Partial Order Multiway Search

LU SHANGQI, Chinese University of Hong Kong, China
WIM MARTENS and MATTHIAS NIEWERTH, University of Bayreuth, Germany
YUFEI TAO, Chinese University of Hong Kong, China

Partial order multiway search (POMS) is a fundamental problem that finds applications in crowdsourcing, distributed file systems, software testing, and more. This problem involves an interaction between an algorithm $\mathcal{A}$ and an oracle, conducted on a directed acyclic graph $\mathcal{G}$ known to both parties. Initially, the oracle selects a vertex t in $\mathcal{G}$ called the *target*. Subsequently, $\mathcal{A}$ must identify the target vertex by probing reachability. In each *probe*, $\mathcal{A}$ selects a set Q of vertices in $\mathcal{G}$, the number of which is limited by a pre-agreed value k. The oracle then 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. We propose an algorithm to solve POMS in $O(\log_{1+k} n + \frac{d}{k} \log_{1+d} n)$ probes, where n represents the number of vertices in $\mathcal{G}$, and d denotes the largest out-degree of the vertices in $\mathcal{G}$. The probing complexity is asymptotically optimal. Our study also explores two new POMS variants: The first one, named *taciturn POMS*, is similar to classical POMS but assumes a weaker oracle, and the second one, named *EM POMS*, is a direct extension of classical POMS to the *external memory* (EM) model. For both variants, we introduce algorithms whose performance matches or nearly matches the corresponding theoretical lower bounds.

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

Additional Key Words and Phrases: Partial order, graph algorithms, data structures, lower bounds

ACM Reference format:
Lu Shangqi, Wim Martens, Matthias Niewerth, and Yufei Tao. 2023. Partial Order Multiway Search. *ACM Trans. Datab. Syst.* 48, 4, Article 10 (November 2023), 31 pages.
https://doi.org/10.1145/3626956

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 [18] exemplifies the multiway version of the above process. In each round, the search algorithm

The research of Shangqi Lu and Yufei Tao was partially supported by GRF projects 14222822, 14203421, and 14207820 from HKRGC. The research of Wim Martens was partially supported by grants 369116833 and 431183758 of the Deutsche Forschungsgemeinschaft (DFG).
Authors' addresses: L. Shangqi and Y. Tao, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, New Territories, Hong Kong; e-mails: {sqlu, taoyf}@cse.cuhk.edu.hk; W. Martens and M. Niewerth, Angewandte Informatik 7, University of Bayreuth, 95447 Bayreuth, Germany; e-mails: {wim.martens, matthias.niewerth}@uni-bayreuth.de.

0362-5915/2023/11-ART10 $15.00
https://doi.org/10.1145/3626956

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 seeks to discover t with the fewest rounds possible. ***Partial order multiway search* (POMS)**, which will be introduced shortly, extends these problems to arbitrary partial orders. This article presents a formal investigation of several variants of POMS.

1.1 Problem Definitions: Three Versions of Partial Order Multiway Search

The Classical Version. POMS can be framed as an interaction 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) that has a path to every other vertex. The interaction begins with the oracle selecting a *target* vertex t from $\mathcal{G}$. Subsequently, $\mathcal{A}$ must determine 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 $\mathcal{G}$.

Let n be the number of vertices in $\mathcal{G}$. It is evident that $\mathcal{A}$ can always discover t with $\lceil n/k \rceil$ probes by inquiring the oracle about each vertex in $\mathcal{G}$ explicitly. The challenge lies in proving a better bound on the number of probes.

Taciturn POMS. This POMS variant is also an interaction between an *oracle* and an *algorithm* $\mathcal{A}$. As before, the oracle first picks a *target* vertex t from a single-rooted DAG $\mathcal{G}$, after which $\mathcal{A}$ aims to find out which vertex is t with probing. To perform a *probe*, $\mathcal{A}$ still chooses a set Q of vertices with $|Q| \leq k$, where k is a problem parameter. However, the oracle returns only a binary answer:

— yes, if at least one vertex in Q can reach t;
— no, otherwise (i.e., none of the vertices in Q can reach t).

Compared to a traditional oracle, the oracle reveals less information (hence, the name "taciturn"). The algorithmic challenge is again to minimize the number of probes.

POMS in External Memory. In the ***external memory* (EM)** model [3], a machine is equipped with (i) a *disk*, which is an unbounded sequence of words divided into *blocks* of $B \geq 2$ words, and (ii) *memory*, which is a sequence of M words. The structure's *space* is the number of blocks occupied. An *I/O operation* reads a block of data from the disk to memory.[1] The value of M is assumed to be larger than B by a sufficiently large constant factor.[2]

In the EM POMS problem, we are permitted to preprocess a single-rooted DAG $\mathcal{G}$ into a disk-resident structure. To start the interaction, the oracle (as in classical POMS) chooses a target t from $\mathcal{G}$. Then, an algorithm $\mathcal{A}$ performs a sequence of *probes*, each of which involves three steps:

(1) $\mathcal{A}$ reads a set Q of vertices from a disk block into memory.
(2) The oracle reveals the reachability (to t) for all the vertices in Q.
(3) $\mathcal{A}$ clears up memory to finish the probe.

Naively, $\mathcal{A}$ can store all the n vertices arbitrarily 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.

[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 article.
[2] The strictest EM model [3] requires an algorithm to work even if $M \geq 2B$. However, as shown in Reference [26], 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.

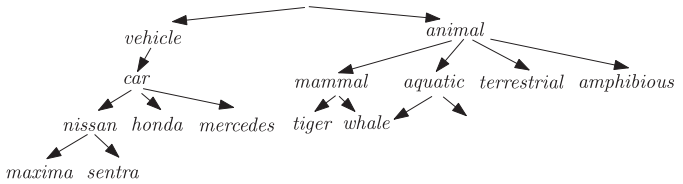


Fig. 1. POMS in image classification with crowdsourcing.

1.2 Motivation

Classical POMS. In the database area, a significant application of POMS is *image classification with crowdsourcing* [39], 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. This application highlights 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 (respectively, positive) answer to the question "*is this a* **vehicle**?", an algorithm can eliminate all the concepts that are (respectively, 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 Reference [37], 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, then (with high probability) the folder 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 References [7, 37, 39] for more POMS applications in software testing, relational databases, and workflow management.

Taciturn POMS. This POMS variant offers a more human-friendly way to implement crowdsourcing algorithms. Consider the image classification scenario described earlier. Under classical POMS, the amount of feedback solicited from human beings can be excessive: Each probe requires a human to answer k questions, which means at least k mouse clicks on a crowdsourcing platform. This task can become rather tedious when k is large, potentially discouraging human participation. Unfortunately, as will become evident later, the value of k cannot be too small to ensure a limited number of probes. Taciturn POMS offers an appealing remedy to this issue. Regardless of how large k is, a human only needs to provide a Boolean answer, which can be done with a single click. Moreover, if any vertex in Q can reach t, then the human can directly answer yes and safely disregard the other vertices in Q.

EM POMS. Making a "conventional" data structure—that is, an index designed for memory-resident data—I/O-efficient is non-trivial, because one must consider the effects of reading and writing in blocks. Ideally, we want a generic reduction that can convert an arbitrary in-memory

structure to an EM version with excellent performance. However, designing such reductions is still a major challenge today.

We observe that a solution to EM POMS offers a reduction applicable to a class of *region-based* structures satisfying the following requirements:

- 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*—denoted as reg_u—which is a subset of the search space $\mathbb{Q}$ and can be described in $O(1)$ words.
- All the leaves (i.e., vertices with an out-degree of 0) have disjoint regions whose union is $\mathbb{Q}$.
- For each vertex u, its reg_u is the union of the regions of all the leaves reachable from u.
- A *query* selects 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.

In a region-based structure, a query can be modeled as an instance of EM POMS. Let t be the leaf whose region contains the query element q. We may 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 Q of vertices in a disk block, then we store Q, as well as the regions of the vertices therein, in $O(|Q|/B) = O(1)$ disk blocks. In answering a query, if $\mathcal{A}$ reads the block on Q, then we read the corresponding $O(1)$ blocks to acquire all the information needed to resolve reachability (to t) for the vertices in Q. This enables us to simulate the execution of $\mathcal{A}$ with asymptotically the same I/O cost. We will demonstrate the reduction's power by employing our solution to EM POMS to obtain an optimal I/O-efficient index for the *vertical ray shooting problem* for free.

1.3 Related Work

A (deterministic) algorithm $\mathcal{A}$ for classical POMS can be modeled as a *decision tree*. Each node in the tree is associated with a set Q of vertices. The set associated with the root represents the set Q of vertices chosen by $\mathcal{A}$ to perform the first probe. Recall that, upon being given a probe with set Q, the oracle must reveal, for each vertex $q \in Q$, whether q can reach the target vertex t. Since each q may or may not reach t, there can be at most $2^{|Q|}$ different outcomes for the probe. For each outcome, node Q has a child node in the tree, whose associated vertex set Q' represents the set of vertices chosen by $\mathcal{A}$ to perform the next probe in that outcome's situation. Specially, if $Q' = \emptyset$, then the child is a leaf of the tree, indicating that $\mathcal{A}$ has already found t. Thus, designing a POMS algorithm amounts to finding such a decision tree.

To understand the POMS literature, it is important to distinguish between the *instance-oriented* and *class-oriented* categories, because they have drastically different objectives.

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

$$maxcost_k^{\text{inst}}(\mathcal{A}, \mathcal{G}) = \max_t cost_k(\mathcal{A}, \mathcal{G}, t),$$

namely, the largest cost over all possible t in $\mathcal{G}$.

As explained earlier, the algorithm $\mathcal{A}$ can be modeled as a decision tree. As the number of possible decision trees is finite, the problem of computing an optimal decision tree—a.k.a. finding an algorithm $\mathcal{A}^*$ with the lowest $maxcost_k^{\text{inst}}(\mathcal{A}^*, \mathcal{G})$—is decidable. In the *instance-oriented* category

of POMS, the main objective is to minimize the amount of time needed to discover an optimal decision tree.

The task of computing an optimal decision tree is best understood when $\mathcal{G}$ is a tree and $k = 1$. In that case, Ben-Asher et al. [7] were the first to show that this can be achieved in time polynomial to the number n of vertices in $\mathcal{G}$. Their work motivated a line of research looking for faster solutions [19, 21, 27, 32–34, 37]. Amazingly, the task turned out to be solvable in $O(n)$ time! This was first stated by Mozes et al. [34]; later, Dereniowski [21] 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 [33].

In contrast, it is NP-hard to compute an optimal decision tree on a DAG $\mathcal{G}$ even if $k = 1$. This opens the door to studying how to compute a decision tree whose corresponding algorithm $\mathcal{A}$ ensures a $maxcost_k^{\text{inst}}(\mathcal{A}, \mathcal{G})$ that is sufficiently close to $maxcost_k^{\text{inst}}(\mathcal{A}^*, \mathcal{G})$. To that end, Arkin et al. [5] showed that, for any DAG $\mathcal{G}$ and $k = 1$, it is possible to compute in polynomial time a decision tree corresponding to an algorithm $\mathcal{A}$ whose $maxcost_k^{\text{inst}}(\mathcal{A}, \mathcal{G})$ is higher than $maxcost_k^{\text{inst}}(\mathcal{A}^*, \mathcal{G})$ by a factor of $O(\log n)$.[3]

We are not aware of any results for $k > 1$ even when $\mathcal{G}$ is a tree.

Class-oriented POMS. In the class-oriented POMS category, the focus shifts from the computability of an algorithm's decision tree to evaluating an algorithm's performance across a class of single-rooted DAGs. To explain, let $\mathscr{C}$ be an arbitrary set of single-rooted DAGs. The following metric provides a way to measure the quality of an algorithm $\mathcal{A}$ with respect to the whole class $\mathscr{C}$:

$$maxcost_k^{\text{class}}(\mathcal{A}, \mathscr{C}) = \max_{\mathcal{G} \in \mathscr{C}} maxcost_k^{\text{inst}}(\mathcal{A}, \mathcal{G}).$$

This metric represents the largest cost that $\mathcal{A}$ incurs on any of the graphs in $\mathscr{C}$. Define

$$minmaxcost_k(\mathscr{C}) = \min_{\mathcal{A}} maxcost_k^{\text{class}}(\mathcal{A}, \mathscr{C}). \tag{1}$$

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

Define

$$\mathscr{G}(n, d) = \{ \text{single-rooted DAG } \mathcal{G} \mid \mathcal{G} \text{ has } n \text{ vertices and maximum out-degree } d \}, \tag{2}$$

$$\mathscr{T}(n, d) = \{\mathcal{G} \in \mathscr{G}(n, d) \mid \mathcal{G} \text{ is a tree}\}. \tag{3}$$

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 [6] 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 [32] tightened the upper bound and proved that $minmaxcost_1(\mathscr{T}(n, d)) \in \Theta(d \log_{1+d} n)$ (see also References [22, 24] where the same result was derived). Regarding $\mathscr{G}(n, d)$ and arbitrary $k \geq 1$, Tao et al. [39] 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 Reference [39] holds even when replacing $\mathscr{G}(n, d)$ with $\mathscr{T}(n, d)$.

Remarks. Taciturn POMS and EM POMS, both introduced in this work, have not been studied previously. Regarding other POMS variants, we note that instance-oriented POMS with $k = 1$ has been studied under various other setups [2, 10–17, 20, 23, 28, 30, 31]. These setups differ in several

[3] A better approximation ratio $O(\log n / \log \log n)$ was claimed in Reference [21] but unfortunately is not correct, as has been confirmed by our personal communication with the author of Reference [21].

aspects, such as: (i) whether the cost of a probe depends on the provided vertex, (ii) whether the goal is to minimize the worst-case cost for a given t or the average cost over a distribution of t, and (iii) whether the oracle's answer can be noisy.

1.4 Our Contributions

Classical POMS. Our first contribution is to settle class-oriented POMS optimally.

Theorem 1. *For the POMS problem, let n represent the number of vertices in the input graph $\mathcal{G}$ and d denote the maximum vertex out-degree in $\mathcal{G}$. Both of the following statements are true:*

- *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 implies:

$$minmaxcost_k(\mathscr{G}(n,d)) \quad = \quad \Theta\left(\log_{1+k} n + \frac{d}{k} \log_{1+d} n\right). \tag{4}$$

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

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

We also deploy Theorem 1 to derive a new result for instance-oriented POMS:

Theorem 3. *Consider any tree $\mathcal{G} \in \mathscr{T}(n,d)$ and an arbitrary integer $k \in [1,n]$. Let $\mathcal{A}^*$ be an algorithm achieving the lowest $maxcost_k^{\text{inst}}(\mathcal{A}^*, \mathcal{G})$. We can compute in $\text{poly}(n)$ time the decision tree of an algorithm $\mathcal{A}$ satisfying*

$$\frac{maxcost_k^{\text{inst}}(\mathcal{A}, \mathcal{G})}{maxcost_k^{\text{inst}}(\mathcal{A}^*, \mathcal{G})} \quad = \quad O\left(\frac{\log n}{\log(1+k) + \log\log n}\right).$$

Note that the ratio in the Theorem 3 is no worse than $O(\log n / \log\log n)$. Furthermore, the ratio is $O(1)$ when $k = \Omega(n^\epsilon)$ for any constant $\epsilon > 0$.

Taciturn POMS. We establish the following for taciturn POMS:

Theorem 4. *For the taciturn POMS problem, let n represent the number of vertices in the input graph $\mathcal{G}$ and d denote the maximum vertex out-degree in $\mathcal{G}$. Both of the following statements are true:*

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

Our algorithm (the first bullet) is optimal up to an $O(\log(1+k))$ factor. It is interesting to compare the two POMS problems: classical vs. taciturn. A classical oracle reveals up to k bits of information (i.e., one bit for each vertex in Q, encoding the vertex's reachability to t), whereas a taciturn oracle reveals only a single bit. Therefore, by trying to simulate a classical oracle with a taciturn oracle, one is forced to do k (taciturn) probes in the worst case, regardless of the strategy. Therefore, naively, one would expect a blow-up factor of k in the probing complexity. However, by comparing Theorems 1 and 4, one can see that taciturn POMS demands more probes than classical POMS by only a polylogarithmic factor.

EM POMS. We establish the following for EM POMS:

Table 1. Summary of the Previous and New Results

POMS	ref.	cost	remark
classical	[6]	$O(d \log n)$	$\mathcal{G}$ is a tree and $k = 1$
classical	[22, 24, 32]	$O(d \log_{1+d} n)$	$\mathcal{G}$ is a tree and $k = 1$
classical	[39]	$O((\log n)(\log_{1+k} n) + \frac{d}{k} \log_{1+d} n)$	any DAG $\mathcal{G}$ and any k
classical	**this article**	$O(\log_{1+k} n + \frac{d}{k} \log_{1+d} n)$	any DAG $\mathcal{G}$ and any k
classical	[6]	$\Omega(d \log_{1+d} n)$	$\mathcal{G}$ is a tree and $k = 1$
classical	[39]	$\Omega(\frac{d}{k} \log_{1+d} n)$	$\mathcal{G}$ is a tree and any k
classical	**this article**	$\Omega(\log_{1+k} n + \frac{d}{k} \log_{1+d} n)$	$\mathcal{G}$ is a tree and any k
taciturn	**this article**	$O(\log n \cdot \log(1 + k) + \frac{d}{k} \log_{1+d} n)$	any DAG $\mathcal{G}$ and any k
taciturn	**this article**	$\Omega(\log n + \frac{d}{k} \log_{1+d} n)$	$\mathcal{G}$ is a tree and any k
EM	**this article**	$O(\log_B n + \frac{d}{B} \log_{1+d} n)$	any DAG $\mathcal{G}$, space $O(n/B)$
EM	**this article**	$\Omega(\log_B n + \frac{d}{B} \log_{1+d} n)$	$\mathcal{G}$ is a tree and regardless of space

Theorem 5. *For the EM POMS problem, let n represent the number of vertices in the input graph $\mathcal{G}$ and d denote the maximum vertex out-degree in $\mathcal{G}$. Both of the following statements are true:*

- *There is a structure of $O(n/B)$ space that can discover 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.*

Interestingly, our structure's space and I/O complexities do not rely on the number of edges in $\mathcal{G}$. When $d = O(B)$, our I/O cost becomes $O(\log_B n)$. Combining Theorem 5 and the discussion in Section 1.2 shows 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! In Section 6.4, we will employ our techniques to develop a simple, optimal, EM index for the vertical ray shooting problem.

Empirical Evaluation for POMS. We present an empirical evaluation that examines the practical efficiency of our algorithm in Theorem 1, using the state-of-the-art [39] as a benchmark. The results are promising: The proposed algorithm exhibited robust performance and consistently outperformed the method of Reference [39] in all scenarios. Our evaluation also covers taciturn POMS, but not EM POMS for which our contributions are theoretical in nature.

Remarks. Table 1 provides a summary of our results and offers comparisons to previous findings where appropriate.

In this article, our discussion assumes that $\mathcal{G}$ is single-rooted; however, all the POMS definitions can be extended to DAGs with multiple roots in a straightforward manner. In general, an algorithm designed for single-rooted DAGs can be utilized to tackle multi-root DAGs as well. Consider, for example, classical POMS. If $\mathcal{G}$ has ρ roots, then we can conceptually add a dummy root that has an edge to each of the ρ original roots. This effectively creates a single-rooted DAG $\mathcal{G}'$ where each vertex has an out-degree at most $\max\{\rho, d\}$, with d being the largest out-degree in $\mathcal{G}$. It should now be rudimentary to adapt a single-rooted algorithm to perform POMS on $\mathcal{G}'$.

A preliminary version of this work appeared in Reference [1]. Besides an improved presentation of the content in the conference version, the current article also introduces taciturn POMS as a new problem, presents a systematic investigation of the problem, and features an experimental study to confirm our algorithms' usefulness in practice.

2 PRELIMINARIES

Basic Concepts and Notations. Henceforth, every "tree"—unless otherwise stated—should be understood as a *rooted* tree. The *size* of a tree T, denoted as $|T|$, is the number of nodes in T. The notation $u \in T$ (respectively, $u \notin T$) indicates that u is (respectively, is not) a node of T. The notation $parent(u)$ gives the parent node of u and is undefined if u is the root. The subtree of a node $u \in T$—denoted as T_u—is the tree induced by the descendants of u in T; the root of T_u is u. We would like to remind the reader that a node is considered a descendant of itself (though not a proper descendant); similarly, a node is an ancestor of itself (but not a proper ancestor).

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, the subgraph of G induced by the vertices in T is denoted as $G[T]$. Note that $G[T]$ must be a single-rooted DAG. If node u can reach node v in G, then we 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, otherwise.

Note that if $v \notin T_u$, then $T_u \ominus \{v\} = T_u$; furthermore, $T_u \ominus \{v\}$ always contains the root of T_u.

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

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

Note that $T_u \ominus S$ is always a non-empty tree, because it always contains u.

Heavy-path Depth First Search Tree. Consider 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)**, however, processes the white out-neighbor v_{best} of u_{top} that is able to G-reach the most white vertices via white paths.[4]

HPDFS defines a tree T—the *HPDFS-tree* [39]—where a node u parents another v if the latter is discovered while the former tops the stack. It also determines a total order $\prec$ on the vertices in G: We define $u \prec v$—read as "*u is smaller than v*" or "*v is 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 6. *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 $u' \prec v'$ for every $u' \in T_u$ and $v' \in T_v$.*
— **(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 for every node v that lies on at least one u-to-w path in G, we have $v \in T_u$.*
— **(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 including only white vertices. If two or more nodes satisfy this condition, then v_{best} can be any of them.

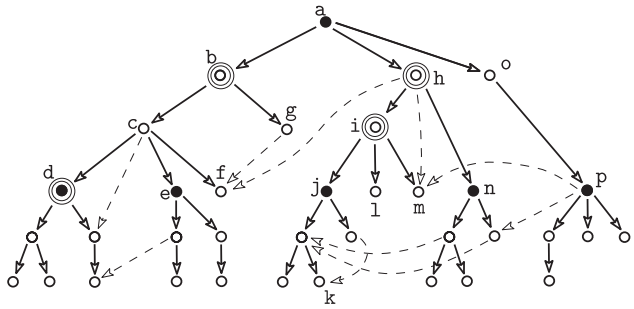


Fig. 2. A running example. G is the graph represented by both the solid and dashed edges. An HPDFS T of G is indicated by the solid edges. The labels on the nodes are consistent with the total order $\prec$. The black nodes constitute the 8-separator $\Sigma = \{\mathsf{a}, \mathsf{d}, \mathsf{e}, \mathsf{j}, \mathsf{n}, \mathsf{p}\}$ of T. The nodes of $\text{LFU}(\Sigma) = \{\mathsf{b}, \mathsf{d}, \mathsf{h}, \mathsf{i}\}$ are shown using concentric circles.

Example. Consider G as the graph that has all the solid and dashed edges in Figure 2. 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 $\mathsf{h} \notin T_{\mathsf{b}}$, the no-cross-reachability property assures us that b cannot G-reach any node in T_{h}. Because $\mathsf{k} \in T_{\mathsf{h}}$, 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

In this section, our discussion will be purely graph theoretic and will revolve around a single-rooted DAG G with n nodes, an arbitrary HPDFS-tree T of G, and an ordering $\prec$ on the vertices of G determined by T. The core of the discussion is:

> **Path Preservation:** Given a target vertex t in G, how to find a vertex u—which is neither t nor the root of G—such that every u-to-t path in G is preserved in $G[T_u]$?

Recall that $G[T_u]$ is the subgraph of G induced by the vertices in the subtree of u in T. It is worth emphasizing that $G[T_u]$ must contain *all* the edges of *every* u-to-t path in G. Our main finding is that such a vertex u can always be found from a small collection of vertices in G, unless the same collection already contains t.

Next, we will need to prove several fundamental properties in Sections 3.1–3.3 before presenting our findings in Section 3.4.

3.1 Separators

It is well known that each tree T must contain a node whose removal disconnects T into trees each having at most $n/2$ nodes (see Reference [29] for a proof). We now prove a more general fact.

Lemma 7. *Let T be an HPDFS-tree of a single-rooted DAG with n vertices. For every $\lambda \in [2, n]$, there is a set S of at most $\lambda - 1$ nodes whose removal disconnects T into trees each having at most n/λ nodes.*

Proof. We can find such a set S using the algorithm below:

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

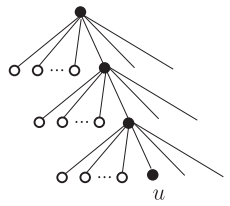


Fig. 3. The left flank of u is the set of white nodes.

4. add u to S; remove T'_u from T'
5. **return** S

It is easy to verify that the removal of S disconnects T into trees each having at most n/λ nodes. It remains to show that $|S| \leq \lambda - 1$. Every time we add a node into S at Line 4, $\lfloor n/\lambda \rfloor + 1 > n/\lambda$ nodes are removed from T'. If $|S| \geq \lambda$, then the total number of nodes removed would be *strictly* larger than $|S| \cdot n/\lambda \geq \lambda \cdot n/\lambda = n$, giving a contradiction. □

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

— if the output S of **construct-separator** contains the root of T, then $\Sigma = S$;
— otherwise, $\Sigma = S \cup \{\text{root of } T\}$.

It holds by Lemma 7 that $|\Sigma| \leq \lambda$.

Example. Assume that T is the tree in solid edges as shown in Figure 2 (T has 36 nodes). The 8-separator of T is $\Sigma = \{\mathsf{a}, \mathsf{d}, \mathsf{e}, \mathsf{j}, \mathsf{n}, \mathsf{p}\}$; the above algorithm finds the nodes of Σ in the order $\mathsf{d}, \mathsf{e}, \mathsf{j}, \mathsf{n}, \mathsf{p}$, and a. Figure 2 colors all the nodes of Σ in black.

3.2 Left Flanks, Left Flank Unions, and Grand Unions

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 $\text{LF}(u)$—as the set of left siblings of the nodes on π. See Figure 3 for an illustration.

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

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

and the *grand union* (GU) of Σ is

$$\text{GU}(\Sigma) = \Sigma \cup \text{LFU}(\Sigma). \tag{5}$$

Example. Consider again the graph G in Figure 2 with $\lambda = 8$. As explained before, $\Sigma = \{\mathsf{a}, \mathsf{d}, \mathsf{e}, \mathsf{j}, \mathsf{n}, \mathsf{p}\}$. The left flank of node p is $\text{LF}(\mathsf{p}) = \{\mathsf{b}, \mathsf{h}\}$, while $\text{LF}(\mathsf{n}) = \{\mathsf{b}, \mathsf{i}\}$. It is easy to verify that $\text{LFU}(\Sigma) = \{\mathsf{b}, \mathsf{d}, \mathsf{h}, \mathsf{i}\}$ and $\text{GU}(\Sigma) = \{\mathsf{a}, \mathsf{b}, \mathsf{d}, \mathsf{e}, \mathsf{h}, \mathsf{i}, \mathsf{j}, \mathsf{n}, \mathsf{p}\}$.

Next, we will prove several properties of the above concepts.

Lemma 8. *For every node $u \in \Sigma$ and node $v \in \text{LF}(u)$, the tree T_v has at least one node in $\Sigma \setminus \{u\}$.*

Proof. By definition of left flank, node v must have a right sibling v' on the root-to-u path in T and this node v' must be an ancestor of u. By construction of Σ using the method **construct-separator** (Section 3.1), we must have $|T_u| \geq \lfloor n/\lambda \rfloor + 1$, implying $|T_{v'}| \geq \lfloor n/\lambda \rfloor + 1$, which in turn yields $|T_v| \geq \lfloor n/\lambda \rfloor + 1$ (subtree-size property of Lemma 6). As the removal of Σ disconnects T into trees each having at most n/λ nodes, T_v must contain at least one node in Σ, which cannot be u (because v' is a right sibling of v and also an ancestor of u). □

Corollary 9. *For every node $v \in \mathrm{LFU}(\Sigma)$, the tree T_v contains at least one node in Σ.*

Proof. The fact $v \in \mathrm{LFU}(\Sigma)$ means that $v \in \mathrm{LF}(u)$ for some $u \in \Sigma$. The claim then follows from Lemma 8. □

Lemma 10. $|\mathrm{LFU}(\Sigma)| \leq |\Sigma| - 1$ *and* $|\mathrm{GU}(\Sigma)| < 2\lambda$.

Proof. We will prove only $|\mathrm{LFU}(\Sigma)| \leq |\Sigma| - 1$, because $|\mathrm{GU}(\Sigma)| < 2\lambda$ will then follow immediately from Equation (5) and Lemma 7.

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; note that T^P is a tree. Consider any node $v \in \mathrm{LFU}(\Sigma)$. By definition of $\mathrm{LFU}(\Sigma)$, we have that $v \in \mathrm{LF}(u)$ for some $u \in \Sigma$. Therefore, T_v contains at least one node in Σ (Lemma 8) and $v \in T^P$. In other words, all the nodes in $\mathrm{LFU}(\Sigma)$ are in T^P.

Root T^P at the root of T. 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 $v \in \mathrm{LFU}(\Sigma)$ satisfies:

(i) some node w in T^P is a right sibling of v in T (by definition of left-flank unions), and
(ii) $\mathit{parent}(v) \in I$ (because both v and w are in T^P).

This implies that each $u \in I$ has at most $c_u - 1$ child nodes in $\mathrm{LFU}(\Sigma)$ (note that the largest child of u in T^P cannot be in $\mathrm{LFU}(\Sigma)$). This yields $|\mathrm{LFU}(\Sigma)| \leq \sum_{u \in I} (c_u - 1)$.

It remains to prove that $\sum_{u \in I} (c_u - 1) \leq |\Sigma| - 1$. Denote by x the number of leaves in T^P and by y the number of internal nodes in T^P that have only one child in T^P. By definition of T^P, every leaf node of T^P must belong to Σ; hence, $x \leq |\Sigma|$.

Now, let us view T^P as an *undirected* tree. Under this view, we have:

$$\begin{aligned} \text{degree sum of all vertices in (the undirected) } T^P &= 2 \cdot \text{number of edges in } T^P \\ \Rightarrow x + 2y + \left(\sum_{u \in I} (c_u + 1) \right) - 1 &= 2 \cdot (|I| + x + y - 1) \\ \text{(note: the ``}-1\text{'' is for the root of } T^P\text{)} & \\ \Rightarrow \sum_{u \in I} (c_u - 1) &= x - 1 \leq |\Sigma| - 1. \end{aligned}$$

□

Lemma 11. *If node u can G-reach node v, then $v \in T_{u^*}$, where u^* is the smallest node (under $\prec$) in $\mathrm{LF}(u) \cup \{u\}$ able to G-reach v.*

Proof. We first show that $v \in T_{u^*}$ for some node $u^* \in \mathrm{LF}(u) \cup \{u\}$. Let π be the root-to-u path in T and p be the lowest ancestor of u such that $v \in T_p$. If $p = u$, then we have found a node $u^* = u \in \mathrm{LF}(u) \cup \{u\}$ satisfying $v \in T_{u^*}$. Consider now $p \neq u$. Define w as the child of p on π (note: w can be u); hence, $v \notin T_w$ by definition of p. Since $p \neq v$ (otherwise, v can G-reach u and, thus, G has a cycle), p must have a child u^* such that $v \in T_{u^*}$. We now argue that u^* is a left sibling of w and, hence, belongs to $\mathrm{LF}(u)$. First observe that w can G-reach v, because u can G-reach v and w is an ancestor of u. Therefore, if u^* would be a right sibling of w, then the nodes w and u^* cause a violation of the no-cross-reachability property of Lemma 6: w can G-reach a node (i.e., v) in T_{u^*}. Moreover, we have that $u^* \neq w$, because $v \in T_{u^*}$ and $v \notin T_w$. It thus follows that u^* must be a left sibling of w.

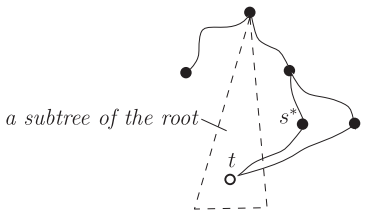


Fig. 4. S is the set of black vertices. The star of S for t is s^*.

As the nodes in $\text{LF}(u) \cup \{u\}$ are not ancestors of each other, there is a unique node $u^* \in \text{LF}(u) \cup \{u\}$ satisfying $v \in T_{u^*}$. By the no-cross-reachability property, no $w' \in \text{LF}(u) \cup \{u\}$ with $w' \prec u^*$ can G-reach v. Hence, u^* is the smallest node in $\text{LF}(u) \cup \{u\}$ that can G-reach v, as claimed. □

3.3 Stars

Next, we introduce *star*, another concept crucial to our technical development. Let S be a non-empty set of vertices in G that includes the root of G. For any vertex t in 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)** no other node $s \in S$ satisfies (i) $s \in T_{s^*}$ and (ii) s can G-reach t.

See Figure 4 for an illustration. Note that the root's presence in S guarantees the existence of s^*.

Example. Consider Figure 2 with $t = \mathsf{k}$ and $S = \{\mathsf{a}, \mathsf{b}, \mathsf{h}, \mathsf{l}, \mathsf{m}, \mathsf{p}\}$. The star s^* of S for t is h.

The next two lemmas present some properties of the star.

Lemma 12. *Let S be a set of vertices in G. For every $u \in S$, the star of S for u must be u itself.*

Proof. This is due to three facts: (i) no proper descendant of u in T can G-reach u (otherwise, there would be a cycle), (ii) no proper ancestor of u can be the star of S for u, because u can G-reach itself, and (iii) if a node v is smaller than u (under $\prec$) but not an ancestor of u, then v cannot G-reach u (no-cross-reachability property of Lemma 6). □

Lemma 13. *Let Σ be a λ-separator of T where λ can be any value at least 2. If node u is a child node of some node in Σ but $u \notin \text{GU}(\Sigma)$, then $parent(u)$ is the star of $\text{GU}(\Sigma)$ for u.*

Proof. The claim will follow from the definition of star, provided that we can show:

— If a node $v \in \text{GU}(\Sigma)$ satisfies $v \prec parent(u)$ and $parent(u) \notin T_v$, then v cannot G-reach u.
— If a node $v \in \text{GU}(\Sigma)$ satisfies $v \in T_{parent(u)}$ and $v \neq parent(u)$, then v cannot G-reach u.

The first bullet is a direct corollary of the no-cross-reachability property (Lemma 6). Next, we focus on the second bullet.

Suppose that some node v as defined in the second bullet can G-reach u. We observe:

— v cannot be a descendant (in T) of any left sibling of u (otherwise, the left sibling of u can G-reach u through v, violating the no-cross-reachability property);
— $v \neq u$ (because $v \in \text{GU}(\Sigma)$ yet $u \notin \text{GU}(\Sigma)$);
— 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 satisfying $v \in T_{u'}$.

We argue that T_v must contain at least one node in Σ. This is obviously true if $v \in \Sigma$. If, however, $v \notin \Sigma$, then the fact $v \in \text{GU}(\Sigma)$ tells us that $v \in \text{LF}(w)$ for some $w \in \Sigma$. In that case, by Lemma 8, T_v must have at least one node in $\Sigma \setminus \{w\}$.

Because T_v has a node in Σ and $v \in T_{u'}$, it follows that $T_{u'}$ also has a node—denoted as w—in Σ. But this means that u (being a left sibling of u') belongs to $\text{LF}(w)$ and, hence, also belongs to $\text{LFU}(\Sigma)$, contradicting that $u \notin \text{GU}(\Sigma)$. □

3.4 Path Preservation Lemmas

Recall that the core of our discussion in this section is path preservation, which, as explained before, is to find a vertex u—which is neither t nor the root of G—such that every u-to-t path in G is preserved in $G[T_u]$. Next, we will prove that such a vertex u can always be identified from a small collection of vertices in G, unless the collection already contains the target t. We will present our findings in three lemmas, which are useful in different scenarios. These lemmas demonstrate the importance of all the concepts introduced earlier: separator, left flanks, grand unions, and stars.

Lemma 14 (Path Preservation Using a Root-Containing Set). *Let t be a vertex in G, let S be a set of vertices in G including the root, and let s^* be the star of S for t. Suppose that $t \in T_{s^*}$ yet $t \neq s^*$. If $s^\#$ is the smallest 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 $G[T_{s^\#} \ominus S]$.*

Proof. To prove the first bullet, notice that $t \notin T_v$ for any left sibling v of $s^\#$; otherwise, $s^\#$ would not be the *smallest* child of s^* that can G-reach t. However, $t \notin T_v$ for any right sibling v of $s^\#$; otherwise, $s^\#$, v, and t cause a violation of the no-cross-reachability property of Lemma 6. Hence, $t \in T_{s^\#}$.

Next, we prove 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 second bullet will then follow, because $G[T_{s^\#} \ominus S]$ is a vertex-induced subgraph of G. Because $t \in T_{s^\#}$, the path-descendants property of Lemma 6 indicates $u \in T_{s^\#}$. If $u \notin T_{s^\#} \ominus S$, then u must be "shielded" by S, namely, there is some node $s \in S$ satisfying $s \neq s^\#$, $s \in T_{s^\#}$, and $u \in T_s$. Given that u can G-reach t, node s must also be able to G-reach t. However, $s \in T_{s^\#}$ tells us that $s \in T_{s^*}$; thus, s^* violates Condition C2 (Section 3.3) in the definition of star, giving a contradiction. □

Example. Consider Figure 2 with $t = \mathsf{k}$ and $S = \{\mathsf{a}, \mathsf{b}, \mathsf{h}, \mathsf{l}, \mathsf{m}, \mathsf{p}\}$. As mentioned, the star s^* of S for t is h. Both child nodes of h (i.e., i and n) can G-reach $t = \mathsf{k}$. Hence, $s^\# = \mathsf{i}$. $T_{s^\#} \ominus S$—the tree obtained by "shielding" T_i with S—consists of the edges in T_j plus the edge (i, j). Note that i has two paths to k in G, both of which are preserved in $G[T_{s^\#} \ominus S]$.

Lemma 15 (Path Preservation Using a Left Flank). *Let t be a vertex in G, let Σ be the k-separator of T, and let s^* be 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 $G[T_{s^{**}} \ominus \Sigma]$.*

Proof. As s^* can G-reach t, the first bullet follows directly from Lemma 11.

Next, we prove the second bullet. Consider an arbitrary s^{**}-to-t path π in G. We argue that every node u on π is in $T_{s^{**}} \ominus \Sigma$. The second bullet will then follow, because $G[T_{s^{**}} \ominus \Sigma]$ is a vertex-induced subgraph of G. By the path-descendants property of Lemma 6, we must have $u \in T_{s^{**}}$ (recall that we have shown $t \in T_{s^{**}}$). If $u \notin T_{s^{**}} \ominus \Sigma$, then u must be "shielded" by Σ, namely, there is some node

$s \in \Sigma$ satisfying $s \neq s^{**}$, $s \in T_{s^{**}}$, and $u \in T_s$. We must have $s^* \neq s^{**}$; otherwise, T_{s^*} has a node—i.e., s, which is different from s^*—in Σ able to G-reach t, violating Condition C2 (Section 3.3) in the definition of star. Thus, $s^{**} \in \text{LF}(s^*)$ and therefore $s^{**} \prec s^*$. It follows that $s \prec s^*$ (order property of Lemma 6). Hence, T_s contains a node (namely, s itself) that is in Σ, can G-reach t, and is smaller than s^*. This contradicts s^* being the star of Σ for t. □

Example. Consider Figure 2 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 $G[T_{s^{**}} \ominus \Sigma]$.

Lemma 16 (Path Preservation Using a Grand Union). *Let t be a vertex in G, let Σ be the k-separator of T, and let s^* be the star of* $\text{GU}(\Sigma)$ *for t. Then*

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

Proof. We first show $\text{LF}(s^*) \subseteq \text{LFU}(\Sigma)$. This holds by definition if $s^* \in \Sigma$. Consider now $s^* \in \text{GU}(\Sigma) \setminus \Sigma$, which means $s^* \in \text{LFU}(\Sigma)$. By Corollary 9, T_{s^*} contains at least one node $u \in \Sigma$; thus, $\text{LF}(s^*) \subseteq \text{LF}(u) \subseteq \text{LFU}(\Sigma)$.

Next, we prove the first bullet of Lemma 16. We claim that no nodes in $\text{LF}(s^*)$ can G-reach t. To understand why, note that every node in $\text{LF}(s^*)$ is smaller than s^*, and $\text{LF}(s^*) \subseteq \text{LFU}(\Sigma) \subseteq \text{GU}(\Sigma)$. Thus, if some node in $\text{LF}(s^*)$ can G-reach t, then s^* cannot be the smallest node in $\text{GU}(\Sigma)$ satisfying Conditions C1 and C2 (Section 3.3), violating the definition of star. Equipped with the claim, we can now obtain $t \in T_{s^*}$ from Lemma 11 (here, we applied the lemma by setting $u = s^*$ and $v = t$, noticing that the node u^* in the lemma's statement is also s^*).

To prove the second bullet, consider an arbitrary s^*-to-t path π in G. We argue that every node u on π is in $T_{s^*} \ominus \text{GU}(\Sigma)$. The second bullet will then follow, because $G[T_{s^*} \ominus \text{GU}(\Sigma)]$ is a vertex-induced subgraph of G. By the path-descendants property of Lemma 6, we have that $u \in T_{s^*}$. If $u \notin T_{s^*} \ominus \text{GU}(\Sigma)$, then there must be some node $s \in \text{GU}(\Sigma)$ satisfying $s \neq s^*$, $s \in T_{s^*}$, and $u \in T_s$. But this contradicts Condition C2 (Section 3) in the definition of star. □

Example. Consider Figure 2 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 CLASSICAL POMS

This section aims to establish the formal results mentioned in Section 1.4 for classical POMS. Specifically, we will introduce a new POMS algorithm in Section 4.1 and analyze its probing cost in Section 4.2. In Section 4.3, we will present a matching lower bound on the probing complexity, which will complete the proofs of Theorem 1 and Corollary 2. Finally, Section 4.4 will prove Theorem 3. Throughout our discussion, we will assume that $k \geq 2$; for cases with $k = 1$, one can manually increase k to 2 (i.e., simulate an oracle for $k = 2$ using the provided oracle of $k = 1$) and then apply our techniques.

4.1 A POMS Algorithm

Our POMS algorithm works by shrinking the input graph $\mathcal{G}$ into smaller single-rooted DAGs $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_h$ (for some $h \geq 1$) where the last DAG $\mathcal{G}_h$ becomes small enough to be solvable with a single probe.

Define $\mathcal{G}_0 = \mathcal{G}$. The algorithm runs in iterations. The ith iteration takes as the input a single-rooted DAG $\mathcal{G}_{i-1}$ with n_{i-1} vertices and produces a single-rooted DAG $\mathcal{G}_i$ having four properties:

— (**subgraph**) $\mathcal{G}_i$ is a subgraph of $\mathcal{G}_{i-1}$;
— (**size reduction**) $\mathcal{G}_i$ has $n_i \leq n_{i-1}/k$ 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 $\mathcal{G}_i$.

These properties ensure:

Lemma 17. *For each vertex u in $\mathcal{G}_i$, it holds that u can $\mathcal{G}_i$-reach the target t if and only if u can $\mathcal{G}$-reach t.*

Proof. As $\mathcal{G}_i$ is a subgraph of $\mathcal{G}$, the "only-if direction" trivially holds. We will focus on the "if direction." In fact, we will prove a stronger claim: *every u-to-t path in the original graph $\mathcal{G}$ is preserved in $\mathcal{G}_i$*. As $\mathcal{G}_0 = \mathcal{G}$, the claim is obvious for $i = 0$. Next, assuming the claim's correctness on $i = j \geq 0$, we will prove the correctness for $i = j + 1$.

Consider an arbitrary u-to-t path π in the original graph $\mathcal{G}$. As $\mathcal{G}_{j+1}$ is single-rooted, the root r of $\mathcal{G}_{j+1}$ can $\mathcal{G}_{j+1}$-reach u. Identify an arbitrary r-to-u path π' in $\mathcal{G}_{j+1}$. By concatenating π' and π, we obtain a path π'' from r to t in $\mathcal{G}$. By the inductive assumption, π is preserved in $\mathcal{G}_j$. The path-preserving property assures us that π'' must also exist in $\mathcal{G}_{j+1}$. This means that π is preserved in $\mathcal{G}_{j+1}$, as claimed. □

Owing to the above guarantee, we can pretend as if $\mathcal{G}_i$ comes with a "dedicated oracle" responsible for reachability probes on $\mathcal{G}_i$. Specifically, when asked if a node u in $\mathcal{G}_i$ can reach t, the $\mathcal{G}_i$-oracle simply passes u and t to the original oracle and then relays the oracle's answer back to the algorithm.

Algorithm. Consider iteration $i \geq 1$. If $\mathcal{G}_{i-1}$ has $n_{i-1} \leq k$ vertices, then 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 $|\Sigma| \leq k$ (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 $\mathcal{G}_{i-1}$, which can definitely $\mathcal{G}_{i-1}$-reach t). We can then identify the star s^* of Σ for t (Section 3.3). By Lemma 10, $|\text{LF}(s^*)| \leq |\text{LFU}(\Sigma)| \leq k - 1$. Thus, with another probe, we can figure out which nodes in $\text{LF}(s^*) \cup \{s^*\}$ can $\mathcal{G}_{i-1}$-reach t. Define s^{**} to be the smallest (under $\prec$) among those nodes.

Phase 2. It must hold that either $s^{**} \notin \Sigma$ or $s^{**} = s^*$. Indeed, if $s^{**} \in \Sigma$ but $s^{**} \neq s^*$, then $s^{**} \prec s^*$ and s^* cannot be the *smallest* node satisfying Conditions C1 and C2 (Section 3.3).

If $s^{**} \notin \Sigma$, then we finalize $\mathcal{G}_i$ to be $\mathcal{G}_{i-1}[T_{s^{**}} \ominus \Sigma]$. Now consider $s^{**} = s^*$. 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, then we declare $t = s^*$ and finish the whole algorithm. Otherwise, we set $\mathcal{G}_i$ to $\mathcal{G}_{i-1}[T_{s^\#} \ominus \Sigma]$. Note that if the algorithm does not finish in this iteration, then the graph $\mathcal{G}_i$ generated contains no vertices in Σ.

Correctness. The lemma below ascertains our algorithm's correctness.

Lemma 18. *If the algorithm finishes in iteration $i \geq 1$, then it correctly finds $t = s^*$. Otherwise, $\mathcal{G}_i$ has the subgraph, size-reduction, target-containment, and path-preserving properties.*

Proof. The algorithm terminates in the ith iteration only when $s^{**} = s^*$. In this case, the first bullet of Lemma 15 indicates $t \in T_{s^*}$. Hence, if none of the child nodes of s^* can $\mathcal{G}_{i-1}$-reach t (this means none of them can $\mathcal{G}$-reach t; see Lemma 17), then t must be s^*.

Next, we consider that the algorithm does not terminate in the iteration. It is obvious that $\mathcal{G}_i$ is a subgraph of $\mathcal{G}_{i-1}$. Furthermore, $\mathcal{G}_i$ includes no vertices from Σ and, thus, can have at most n_{i-1}/k nodes (Lemma 7). Next, we prove the claim that $\mathcal{G}_i$ contains t and is path-preserving. If $s^{**} \notin \Sigma$, then the claim follows from Lemma 15. Otherwise, we must have $s^{**} = s^*$, and the first bullet of Lemma 15 assures us $t \in T_{s^*}$. As the existence of $s^{\#}$ indicates $t \neq s^*$, the claim now follows from Lemma 14. □

4.2 Cost Analysis

In this subsection, we will show that our algorithm does $O(\log_{1+k} n + \frac{d}{k} \log_{1+d} n)$ probes, as claimed in the first bullet of Theorem 1. Given $\mathcal{G}_{i-1}$ (for $i \geq 1$), the ith iteration of our algorithm either finds t or outputs $\mathcal{G}_i$. Suppose that the algorithm finds t at iteration h for some $h \geq 1$.

Analysis of One Iteration. Consider the ith iteration where $i \in [1, h-1]$. Let T, $\prec$, s^*, s^{**}, and $s^{\#}$ be defined as in Section 4.1. Set n_{i-1} (respectively, n_i) to the number of vertices in $\mathcal{G}_{i-1}$ (respectively, $\mathcal{G}_i$). Define an integer x_i as follows:

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

The ith 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 19. *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 Lemma 18. Next, we will prove $n_i \leq n_{i-1}/(x_i+1)$. This is obviously true if $x_i = 0$. Consider now $x_i > 0$. In this case, $s^{\#}$ has x_i left siblings v satisfying $|T_v| \geq |T_{s^{\#}}|$ (subtree-size property of Lemma 6). 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 Equation (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 Equation (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, Equation (9) is bounded by $O(\log_k n)$. Assuming $d \geq k + 1$, the rest of the proof will show

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

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

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

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

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

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

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

Lemma 20. *When $\sum_{i=1}^{h-1} x_i$ is maximized, at most one of $x_1, \ldots, x_{h-1}$ 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 $x_{i_1} \ge x_{i_2}$. 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 $c > dk$, then 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 $k-1 \le x_{i_2} \le x_{i_1} \le d-1$ and $(x_{i_1}+1)(x_{i_2}+1) = c$. This contradicts the claim that the original $x_1, \ldots, x_{h-1}$ maximize $\sum_{i=1}^{h-1} x_i$. □

Consider a set of $x_1, \ldots, x_{h-1}$ that maximizes $\sum_{i=1}^{h-1} x_i$. Let y_1 (or y_2) be the number of variables among $x_1, \ldots, x_{h-1}$ that are set to $k-1$ (or $d-1$, respectively). Because of Equation (12), we have $y_2 = O(\log_d n)$; however, trivially, $y_1 \le h-1$. Hence:

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

4.3 A Lower Bound

Consider $\mathcal{G}$ as a tree with n vertices. We will show that

$$maxcost_k^{\text{inst}}(\mathcal{A}, \mathcal{G}) \quad = \quad \Omega(\log_{1+k} n) \tag{13}$$

holds for any POMS algorithm $\mathcal{A}$, where $maxcost_k^{\text{inst}}(\mathcal{A}, \mathcal{G})$ is defined in Section 1.3. This lower bound applies to an arbitrary tree $\mathcal{G}$ with n vertices.

Consider a probe with a set Q of $k \ge 1$ vertices $q_1, q_2, \ldots, q_k$. The oracle returns an *outcome sequence* $a_1, a_2, \ldots, a_k$, 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.

Lemma 21. *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 $\mathcal{G}$.*

Proof. 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 $z \geq 1$), next, we prove its correctness for $k = z + 1$. As before, let the query sequence Q be $q_1, q_2, \ldots, q_{z+1}$. 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 $a_1(t), a_2(t), \ldots, a_{z+1}(t)$ 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 $a_1(t), a_2(t), \ldots, a_{z+1}(t)$ 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 $a_1(t), a_2(t), \ldots, a_{z+1}(t)$. □

We now prove Equation (13) with an information theoretic argument. By Lemma 21, 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.

As mentioned in Section 1.3, it has been proved in Reference [39] that $minmaxcost_k(\mathcal{T}(n, d)) = \Omega(\frac{d}{k} \log_{1+d} n)$, where $minmaxcost(.)$ is defined in Equation (1), and $\mathcal{T}(n, d)$ is defined in Equation (3). As the above argument holds for any tree with n vertices, we can now conclude that $minmaxcost_k(\mathcal{T}(n, d)) = \Omega(\log_{1+k} n + \frac{d}{k} \log_{1+d} n)$. This proves the second bullet of Theorem 1 and also Corollary 2.

4.4 Discussion on Instance-oriented POMS

The above discussion focused on *class-oriented* POMS. This subsection will explain an implication of our results on *instance-oriented* POMS (see Section 1.3) when the input graphs are trees.

Let $\mathcal{G}$ be an arbitrary tree with n vertices and maximum out-degree d. We will show that

$$maxcost_k^{\text{inst}}(\mathcal{A}, \mathcal{G}) \quad = \quad \Omega(\log_{1+k} n + d/k) \tag{14}$$

holds for any POMS algorithm $\mathcal{A}$. In Section 4.3, we have already proved $maxcost_k^{\text{inst}}(\mathcal{A}, \mathcal{G}) = \Omega(\log_{1+k} n)$; see Equation (13). It remains to show that $maxcost_k^{\text{inst}}(\mathcal{A}, \mathcal{G}) = \Omega(1 + d/k)$. To do so, identify an arbitrary node u in $\mathcal{G}$ with d child nodes $v_1, v_2, \ldots, v_d$. Define $S = \{v_1, \ldots, v_d\}$. When asked if a node $q \in \mathcal{G}$ can reach t, the oracle acts in the following manner until $|S| = 1$: (i) if $q \in S$, then return "no" and then remove q from S; (ii) if $q \notin S$ and q can reach u, then 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. This establishes the lower bound in Equation (14).

Our algorithm $\mathcal{A}$ in Theorem 1 ensures $maxcost_k^{\text{inst}}(\mathcal{A}, \mathcal{G}) = O(\log_{1+k} n + \frac{d}{k} \log_{1+d} n)$, which is greater than the right-hand side of Equation (14) by a multiplicative 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 $d \leq \frac{k \log_2 n}{\log_2(1+k)}$, then 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 $O(\frac{\log n}{\log(1+k)+\log\log n})$. This completes the proof of Theorem 3.

5 TACITURN POMS

This section is dedicated to establishing the formal results outlined in Section 1.4 for taciturn POMS. Specifically, an algorithm for solving taciturn POMS will be introduced in Section 4.1, followed by an analysis of its probing cost in Section 4.2. In Section 4.3, we will present a matching lower bound on the probing complexity, which will complete the proof of Theorem 4.

5.1 An Algorithm

We will simulate our classical-POMS algorithm in Section 4.1—referred to as POMS(k) henceforth—under a taciturn oracle.

Recall that, given $\mathcal{G}_0 = \mathcal{G}$, POMS($k$) iteratively produces DAGs $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_h$ (for some $h \geq 1$) such that the number of vertices in $\mathcal{G}_i$ is at most $1/k$ of that in $\mathcal{G}_{i-1}$, for $i \in [1, h]$. To solve the taciturn POMS problem, we will simulate POMS(2), but using a taciturn oracle with parameter k. To avoid confusion, we will use k_{old} for the value of k in classical POMS (i.e., k_{old} will be fixed to 2), while reserving k for taciturn POMS.

Next, we will explain how to implement the ith ($1 \leq i \leq h$) iteration of POMS(2), which takes as the input $\mathcal{G}_{i-1}$ and outputs $\mathcal{G}_i$. If $\mathcal{G}_{i-1}$ has at most $k_{old} = 2$ vertices, then we use two taciturn probes to find out which vertex is t. Otherwise, the iteration performs two phases. The following discussion will clarify how each phase is implemented on a taciturn oracle:

Phase 1. POMS(2) starts by computing an HPDFS T of $\mathcal{G}_{i-1}$; let $\prec$ be the total order that T defines on the vertices of $\mathcal{G}_{i-1}$. After finding the 2-separator Σ of T, POMS(2) looks for (i) s^*, the star of Σ for t and (ii) s^{**}, the smallest node in $\mathrm{LF}(s^*) \cup \{s^*\}$ able to $\mathcal{G}_{i-1}$-reach t.

As $|\Sigma| \leq k_{old} = 2$, we can use $O(1)$ taciturn probes to decide the vertices of Σ that can $\mathcal{G}_{i-1}$-reach t and, thereby, identify the star s^* of Σ for t. Similarly, as $\mathrm{LF}(s^*) \leq |\mathrm{LFU}(\Sigma)| \leq k_{old} - 1 = 1$ (Lemma 10), using another $O(1)$ taciturn probes, we can identify s^{**}.

Phase 2. As explained in Section 4.1, either $s^{**} \notin \Sigma$ or $s^{**} = s^*$. In the former case, the iteration finishes by generating $\mathcal{G}_i = \mathcal{G}_{i-1}[T_{s^{**}} \ominus \Sigma]$; no more probes are necessary. In the latter case, POMS(2) finds the smallest (under $\prec$) child $s^\#$ of s^* (in T) that can $\mathcal{G}_{i-1}$-reach t. If $s^\#$ does not exist, then POMS(2) returns $t = s^*$; otherwise, it generates $\mathcal{G}_i = \mathcal{G}_{i-1}[T_{s^\#} \ominus \Sigma]$.

Next, focusing on the case $s^{**} = s^*$, we will explain how to use taciturn probes to find $s^\#$ or declare its non-existence. Suppose that s^* has y child nodes in T, which are $u_1, u_2, \ldots, u_y$ in ascending order of $\prec$. Divide the ordered child list into $\lceil y/k \rceil$ groups, such that group 1 contains the smallest k child nodes, group 2 the next smallest k, and so on. Each group has exactly k nodes except possibly the last group. Use taciturn probes to identify the smallest j such that group j contains at least one node that can $\mathcal{G}_{i-1}$-reach t. If j does not exist, then we conclude that there is no $s^\#$.

Now consider that j is has been obtained. Thus, $s^\#$ must be the smallest node in group j that can $\mathcal{G}_{i-1}$-reach t. We can find $s^\#$ with binary search as follows: Let S be the set of nodes in group j and S_1 be the set of the $\lfloor |S|/2 \rfloor$ smallest nodes in S. Issue a taciturn probe with S_1. If the probe returns yes, then we recursively look for $s^\#$ in S_1; otherwise, we do so in $S \setminus S_1$. The process continues until only one node is left. The binary search costs $1 + \lceil \log_2 k \rceil$ probes.

In summary, if $s^\#$ does not exist, then Phase 2 issues $\lceil d/k \rceil$ probes and terminates the whole algorithm. Otherwise, the number of probes is $\lceil \frac{1+x}{k} \rceil + 1 + \lceil \log_2 k \rceil$, where x is the number of left siblings of $s^\#$.

5.2 Cost Analysis

Suppose that our algorithm in the previous subsection finds t in the hth iteration. As analyzed in Section 4.2, the value of h is $O(\log_{k_{old}} n)$, which is $O(\log n)$.

For each $i \in [1, h-1]$, define x_i as in Section 4.2. Specifically, if $s^{**} \neq s^*$, then $x_i = 0$; otherwise, x_i is the number of left siblings of $s^{\#}$. As discussed in Section 5.1, the ith iteration performs

$$O\left(1 + \frac{x_i}{k} + \log k\right)$$

probes. Therefore, the total probing cost is at the order of

$$\sum_{i=1}^{h}\left(1 + \frac{x_i}{k} + \log k\right) \quad = \quad O(\log n \cdot \log(1+k)) + \frac{1}{k}\sum_{i=1}^{h} x_i. \tag{15}$$

The analysis of $\sum_{i=1}^{h} x_i$ is identical to that of Section 4.2. From Equation (10), we get

$$\sum_{i=1}^{h} x_i = O(d \log_d n + k_{old} \log_{k_{old}} n) = O(d \log_d n + \log n).$$

Plugging the above into Equation (15) shows that our total probing cost is $O(\log n \cdot \log(1+k) + \frac{d}{k}\log_{1+d} n)$, as claimed in the first bullet of Theorem 4.

5.3 A Lower Bound

In this section, we will prove the lower bound in the second bullet of Theorem 4. As mentioned in Section 4.3, it has been proved in Reference [39] that, for classical POMS, any algorithm must perform $\Omega(\frac{d}{k}\log_{1+d} n)$ probes in the worst case. The same lower bound must also apply to taciturn POMS because one can utilize the oracle in classical POMS to simulate a taciturn oracle (return no for taciturn POMS if and only if the traditional oracle replies no to every vertex in the probe).

Next, we establish another lower bound of $\Omega(\log n)$ on the probing complexity of taciturn POMS. Consider $\mathcal{G}$ to be a chain of n vertices (i.e., a rooted tree where every non-leaf vertex has an out-degree 1). We observe that if we perform a taciturn probe with a set Q of vertices, then the probe's outcome is uniquely determined by only one vertex in Q: the one "highest" in the chain (that can reach every other vertex in Q). Thus, we can as well limit the size of Q to 1, in which case a lower bound of $\lceil \log_2 n \rceil$ becomes obvious (this is the same as classical POMS with $k = 1$, and thus the discussion in Section 4.3 applies).

Finally, we can obtain a lower bound $\Omega(\log n + \frac{d}{k}\log_{1+d} n)$ by merging our argument with that of Reference [39] as follows: The hard input in Reference [39] (for proving $\Omega(\frac{d}{k}\log_{1+d} n)$) was a perfect d-ary ($d \geq 2$) tree with n vertices; denote such a tree as $T_1(d, n)$. However, let $T_2(n)$ represent a chain with n vertices. Now, given valid parameters d and n for $T_1(d, n)$, we construct a tree $T_3(d, n)$ as follows:

— The root r of $T_3(d, n)$ has two child nodes.
— The left subtree of r is $T_1(d, n)$, while its right subtree is $T_2(n)$.

It is clear that $T_3(d, n)$ has $2n + 1$ nodes and a maximum vertex out-degree of d. We argue that any taciturn POMS algorithm $\mathcal{A}$ must perform $\Omega(\log n + \frac{d}{k}\log_{1+d} n)$ probes to find the target vertex t in the worst case. Indeed, if $\log_2 n \leq \frac{d}{k}\log_{1+d} n$, then $\mathcal{A}$ needs $\Omega(\frac{d}{k}\log_{1+d} n)$ probes to guarantee finding t in $T_1(d, n)$; otherwise, $\mathcal{A}$ needs $\Omega(\log n)$ probes to guarantee finding t in $T_2(n)$.

6 EM POMS

This section aims to establish the formal results outlined in Section 1.4 for EM POMS. To achieve this, we will first design another algorithm for classical POMS in Section 6.1, which shares the same performance guarantees as Theorem 1. However, this new algorithm possesses additional properties not found in our algorithm from Section 4.1. These properties are crucial for designing a

structure for EM POMS, as demonstrated in Section 6.2. In Section 6.3, we will present a matching lower bound on the probing complexity, completing the proof of Theorem 4. To showcase the power of our techniques, Section 6.4 will utilize the theorem to develop a new optimal structure for the vertical ray shooting problem.

6.1 Another Algorithm for Classical POMS

In this subsection, we revisit classical POMS and present another algorithm to achieve the probing complexity in Theorem 1.

Algorithm. Our new POMS algorithm follows the same iterative framework in Section 4. The ith iteration ($i \geq 1$) 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 4.1.

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 (Section 3.1). As $|\text{GU}(\Sigma)| < 2k$ (Section 3.2), with at most two probes, we can find the nodes in $\text{GU}(\Sigma)$ capable of $\mathcal{G}_{i-1}$-reaching t and, hence, the star s^* of $\text{GU}(\Sigma)$ for t.

Phase 2. If $s^* \notin \Sigma$, then the iteration outputs $\mathcal{G}_i = \mathcal{G}_{i-1}[T_{s^*} \ominus \text{GU}(\Sigma)]$. Otherwise, we find the smallest (under $\prec$) child $s^\#$ of s^* in T that can $\mathcal{G}_{i-1}$-reach t. If $s^\#$ exists, then the iteration outputs $\mathcal{G}_i = \mathcal{G}_{i-1}[T_{s^\#} \ominus \text{GU}(\Sigma)]$; if $s^\#$ does not exist, then the algorithm finishes with $t = s^*$.

Correctness and Cost. By resorting to Lemmas 14 and 16 and adapting the cost analysis of our first POMS algorithm, we prove in Appendix B:

Lemma 22. *The above algorithm is correct and achieves the same guarantees as in Theorem 1.*

New Properties. Let us concentrate on an arbitrary iteration—say, the ith (with $i \geq 1$)—of the algorithm. Denote by $\mathcal{G}_{i-1}$ the iteration's input, by T an HPDFS-tree of $\mathcal{G}_{i-1}$, and by Σ the k-separator of T. 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). Define

$$\text{OUT}(\mathcal{G}_{i-1}, t) = \begin{cases} \text{an empty DAG} & \text{if the iteration finds } t \\ \mathcal{G}_i & \text{otherwise.} \end{cases}$$

Define further:

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

Lemma 23. *No two DAGs in* $\text{OUT}(\mathcal{G}_{i-1})$ *share any common vertex.*

Proof. Consider any two different non-empty DAGs G_1 and G_2 in $\text{OUT}(\mathcal{G}_{i-1})$. Denote by r_1 (respectively, r_2) the root of G_1 (respectively, G_2). In other words, $G_1 = \mathcal{G}_{i-1}[T_{r_1} \ominus \text{GU}(\Sigma)]$ and $G_2 = \mathcal{G}_{i-1}[T_{r_2} \ominus \text{GU}(\Sigma)]$, which implies $r_1 \neq r_2$. Next, we show that $T_{r_1} \ominus \text{GU}(\Sigma)$ and $T_{r_2} \ominus \text{GU}(\Sigma)$ do not share any common vertex. Since these graphs are trees, the claim is true 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) $r_2 \in \text{GU}(\Sigma) \setminus \Sigma$ or
- (Case 2) $\mathit{parent}(r_2) \in \Sigma$.

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

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

Consider now Case 2. In general, the $\mathcal{G}_i$ produced by iteration i cannot be rooted at a vertex in Σ. To understand why, first note that $\mathcal{G}_i$ 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 $s^\# \in \text{GU}(\Sigma)$, which contradicts that s^* satisfies condition C2 (Section 3.3).

As the root $\mathcal{G}_i$ cannot be in Σ, we know $r_1 \notin \Sigma$, which means that $parent(r_2)$ is a proper descendant of r_1 in T. Because $parent(r_2) \in \Sigma \subseteq \text{GU}(\Sigma)$, we have that $T_{r_1} \ominus \text{GU}(\Sigma)$ is contained in $T_{r_1} \ominus \{parent(r_2)\}$. Furthermore, $T_{r_1} \ominus \{parent(r_2)\}$ shares no vertices with $T_{parent(r_2)}$, whereas $T_{parent(r_2)}$ contains $T_{r_2} \ominus \text{GU}(\Sigma)$. Therefore, we can conclude that no common vertex can exist in $T_{r_1} \ominus \text{GU}(\Sigma)$ and $T_{r_2} \ominus \text{GU}(\Sigma)$. □

Define the *children set of* Σ as

$$C \quad = \quad \{\text{node } u \in T \mid u \text{ is a child of some node in } \Sigma\}.$$

We can bound the size of C as follows:

Lemma 24. $|C| \leq |\Sigma| + |\text{OUT}(\mathcal{G}_{i-1})|$.

Proof. We will prove that, for each node $u \in C \setminus \Sigma$, the set $\text{OUT}(\mathcal{G}_{i-1})$ contains at least one DAG rooted at u. Because each graph in $\text{OUT}(\mathcal{G}_{i-1})$ is single-rooted, we have $|C \setminus \Sigma| \leq |\text{OUT}(\mathcal{G}_{i-1})|$, which yields $|C| \leq |\Sigma| + |\text{OUT}(\mathcal{G}_{i-1})|$.

We will prove a more specific claim: *When the target t equals u,* $\text{OUT}(\mathcal{G}_{i-1}, t)$ *must be a DAG rooted at u.* Let us start with the scenario where $t = u \in \text{LFU}(\Sigma)$. By Lemma 12, u is the star of $\text{GU}(\Sigma)$ for u itself. Hence, Phase 1 of the algorithm returns $s^* = u$. As $s^* = u \notin \Sigma$, Phase 2 generates $\mathcal{G}_i = \mathcal{G}_{i-1}[T_u \ominus \text{GU}(\Sigma)]$, which is rooted at u. Consider now the scenario where $t = u \notin \text{LFU}(\Sigma)$. This, together with the fact $u \in C \setminus \Sigma$, indicates $u \in C \setminus \text{GU}(\Sigma)$. By Lemma 13, Phase 1 returns $s^* = parent(u)$, which is in Σ (by definition of C). Phase 2 sets $s^\# = u$ (no left sibling of u can $\mathcal{G}_{i-1}$-reach u, by the no-cross-reachability property of Lemma 6) and outputs $\mathcal{G}_i = \mathcal{G}_{i-1}[T_u \ominus \text{GU}(\Sigma)]$, which is rooted at u. □

6.2 An EM Structure

To find the target t, our EM structure deploys the algorithm $\mathcal{A}$ of Lemma 22 by setting its parameter k to the block size B. Specifically, 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 $O(n/B)$. For that purpose, we will create the structure recursively and leverage Lemmas 23 and 24 to obtain a non-conventional recurrence on the space consumption, which will solve to $O(n/B)$.

As mentioned, the output of the ith ($i \geq 1$) iteration of $\mathcal{A}$ depends on $\mathcal{G}_{i-1}$ and t. More specifically, conditioned on each $\mathcal{G}_{i-1}$, the ith iteration can have $|\text{OUT}(\mathcal{G}_{i-1})|$ different outputs, as discussed in Section 6.1. We will create a structure $\text{POMS}(\mathcal{G}_{i-1}, i)$, which gives $\mathcal{A}$ all the information needed to execute the iteration on $\mathcal{G}_{i-1}$. The iteration either terminates or outputs a DAG $\mathcal{G}_i \in \text{OUT}(\mathcal{G}_{i-1})$ for the $(i+1)$-th iteration. This $\mathcal{G}_i$ will then be recursively handled by a structure $\text{POMS}(\mathcal{G}_i, i+1)$. The entry point to the whole recursion is $\text{POMS}(\mathcal{G}, 1) = \text{POMS}(\mathcal{G}_0, 1)$. Next, we explain the details of $\text{POMS}(\mathcal{G}_{i-1}, i)$.

Structure POMS($\mathcal{G}_{i-1}$, i): When $\mathcal{G}_{i-1}$ Is Large. Let us start from the scenario where $\mathcal{G}_{i-1}$ 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 $\text{GU}(\Sigma)$ and encode their ancestor-descendant relationships in T (it is well known that the ancestor-descendant relationships of x nodes in a tree can be encoded in $O(x)$ words). 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 $u \in \text{GU}(\Sigma)$, whether u can $\mathcal{G}_{i-1}$-reach t. From that information, $\mathcal{A}$ obtains the star s^* of $\text{GU}(\Sigma)$ for t.

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

Consider now an arbitrary node $u \in \Sigma$. If $s^* = u$, then 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, then the algorithm terminates with $t = u$. Otherwise, it should operate on $\mathcal{G}_i = \mathcal{G}_{i-1}[T_{s^\#} \ominus \text{GU}(\Sigma)]$ in the next iteration. We build $\text{POMS}(\mathcal{G}_i, i+1)$ recursively and store a pointer to $\text{POMS}(\mathcal{G}_i, i+1)$ at $s^\#$ inside the children blocks. This completes the description of $\text{POMS}(\mathcal{G}_{i-1}, i)$.

Structure POMS($\mathcal{G}_{i-1}$, i): When $\mathcal{G}_{i-1}$ Is Small. If $\mathcal{G}_{i-1}$ has less than B vertices, then $\mathcal{A}$ 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 overcome 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 the oracle, and then identify the star s^* of the set of vertices in T for t. The no-cross-reachability property of Lemma 6 ensures $s^* = t$.

I/O cost. The algorithm performs $O(1)$ I/Os for every probe issued by the POMS algorithm of Lemma 22. The overall I/O cost is thus $O(\log_B n + (d/B)\log_{1+d} n)$.

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. 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 $n \leq B$. Next, we discuss the scenario $n > B$.

Let us focus on the structure $\text{POMS}(\mathcal{G}, 1)$, i.e., the entry structure of the whole recursion. The number of words in the grand-union blocks is $O(|\text{GU}(\Sigma)|) = O(|\Sigma|)$ (Lemma 3). Let C be the children set (Section 6.1) of the B-separator Σ used in $\text{POMS}(\mathcal{G}, 1)$. 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 $\mathcal{G}_1 \in \text{OUT}(\mathcal{G}, 1)$. Denoting by $|\mathcal{G}_1|$ the number of vertices in $\mathcal{G}_1$, we have:

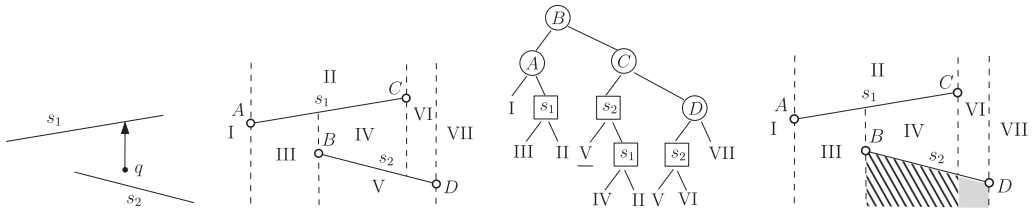
$$\begin{aligned} f(n) &= O(1 + |\Sigma| + |C|) + \sum_{\mathcal{G}_1 \in \text{OUT}(\mathcal{G},1)} f(|\mathcal{G}_1|) \\ &= O(1 + |\Sigma| + |\text{OUT}(\mathcal{G}, 1)|) + \sum_{\mathcal{G}_1 \in \text{OUT}(\mathcal{G},1)} f(|\mathcal{G}_1|), \end{aligned} \tag{16}$$

where the last equality applied Lemma 24. The recurrence is constrained by

$$|\Sigma| + \sum_{\mathcal{G}_1 \in \text{OUT}(\mathcal{G},1)} |\mathcal{G}_1| \leq n,$$

because (i) Σ has no vertices in any $\mathcal{G}_1 \in \text{OUT}(\mathcal{G}, 1)$ and (ii) no two DAGs in $\text{OUT}(\mathcal{G}, 1)$ share any common vertices (Lemma 23). Appendix C shows that the recurrence (16) gives $f(n) = O(n)$.

This concludes the proof for the first bullet of Theorem 5.



(a) A VRS query (b) Trapezoidal map (c) A VRS structure (d) Region illustration

Fig. 5. A region-based structure on the VRS problem.

6.3 A Lower Bound

This subsection will prove the second bullet in Theorem 5. Fix a value of n and consider any DAG $\mathcal{G}$ with n vertices. We claim that, in general, given an EM-POMS structure $\mathcal{I}$ that can find any target t in $F(B)$ I/Os, we can obtain a POMS algorithm $\mathcal{A}$ that finds t with at most $F(k)$ probes. It will then follow from the second bullet of Theorem 1 that $F(B) = \Omega(\log_B n + (d/B)\log_{1+d} n)$.

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).

6.4 Application: A New EM Structure for Vertical Ray Shooting

In Section 1.2, we introduced the class of *region-based* data structures in the RAM model and presented a generic black-box reduction that converts any such structure into an I/O-efficient counterpart. Numerous well-known indexes are region-based. The binary search tree serves as one example, where a node's region is an interval of the form $[x, y)$, with x and y being real values. The quad-tree and the kd-tree are further examples, where a node's region is a multidimensional rectangle. Next, we will discuss another, more sophisticated, region-based structure that better illustrates the strength of our black-box reduction.

The ***vertical ray shooting* (VRS)** problem is 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 5(a) shows an example where the query answer is s_1. The objective is to store S in a structure to answer all queries efficiently. This is a fundamental problem with profound significance to database systems; see Reference [26] for its relevance to point location queries and nearest neighbor search and Reference [8] 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 5(b) shows the trapezoidal map for the input in Figure 5(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 5(b)).

In Reference [35], 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 5(c) shows a binary tree for our example. Consider the point

q in Figure 5(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, then the region of its left (respectively, right) child includes all the points in reg_u whose x-coordinates are smaller (respectively, larger) than that of p; (ii) if an internal node u stores a segment s, then the region of its left (right, respectively) child includes all the points in reg_u below (respectively, above) of s. In Figure 5(d), we have divided trapezoid V into two parts such that the left (respectively, right) part is the region of the leaf labeled as $\underline{\text{V}}$ (respectively, V) in Figure 5(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 can have $\Theta(n^2)$ nodes where $n = |S|$. In Reference [38], 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 5 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). The algorithm of Reference [38] may yield a binary tree with a large height (even when the tree has size $O(n)$). Seidel [38] gave a non-trivial analysis on how likely the height is small. In contrast, we can take an any 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 References [9, 26, 36] for a full literature review) achieving $O(n/B)$ space and $O(\log_B n)$ query cost were obtained using the *partial persistence* [4, 26] and the *topology tree* [9, 25] techniques. Our method is drastically different and conceptually neater.

7 EXPERIMENTS

This section presents an experimental evaluation of the proposed POMS algorithms. We will concentrate on traditional and taciturn POMS (our contribution to EM POMS, a generic transformation for converting an in-memory structure to an external memory counterpart, is theoretical in nature). The objectives of our empirical study are two-fold. First, we want to understand how competitive our POMS algorithm is with regard to the state-of-the-art [39] in practical performance. Second, we want to understand how many more probes one should expect from our taciturn algorithm compared to classical POMS. After all, one motivation for taciturn POMS is to increase interaction rounds (a.k.a. probes) in exchange for simplified human inputs.

Data. We deployed precisely the same datasets used in the experiments of Reference [39].

- `Amazon`: The input $\mathcal{G}$ is a tree with 29,240 vertices representing Amazon's product hierarchy.
- `ImageNet`: The input $\mathcal{G}$ is a (non-tree) DAG with 27,714 vertices representing an annotation ontology over a collection of images.

We refer the reader to Reference [39] for additional details of the two datasets, e.g., the semantics of the vertices and edges in $\mathcal{G}$, where the data can be downloaded, and how they were post-processed from the original raw versions. Table 2 shows the out-degree statistics for each dataset. Here, we define the *level* of a vertex u in $\mathcal{G}$ as the length (in number of edges) of a shortest path from the root of $\mathcal{G}$ to u. The average/max out-degree at a level ℓ is calculated from all the nodes in $\mathcal{G}$ at level ℓ. A *leaf* of $\mathcal{G}$ is a node with out-degree 0. The number of leaves is 24,329 and 21,427 for `Amazon` and `ImageNet`, respectively.

Table 2. Out-degree Statistics

	(a) Amazon		(b) ImageNet	
level	avg. out-degree	max out-degree	avg. out-degree	max out-degree
0	84	84	8	8
1	11	225	83	402
2	4.6	90	3.4	173
3	2.4	49	2.2	357
4	0.97	78	1.4	304
5	0.33	27	0.87	123
6	0.17	14	0.71	87
7	0.13	14	0.59	31
8	0.11	2	0.54	24
9	0	0	0.48	54
10	–	–	0.69	21
11	–	–	0.44	12
12	–	–	0	0

Competing Algorithms. Our evaluation examined three algorithms:

— `POMS-ours`: the POMS algorithm described in Section 4.1;
— `TLL19`: the state-of-the-art POMS algorithm in Reference [39];
— `Taciturn`: the taciturn algorithm described in Section 5.1.

The reader should bear in mind that, while `POMS-ours` and `TLL19` use the same oracle, `Taciturn` executes on a weaker oracle. The comparison between the first two algorithms demonstrates the efficiency of two solutions to the same problem, whereas the comparison between `Taciturn` and the rest demonstrates the effects of weakening the oracle's power.

Workloads and Metrics. Given an input graph $\mathcal{G}$, each target vertex t defines a problem instance. We considered the instances defined by all the leaves of $\mathcal{G}$ and will refer to the collection of those instances as a *workload*. In other words, for $\mathcal{G}$ = `Amazon` (respectively, `ImageNet`), a workload contains 24,329 (respectively, 21,427) instances. Given an integer $\ell \geq 1$, we use the term *level-ℓ workload* for the set of instances defined by all the level-ℓ leaves of $\mathcal{G}$. For each algorithm $\mathcal{A}$ (i.e., `POMS-ours`, `TLL19`, and `Taciturn`), we will report its

— *average cost*: how many probes $\mathcal{A}$ performs on average answering an instance in a workload;
— *max cost*: the largest cost $\mathcal{A}$ incurs answering an instance in a workload;
— *level-average cost*: given a level ℓ, how many probes $\mathcal{A}$ performs on average answering an instance in a level-ℓ workload;
— *level-max cost*: given a level ℓ, the largest cost $\mathcal{A}$ incurs answering an instance in a level-ℓ workload.

The workload design and the definitions of average cost and level-average cost are the same as in Reference [39]. Max cost and level-max cost are new.

Results. In the first experiment, we inspected the average cost of each algorithm as the parameter k grew from 1 to 10. Tables 3(a) and 3(b) present the results for `Amazon` and `ImageNet`, respectively. For both datasets, `POMS-ours` outperformed `TLL19` under all values of k. The performance gap between the two algorithms was quite significant for small k, but gradually narrowed as k increased. `Taciturn` had the same cost as `POMS-ours` for $k = 1$, because the two algorithms behave exactly

Table 3. Average Cost vs. k

	(a) Amazon			(b) ImageNet		
k	POMS-ours	TLL19	Taciturn	POMS-ours	TLL19	Taciturn
1	26	36	26	35	45	35
2	14	21	19	19	26	35
3	11	17	19	15	19	25
4	9.1	15	19	12	17	24
5	8.2	13	18	11	15	23
6	7.6	11	18	9.9	14	23
7	8.0	10	18	9.3	13	23
8	7.7	10	18	8.9	12	23
9	7.2	9.7	17	8.5	12	22
10	7.0	9.5	17	8.0	10	22

Table 4. Maximum Cost vs. k

	(a) Amazon			(b) ImageNet		
k	POMS-ours	TLL19	Taciturn	POMS-ours	TLL19	Taciturn
1	228	236	228	402	412	402
2	115	121	116	201	209	203
3	76	82	79	136	141	138
4	58	64	61	102	107	105
5	47	52	51	82	87	87
6	40	44	43	69	74	73
7	35	38	39	60	65	64
8	31	34	35	52	57	58
9	27	30	33	47	52	55
10	25	28	31	42	45	52

the same at that value of k. As expected, Taciturn required more probes than POMS-ours at higher values of k, but by a factor far less than k.

Recall that POMS-ours, TLL19, and Taciturn all have non-trivial worst-case guarantees. Thus, it makes sense to compare them by max cost. Table 4 presents each algorithm's max cost as a function of k, obtained from the experiment of Table 3. In general, the max cost of each algorithm is closely related to the maximum out-degree in the input graph $\mathcal{G}$. The main observation here is that employing a large k is highly effective in reducing the max cost. This phenomenon supports the motivation behind taciturn POMS (i.e., large k values are important in reality).

The next experiment zoomed into specific levels and compared different algorithms by their level-average cost. For that purpose, we set the parameter k to 5 (the median value in the experiments of Tables 3 and 4) and ran all algorithms using level-ℓ workloads for every possible ℓ. Tables 5(a) and 5(b) present the results as a function of ℓ for Amazon and ImageNet, respectively. In Table 5(b), the level number starts from 2, because no level-1 leaves exist in this dataset. Once again, POMS-ours consistently outperformed TLL19 in all scenarios, and the ratio between the costs of Taciturn and POMS-ours was far less than k. The reader can observe a clear correlation between the level-average costs of each algorithm and the average out-degrees shown in Table 2.

Table 5. Level-average Cost vs. Level ($k = 5$)

	(a) Amazon			(b) ImageNet		
level	POMS-ours	TLL19	Taciturn	POMS-ours	TLL19	Taciturn
1	14	18	17	–	–	–
2	17	22	21	41	46	45
3	9.4	15	16	13	18	20
4	7.9	13	17	13	18	22
5	7.8	13	19	11	16	22
6	7.7	12	21	9.2	13	23
7	8.0	12	22	9.0	12	24
8	8.2	12	23	8.8	12	25
9	8.0	10	27	9.0	12	26
10	–	–	–	9.5	13	27
11	–	–	–	9.4	14	26
12	–	–	–	9.0	13	27

Table 6. Level-max Cost vs. Level ($k = 5$)

	(a) Amazon			(b) ImageNet		
level	POMS-ours	TLL19	Taciturn	POMS-ours	TLL19	Taciturn
1	17	21	19	–	–	–
2	47	52	51	82	87	87
3	22	28	29	38	42	46
4	14	20	24	76	82	83
5	20	23	31	66	70	78
6	12	17	27	30	35	38
7	11	17	28	23	28	34
8	9.0	17	28	14	19	34
9	8.0	12	27	14	18	34
10	–	–	–	16	21	36
11	–	–	–	13	18	36
12	–	–	–	11	17	38

In Table 6, we report the level-max cost of each algorithm as a function of ℓ in the experiment of Table 5. An interesting observation here is that, for both datasets, the difference between POMS-ours and TLL19 tended to be more significant at a deeper level ℓ. We believe that this phenomenon reflects the superiority of POMS-ours in having a lower probing complexity. Such superiority manifests itself better when the target vertex is "buried" deep in the input graph.

8 CONCLUSIONS

Partial order multiway search (POMS) is a classical problem in computer science that has been extensively studied. In this article, we settle the problem by presenting new upper and lower bounds matching each other asymptotically. Central to the proposed algorithms is a suite of graph-theoretic results, which revolve around several novel concepts introduced in this work: left flank, grand union, and star. Our graph-theoretic lemmas provide new mathematic tools for reasoning about reachability in a **directed acyclic graph (DAG)** and, we believe, are of independent interest. We have also studied two non-trivial variants of classical POMS. The first one, called taciturn POMS, aims to better understand how the power of the oracle of classical POMS affects the probing

complexity. The second one, called EM POMS, is at the core of a generic reduction that can convert many in-memory data structures to their I/O-efficient counterparts in external memory. For both variants, we present non-trivial upper bounds that match or nearly match the corresponding lower bounds.

APPENDICES

A PROOF OF LEMMA 6

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

THEOREM 25 ([18]). *For any nodes u and v, it holds that $v \in T_u$ if and only if G has a white path from u to v right before u enters the stack.*

The theorem implies:

COROLLARY 26. *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 25, no white path exists from u to v when u enters the stack; thus, v enters the stack after u is popped (Corollary 26, applying the fact that $u \prec v$). Furthermore, $u' \in T_u$ indicates that u is in the stack when u' enters the stack. The two facts together indicate $u' \prec v$. However, $v' \in T_v$ means that $v \prec v'$, which leads to $u' \prec v'$.

(No-cross-reachability property) Take an arbitrary $v' \in T_v$. When u enters the stack, node v' must be white (by the order property, we have $u \prec v'$, which means that v' has never been en-stacked at the moment). 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 25, contradicting $v' \in T_v$). Hence, when v'' entered the stack, a white path existed from v'' to v' but not from v'' to u (as mentioned, u can G-reach v''; hence, a path from v'' to u implies a cycle in G, contradicting the fact that G is a DAG). By Corollary 26, we have $v' \prec u$, 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 25); 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 25, $w \in T_v$. Because $w \in T_u$ and $v \notin T_u$, 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, which contradicts the fact that G is a DAG.

(Subtree-size property) This property immediately follows from the definition of HPDFS and Theorem 25.

B PROOF OF LEMMA 22

To prove the algorithm's correctness, it suffices to show that if the algorithm does not finish at iteration i, then $\mathcal{G}_i$ has the size-reduction, target-containment, and path-preserving properties described in Section 4.1 (the subgraph property is obvious).

- **(size-reduction)** As $\mathcal{G}_i$ includes no vertices from Σ, $\mathcal{G}_i$ can have at most n_{i-1}/k nodes (Lemma 7).
- **(target-containment and path-preserving)** If $s^* \notin \Sigma$, then the two properties follow directly from Lemma 16. Now, consider $s^* \in \Sigma$. By Lemma 16, $t \in T_{s^*}$. Hence, if $s^\#$ does not exist, then $s^* = t$ and the algorithm finishes. Otherwise, by Lemma 14, $\mathcal{G}_i = \mathcal{G}_{i-1}[T_{s^\#} \ominus \mathrm{GU}(\Sigma)]$ 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 Section 4.2 in a straightforward manner.

C SOLVING THE FUNCTION $f(n)$ IN SECTION 6.2

We consider, w.l.o.g., that $f(n) \leq c_1 n$ for $n \leq B$ where c_1 is a constant. Rewrite Equation (16) into:

$$f(n) \quad \leq \quad c_2(1 + |\Sigma| + |\text{OUT}(\mathcal{G}, 1)|) + \sum_{\mathcal{G}_1 \in \text{OUT}(\mathcal{G},1)} f(|\mathcal{G}_1|) \tag{17}$$

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

Consider any $\mathcal{G}$ with z vertices. If $|\text{OUT}(\mathcal{G}, 1)| = 0$, then Equation (17) gives $f(z) \leq cz + c$, which is at most $4cz - 3c$ as long as $z \geq 2$. When $|\text{OUT}(\mathcal{G}, 1)| \geq 1$, we get from Equation (17):

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

Recall from Section 6.2 that $|\Sigma| + \sum_{\mathcal{G}_1 \in \text{OUT}(\mathcal{G},1)} |\mathcal{G}_1| \leq n = z$. Hence:

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

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

REFERENCES

[1] Shangqi Lu, Wim Martens, Matthias Niewerth, and Yufei Tao. 2022. Optimal algorithms for multiway search on partial orders. *PODS 2022*, 175–187.

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

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

[4] Lars Arge, Andrew Danner, and Sha-Mayn Teh. 2003. I/O-efficient point location using persistent B-trees. *ACM J. Experim. Algor.* 8 (2003).

[5] Esther M. Arkin, Henk Meijer, Joseph S. B. Mitchell, David Rappaport, and Steven Skiena. 1998. Decision trees for geometric models. *Int. J. Comput. Geom. Applic.* 8, 3 (1998), 343–364.

[6] Yosi Ben-Asher and Eitan Farchi. 1997. *The Cost of Searching in General Trees versus Complete Binary Trees.* Technical Report. https://researcher.watson.ibm.com/researcher/view_person_pubs.php?person=il-FARCHI&t=1

[7] Yosi Ben-Asher, Eitan Farchi, and Ilan Newman. 1999. Optimal search in trees. *SIAM J. Comput.* 28, 6 (1999), 2090–2102.

[8] Elisa Bertino, Barbara Catania, and Boris Shidlovsky. 1998. Towards optimal indexing for segment databases. In *Proceedings of the Conference on Extending Database Technology (EDBT'98).* 39–53.

[9] Paul B. Callahan, Michael T. Goodrich, and Kumar Ramaiyer. 1995. Topology B-trees and their applications. In *Proceedings of the Algorithms and Data Structures Workshop (WADS'95).* 381–392.

[10] Renato Carmo, Jair Donadelli, Yoshiharu Kohayakawa, and Eduardo Sany Laber. 2004. Searching in random partially ordered sets. *Theor. Comput. Sci.* 321, 1 (2004), 41–57.

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

[12] Venkatesan T. Chakaravarthy, Vinayaka Pandit, Sambuddha Roy, and Yogish Sabharwal. 2009. Approximating decision trees with multiway branches. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP'09)*. 210–221.
[13] Ferdinando Cicalese, Tobias Jacobs, Eduardo Sany Laber, and Marco Molinaro. 2010. On greedy algorithms for decision trees. In *Proceedings of the International Symposium on Algorithms and Computation (ISAAC'10)*. 206–217.
[14] Ferdinando Cicalese, Tobias Jacobs, Eduardo Sany Laber, and Marco Molinaro. 2011. On the complexity of searching in trees and partially ordered structures. *Theor. Comput. Sci.* 412, 50 (2011), 6879–6896.
[15] Ferdinando Cicalese, Tobias Jacobs, Eduardo Sany Laber, and Marco Molinaro. 2014. Improved approximation algorithms for the average-case tree searching problem. *Algorithmica* 68, 4 (2014), 1045–1074.
[16] Ferdinando Cicalese, Tobias Jacobs, Eduardo Sany Laber, and Caio Dias Valentim. 2012. The binary identification problem for weighted trees. *Theor. Comput. Sci.* 459 (2012), 100–112.
[17] Ferdinando Cicalese, Balázs Keszegh, Bernard Lidický, Dömötör Pálvölgyi, and Tomás Valla. 2016. On the tree search problem with non-uniform costs. *Theor. Comput. Sci.* 647 (2016), 22–32.
[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms, Second Edition*. The MIT Press.
[19] Pilar de la Torre, Raymond Greenlaw, and Alejandro A. Schäffer. 1995. Optimal edge ranking of trees in polynomial time. *Algorithmica* 13, 6 (1995), 592–618.
[20] Dariusz Dereniowski. 2006. Edge ranking of weighted trees. *Discr. Appl. Math.* 154, 8 (2006), 1198–1209.
[21] Dariusz Dereniowski. 2008. Edge ranking and searching in partial orders. *Discr. Appl. Math.* 156, 13 (2008), 2493–2500.
[22] Dariusz Dereniowski and Marek Kubale. 2006. Efficient parallel query processing by graph ranking. *Fundam. Inform.* 69, 3 (2006), 273–285.
[23] Dariusz Dereniowski, Stefan Tiegel, Przemyslaw Uznanski, and Daniel Wolleb-Graf. 2019. A framework for searching in graphs in the presence of errors. In *Proceedings of the Symposium on Simplicity in Algorithms (SOSA)*, 4:1–4:17.
[24] Ehsan Emamjomeh-Zadeh, David Kempe, and Vikrant Singhal. 2016. Deterministic and probabilistic binary search in graphs. In *Proceedings of the ACM Symposium on Theory of Computing (STOC'16)*. 519–532.
[25] Greg N. Frederickson. 1997. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput.* 26, 2 (1997), 484–538.
[26] Xiaocheng Hu, Cheng Sheng, and Yufei Tao. 2019. Building an optimal point-location structure in O(sort(n)) I/Os. *Algorithmica* 81, 5 (2019), 1921–1937.
[27] Ananth V. Iyer, H. Donald Ratliff, and Gopalakrishnan Vijayan. 1991. On an edge ranking problem of trees and graphs. *Discr. Appl. Math.* 30, 1 (1991), 43–52.
[28] Tobias Jacobs, Ferdinando Cicalese, Eduardo Sany Laber, and Marco Molinaro. 2010. On the complexity of searching in trees: Average-case minimization. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP'10)*. 527–539.
[29] Camille Jordan. 1869. Sur les assemblages de lignes. *Journal für die reine und angewandte Mathematik* 70 (1869), 185–190.
[30] S. Rao Kosaraju, Teresa M. Przytycka, and Ryan S. Borgstrom. 1999. On an optimal split tree problem. In *Proceedings of the Algorithms and Data Structures Workshop (WADS'99)*. 157–168.
[31] Eduardo Sany Laber and Marco Molinaro. 2011. An approximation algorithm for binary searching in trees. *Algorithmica* 59, 4 (2011), 601–620.
[32] Eduardo Sany Laber and Loana Tito Nogueira. 2001. Fast searching in trees. *Electron. Notes Discr. Math.* 7 (2001), 90–93.
[33] Tak Wah Lam and Fung Ling Yue. 2001. Optimal edge ranking of trees in linear time. *Algorithmica* 30, 1 (2001), 12–33.
[34] Shay Mozes, Krzysztof Onak, and Oren Weimann. 2008. Finding an optimal tree searching strategy in linear time. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'08)*. 1096–1105.
[35] Ketan Mulmuley. 1990. A fast planar partition algorithm, I. *J. Symb. Comput.* 10, 3/4 (1990), 253–280.
[36] J. Ian Munro and Yakov Nekrich. 2019. Dynamic planar point location in external memory. In *Proceedings of the Symposium on Computational Geometry (SoCG)*, Vol. 129, 52:1–52:15.
[37] Krzysztof Onak and Pawel Parys. 2006. Generalization of binary search: Searching in trees and forest-like partial orders. In *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*. 379–388.
[38] Raimund Seidel. 2010. Reprint of: A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom.* 43, 6–7 (2010), 556–564.
[39] Yufei Tao, Yuanbing Li, and Guoliang Li. 2019. Interactive graph search. In *Proceedings of the ACM Management of Data Conference (SIGMOD'19)*. 1393–1410.

Received 15 December 2022; revised 29 July 2023; accepted 14 September 2023