the target t in $\mathcal{G}$. The objective is to understand the function $minmaxcost_k(\mathscr{C})$ for important classes $\mathscr{C}$. Define

$$\begin{aligned} \mathscr{G}(n,d) \quad &= \quad \{ \text{single-rooted DAG } G \mid G \text{ has } n \text{ vertices and maximum out-degree } d \} \\ \mathscr{T}(n,d) \quad &= \quad \{ \mathcal{G} \in \mathscr{G}(n,d) \mid \mathcal{G} \text{ is a tree} \} \end{aligned}$$

Clearly, $minmaxcost_k(\mathscr{T}(n,d)) \leq minmaxcost_k(\mathscr{G}(n,d))$.

Focusing on $\mathscr{T}(n,d)$ and $k = 1$, Ben-Asher and Farchi [5] showed that $minmaxcost_1(\mathscr{T}(n,d))$ is $\Omega(d \log_{1+d} n)$ but $O(d \log n)$, leaving a gap of $\Theta(\log(1+d))$ in between. Laber and Nogueira [31] tightened the upper bound and proved that $minmaxcost_1(\mathscr{T}(n,d)) \in \Theta(d \log_{1+d} n)$ (see also [21, 23] where the same result was derived). Regarding $\mathscr{G}(n,d)$ and arbitrary $k \geq 1$, Tao et al. [38] obtained $minmaxcost_k(\mathscr{G}(n,d)) = \Omega(\frac{d}{k} \log_{1+d} n)$ and $minmaxcost_k(\mathscr{G}(n,d)) = O((\log n)(\log_{1+k} n) + \frac{d}{k} \log_{1+d} n)$. In fact, the lower bound of [38] holds even when replacing $\mathscr{G}(n,d)$ with $\mathscr{T}(n,d)$.

EM. We are aware of no solutions to the EM version of POMS even when $\mathcal{G}$ is a tree.

1.4 Our Results

Table 1 compares this paper's results to the previous ones. Next, we will discuss our findings and their implications in detail.

POMS. Our first main contribution is to settle POMS optimally:

Theorem 1. *Both statements below are true about POMS:*

- *There is an algorithm that can find the target in $O(\log_{1+k} n + \frac{d}{k} \log_{1+d} n)$ probes.*
- *Any POMS algorithm must perform $\Omega(\log_{1+k} n + \frac{d}{k} \log_{1+d} n)$ probes to find the target in the worst case.*

The theorem essentially shows

$$minmaxcost_k(\mathscr{G}(n,d)) \quad = \quad \Theta(\log_{1+k} n + (d/k) \log_{1+d} n). \quad (2)$$

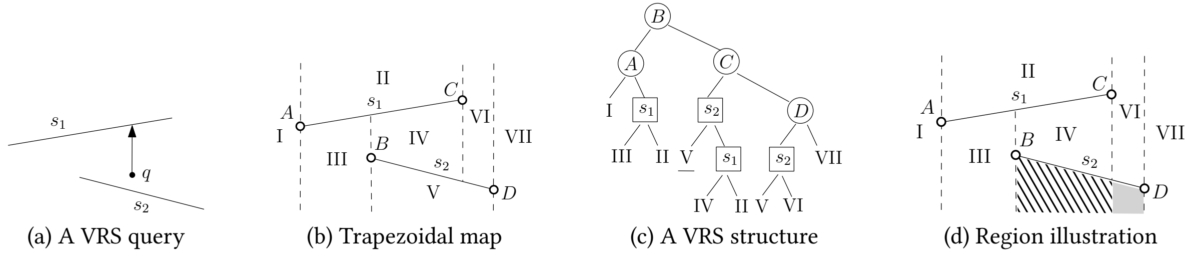

Figure 2: A region-based structure on the VRS problem

Our lower bound in the second bullet holds even if $\mathcal{G}$ comes from $\mathscr{T}(n, d)$. This reveals a somewhat unexpected fact: POMS on *trees* is as hard as on *DAGs*, or formally:

Corollary 2. *$minmaxcost_k(\mathscr{T}(n, d)) = \Theta(minmaxcost_k(\mathscr{G}(n, d)))$.*

Theorem 1 has a further implication in the instance-oriented category:

Corollary 3. *Consider any tree $\mathcal{G} \in \mathscr{T}(n, d)$ and an arbitrary integer $k \in [1, n]$. Denote by $\mathcal{A}^*$ an algorithm achieving the lowest $maxcost^{\mathrm{I}}_k(\mathcal{A}^*, \mathcal{G})$. In time polynomial to n, we can obtain an algorithm $\mathcal{A}$ satisfying*

$$\frac{maxcost^{\mathrm{I}}_k(\mathcal{A}, \mathcal{G})}{maxcost^{\mathrm{I}}_k(\mathcal{A}^*, \mathcal{G})} = O\left(\frac{\log n}{\log(1 + k) + \log\log n}\right).$$

The EM version. Our second main contribution is to optimally settle POMS in EM:

Theorem 4. *Both statements below are true about EM POMS:*

- *There is a structure of $O(n/B)$ space that guarantees discovering the target in $O(\log_B n + \frac{d}{B}\log_{1+d} n)$ I/Os.*
- *In the worst case, every structure must incur $\Omega(\log_B n + \frac{d}{B}\log_{1+d} n)$ I/Os to find the target, regardless of the space usage.*

Note that our structure's space and I/O complexities (the first bullet) do not depend on the number of edges in the input DAG. Furthermore, when $d = O(B)$, the I/O cost becomes $O(\log_B n)$.

Generic EM reduction for region-based structures. Combining Theorem 4 with the reduction in Section 1.2, we can now assert that *any* region-based structure with n vertices has an EM counterpart that uses $O(n/B)$ space and answers a query in $O(\log_B n + \frac{d}{B}\log_{d+1} n)$ I/Os. Next, we illustrate the reduction using an internal-memory structure that (i) is well known, (ii) solves an important problem, and (iii) contains an appropriate amount of sophistication to demonstrate the reduction's power.

Vertical ray shooting (VRS) is a problem defined as follows. The input is a set S of disjoint line segments in $\mathbb{R}^2$. Given a point q in $\mathbb{R}^2$, a query reports the first segment in S (if any) hit by the upward ray emanating from q. Figure 2(a) shows an example where the query answer is s_1. The objective is to store S in a data structure to answer all queries efficiently. This is a fundamental problem with profound significance to database systems; see [25] for how it stands at the core of point location queries and nearest neighbor search, and [7] for its relevance to temporal databases.

For each segment $s \in S$, shoot upward and downward rays from each of its two endpoints. Each ray stops as soon as hitting a segment in S and, accordingly, turns into a segment. These rays (some have turned into segments) together with S form a planar subdivision of $\mathbb{R}^2$, which is called the *trapezoidal map* on S. Figure 2(b) shows the trapezoidal map for the input in Figure 2(a). Answering a query with some point q is identical to finding the trapezoid in the trapezoidal map covering q (e.g., trapezoid IV in Figure 2(b)).

In [34], Mulmuley introduced the idea of building a binary tree where (i) each internal node stores either a segment in S or an endpoint of such a segment, and (ii) each leaf node stores a trapezoid in the trapezoidal map. Given a point q, we can identify the trapezoid containing q by traversing a root-to-leaf path. Figure 2(c) shows a binary tree for our example. Consider the point q in Figure 2(a). At the root B, we navigate to the right child C because q is on the right of B (by x-coordinate). From node C, we descend to the left child s_2 because q is on the left of C. At node s_2, we check whether q is below or above s_2; since the answer is "above", we move to the right child s_1. At node s_1, we go to the left child because q is below s_1. This takes us to the target trapezoid IV.

Each node u in the binary tree is implicitly associated with a region reg_u in $\mathbb{R}^2$. The root is associated with the entire $\mathbb{R}^2$. Inductively, (i) if an internal node u stores a point p, the region of its left (resp., right) child includes all the points in reg_u whose x-coordinates are smaller (resp., larger) than that of p; (ii) if an internal node u stores a segment s, the region of its left (right, resp.) child includes all the points in reg_u below (resp., above) of s. In Figure 2(d), we have divided trapezoid V into two parts such that the left (resp., right) part is the region of the leaf labeled as $\underline{\mathrm{V}}$ (resp., V) in Figure 2(c). It is then easy to verify that the binary tree is indeed a region-based structure.

Binary trees satisfying Mulmuley's description are not unique. Some of them can have $\Theta(n^2)$ nodes where $n = |S|$. In [37], Seidel gave a randomized algorithm to produce a binary tree of size $O(n)$ in expectation. This proves the existence of at least one binary tree having $O(n)$ nodes. Theorem 4 immediately gives an EM structure of $O(n/B)$ space that answers any query in $O(\log_B n + \frac{d}{B}\log_{d+1} n) = O(\log_B n)$ I/Os, noticing that the parameter d is 2 (binary tree).

Several remarks are in order:

- The algorithm of [37] may yield a binary tree with a large height (even when the tree has size $O(n)$). Seidel [37] gave a non-trivial analysis on how likely the height is small. In contrast, we can take an *arbitrarily* unbalanced binary tree with $O(n)$ nodes and obtain an EM structure of $O(\log_B n)$ query cost.
- In EM, the known VRS structures (see [8, 25, 35] for a full literature review) achieving $O(n/B)$ space and $O(\log_B n)$ query cost were obtained using the *partial persistence* [3, 25] and

the *topology tree* [8, 24] techniques. Our method is drastically different and conceptually neater.

- Our reduction in Section 1.2 is actually more powerful than demonstrated above because it applies even if the (internal-memory) structure is a DAG.

2 PRELIMINARIES

Basic concepts and notation. Henceforth, every "tree" should be understood as a *rooted* tree. The *size* of a tree T, denoted as $|T|$, is the number of nodes. Notation $u \in T$ (resp., $u \notin T$) indicates that u is (resp., is not) a node of T. Notation $parent(u)$ gives the parent node of u. The subtree of a node $u \in T$ — denoted as T_u — is the tree that is induced by the descendants of u in T and is rooted at u.

Reserving $\mathcal{G}$ for the input graph of POMS, we will use symbol G when referring to a general single-rooted DAG. A tree T is *contained* in G if every edge of T belongs to G. Given such a tree T, $G[T]$ represents the subgraph of G induced by the vertices in T; note that $G[T]$ must be a single-rooted DAG. If node u can reach node v in G, we will say that u can *G-reach* v.

Shielding. Given nodes u and v in a tree T, we define $T_u \ominus \{v\}$ as:

- T_u if $u = v$;
- what remains in T_u after removing T_v (note that if $v \notin T_u$, $T_u \ominus \{v\} = T_u$).

We will refer to $\ominus$ as the *shield* operator. Given a node $u \in T$ and a set $S = \{v_1, v_2, ..., v_x\}$ where $v_i \in T$ for each $i \in [1, x]$, we define

$$T_u \ominus S \quad = \quad T_u \ominus \{v_1\} \ominus \{v_2\} ... \ominus \{v_x\}.$$

Note that $T_u \ominus S$ is always a non-empty tree because it must contain u itself.

Heavy-path depth first search tree. Consider performing a depth first search (DFS) on a single-rooted DAG G starting from its root. Recall that DFS uses a stack to manage the vertices that have been discovered but may still have undiscovered out-neighbors. Vertices are assigned three colors: *white* (never in stack), *gray* (in stack), and *black* (already popped out). At each step, the traditional DFS would process an arbitrary white out-neighbor v of the vertex u_{top} that currently tops the stack. The *heavy path depth first search* (HPDFS), on the other hand, processes the white out-neighbor v_{best} of u_{top} able to G-reach the most white vertices via white paths[4].

HPDFS defines a tree T — the *HPDFS-tree* [38] — where a node u parents another v if the latter is discovered while the former tops the stack. It also defines a total order $\prec$ on the vertices in G: $u \prec v$ (read as u *smaller than* v or v *larger than* u) if u enters the stack before v. For two sibling nodes u and v in T such that $u \prec v$, we call u a *left sibling* of v and, conversely, v a *right sibling* of u.

Appendix A proves the following properties of T:

Lemma 5. *Let T be an HPDFS-tree of a single-rooted DAG G.*

- **(Order property)** *If u is a left sibling of v in T, then (i) $u' \prec v$ for every $u' \in T_u$ and (ii) $u \prec v'$ for every $v' \in T_v$.*
- **(No-cross-reachability property)** *If $u \prec v$ and $v \notin T_u$, then u cannot G-reach v' for any $v' \in T_v$.*
- **(Path-descendants property)** *If $w \in T_u$, then $v \in T_u$ for every node v that lies on at least one u-to-w path in G.*
- **(Subtree-size property)** *If u is a left sibling of v in T, then $|T_u| \geq |T_v|$.*

[4] A white path is a path that includes only white vertices. If two or more nodes satisfy this condition, v_{best} can be any of them.

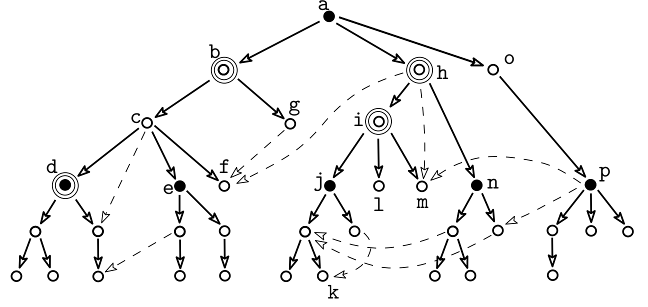

Figure 3: A running example. G is the graph represented by both the solid and dashed edges. An HPDFS T of G is indicated by the solid edges. The labels on the nodes are consistent with the total order $\prec$. The black nodes constitute the 8-separator $\Sigma = \{a, d, e, j, n, p\}$ of T. The nodes of LFU(Σ) = {b, d, h, i} are shown using concentric circles.

Example. Consider G as the graph that has all the solid and dashed edges in Figure 3. The tree in solid edges represents an HPDFS-tree T of G. The alphabetic order of the node labels reflects the total order $\prec$ (the labels on some nodes are omitted). Because node b precedes h in $\prec$ and h $\notin T_{\mathsf{b}}$, the no-cross-reachability property assures us that b cannot G-reach any node in T_{h}. Because k $\in T_{\mathsf{h}}$, the path-descendants property asserts that every path from h to k in G can contain only nodes in T_{h}. The other two properties are easy to understand. □

3 NEW RESULTS IN GRAPH THEORY

Crucial to our POMS algorithms is a suite of new graph-theoretic results which we present in this section. Our discussion will revolve around a single-rooted graph G having n nodes, an arbitrary HPDFS-tree T of G, and an ordering $\prec$ on the vertices of G decided by T.

3.1 Separators

It is well-known [28] that T must contain a node whose removal disconnects T into trees each of at most $n/2$ nodes. In Appendix B, we prove a more general fact:

Lemma 6. *Let T be an HPDFS-tree of a single-rooted DAG with n vertices. For any $\lambda \in [2, n]$, there exists a set S of at most $\lambda - 1$ nodes whose removal disconnects T into trees each of at most n/λ nodes.*

More specifically, Appendix B shows that such an S can be found using the algorithm below:

construct-separator
1. $S \leftarrow \emptyset$; $T' \leftarrow T$; $\tau \leftarrow n/\lambda$
2. **while** $|T'| \geq \lfloor \tau \rfloor + 1$ **do**
3. $u \leftarrow$ the smallest node (under $\prec$) in T' s.t. $|T'_u| \geq \lfloor \tau \rfloor + 1$ but $|T'_v| \leq \lfloor \tau \rfloor$ for each child v of u /* remark: u definitely exists */
4. add u to S; remove T'_u from T'
5. **return** S

We define the λ*-separator* of T to be a set Σ determined as follows:

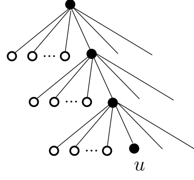

Figure 4: The left flank of u includes all the white nodes.

- if S (the above algorithm's output) contains the root of T, then $\Sigma = S$;
- otherwise, $\Sigma = S \cup \{\text{root of } T\}$.

Clearly, $|\Sigma| \le \lambda$ by Lemma 6.

Example. Assume that T is the tree in solid edges as shown in Figure 3 (T has 36 nodes). The 8-separator of T is $\Sigma = \{\texttt{a}, \texttt{d}, \texttt{e}, \texttt{j}, \texttt{n}, \texttt{p}\}$; the above algorithm finds the nodes of Σ in the order d, e, j, n, p, a. Figure 3 colors all the nodes of Σ in black. □

3.2 Left Flanks and Flank/Grand Unions

In this subsection, we introduce three concepts important to our discussion: *left flank*, *left-flank union*, and *grand union*.

Fix an arbitrary node $u \in T$; and consider the root-to-u path π in T. We define the *left flank* of u — denoted as LF(u) — using the following procedure:

- Initialize an empty set S.
- Consider each node v on π with $v \ne u$. Let v' be the child of v on π. Add to S all the left siblings of v' in T.
- The S after the previous step is LF(u).

See Figure 4 for an illustration.

Let Σ be the λ-separator of T. The *left-flank union* (LFU) of Σ is

$$\text{LFU}(\Sigma) \quad = \quad \bigcup_{u \in \Sigma} \text{LF}(u)$$

and the *grand union* (GU) of Σ is

$$\text{GU}(\Sigma) \quad = \quad \Sigma \cup \text{LFU}(\Sigma).$$

Example. Consider again the graph G in Figure 3 with $\lambda = 8$. As explained before, $\Sigma = \{\texttt{a}, \texttt{d}, \texttt{e}, \texttt{j}, \texttt{n}, \texttt{p}\}$. The left flank of node p is $\text{LF}(\texttt{p}) = \{\texttt{b}, \texttt{h}\}$, while $\text{LF}(\texttt{n}) = \{\texttt{b}, \texttt{i}\}$. It is easy to verify that $\text{LFU}(\Sigma) = \{\texttt{b}, \texttt{d}, \texttt{h}, \texttt{i}\}$ and $\text{GU}(\Sigma) = \{\texttt{a}, \texttt{b}, \texttt{d}, \texttt{e}, \texttt{h}, \texttt{i}, \texttt{j}, \texttt{n}, \texttt{p}\}$. □

In Appendix C, we prove:

Lemma 7. *$|\text{LFU}(\Sigma)| \le |\Sigma| - 1$.*

It is clear from Lemmas 6 and 7 that $|\text{GU}(\Sigma)| < 2\lambda$.

3.3 Stars

In this subsection, we introduce *star*, which is yet another concept crucial to our technical development.

Fix a vertex t in G and a non-empty set S of vertices in G that includes the root of G. The *star of S for t* is the smallest (under $\prec$) node $s^* \in S$ satisfying:

- **(Condition C1)** s^* can G-reach t;
- **(Condition C2)** there does not exist another node $s \in S$ such that $s \in T_{s^*}$ and s can G-reach t.

See Figure 5 for an illustration. Note that the root's presence in S guarantees the existence of s^*.

Example. Consider Figure 3 with $t = \texttt{k}$ and $S = \{\texttt{a}, \texttt{b}, \texttt{h}, \texttt{l}, \texttt{m}, \texttt{p}\}$. The star s^* of S for t is h. □

3.4 Three Path Preservation Lemmas

This subsection will demonstrate the importance of left flanks, grand unions, and stars in preserving reachability. We will establish three lemmas that are useful in various scenarios.

Preservation using a root-containing set. Appendix D proves:

Lemma 8. *Let T be an HPDFS-tree of a single-rooted DAG G. Let S be a set of vertices that includes the root of G, t be a vertex in G, and s^* be the star of S for t. Suppose that $t \in T_{s^*}$ and $t \ne s^*$. If $s^\#$ is the smallest (under the total order decided by T) child of s^* in T that can G-reach t, then*

- *$t \in T_{s^\#}$;*
- *every $s^\#$-to-t path in G is present in $G[T_{s^\#} \ominus S]$.*

Example. Consider Figure 3 with $t = \texttt{k}$ and $S = \{\texttt{a}, \texttt{b}, \texttt{h}, \texttt{l}, \texttt{m}, \texttt{p}\}$. As mentioned, the star s^* of S for t is h. Both child nodes of h (i.e., i and n) can G-reach $t = \texttt{k}$. Hence, $s^\# = \texttt{i}$. $T_{s^\#} \ominus S$ — the tree obtained by "shielding" $T_{\texttt{i}}$ with S — consists of the edges in $T_{\texttt{j}}$ plus the edge (i, j). Note that i has two paths to k in G, both of which are preserved in $G[T_{s^\#} \ominus S]$. □

Preservation using a separator. We prove in Appendix E:

Lemma 9. *Let T be an HPDFS-tree of a single-rooted DAG G, t be a vertex in G, Σ be the k-separator of T, and s^* the star of Σ for t. If s^{**} is the smallest node in $\text{LF}(s^*) \cup \{s^*\}$ able to G-reach t, then*

- *$t \in T_{s^{**}}$;*
- *every s^{**}-to-t path in G is present in $G[T_{s^{**}} \ominus \Sigma]$.*

Example. Consider Figure 3 with $\lambda = 8$ and $t = \texttt{m}$. Recall that $\Sigma = \{\texttt{a}, \texttt{d}, \texttt{e}, \texttt{j}, \texttt{n}, \texttt{p}\}$ and, hence, $s^* = \texttt{p}$. Thus, $\text{LF}(s^*) \cup \{s^*\} = \{\texttt{b}, \texttt{h}, \texttt{p}\}$, giving $s^{**} = \texttt{h}$. $T_{s^{**}} \ominus \Sigma$ is a tree with four nodes: h, i, l and m. G has two h-to-m paths in G, both of which are preserved in $G[T_{s^{**}} \ominus \Sigma]$. □

Preservation using a grand union. We prove in Appendix F:

Lemma 10. *Let T be an HPDFS-tree of a single-rooted DAG G, t be a vertex in G, Σ be the k-separator of T, and s^* is the star of $\text{GU}(\Sigma)$ for t. Then*

- *$t \in T_{s^*}$;*
- *every s^*-to-t path in G is preserved in $G[T_{s^*} \ominus \text{GU}(\Sigma)]$.*

Example. Consider Figure 3 with $\lambda = 8$. As explained in Section 3.2, $\text{GU}(\Sigma) = \{\texttt{a}, \texttt{b}, \texttt{d}, \texttt{e}, \texttt{h}, \texttt{i}, \texttt{j}, \texttt{n}, \texttt{p}\}$. If $t = \texttt{f}$, then $s^* = \texttt{b}$. The tree $T_{\texttt{b}} \ominus \text{GU}(\Sigma)$ has nodes b, c, f and g. G has two b-to-f paths, both preserved in $G[T_{s^*} \ominus \text{GU}(\Sigma)]$. □

4 A POMS ALGORITHM

This section will present our first POMS algorithm, assuming $k \ge 2$ (if $k = 1$, manually increase it to 2).

Define $\mathcal{G}_0 = \mathcal{G}$. We perform POMS by shrinking the input graph $\mathcal{G}$ into smaller single-rooted DAGs $\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_h$ (for some $h \ge 1$) where the last DAG $\mathcal{G}_h$ becomes small enough to be solvable with a single probe.

Specifically, in the i-th iteration, we are given a single-rooted DAG $\mathcal{G}_{i-1}$ with n_{i-1} vertices and produce a single-rooted DAG $\mathcal{G}_i$ having three properties:

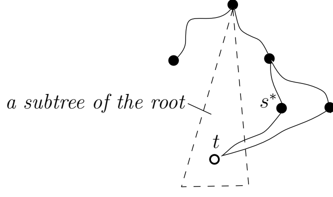

Figure 5: S is the set of black vertices. The star of S for t is s^*.

- (**size reduction**) $\mathcal{G}_i$ has $n_i \le n_{i-1}/k$ vertices;
- (**target containment**) $\mathcal{G}_i$ contains the target t;
- (**path preserving**) if r is the root of $\mathcal{G}_i$, then every r-to-t path in $\mathcal{G}_{i-1}$ is present in $\mathcal{G}_i$.

Graph $\mathcal{G}_i$ has an important property:

> A vertex u can $\mathcal{G}_i$-reach the target t
> if and only if u can $\mathcal{G}$-reach t.

The "only-if direction" is trivial. The "if direction" follows from the fact that *every r-to-t path in the original graph $\mathcal{G}$ is preserved in $\mathcal{G}_i$ for every i*, which can proved with a simple inductive argument (induction on i) applying the path-preserving property.

Next, we prove the existence of such a nice $\mathcal{G}_i$ by giving an algorithm for its computation. Thanks to the above property, we can speak about "performing" a reachability probe on $\mathcal{G}_{i-1}$, with the understanding that the probe should really be carried out on $\mathcal{G}$.

Consider iteration $i \ge 1$. If $\mathcal{G}_{i-1}$ has $n_{i-1} \le k$ vertices, t can be found trivially with one probe. Otherwise, we generate $\mathcal{G}_i$ in two phases.

Phase 1. Construct an HPDFS-tree T of $\mathcal{G}_{i-1}$ and the k-separator Σ of T. Let $\prec$ be the total order defined by T on the vertices of $\mathcal{G}_{i-1}$. As $|\Sigma| \le k$ (Section 3.1), with a single probe we can obtain all the vertices in Σ able to $\mathcal{G}_{i-1}$-reach t (there must be at least one such vertex because Σ includes the root of $\mathcal{G}_{i-1}$). We can then identify the star s^* of Σ for t (Section 3). By Lemma 7, $|\mathrm{LF}(s^*)| \le |\mathrm{LFU}(\Sigma)| \le k-1$. Thus, with another probe we can figure out which nodes in $\mathrm{LF}(s^*) \cup \{s^*\}$ can $\mathcal{G}_{i-1}$-reach t; let s^{**} be the smallest (under $\prec$) among those nodes.

Phase 2. It must hold that either $s^{**} \notin \Sigma$ or $s^{**} = s^*$.[5] In the former case, we finalize $\mathcal{G}_i$ to be $\mathcal{G}_{i-1}[T_{s^{**}} \ominus \Sigma]$.

Now consider the case where $s^{**} = s^*$. We aim to find the smallest (under $\prec$) child $s^\#$ of s^* (in T) that can $\mathcal{G}_{i-1}$-reach t. For this purpose, it suffices to probe the reachability (to t) for the child nodes of s^* in ascending order of $\prec$ (each probe includes k nodes, except possibly the last probe) and stop as soon as encountering $s^\#$. If $s^\#$ does not exist, we declare $t = s^*$ and finish the whole algorithm. Otherwise, we set $\mathcal{G}_i$ to $\mathcal{G}_{i-1}[T_{s^\#} \ominus \Sigma]$.

The lemma below, whose proof (Appendix G) utilizes Lemmas 8 and 9, ascertains our algorithm's correctness:

Lemma 11. *If the algorithm does not finish in iteration i, $\mathcal{G}_i$ has the size-reduction, target-containment, and path-preserving properties.*

As analyzed in Appendix H, our algorithm performs $O(\log_{1+k} n + \frac{d}{k}\log_{1+d} n)$ probes, which establishes the first bullet of Theorem 1.

[5] If $s^{**} \in \Sigma$ but $s^{**} \ne s^*$, then $s^{**} \prec s^*$ in which case s^* cannot be the *smallest* node satisfying Conditions C1 and C2 (Section 3).

5 AN EM POMS ALGORITHM

This section will settle POMS in the EM model. In Section 5.1, we design another POMS algorithm with the performance guarantees in Theorem 1. Compared to the one in Section 4, the new algorithm is more complex but is endowed with two special properties and, thus, can be adapted to work in EM as shown in Section 5.2.

5.1 Another POMS algorithm

Our new POMS algorithm follows the same iterative framework as in Section 4. The i-th iteration ($i \ge 1$) finds t with one probe if $\mathcal{G}_{i-1}$ has at most k vertices; otherwise, it produces $\mathcal{G}_i$ in two phases but in a way different from Section 4.

Phase 1. Construct an HPDFS-tree T of $\mathcal{G}_{i-1}$ (which determines a total order $\prec$) and find the k-separator Σ of T. As $|\mathrm{GU}(\Sigma)| < 2k$ (Section 3), with at most two probes we can find the nodes in $\mathrm{GU}(\Sigma)$ capable of $\mathcal{G}_{i-1}$-reaching t and, hence, the star s^* of $\mathrm{GU}(\Sigma)$ for t.

Phase 2. If $s^* \notin \Sigma$, the iteration outputs $\mathcal{G}_i = \mathcal{G}_{i-1}[T_{s^*} \ominus \mathrm{GU}(\Sigma)]$. Otherwise, we find the smallest (under $\prec$) child $s^\#$ of s^* in T that can $\mathcal{G}_{i-1}$-reach t. If $s^\#$ exists, the iteration outputs $\mathcal{G}_i = \mathcal{G}_{i-1}[T_{s^\#} \ominus \mathrm{GU}(\Sigma)]$; otherwise, the algorithm finishes with $t = s^*$.

By resorting to Lemmas 8 and 10 and adapting the cost analysis of our first POMS algorithm, we prove in Appendix I:

Lemma 12. *The above algorithm is correct and achieves the same guarantees as in Theorem 1.*

Two properties. Let us concentrate on an arbitrary iteration — say the i-th (with $i \ge 1$) — of the algorithm and fix its input $\mathcal{G}_{i-1}$. The iteration's execution is determined by $\mathcal{G}_{i-1}$ and the target t (which must be in $\mathcal{G}_{i-1}$ due to the target-containment property in Section 4). We define a function $\mathrm{OUT}(\mathcal{G}_{i-1}, t)$ to return

- an empty graph if the iteration finds t, or
- the DAG $\mathcal{G}_i$, otherwise.

Define further:

$$\mathrm{OUT}(\mathcal{G}_{i-1}) \quad = \quad \{\mathrm{OUT}(\mathcal{G}_{i-1}, v) \mid v \text{ in } \mathcal{G}_{i-1}\}.$$

Appendix J proves:

Lemma 13. *No two DAGs in $\mathrm{OUT}(\mathcal{G}_{i-1})$ share any common vertex.*

Let T be an HPDFS-tree of $\mathcal{G}_{i-1}$ and Σ be the λ-separator of T. Let us collect the children of all the nodes in Σ into a set C; we will call C the *children set of* Σ. We prove in Appendix K:

Lemma 14. $|C| \le |\Sigma| + |\mathrm{OUT}(\mathcal{G}_{i-1})|$.

5.2 An EM Structure

To find the target t, our EM structure deploys the algorithm $\mathcal{A}$ of Lemma 12 by setting its parameter k to the block size B. Towards this purpose, we precompute all the probes that $\mathcal{A}$ can possibly perform. For every probe, the structure stores the at most B vertices (requested by the probe) in $O(1)$ blocks. Thus, no matter which probe $\mathcal{A}$ needs to make, $\mathcal{A}$ can always load the corresponding vertices into memory with $O(1)$ I/Os.

The main challenge is to argue that the space complexity is $O(n/B)$. Towards that purpose, we will create the structure recursively and leverage Lemmas 13 and 14 to obtain a non-conventional recurrence on the space consumption which will solve to $O(n/B)$.

Fix an arbitrary i. As mentioned, the i-th iteration of $\mathcal{A}$ depends only on $\mathcal{G}_{i-1}$ and where t is in $\mathcal{G}_{i-1}$. Now, fix a $\mathcal{G}_{i-1}$ that can possibly occur. Given this $\mathcal{G}_{i-1}$, the i-th iteration can have $|\mathrm{OUT}(\mathcal{G}_{i-1})|$ different outputs, as discussed in Section 5.1. We will create a structure POMS$(\mathcal{G}_{i-1}, i)$ which gives $\mathcal{A}$ all the information needed to carry out the iteration on $\mathcal{G}_{i-1}$. Recall that, when unable to find the target t, the iteration outputs a DAG $\mathcal{G}_i \in \mathrm{OUT}(\mathcal{G}_{i-1})$ for the $(i+1)$-th iteration. This $\mathcal{G}_i$ will then be recursively handled by structure POMS$(\mathcal{G}_i, i+1)$. The entry point to the whole recursion is POMS$(\mathcal{G}, 1)$ = POMS$(\mathcal{G}_0, 1)$.

Structure POMS$(\mathcal{G}_{i-1}, i)$: when $\mathcal{G}_{i-1}$ is large. We now explain the details of POMS$(\mathcal{G}_{i-1}, i)$, assuming first that $\mathcal{G}_{i-1}$ has more than B vertices. Let T be an HPDFS-tree of $\mathcal{G}_{i-1}$ and Σ be the B-separator of T. In $O(1)$ blocks, we store all the vertices GU(Σ) and encode their ancestor-descendant relationships in T;[6] they will be referred to as the *grand-union blocks*. By reading these blocks into memory, $\mathcal{A}$ can execute Phase 1 to decide, for each $u \in \mathrm{GU}(\Sigma)$, whether u can $\mathcal{G}_{i-1}$-reach t. After that, $\mathcal{A}$ must have obtained the star s^* of GU(Σ) for t.

Fix an arbitrary node $u \in \mathrm{GU}(\Sigma) \setminus \Sigma$. If $s^* = u$, Phase 2 of the iteration generates $\mathcal{G}_i = \mathcal{G}_{i-1}[T_{s^*} \ominus \mathrm{GU}(\Sigma)]$ to be processed by iteration $i+1$. We build POMS$(\mathcal{G}_i, i+1)$ recursively and store a pointer (i.e., a disk address) to POMS$(\mathcal{G}_i, i+1)$ at u inside the grand-union blocks of POMS$(\mathcal{G}_{i-1}, i)$.

Now, fix an arbitrary node $u \in \Sigma$. If $s^* = u$, Phase 2 needs to identify the smallest child $s^\#$ of s^* able to $\mathcal{G}_{i-1}$-reach t or declare the absence of $s^\#$. For this purpose, we store the children of u in ascending order of $\prec$ in consecutive blocks — call them the *children blocks* — which $\mathcal{A}$ reads until either having found $s^\#$ or having exhausted all the children of s^*. If $s^\#$ is not found, the algorithm terminates with $t = u$. Otherwise, it generates $\mathcal{G}_i = \mathcal{G}_{i-1}[T_{s^\#} \ominus \mathrm{GU}(\Sigma)]$. We build POMS$(\mathcal{G}_i, i+1)$ recursively and store a pointer to POMS$(\mathcal{G}_i, i+1)$ at $s^\#$ inside the children blocks. This completes the description of POMS$(\mathcal{G}_{i-1}, i)$.

Structure POMS$(\mathcal{G}_{i-1}, i)$: when $\mathcal{G}_{i-1}$ is small. If $\mathcal{G}_{i-1}$ has less than B vertices, $\mathcal{A}$ finishes with a single probe. The situation is slightly more complex in EM because we do not have access to the edges in $\mathcal{G}_{i-1}$. Fortunately, we can eliminate the barrier by resorting to an HPDFS-tree T of $\mathcal{G}_{i-1}$. Note that T has at most B nodes and, therefore, fits in $O(1)$ blocks. To find t, we read those blocks into memory, acquire their $\mathcal{G}_{i-1}$-reachability to t from oracle, and then identify the star s^* of the set of vertices in T for t. The no-cross-reachability property of Lemma 5 ensures $s^* = t$.

I/O cost. The algorithm performs $O(1)$ I/Os for every probe issued by the POMS algorithm of Lemma 12. It follows immediately that the overall I/O cost is $O(\log_B n + (d/B)\log_{1+d} n)$.

Space. We make sure that all blocks, except possibly one, are full. This can be achieved by first generating the sequence of words needed to represent the structure, then chopping the sequence into blocks of size B, and finally making one more pass over the sequence to fix the pointers. Hence, it suffices to analyze how many words are used by our structure.

[6] It is well-known that the ancestor-descendant relationships of x nodes in a tree can be encoded in $O(x)$ words.

Let function $f(n)$ be the number of words necessary (in the worst case) when $\mathcal{G}$ has n vertices. Trivially, $f(n) = O(n)$ when $n \le B$. Next, we discuss the scenario $n > B$.

Let us focus on the structure POMS$(\mathcal{G}, 1)$, i.e., the entry structure of the whole recursion. The number of words in the grand-union blocks is $O(|\mathrm{GU}(\Sigma)|) = O(|\Sigma|)$ (Lemma 3). Let C be the children set (Section 5.1) of the B-separator Σ used in POMS$(\mathcal{G}, 1)$. The children blocks of all the nodes in Σ use $O(|C|)$ words in total.

We still need to account for the space of the recursive structure on every possible $\mathcal{G}_1 \in \mathrm{OUT}(\mathcal{G}, 1)$. Denoting by $|\mathcal{G}_1|$ the number of vertices in $\mathcal{G}_1$, we have:

$$\begin{aligned} f(n) &= O(1 + |\Sigma| + |C|) + \sum_{\mathcal{G}_1 \in \mathrm{OUT}(\mathcal{G},1)} f(|\mathcal{G}_1|) \\ &= O(1 + |\Sigma| + |\mathrm{OUT}(\mathcal{G},1)|) + \sum_{\mathcal{G}_1 \in \mathrm{OUT}(\mathcal{G},1)} f(|\mathcal{G}_1|) \quad (3) \end{aligned}$$

where the last equality applied Lemma 14. The recurrence is constrained by $|\Sigma| + \sum_{\mathcal{G}_1 \in \mathrm{OUT}(\mathcal{G},1)} |\mathcal{G}_1| \le n$ because (i) no vertex of Σ can appear in any $\mathcal{G}_1 \in \mathrm{OUT}(\mathcal{G}, 1)$, and (ii) no two DAGs in OUT$(\mathcal{G}, 1)$ share any common vertices (Lemma 13). Appendix L shows that the recurrence (3) gives $f(n) = O(n)$.

This concludes the proof for the first bullet of Theorem 4.

6 LOWER BOUNDS

A lower bound on instance-oriented POMS. Consider $\mathcal{G}$ as a tree with n vertices and maximum out-degree d. We will show that

$$maxcost^{\mathrm{I}}_k(\mathcal{A}, \mathcal{G}) = \Omega(\log_{1+k} n + d/k) \quad (4)$$

hold for any POMS algorithm $\mathcal{A}$, where $maxcost^{\mathrm{I}}_k(\mathcal{A}, \mathcal{G})$ is defined in Section 1.3.

Consider a probe with a set Q of $k \ge 1$ vertices $q_1, q_2, ..., q_k$. The oracle returns an *outcome sequence* $a_1, a_2, ..., a_k$, where $a_i = 1$ if q_i can $\mathcal{G}$-reach the target t or 0 otherwise. With Q fixed, the outcome sequence solely depends on t. In Appendix M, we prove:

LEMMA 15. *When $\mathcal{G}$ is a tree, there are at most $k+1$ distinct output sequences for a specific Q, as t ranges over all the vertices in $\mathcal{G}$.*

We now prove $maxcost^{\mathrm{I}}_k(\mathcal{A}, \mathcal{G}) = \Omega(\log_{1+k} n)$ with an information theoretic argument. By Lemma 15, each outcome sequence can be encoded in $O(\log(k+1))$ bits. At least $\log_2 n$ bits are needed to encode the n possible targets t. Thus, $\Omega(\frac{\log n}{\log(1+k)})$ probes are needed for at least one t.

To establish (4), it remains to show that $maxcost^{\mathrm{I}}_k(\mathcal{A}, \mathcal{G}) = \Omega(1 + d/k)$. We achieve the purpose with an alternative argument. Let u be an arbitrary node in $\mathcal{G}$ with d child nodes $v_1, v_2, ..., v_d$. Define $S = \{v_1, ..., v_d\}$. When asked if a node $q \in \mathcal{G}$ can reach t, the oracle acts in the following manner until $|S| = 1$: (i) if $q \in S$, return "no" and then remove q from S; (ii) if $q \notin S$ and q can reach u, return "yes"; (iii) otherwise, return no. When $|S|$ drops to 1, the oracle finalizes t to the only node left in S. It is now clear that $\mathcal{A}$ must set q to at least $d-1$ distinct nodes throughout the execution, which necessitates at least $\lceil (d-1)/k \rceil$ probes.

Proof of the second bullet in Theorem 1. It has been proved in [38] that $minmaxcost_k(\mathscr{T}(n, d))$, defined in (1), is $\Omega((d/k)\log_{1+d} n)$. Next, we show that it is also $\Omega(\log_{1+k} n)$, which will give

$$minmaxcost_k(\mathscr{T}(n, d)) = \Omega(\log_{1+k} n + (d/k)\log_{1+d} n) \quad (5)$$

and will complete the proof of Theorem 1.

Let $\mathcal{A}$ be an algorithm achieving the minimum in the right hand side of (1), namely, $maxcost_k^C(\mathcal{A}, \mathscr{T}(n,d)) = minmaxcost_k(\mathscr{T}(n,d))$. Take an arbitrary $\mathcal{G} \in \mathscr{T}(n,d)$. Clearly, $maxcost_k^C(\mathcal{A}, \mathscr{T}(n,d)) \geq maxcost_k^I(\mathcal{A}, \mathcal{G})$. We have proved earlier that $maxcost_k^I(\mathcal{A}, \mathcal{G})$ must be $\Omega(\log_{1+k} n)$.

Proof of Corollary 2. Immediate from (2) and (5).

Proof of Corollary 3. Our algorithm $\mathcal{A}$ in Theorem 1 ensures $maxcost_k^I(\mathcal{A}, \mathcal{G}) = O(\log_{1+k} n + \frac{d}{k} \log_{1+d} n)$, which is greater than the right hand side of (4) by a factor of $O(\frac{\log n}{\log(1+k)+\log\log n})$. To see why, note that the factor is always bounded by $O(\log_{1+d} n)$, which is $O(\frac{\log n}{\log(1+k)+\log\log n})$ if $d \geq \frac{k \log_2 n}{\log_2(1+k)}$. If $d \leq \frac{k \log_2 n}{\log_2(1+k)}$, the factor is $O(\frac{(d/k)\log_{1+d} n}{\log_{1+k} n}) = O(\frac{d}{k} \frac{\log(1+k)}{\log(1+d)})$, which is $O(\frac{\log n}{\log(1+k)+\log\log n})$.

Proof of the second bullet in Theorem 4. Fix a value of n and consider any DAG $\mathcal{G}$ with n vertices. We claim that, in general, given a structure $\mathcal{I}$ that ensures finding the target t in $\mathcal{G}$ using $F(B)$ I/Os, we can obtain a POMS algorithm $\mathcal{A}$ that finds t with at most $F(k)$ probes. It will then follows from the second bullet of Theorem 1 that $F(B) = \Omega(\log_B n + (d/B)\log_{1+d} n)$, which will complete the proof of Theorem 4.

We design $\mathcal{A}$ as follows. First, $\mathcal{A}$ builds a structure $\mathcal{I}$ on $\mathcal{G}$ by setting $B = k$. Then, $\mathcal{A}$ interacts with the oracle by emulating the algorithm of $\mathcal{I}$. Specifically, whenever $\mathcal{I}$ performs an I/O to read a set S of at most B vertices, $\mathcal{A}$ probes the oracle about the $\mathcal{G}$-reachability (to t) of every vertex in S. This way, $\mathcal{A}$ acquires as much information as $\mathcal{I}$ and, thus, will terminate after $F(k)$ probes (a.k.a. I/Os).

ACKNOWLEDGEMENTS

The research of Shangqi Lu and Yufei Tao was partially supported by GRF projects 142034/21 and 142078/20 from HKRGC. The research of Wim Martens was partially supported by grants 369116833 and 431183758 of the Deutsche Forschungsgemeinschaft (DFG).

REFERENCES

[1] Micah Adler and Brent Heeringa. Approximating optimal binary decision trees. *Algorithmica*, 62(3-4):1112–1121, 2012.

[2] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM (CACM)*, 31(9):1116–1127, 1988.

[3] Lars Arge, Andrew Danner, and Sha-Mayn Teh. I/O-efficient point location using persistent B-trees. *ACM Journal of Experimental Algorithmics*, 8, 2003.

[4] Esther M. Arkin, Henk Meijer, Joseph S. B. Mitchell, David Rappaport, and Steven Skiena. Decision trees for geometric models. *International Journal of Computational Geometry and Applications*, 8(3):343–364, 1998.

[5] Yosi Ben-Asher and Eitan Farchi. The cost of searching in general trees versus complete binary trees. Technical report, 1997.

[6] Yosi Ben-Asher, Eitan Farchi, and Ilan Newman. Optimal search in trees. *SIAM Journal of Computing*, 28(6):2090–2102, 1999.

[7] Elisa Bertino, Barbara Catania, and Boris Shidlovsky. Towards optimal indexing for segment databases. In *Proceedings of Extending Database Technology (EDBT)*, pages 39–53, 1998.

[8] Paul B. Callahan, Michael T. Goodrich, and Kumar Ramaiyer. Topology B-trees and their applications. In *Algorithms and Data Structures Workshop (WADS)*, pages 381–392, 1995.

[9] Renato Carmo, Jair Donadelli, Yoshiharu Kohayakawa, and Eduardo Sany Laber. Searching in random partially ordered sets. *Theoretical Computer Science*, 321(1):41–57, 2004.

[10] Venkatesan T. Chakaravarthy, Vinayaka Pandit, Sambuddha Roy, Pranjal Awasthi, and Mukesh K. Mohania. Decision trees for entity identification: Approximation algorithms and hardness results. *ACM Transactions on Algorithms*, 7(2):15:1–15:22, 2011.

[11] Venkatesan T. Chakaravarthy, Vinayaka Pandit, Sambuddha Roy, and Yogish Sabharwal. Approximating decision trees with multiway branches. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, pages 210–221, 2009.

[12] Ferdinando Cicalese, Tobias Jacobs, Eduardo Sany Laber, and Marco Molinaro. On greedy algorithms for decision trees. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 206–217, 2010.

[13] Ferdinando Cicalese, Tobias Jacobs, Eduardo Sany Laber, and Marco Molinaro. On the complexity of searching in trees and partially ordered structures. *Theoretical Computer Science*, 412(50):6879–6896, 2011.

[14] Ferdinando Cicalese, Tobias Jacobs, Eduardo Sany Laber, and Marco Molinaro. Improved approximation algorithms for the average-case tree searching problem. *Algorithmica*, 68(4):1045–1074, 2014.

[15] Ferdinando Cicalese, Tobias Jacobs, Eduardo Sany Laber, and Caio Dias Valentim. The binary identification problem for weighted trees. *Theoretical Computer Science*, 459:100–112, 2012.

[16] Ferdinando Cicalese, Balázs Keszegh, Bernard Lidický, Dömötör Pálvölgyi, and Tomás Valla. On the tree search problem with non-uniform costs. *Theoretical Computer Science*, 647:22–32, 2016.

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[18] Pilar de la Torre, Raymond Greenlaw, and Alejandro A. Schäffer. Optimal edge ranking of trees in polynomial time. *Algorithmica*, 13(6):592–618, 1995.

[19] Dariusz Dereniowski. Edge ranking of weighted trees. *Discrete Applied Mathematics*, 154(8):1198–1209, 2006.

[20] Dariusz Dereniowski. Edge ranking and searching in partial orders. *Discrete Applied Mathematics*, 156(13):2493–2500, 2008.

[21] Dariusz Dereniowski and Marek Kubale. Efficient parallel query processing by graph ranking. *Fundamenta Informaticae*, 69(3):273–285, 2006.

[22] Dariusz Dereniowski, Stefan Tiegel, Przemyslaw Uznanski, and Daniel Wolleb-Graf. A framework for searching in graphs in the presence of errors. In *Proceedings of Symposium on Simplicity in Algorithms (SOSA)*, pages 4:1–4:17.

[23] Ehsan Emamjomeh-Zadeh, David Kempe, and Vikrant Singhal. Deterministic and probabilistic binary search in graphs. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 519–532, 2016.

[24] Greg N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM Journal of Computing*, 26(2):484–538, 1997.

[25] Xiaocheng Hu, Cheng Sheng, and Yufei Tao. Building an optimal point-location structure in O(sort(n)) I/Os. *Algorithmica*, 81(5):1921–1937, 2019.

[26] Ananth V. Iyer, H. Donald Ratliff, and Gopalakrishnan Vijayan. On an edge ranking problem of trees and graphs. *Discrete Applied Mathematics*, 30(1):43–52, 1991.

[27] Tobias Jacobs, Ferdinando Cicalese, Eduardo Sany Laber, and Marco Molinaro. On the complexity of searching in trees: Average-case minimization. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, pages 527–539, 2010.

[28] Camille Jordan. Sur les assemblages de lignes. *Journal für die reine und angewandte Mathematik*, 70:185–190, 1869.

[29] S. Rao Kosaraju, Teresa M. Przytycka, and Ryan S. Borgstrom. On an optimal split tree problem. In *Algorithms and Data Structures Workshop (WADS)*, pages 157–168, 1999.

[30] Eduardo Sany Laber and Marco Molinaro. An approximation algorithm for binary searching in trees. *Algorithmica*, 59(4):601–620, 2011.

[31] Eduardo Sany Laber and Loana Tito Nogueira. Fast searching in trees. *Electronic Notes in Discrete Mathematics*, 7:90–93, 2001.

[32] Tak Wah Lam and Fung Ling Yue. Optimal edge ranking of trees in linear time. *Algorithmica*, 30(1):12–33, 2001.

[33] Shay Mozes, Krzysztof Onak, and Oren Weimann. Finding an optimal tree searching strategy in linear time. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1096–1105, 2008.

[34] Ketan Mulmuley. A fast planar partition algorithm, I. *J. Symb. Comput.*, 10(3/4):253–280, 1990.

[35] J. Ian Munro and Yakov Nekrich. Dynamic planar point location in external memory. In *Proceedings of Symposium on Computational Geometry (SoCG)*, volume 129, pages 52:1–52:15.

[36] Krzysztof Onak and Pawel Parys. Generalization of binary search: Searching in trees and forest-like partial orders. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 379–388, 2006.

[37] Raimund Seidel. Reprint of: A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom.*, 43(6-7):556–564, 2010.

[38] Yufei Tao, Yuanbing Li, and Guoliang Li. Interactive graph search. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 1393–1410, 2019.

APPENDIX

A PROOF OF LEMMA 5

The following is known as the *white path theorem* of DFS:

Theorem 16 ([17]). *For any nodes u and v, $v \in T_u$ if and only if G has a white path from u to v right before u enters the stack.*

The theorem implies:

Corollary 17. *Consider the moment when u is about to enter the stack; if (i) v is white and (ii) G has no white path from u to v, then v enters the stack after u is popped. If in addition G has a white path from u to u' at that moment, then $u' \prec v$.*

(Order property) As $v \notin T_u$, by Theorem 16, no white path exists from u to v when u enters the stack; thus, v enters the stack after u is popped (Corollary 17, applying the fact that $u \prec v$) . $u' \in T_u$ indicates that u is in the stack when u' enters the stack. The two facts together indicate $u' \prec v$. On the other hand, $v' \in T_v$ means that $v \prec v'$, which leads to $u \prec v'$.

(No-cross-reachability property) Take an arbitrary $v' \in T_v$; it must be white when u enters the stack (order property). Assume that G has a path π from u to v'. Let v'' be the last non-white node on π at the moment when u enters the stack (v'' must exist because otherwise $v' \in T_u$ by Theorem 16, contradicting $v' \in T_v$). Hence, when v'' enters the stack, a white path exists from v'' to v' but not from v'' to u (otherwise, there would be a cycle). By Corollary 17, $v' \prec u$, which contradicts the order property.

(Path-descendants property) Assume that π is a u-to-w path containing at least one node outside T_u. When u enters the stack, some nodes on π must be non-white (Theorem 16); let v be such a node on π closest to w. When v enters the stack, there must be a white path from v to w; by Theorem 16, $w \in T_v$. Because (i) $w \in T_u$ and (ii) $v \notin T_u$, v must be a proper ancestor of u in T. But this means that G has a path from v to u, thereby creating a cycle.

(Subtree-size property) This property immediately follows from the definition of HPDFS and Theorem 16.

B PROOF OF LEMMA 6

It is obvious from the algorithm that the removal of Σ disconnects T into trees of at most n/λ nodes. It remains to show $|\Sigma| \le \lambda - 1$. Suppose that $|\Sigma| \ge \lambda$. Every time when we add a node into Σ at Line 4, $\lfloor \tau \rfloor + 1 > \tau$ nodes are removed from T'. The total number of nodes removed is *strictly* larger $|\Sigma|\tau \ge \lambda\tau = n$, giving a contradiction.

C PROOF OF LEMMA 7

We will start with a property of left flanks:

Lemma 18. *For any $u \in \Sigma$ and $v \in$ LF(u), $\Sigma \setminus \{u\}$ has at least one node in T_v.*

Proof. Node v must have a right sibling v' on the path from the root to u; note that v' is an ancestor of u. By the way Σ is produced from **construct-separator** (Section 2), we must have $|T_{v'}| \ge \lfloor n/\lambda \rfloor + 1$, which leads to $|T_v| \ge \lfloor n/\lambda \rfloor + 1$ (subtree-size property of Lemma 5). The design of **construct-separator** guarantees that Σ contains at least one node in T_v, which obviously cannot be u. □

Define P as the set of edges e in T such that e is on the root-to-u path for at least one $u \in \Sigma$. Denote by T^P the subgraph of T induced by P; clearly, T^P is a tree. Consider any node $v \in$ LFU(Σ). By definition of LFU(Σ), $v \in$ LF(u) for some $u \in \Sigma$, because of which T_v contains at least one node in Σ (Lemma 18). Hence, $v \in T^P$. In other words, all the nodes in LFU(Σ) are in T^P.

Let I be the set of internal nodes in T^P that have two or more child nodes (in T^P). For each $u \in I$, denote by c_u the number of its child nodes in T^P. Every node $v \in$ LFU(Σ) satisfies: (i) *parent*(v) $\in I$; and (ii) v has a right sibling in T^P. This implies that each $u \in I$ has at most $c_u - 1$ child nodes in LFU(Σ), leading to $|\text{LFU}(\Sigma)| \le \sum_{u \in I} (c_u - 1)$.

It remains to prove $\sum_{u \in I} (c_u - 1) \le |\Sigma| - 1$. Denote by x the number of leaves in T^P and by y the number of internal nodes in T^P with only one child (in T^P). By how T^P is built, every leaf node of T^P must belong to Σ; hence, $x \le |\Sigma|$. Since the degree sum of the vertices in T^P must be equivalent to twice the number of edges in T^P, we have

$$\begin{aligned} x + 2y + \left(\sum_{u \in I} (c_u + 1) \right) - 1 &= 2(|I| + x + y - 1) \\ \Rightarrow \sum_{u \in I} (c_u - 1) &= x - 1 \le |\Sigma| - 1. \end{aligned}$$

D PROOF OF LEMMA 8

Proof of the first bullet. Clearly, $t \notin T_v$ for any left sibling v of $s^\#$ (recall that $s^\#$ is the *smallest* child of s^* that can G-reach t). If $t \in T_v$ for some right sibling v of $s^\#$, we have a violation of the no-cross-reachability property.

Proof of the second bullet. Consider an arbitrary $s^\#$-to-t path π in G. We argue that *every* node u on π is in $T_{s^\#} \ominus S$. The claim in the second bullet will then follow because $G[T_{s^\#} \ominus S]$ is a vertex-induced subgraph of G.

To this end, assume that u is a node on π. The path-descendants property of Lemma 5 indicates $u \in T_{s^\#}$ due to the existence of π. Therefore, if $u \notin T_{s^\#} \ominus S$, then u must have been "shielded" by S, namely, there is some $s \in S$ with $s \ne s^\#$ such that $s \in T_{s^\#}$ and $u \in T_s$. Given that u can G-reach t, s must also be able to G-reach t. However, $s \in T_{s^\#}$ tells us that $s \in T_{s^*}$, which means that s violates Condition C2 in the definition of s^*, giving a contradiction.

E PROOF OF LEMMA 9

We will establish a useful fact:

Lemma 19. *If u can G-reach w, then $w \in T_{u^*}$ where u^* is the smallest node in* LF(u) $\cup \{u\}$ *able to G-reach w.*

Proof. We will show first that $w \in T_{u^*}$ for some node $u^* \in$ LF(u) $\cup \{u\}$. Let π be the path in T from the root to u, and p be the lowest ancestor of u such that $w \in T_p$. If $p = u$, $w \in T_{u^*}$ holds by setting $u^* = u$. Otherwise, let v be the child of p on π; we must have $w \notin T_v$. Since $p \ne w$ (otherwise G has a cycle), p must have a child u^* such that $w \in T_{u^*}$. We now argue that u^* is a *left* sibling of v and, hence, belongs to LF(u). Indeed, if u^* is a right sibling, then nodes v and u^* cause a violation of the no-cross-reachability property of Lemma 5: v (which can G-reach w via u) is able to G-reach a node (i.e., w) in T_{u^*}.

As none of the nodes in $\mathrm{LF}(u) \cup \{u\}$ are ancestors of each other, there is a unique node $u^* \in \mathrm{LF}(u) \cup \{u\}$ satisfying $w \in T_{u^*}$. By the no-cross-reachability property, no $v' \in \mathrm{LF}(u) \cup \{u\}$ with $v' \prec u^*$ can G-reach w: if such a v' exists, the property is violated for the nodes v' and w. This completes the proof. □

Proof of the first bullet of Lemma 9. Immediate from Lemma 19: since s^* can G-reach t, we know $t \in T_{s^{**}}$.

Proof of the second bullet. We will argue that every s^{**}-to-t path π in G must have all its nodes in $T_{s^{**}} \ominus \Sigma$. This will establish Lemma 9 by the definition of $G[T_{s^{**}} \ominus \Sigma]$. Let u be an arbitrary node on π. By the path-descendants property of Lemma 5, we must have $u \in T_{s^{**}}$. If $u \notin T_{s^{**}} \ominus \Sigma$, then u must have been "shielded" by Σ, namely, some node $s \in \Sigma$ satisfies $s \neq s^{**}$, $s \in T_{s^{**}}$, and $u \in T_s$ (which indicates that s can G-reach u). The existence of s rules out the possibility that $s^{**} = s^*$; otherwise, s violates the Condition C2 in the definition of s^* (Section 3). Thus, $s^{**} \in \mathrm{LF}(s^*)$ and therefore $s^{**} \prec s^*$. But in this case we must have $s \prec s^*$ (order property of Lemma 5), which means that $s \in \Sigma$ is a smaller node than s^* satisfying Condition C1. Furthermore, since every node in T_s is smaller than s^* (order property), T_s must contain a smaller node than s^* that is in Σ and satisfies both C1 and C2. This contradicts that s^* is the smallest such node.

F PROOF OF LEMMA 10

We will first show $\mathrm{LF}(s^*) \subseteq \mathrm{LFU}(\Sigma)$. This holds directly by definition if $s^* \in \Sigma$. Consider now $s^* \in \mathrm{GU}(\Sigma) \setminus \Sigma$, which means $s^* \in \mathrm{LFU}(\Sigma)$. By Lemma 18, at least one node $u \in \Sigma$ appears in T_{s^*}; thus, $\mathrm{LF}(s^*) \subseteq \mathrm{LF}(u) \subseteq \mathrm{LFU}(\Sigma)$.

We now prove the first bullet of Lemma 10. Every node in $\mathrm{LF}(s^*)$ is smaller than s^* (under $\prec$). Applying $\mathrm{LF}(s^*) \subseteq \mathrm{LFU}(\Sigma)$ and the definition of s^*, we know that $\mathrm{LF}(s^*)$ has no nodes able to G-reach t. Indeed, if some node in $\mathrm{LF}(s^*)$ is able to G-reach t, then s^* is not the smallest node in $\mathrm{GU}(\Sigma)$ satisfying Conditions C1 and C2 (Section 3). Then, $t \in T_{s^*}$ by Lemma 19.

To prove the second bullet, consider an arbitrary s^*-to-t path π in G. It suffices to show $u \in T_{s^*} \ominus \mathrm{GU}(\Sigma)$ for every node u on π. By the path-descendants property of Lemma 5, we have that $u \in T_{s^*}$. If $u \notin T_{s^*} \ominus \mathrm{GU}(\Sigma)$, there must be some $s \in \mathrm{GU}(\Sigma)$ with $s \neq s^*$ such that $s \in T_{s^*}$ and $u \in T_s$. But this means that s can G-reach t (by way of u), contradicting the Condition C2 (Section 3) in the definition of s^*.

G PROOF OF LEMMA 11

$\mathcal{G}_i$ includes no vertices from Σ and, thus, can have at most n_{i-1}/k nodes (Lemma 6). Next, we prove the claim that $\mathcal{G}_i$ contains t and is path preserving. If $s^{**} \notin \Sigma$, the claim follows immediately from Lemma 9. In the case where $s^{**} = s^*$, the first bullet of Lemma 9 assures us that t must appear in T_{s^*}. If $s^\#$ does not exist, s^* must be t and the algorithm finishes correctly. Otherwise, the claim follows from Lemma 8.

H COST ANALYSIS FOR THE ALGORITHM IN SECTION 4

Given $\mathcal{G}_{i-1}$, the i-th ($i \geq 1$) iteration of our algorithm either finds t or outputs $\mathcal{G}_i$. Suppose that the algorithm finds t at iteration h for some $h \geq 1$. Since our goal is to bound the worst-case cost, it suffices to discuss only the case where $\mathcal{G}_{h-1}$ has no more than k vertices. We will focus on $k \geq 2$ (for $k = 1$, manually increase it to 2).

Analysis of one iteration. Consider the i-th iteration of any $i \in [1, h-1]$. Let T, $\prec$, s^*, s^{**} and $s^\#$ be defined in the same way as in Section 4. Set n_{i-1} (resp., n_i) to the number of vertices in $\mathcal{G}_{i-1}$ (resp., $\mathcal{G}_i$). Define an integer x_i as follows:

- if $s^{**} \neq s^*$, $x_i = 0$;
- otherwise, x_i equals how many child nodes of s^* (in T) are smaller than $s^\#$.

The i-th iteration issues at most

$$2 + \left\lceil \frac{x_i + 1}{k} \right\rceil \tag{6}$$

queries (two queries in Phase 1 and the rest in Phase 2).

Lemma 20. *For every $i \in [1, h-1]$:*

$$n_i \leq \frac{n_{i-1}}{\max\{k, x_i + 1\}}. \tag{7}$$

Proof. The fact $n_i \leq n_{i-1}/k$ has been proved in Appendix G. Next, we will prove $n_i \leq n_{i-1}/(x_i + 1)$. This is obviously true if $x_i = 0$. The rest of the proof assumes $x_i > 0$. In this case, $s^\#$ has x_i left siblings v satisfying $|T_v| \geq |T_{s^\#}|$ (subtree-size property of Lemma 5). Hence, $|T_{s^\#}| = (x_i + 1)|T_{s^\#}| \,/\, (x_i + 1) \leq n_{i-1}/(x_i + 1)$. The lemma then follows from $n_i \leq |T_{s^\#}|$. □

Total cost. Applying (7) for each $i \in [1, h-1]$ yields

$$\frac{n}{\prod_{i=1}^{h-1} \max\{k, x_i + 1\}} \geq n_{h-1} \geq 1. \tag{8}$$

Therefore, $h = O(\log_k n)$. By (6), the total cost of the algorithm is at most

$$1 + \sum_{i=1}^{h-1} \left(2 + \left\lceil \frac{x_i + 1}{k} \right\rceil\right) \leq 1 + \sum_{i=1}^{h-1} \left(3 + \frac{1}{k} + \frac{x_i}{k}\right) = O(\log_k n) + \frac{1}{k} \sum_{i=1}^{h-1} x_i. \tag{9}$$

If $d \leq k$, then $x_i \leq d \leq k$; hence, (9) is bounded by $O(\log_k n)$. Assuming $d \geq k + 1$, the rest of the proof will show

$$\sum_{i=1}^{h-1} x_i = O(d \log_d n + k \log_k n) \tag{10}$$

which will yield the conclusion that our algorithm performs $O(\log_k n + \frac{d}{k} \log_d n)$ probes in total.

Proof of (10)**.** The integers $x_1, ..., x_{h-1}$ satisfy $0 \leq x_i \leq d - 1$ and

$$\prod_{i=1}^{h-1} \max\{k, x_i + 1\} \leq n \tag{11}$$

because of (8). We will prove (10) under the relaxation that $x_1, ..., x_h$ are real values (instead of integers) in $[0, d-1]$. In such a case, the constraint (11) can be replaced by

$$\prod_{i=1}^{h-1} (x_i + 1) \leq n \tag{12}$$

by requiring $x_i \geq k-1$, noticing that if $x_i < k-1$, raising it to $k-1$ always increases the left hand side of (10). Thus, the goal now is to maximize $\sum_{i=1}^{h-1} x_i$ subject to (12) and $x_i \in [k-1, d-1]$.

Lemma 21. *When $\sum_{i=1}^{h-1} x_i$ is maximized, at most one of $x_1, ..., x_{h-1}$ can be strictly larger than $k-1$ but strictly smaller than $d-1$.*

Proof. Suppose that there are distinct $i_1, i_2 \in [1, h-1]$ such that x_{i_1} and x_{i_2} are both strictly larger than $k-1$ but strictly smaller than $d-1$. Without loss of generality, assume $x_{i_1} \geq x_{i_2}$. Set $c = (x_{i_1}+1)(x_{i_2}+1)$. Clearly, $k^2 < c < d^2$. We can increase $x_{i_1} + x_{i_2}$ as follows:

- if $c > dk$, modify x_{i_1} to $d-1$ and x_{i_2} to $c/d-1$;
- otherwise, modify x_{i_1} to $c/k-1$ and x_{i_2} to $k-1$.

After the modification, $x_{i_1} = d-1$ or $x_{i_2} = k-1$; and no constraints are violated because $k-1 \leq x_{i_2} \leq x_{i_1} \leq d-1$ and $(x_{i_1}+1)(x_{i_2}+1) = c$. This contradicts the claim that the original $x_1, ..., x_{h-1}$ maximize $\sum_{i=1}^{h-1} x_i$. □

Consider a set of $x_1, ..., x_{h-1}$ that maximizes $\sum_{i=1}^{h-1} x_i$. Let y_1 (or y_2) be the number of variables among $x_1, ..., x_{h-1}$ that are set to $k-1$ (or $d-1$, resp.). Because of (12), $y_2 = O(\log_d n)$; on the other hand, trivially $y_1 \leq h-1$. Hence:

$$\sum_{i=1}^{h-1} x_i \leq y_1(k-1) + (1+y_2)(d-1)$$
$$= O(hk + d\log_d n) = O(k \log_k n + d \log_d n).$$

I PROOF OF LEMMA 12

To prove the algorithm's correctness, it suffices to show that if the algorithm does not finish at iteration i, $\mathcal{G}_i$ has the size-reduction, target-containment, and path-preserving properties described in Section 4.

- **(size-reduction)** As $\mathcal{G}_i$ includes no vertices from Σ, $\mathcal{G}_i$ can have at most n_{i-1}/k nodes (Lemma 6).
- **(target-containment and path-preserving)** If $s^* \notin \Sigma$, the two properties follow directly from Lemma 10. Now, consider $s^* \in \Sigma$. By Lemma 10, $t \in T_{s^*}$. Hence, if $s^\#$ does not exist, $s^* = t$ and the algorithm finishes. Otherwise, by Lemma 8, $\mathcal{G}_i = \mathcal{G}_{i-1}[T_{s^\#} \ominus \mathrm{GU}(\Sigma)]$ contains t and is path preserving.

We can prove that the algorithm performs $O(\log_{1+k} n + \frac{d}{k} \log_{1+d} n)$ probes by adapting the argument of Appendix H in a straightforward manner.

J PROOF OF LEMMA 13

Consider any two different and non-empty graphs G_1 and G_2 in $\mathrm{OUT}(\mathcal{G}_{i-1})$. Denote by r_1 (resp., r_2) the root of G_1 (resp., G_2). In other words, $G_1 = \mathcal{G}_{i-1}[T_{r_1} \ominus \mathrm{GU}(\Sigma)]$ and $G_2 = \mathcal{G}_{i-1}[T_{r_2} \ominus \mathrm{GU}(\Sigma)]$, which implies $r_1 \neq r_2$. Next, we show that $T_{r_1} \ominus \mathrm{GU}(\Sigma)$ and $T_{r_2} \ominus \mathrm{GU}(\Sigma)$ do not share any common vertex. Since these graphs are trees, this is obvious if r_1 and r_2 have no ancestor-descendant relationship in T.

Assume, without loss of generality, that r_1 is a proper ancestor of r_2 in T. By the way our algorithm runs, we must have either

- **(Case 1)** $r_2 \in \mathrm{GU}(\Sigma) \setminus \Sigma$ or
- **(Case 2)** $parent(r_2) \in \Sigma$.

Specifically, Case 1 can happen only if $s^* = r_2$, while Case 2 can happen only if $s^\# = r_2$.

In Case 1, $T_{r_1} \ominus \mathrm{GU}(\Sigma)$ is contained in $T_{r_1} \ominus \{r_2\}$ which shares no vertices with T_{r_2}. It thus follows that $T_{r_1} \ominus \mathrm{GU}(\Sigma)$ shares no vertices with $T_{r_2} \ominus \mathrm{GU}(\Sigma)$.

Consider now Case 2. In general, the $\mathcal{G}_i$ produced by iteration i is never rooted at a vertex in Σ.[7] Hence, $r_1 \notin \Sigma$, which means that $parent(r_2)$ is a proper descendant of r_1 in T. Because $parent(r_2) \in \Sigma \in \mathrm{GU}(\Sigma)$, $T_{r_1} \ominus \mathrm{GU}(\Sigma)$ is contained in $T_{r_1} \ominus \{parent(r_2)\}$. $T_{r_1} \ominus \{parent(r_2)\}$ shares no vertices with $T_{parent(r_2)}$, whereas $T_{parent(r_2)}$ contains $T_{r_2} \ominus \mathrm{GU}(\Sigma)$. Therefore, we can conclude that no common vertex can exist in $T_{r_1} \ominus \mathrm{GU}(\Sigma)$ and $T_{r_2} \ominus \mathrm{GU}(\Sigma)$.

K PROOF OF LEMMA 14

We will prove that for each node $u \in C \setminus \Sigma$, there exists a graph in $\mathrm{OUT}(\mathcal{G}_{i-1})$ rooted at u. Because each graph in $\mathrm{OUT}(\mathcal{G}_{i-1})$ is single-rooted, we have $|C \setminus \Sigma| \leq |\mathrm{OUT}(\mathcal{G}_{i-1})|$, leading to $|C| \leq |\Sigma| + |\mathrm{OUT}(\mathcal{G}_{i-1})|$.

Fix any node $u \in C \setminus \Sigma$. Recall that $\mathrm{OUT}(\mathcal{G}_{i-1}, t)$ is (i) empty if iteration i finds the target t or (ii) a single-rooted $\mathcal{G}_i$ otherwise. We will focus on $t = u$ and prove that in this case $\mathrm{OUT}(\mathcal{G}_{i-1}, t)$ is a DAG rooted at u. As $\mathrm{OUT}(\mathcal{G}_{i-1}, t) \in \mathrm{OUT}(\mathcal{G}_{i-1})$, it will then follow that $\mathrm{OUT}(\mathcal{G}_{i-1})$ has a DAG rooted at u, completing the proof of Lemma 14.

Let us start with the scenario where $(t =)\ u \in \mathrm{LFU}(\Sigma)$. The crucial observation is that u must be the star of $\mathrm{GU}(\Sigma)$ for u itself. This is due to three facts: (i) no proper descendants of u (in T) can $\mathcal{G}_{i-1}$-reach u (otherwise, there would be a cycle), (ii) no proper ancestor of u can be the star (because u can $\mathcal{G}_{i-1}$-reach itself), and (iii) if a node v is smaller than u (under $\prec$) but not an ancestor of u, then v cannot $\mathcal{G}_{i-1}$-reach u (no-cross-reachability property of Lemma 5). Hence, Phase 1 of the algorithm returns $s^* = u$. As $s^* = u \notin \Sigma$, Phase 2 generates $\mathcal{G}_i = \mathcal{G}_{i-1}[T_u \ominus \mathrm{GU}(\Sigma)]$, which is clearly rooted at u.

The subsequent discussion will focus on the situation $(t =)\ u \notin \mathrm{LFU}(\Sigma)$. The following is a general fact (which holds regardless of if $t = u$).

Lemma 22. *If $u \in C \setminus \Sigma$ but $u \notin \mathrm{LFU}(\Sigma)$, then*

- *$parent(u)$ is the star of $\mathrm{GU}(\Sigma)$ for u;*
- *u is the smallest child of $parent(u)$ able to $\mathcal{G}_{i-1}$-reach u.*

Proof. The second bullet is a direct corollary from the no-cross-reachability property of Lemma 5.

The first bullet will follow from the definition of the star of $\mathrm{GU}(\Sigma)$, provided that we can establish:

- **(Fact 1)** If a node $v \in \mathrm{GU}(\Sigma)$ satisfies $v \prec parent(u)$ and $parent(u) \notin T_v$, then v cannot $\mathcal{G}_{i-1}$-reach u.
- **(Fact 2)** If a node $v \in \mathrm{GU}(\Sigma)$ satisfies $v \in T_{parent(u)}$ and $v \neq parent(u)$, then v cannot $\mathcal{G}_{i-1}$-reach u.

Fact 1 is also a direct corollary from the no-cross-reachability property. It remains to prove Fact 2.

Suppose that some v as defined in Fact 2 is able to $\mathcal{G}_{i-1}$-reach u. We observe:

[7] $\mathcal{G}_i$ is rooted either at s^* or $s^\#$. In the former case, we must have $s^* \notin \Sigma$. In the latter case, we must have $s^\# \notin \Sigma$: if $s^\# \in \Sigma$ then $s^\# \in \mathrm{GU}(\Sigma)$, which contradicts that s^* satisfies condition C2 (Section 3).

- v cannot be a descendant (in T) of any left sibling of u (otherwise that left sibling can $\mathcal{G}_{i-1}$-reach u, violating the second bullet of Lemma 22);
- $v \neq u$ (because $u \notin \Sigma$ and $u \notin \text{LFU}(\Sigma)$ tell us $u \notin \text{GU}(\Sigma)$);
- v cannot be a proper descendant of u in T (otherwise there is a cycle).

Thus, there must exist a right sibling u' of u such that $v \in T_{u'}$.

By Lemma 18, T_v must contain at least one node in Σ.[8] Hence, $T_{u'}$ also contains at least one node, say w, in Σ. But this means that u (being a left sibling of u') belongs to LF(w) and, hence, LFU(Σ), causing a contradiction. □

By the first bullet of the above lemma, Phase 1 returns $s^* = \textit{parent}(u)$. Since $u \in C$, we know $\textit{parent}(u) \in \Sigma$. By the lemma's second bullet, Phase 2 sets $s^\# = u$ and outputs $\text{OUT}(\mathcal{G}_{i-1}, t) = \mathcal{G}_{i-1}[T_u \ominus \text{GU}(\Sigma)]$, which is rooted at u.

L SOLVING THE FUNCTION $f(n)$ IN SEC. 5.2

We consider, without loss of generality, that $f(n) \le c_1 n$ for $n \le B$ where c_1 is a constant. Meanwhile, rewrite (3) into:

$$f(n) \quad \le \quad c_2(1 + |\Sigma| + |\text{OUT}(\mathcal{G}, 1)|) + \sum_{\mathcal{G}_1 \in \text{OUT}(\mathcal{G},1)} f(|\mathcal{G}_1|) \quad (13)$$

for some constant c_2. Set $c = \max\{c_1, c_2\}$. We will show $f(n) \le 4cn - 3c$. Assuming that this holds for all $n \le z - 1$ where integer z satisfies $z \ge B + 1 \ge 2$, we will prove its correctness for $n = z$.

Consider any $\mathcal{G}$ with z vertices. If $|\text{OUT}(\mathcal{G}, 1)| = 0$, then (13) gives $f(z) \le cz + c$ which is at most $4cz - 3c$ as long as $z \ge 2$. When $|\text{OUT}(\mathcal{G}, 1)| \ge 1$, we get from (13):

$$\begin{aligned} f(z) \quad &\le \quad c(1 + |\Sigma| + |\text{OUT}(\mathcal{G}, 1)|) + \sum_{\mathcal{G}_1 \in \text{OUT}(\mathcal{G},1)} f(|\mathcal{G}_1|) \\ &\le \quad c(1 + |\Sigma| + |\text{OUT}(\mathcal{G}, 1)|) + \sum_{\mathcal{G}_1 \in \text{OUT}(\mathcal{G},1)} (4c|\mathcal{G}_1| - 3c) \\ &= \quad c - 2c|\text{OUT}(\mathcal{G}, 1)| - 3c|\Sigma| + 4c\left(|\Sigma| + \sum_{\mathcal{G}_1 \in \text{OUT}(\mathcal{G},1)} |\mathcal{G}_1|\right) \end{aligned}$$

Recall from Section 5.2 that $|\Sigma| + \sum_{\mathcal{G}_1 \in \text{OUT}(\mathcal{G},1)} |\mathcal{G}_1| \le n = z$. Hence:

$$\begin{aligned} f(z) \quad &\le \quad c - 2c|\text{OUT}(\mathcal{G}, 1)| - 3c|\Sigma| + 4cz \\ &\le \quad 4cz + c - 2c(|\text{OUT}(\mathcal{G}, 1)| + |\Sigma|) \\ &\le \quad 4cz + c - 4c \end{aligned}$$

where the last inequality used $|\Sigma| \ge 1$ (root of $\mathcal{G}$ always in Σ) and $|\text{OUT}(\mathcal{G}, 1)| \ge 1$. Hence, $f(z) \le 4cz - 3c$, which completes the proof.

M PROOF OF LEMMA 15

We prove the lemma by induction on k. Obviously, a query under $k = 1$ has two outcome sequences. Assuming that the lemma is true for $k = z$ (for some $z \ge 1$), next we prove its correctness for $k = z + 1$. As before, let the query sequence Q be $q_1, q_2, ..., q_{z+1}$. At least one node in Q has the property that its subtree in $\mathcal{G}$ contains no other nodes in Q. Assume that q_{z+1} is such a node (otherwise, rename the nodes in Q).

For each selection of $t \in \mathcal{G}$, denote by $a_1(t), a_2(t), ..., a_{z+1}(t)$ the corresponding output sequence. If nodes t and t' both belong to the subtree of q_{z+1} (in $\mathcal{G}$), then $a_i(t) = a_i(t')$ for all $i \in [1, z]$. This is true because the subtree of q_{z+1} is either contained in that of q_i (in which case $a_i(t) = a_i(t') = 1$) or disjoint with that of q_i (in which case $a_i(t) = a_i(t') = 0$).

Consider the set of all output sequences $a_1(t), a_2(t), ..., a_{z+1}(t)$ as t ranges over all the vertices in $\mathcal{G}$. Divide the set into Group 1 where $a_{z+1}(t) = 1$ and Group 0 where $a_{z+1}(t) = 0$. Our earlier discussion implies that Group 1 has exactly one sequence. By the inductive assumption, Group 0 has at most $z + 1$ sequences. We thus conclude that there are at most $z + 2$ distinct $a_1(t), a_2(t), ..., a_{z+1}(t)$.

[8] This is obviously true if $v \in \Sigma$. Otherwise, the fact $v \in \text{GU}(\Sigma)$ tells us that $v \in \text{LFU}(w)$ for some $w \in \Sigma$. Lemma 18 shows that T_v must have at least one node in $\Sigma \setminus \{w\}$.