An Optimal Algorithm for Partial Order Multiway Search

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

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

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

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

ABSTRACT

Partial order multiway search (POMS) is an important problem that finds use in crowdsourcing, distributed file systems, software testing, etc. In this problem, a game is played between an algorithm $\mathcal{A}$ and an oracle, based on a directed acyclic graph $\mathcal{G}$ known to both parties. First, the oracle picks a vertex t in $\mathcal{G}$ called the *target*; then, $\mathcal{A}$ aims to figure out which vertex is t by probing reachability. In each *probe*, $\mathcal{A}$ selects a set Q of vertices in $\mathcal{G}$ whose size is bounded by a pre-agreed value k, and 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. This article presents an algorithm to solve POMS in $O(\log_{1+k} n + \frac{d}{k} \log_{1+d} n)$ probes, where n is the number of vertices in $\mathcal{G}$, and d is the largest out-degree of the vertices in $\mathcal{G}$. The probing complexity is asymptotically optimal.

1. INTRODUCTION

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

Partial order multiway search (POMS) generalizes the aforementioned problems to arbitrary partial orders. The problem can be cast as a game between an *oracle* and an *algorithm* $\mathcal{A}$, both of which are given a *single-rooted* DAG $\mathcal{G}$, i.e., $\mathcal{G}$ has a unique *root* (a vertex with in-degree 0). The game starts by having the oracle pick a *target* vertex t from $\mathcal{G}$. Then, $\mathcal{A}$ needs to find out which vertex is t by issuing (reachability) *probes*. Specifically, in each probe, $\mathcal{A}$ chooses a set Q of vertices with $|Q| \leq k$, where k is a problem parameter. The oracle then reveals, for each vertex $q \in Q$, whether q can reach t in $\mathcal{G}$. Clearly, $\mathcal{A}$ can always discover t with $\lceil n/k \rceil$ probes where n is the number of vertices in $\mathcal{G}$. The challenge is to prove a better bound on the number of probes.

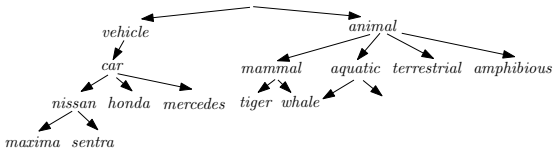

Figure 1: POMS in image classification with crowdsourcing

1.1 Motivation

In the database area, a major application of POMS is *image classification with crowdsourcing* [29], where the objective is to assign an appropriate label from a concept ontology to an image. As illustrated in Figure 1, an ontology is a DAG where each vertex is associated with a concept. Furthermore, as we move down in the ontology, the concepts encountered are increasingly specialized. The application manifests the power of a crowdsourcing system where human beings are summoned to assist problem solving by answering (simple) questions with monetary rewards. Every question has the form "*is this an* **x***?*" where **x** is a concept. Receiving a negative (resp., positive) answer to the question "*is this a* **vehicle**?", an algorithm can eliminate all the concepts that are (resp., are not) reachable from the vertex **vehicle**. The target t here is the concept eventually returned (e.g., **sentra**). As a crucial observation, although a human being is not aware of t or even the DAG, 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 [28], 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 syn-

ref.	cost	remark
[3]	$O(d \log n)$	$\mathcal{G}$ is a tree and $k = 1$
[16, 18, 24]	$O(d \log_{1+d} n)$	$\mathcal{G}$ is a tree and $k = 1$
[29]	$O((\log n)(\log_{1+k} n) + \frac{d}{k} \log_{1+d} n)$	any DAG $\mathcal{G}$ and any k
this article	$O(\log_{1+k} n + \frac{d}{k} \log_{1+d} n)$	any DAG $\mathcal{G}$ and any k
[3]	$\Omega(d \log_{1+d} n)$	$k = 1$
[29]	$\Omega(\frac{d}{k} \log_{1+d} n)$	any k
this article	$\Omega(\log_{1+k} n + \frac{d}{k} \log_{1+d} n)$	any k

Table 1: Summary of the previous and new results on class-oriented POMS (n is the number of vertices, d is the maximum out-degree, and k is the number of vertices in a probe)

chronization. If a folder has an identical checksum at the two servers, with high probability, it and its subfolders have incurred no changes. Based on this property, a POMS algorithm can find a modified folder with small communication between the two servers.

The reader may refer to [4, 28, 29] for more POMS applications in software testing, relational databases, and workflow management.

1.2 Previous Results and Related Work

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

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

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

namely, the largest cost over all possible t in $\mathcal{G}$. There are only a finite number of possible algorithms $\mathcal{A}$. To see why, note that, given $\mathcal{G}$ and t, the execution of $\mathcal{A}$ is merely a sequence of probes. Hence, any $\mathcal{A}$ can be fully described by n execution sequences (one for each t), suggesting that the number of different algorithms is finite. The problem of finding an optimal algorithm $\mathcal{A}^*$ (with the lowest $maxcost_k^{\mathrm{I}}(\mathcal{A}^*, \mathcal{G})$) is thus decidable. The challenge is to understand how much we can reduce the computation time.

The problem is best understood when $\mathcal{G}$ is a tree and $k = 1$. In that case, Ben-Asher et al. [4] were the first to show that $\mathcal{A}^*$ can be found in polynomial time. Their work motivated a line of research looking for faster algorithms [28, 15, 19, 24, 25, 27, 13]. The problem turned out to be solvable in $O(n)$ time. This was first stated by Mozes et al. [27]; later, Dereniowski [15] 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 [25]. In contrast, the problem of computing $\mathcal{A}^*$ on a DAG $\mathcal{G}$ is NP-hard [5] even if $k = 1$. Arkin et al. [2] showed how to obtain, for any DAG $\mathcal{G}$ and $k = 1$, in polynomial time an $\mathcal{A}$ whose $maxcost_k^{\mathrm{I}}(\mathcal{A}, \mathcal{G})$ is higher than $maxcost_k^{\mathrm{I}}(\mathcal{A}^*, \mathcal{G})$ by a factor of $O(\log n)$ (an approximation ratio $O(\log n / \log\log n)$ was claimed in [15] but unfortunately is incorrect, as has been confirmed by our communication with the author of [15]). We are aware of no results for $k > 1$ even when $\mathcal{G}$ is a tree.

For $k = 1$, instance-oriented POMS has also been studied under other variants [1, 5, 11, 8, 9, 10, 12, 6, 7, 14, 17, 20, 22, 23], which differ in whether (i) the cost of a probe depends on the vertex supplied, (ii) the goal is to minimize the cost of the worst t or the average cost over a distribution of t, and (iii) the answer of the oracle can be noisy.

Class-Oriented POMS. Unlike the instance-oriented category that focuses on *computability*, the class-oriented category is *graph theoretic* in nature. Given a set $\mathscr{C}$ of single-rooted DAGs, the quality of $\mathcal{A}$ is measured by

$$maxcost_k^{\mathrm{C}}(\mathcal{A}, \mathscr{C}) = \max_{\mathcal{G} \in \mathscr{C}} maxcost_k^{\mathrm{I}}(\mathcal{A}, \mathcal{G})$$

namely, the largest cost on the worst $\mathcal{G}$ in $\mathscr{C}$. Define

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

that is, 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

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

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

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 [3] proved $minmaxcost_1(\mathscr{T}(n, d)) = \Omega(d \log_{1+d} n)$ and $minmaxcost_1(\mathscr{T}(n, d)) = O(d \log n)$, leaving a gap of $\Theta(\log(1 + d))$ in between. Laber and Nogueira [24] tightened the upper bound to $minmaxcost_1(\mathscr{T}(n, d)) = \Theta(d \log_{1+d} n)$ (see also [16, 18] where the same result was derived). Regarding $\mathscr{G}(n, d)$ and arbitrary $k \geq 1$, Tao et al. [29] 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 [29] holds even if $\mathscr{T}(n, d)$ is replaced with $\mathscr{G}(n, d)$.

1.3 New Results

This article's main contribution is to settle class-oriented POMS optimally.

THEOREM 1. *For the POMS problem, let n be the number of vertices in the input graph $\mathcal{G}$ and d be the maximum vertex out-degree in $\mathcal{G}$. Both statements below 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.*

Table 1 gives a comparison of our results against the previous ones. The above theorem implies:

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

Our lower bound in the second bullet holds even if $\mathcal{G}$ comes from $\mathscr{T}(n,d)$.

Remark. The reader may refer to [26], the full version of this extended abstract, for additional results related to POMS.

2. PRELIMINARIES

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

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

Shielding. We introduce a *shield* operator $\ominus$. Given nodes u and v in a tree T, the expression $T_u \ominus \{v\}$ returns

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

Given a node $u \in T$ and a set $S = \{v_1, v_2, ..., v_x\}$ where $v_i \in T$ for each $i \in [1, x]$, we define

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

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), on the other hand, processes the white out-neighbor v_{best} of u_{top} that is able to G-reach the most white vertices via white paths, where a white path is a path including only white vertices. If two or more nodes satisfy this condition, v_{best} can be any of them.

HPDFS defines a tree T — the *HPDFS-tree* [29] — where a node u parents another node v if the latter is discovered

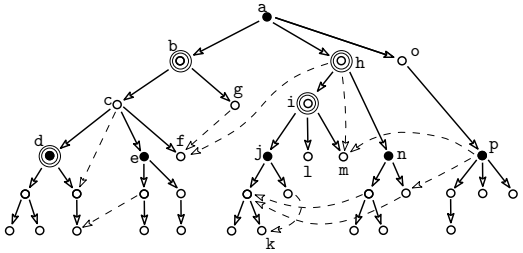

Figure 2: A running example. G is the graph represented by both the solid and dashed edges. An HPDFS T of G is indicated by the solid edges. The labels on the nodes are consistent with the total order $\prec$. The black nodes constitute the 8-separator $\Sigma = \{\mathtt{a}, \mathtt{d}, \mathtt{e},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

while the former tops the stack. It also defines a total order $\prec$ on the vertices in G: $u \prec v$ if u enters the stack before v (we read $u \prec v$ as *u smaller than v* or *v larger than u*). 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.

The lemma below gives several useful properties of T (see [26] for the proof):

LEMMA 2. *Let T be an HPDFS-tree of a single-rooted DAG G.*

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

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 $\mathtt{b}$ precedes $\mathtt{h}$ in $\prec$ and $\mathtt{h} \notin T_{\mathtt{b}}$, the no-cross-reachability property assures us that $\mathtt{b}$ cannot G-reach any node in $T_{\mathtt{h}}$. Because $\mathtt{k} \in T_{\mathtt{h}}$, the path-descendants property asserts that every path from $\mathtt{h}$ to $\mathtt{k}$ in G can contain only nodes in $T_{\mathtt{h}}$. The other two properties are easy to understand. □

3. NEW RESULTS IN GRAPH THEORY

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

3.1 Separators

It is well-known [21] that T must contain a node whose removal disconnects T into trees, each having at most $n/2$ nodes. We now prove a more general fact.

LEMMA 3. *Let T be an HPDFS-tree of a single-rooted DAG with n vertices. For any $\lambda \in [2, n]$, 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 an 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 */
4. $\quad$ 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 $|S| \leq \lambda - 1$. Every time we add a node into S at Line 4, we remove $\lfloor n/\lambda \rfloor + 1 > n/\lambda$ nodes from T'. If $|S| \geq \lambda$, the total number of nodes removed would be *strictly* larger $|S| \cdot n/\lambda \geq \lambda \cdot n/\lambda = n$, 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 3 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 $\Sigma = \{\mathtt{a}, \mathtt{d}, \mathtt{e}, \mathtt{j}, \mathtt{n}, \mathtt{p}\}$; the above algorithm finds the nodes of Σ in the order $\mathtt{d}, \mathtt{e}, \mathtt{j}, \mathtt{n}, \mathtt{p}$, and $\mathtt{a}$. 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 $\text{LF}(u)$ — using the following procedure:

- Initialize an empty set S.
- For each node v on π with $v \neq u$: let v' be the child of v on π; add to S all the left siblings of v' in T.
- The S after the previous step is $\text{LF}(u)$.

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

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

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

LEMMA 4. *For any node $u \in \Sigma$ and node $v \in \text{LF}(u)$, T_v has at least one node in $\Sigma \setminus \{u\}$.*

PROOF. Node v must have a right sibling v' on the root-to-u path in T; note that v' is an ancestor of u. By the way Σ is produced from **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 2). 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 obviously cannot be u. □

COROLLARY 5. *For any node $v \in \text{LFU}(\Sigma)$, we have that T_v contains at least one node in Σ.*

PROOF. The fact $v \in \text{LFU}(\Sigma)$ means that $v \in \text{LF}(u)$ for some $u \in \Sigma$. The claim then follows from Lemma 4. □

LEMMA 6. $|\text{LFU}(\Sigma)| \leq |\Sigma| - 1$ *and* $|\text{GU}(\Sigma)| < 2\lambda$.

PROOF. We will prove only $|\text{LFU}(\Sigma)| \leq |\Sigma| - 1$ because $|\text{GU}(\Sigma)| < 2\lambda$ will then follow immediately from (5) and Lemma 3.

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 $v \in \text{LFU}(\Sigma)$. By definition of $\text{LFU}(\Sigma)$, $v \in \text{LF}(u)$ for some $u \in \Sigma$, because of which T_v contains at least one node in Σ (Lemma 4). Hence, $v \in T^P$. In other words, all the nodes in $\text{LFU}(\Sigma)$ 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 $v \in \text{LFU}(\Sigma)$ satisfies: (i) some node in T^P is a right sibling of v in T, and (ii) $\textit{parent}(v) \in I$. This implies that each $u \in I$ has at most $c_u - 1$ child nodes in $\text{LFU}(\Sigma)$, leading to $|\text{LFU}(\Sigma)| \leq \sum_{u \in I} (c_u - 1)$.

It remains to prove $\sum_{u \in I} (c_u - 1) \leq |\Sigma| - 1$. 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 how T^P is defined, 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, the degree sum of all vertices in (the undirected) T^P is twice the number of edges in T^P. This fact implies:

$$x + 2y + \left(\sum_{u \in I} (c_u + 1)\right) - 1 = 2 \cdot (|I| + x + y - 1)$$
$$\Rightarrow \sum_{u \in I} (c_u - 1) = x - 1 \leq |\Sigma| - 1.$$

□

LEMMA 7. *If node u can G-reach node v, then $v \in T_{u^*}$, where u^* is the smallest node in $\text{LF}(u) \cup \{u\}$ that can G-reach v.*

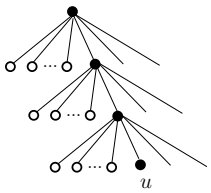

Figure 3: The left flank of u includes all the white nodes.

PROOF. We first show that $v \in T_{u^*}$ for some node $u^* \in \text{LF}(u) \cup \{u\}$. 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 $u^* = u \in \text{LF}(u) \cup \{u\}$ 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, G has a cycle as u can G-reach v), p must have a child u^* such that $w \in T_{u^*}$. We now argue that u^* is a left sibling of w and, hence, belongs to $\text{LF}(u)$. Indeed, if u^* is a right sibling, then nodes w and u^* cause a violation of the no-cross-reachability property of Lemma 2: w can G-reach a node (i.e., v) in T_{u^*}.

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

The next two lemmas present some properties of star.

LEMMA 8. *Let S be any set of vertices in G that contain the root of G. For any $u \in S$, the star of S for u is u itself.*

PROOF. This is due to three facts: (i) no proper descendant of u (in T) can $\mathcal{G}$-reach u (otherwise, there would be a cycle), (ii) no proper ancestor of u can be the lowest node in S able to $\mathcal{G}$-reach u (because u can $\mathcal{G}$-reach itself), and (iii) if a node v is smaller than u (under $\prec$) but not an ancestor of u, then v cannot $\mathcal{G}$-reach u (no-cross-reachability property of Lemma 2). □

LEMMA 9. *Let Σ be a k-separator of T. 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:

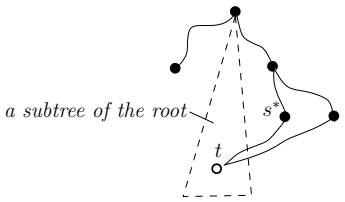

Figure 4: If S is the set of black vertices, then s^* is the star of S for t

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

The first bullet is a direct corollary of the no-cross-reachability property (Lemma 2). Next, we focus on the second bullet.

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

- v cannot be a descendant (in T) of any left sibling of u (otherwise the left sibling of u can $\mathcal{G}$-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 $v \in T_{u'}$.

By Lemma 4, T_v must contain at least one node in Σ (this is obvious true if $v \in \Sigma$. Otherwise, the fact $v \in \text{GU}(\Sigma)$ tells us that $v \in \text{LF}(w)$ for some $w \in \Sigma$. Lemma 4 shows that T_v must have at least one node in $\Sigma \setminus \{w\}$). Hence, $T_{u'}$ also contains at least one node, say w, in Σ. But this means that u (being a left sibling of u') belongs to $\text{LF}(w)$ and, hence, also belongs to $\text{LFU}(\Sigma)$, causing a contradiction. □

3.4 Path Preservation Lemmas

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

LEMMA 10. [Path Preservation Using a Root-Containing Set] *Let t be a vertex in G and S be a set of vertices in G including the root. Denote by s^* 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. On the other hand, $t \notin T_v$ for any right sibling v of $s^{\#}$; otherwise, $s^{\#}$, v, and t cause a violation of the no-cross-reachability property of Lemma 2. 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 2 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, also s must 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 $t = \mathtt{k}$ and $S = \{\mathtt{a}, \mathtt{b}, \mathtt{h}, \mathtt{l}, \mathtt{m}, \mathtt{p}\}$. As mentioned, the star s^* of S for t is $\mathtt{h}$. Both child nodes of $\mathtt{h}$ (i.e., $\mathtt{i}$ and $\mathtt{n}$) can G-reach $t = \mathtt{k}$. Hence, $s^{\#} = \mathtt{i}$. $T_{s^{\#}} \ominus S$ — the tree obtained by "shielding" $T_{\mathtt{i}}$ with S — consists of the edges in $T_{\mathtt{j}}$ plus the edge $(\mathtt{i}, \mathtt{j})$. Note that $\mathtt{i}$ has two paths to $\mathtt{k}$ in G, both of which are preserved in $G[T_{s^{\#}} \ominus S]$. □

The proofs of the next two lemmas can be found in [26].

LEMMA 11. [Path Preservation Using a Left Flank] *Let t be a vertex in G and Σ be a k-separator of T. Denote by s^* the star of Σ for t. If s^{**} is the smallest node in $\mathrm{LF}(s^*) \cup \{s^*\}$ able to G-reach t, then*

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

Example. Consider Figure 2 with $\lambda = 8$ and $t = \mathtt{m}$. Recall that $\Sigma = \{\mathtt{a}, \mathtt{d}, \mathtt{e}, \mathtt{j}, \mathtt{n}, \mathtt{p}\}$ and, hence, $s^* = \mathtt{p}$. Thus, $\mathrm{LF}(s^*) \cup \{s^*\} = \{\mathtt{b}, \mathtt{h}, \mathtt{p}\}$, giving $s^{**} = \mathtt{h}$. $T_{s^{**}} \ominus \Sigma$ is a tree with four nodes: $\mathtt{h}$, $\mathtt{i}$, $\mathtt{l}$ and $\mathtt{m}$. There are two $\mathtt{h}$-to-$\mathtt{m}$ paths in G, both of which are preserved in $G[T_{s^{**}} \ominus \Sigma]$. □

LEMMA 12. [Path Preservation Using a Grand Union] *Let t be a vertex in G and Σ be a k-separator of T. Denote by s^* the star of $\mathrm{GU}(\Sigma)$ for t. Then*

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

Example. Consider Figure 2 with $\lambda = 8$. As explained in Section 3.2, $\mathrm{GU}(\Sigma) = \{\mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{e}, \mathtt{h}, \mathtt{i}, \mathtt{j}, \mathtt{n}, \mathtt{p}\}$. If $t = \mathtt{f}$, then $s^* = \mathtt{b}$. The tree $T_{\mathtt{b}} \ominus \mathrm{GU}(\Sigma)$ has nodes $\mathtt{b}$, $\mathtt{c}$, $\mathtt{f}$ and $\mathtt{g}$. G has two $\mathtt{b}$-to-$\mathtt{f}$ paths, both preserved in $G[T_{s^*} \ominus \mathrm{GU}(\Sigma)]$. □

4. PROBING COMPLEXITY OF POMS

This section will establish the formal results stated in Section 1.3. Specifically, we will present a new POMS algorithm in Section 4.1 and analyze its probing cost in Section 4.2. In Section 4.3, we will provide a matching lower bound on the probing complexity, which will complete the proof of Theorem 1. Our discussion will assume $k \geq 2$; for $k = 1$, one can manually increase k to 2 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 $\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_h$ (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 i-th 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 13. *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 any u-to-t path π in the original graph $\mathcal{G}$. By the inductive assumption, π is preserved in $\mathcal{G}_j$. Let r be the root of $\mathcal{G}_j$. As $\mathcal{G}_j$ is single-rooted, r can $\mathcal{G}_j$-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 $\mathcal{G}_j$. The path preserving property tells us that π'' must 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, 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 $\mathcal{G}_{i-1}$). We can then identify the star s^* of Σ for t (Section 3.3). By Lemma 6, $|\mathrm{LF}(s^*)| \leq |\mathrm{LFU}(\Sigma)| \leq k - 1$. Thus, with another probe we can figure out which nodes in $\mathrm{LF}(s^*) \cup \{s^*\}$ can $\mathcal{G}_{i-1}$-reach t. Define s^{**} to be the smallest (under $\prec$) among those nodes.

Phase 2. It must hold that either $s^{**} \notin \Sigma$ or $s^{**} = s^*$ (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)). In the former case, 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, 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, the graph $\mathcal{G}_i$ generated contains no vertices in Σ.

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

LEMMA 14. *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 i-th iteration only when $s^{**} = s^*$. In this case, the first bullet of Lemma 11 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 13), 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 3). Next, we prove the claim that $\mathcal{G}_i$ contains t and is path preserving. If $s^{**} \notin \Sigma$, the claim follows from Lemma 11. Otherwise, we must have $s^{**} = s^*$, and the first bullet of Lemma 11 assures us $t \in T_{s^*}$. As the existence of $s^\#$ indicates $t \neq s^*$, the claim now follows from Lemma 10. □

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 i-th 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 i-th 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} (resp., n_i) to the number of vertices in $\mathcal{G}_{i-1}$ (resp., $\mathcal{G}_i$). Define an integer x_i as follows:

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

The i-th iteration issues at most

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

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

LEMMA 15. *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 14. 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 2). Hence, $|T_{s^\#}| = (x_i + 1)|T_{s^\#}| \,/\, (x_i + 1) \leq n_{i-1}/(x_i + 1)$. The lemma then follows from $n_i \leq |T_{s^\#}|$. □

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

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

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

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

If $d \leq k$, then $x_i \leq d \leq k$; hence, (9) is $O(\log_k n)$. 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, h-th, iteration obviously performs $O(d/k)$ probes).

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

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

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

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

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

LEMMA 16. *When $\sum_{i=1}^{h-1} x_i$ is maximized, at most one of $x_1, ..., x_{h-1}$ can be strictly larger than $k - 1$ but strictly smaller than $d - 1$.*

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

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

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

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

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

4.3 A Lower Bound

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

$$maxcost^{\mathrm{I}}_k(\mathcal{A}, \mathcal{G}) \quad = \quad \Omega(\log_{1+k} n) \tag{13}$$

holds for any POMS algorithm $\mathcal{A}$, where $maxcost^{\mathrm{I}}_k(\mathcal{A}, \mathcal{G})$ is defined in Section 1.2. Consider a probe with a set Q of $k \geq 1$ vertices $q_1, q_2, ..., q_k$. The oracle returns an *outcome sequence* $a_1, a_2, ..., a_k$, where $a_i = 1$ if q_i can $\mathcal{G}$-reach the target t or 0 otherwise. With Q fixed, the outcome sequence solely depends on t.

LEMMA 17. *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 $q_1, q_2, ..., q_{z+1}$. At least one node in Q has the property that its subtree in $\mathcal{G}$ contains no other nodes in Q. Assume that q_{z+1} is such a node (otherwise, rename the nodes in Q).

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

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

We now prove (13) with an information theoretic argument. By Lemma 17, 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 1.2, it has been proved in [29] that $minmaxcost_k(\mathscr{T}(n, d)) = \Omega(\frac{d}{k} \log_{1+d} n)$, where $minmaxcost(.)$ is defined in Equation (1), and $\mathscr{T}(n, d)$ is defined in Equation (3). As the above argument holds for any tree with n vertices, we can now conclude that $minmaxcost_k(\mathscr{T}(n, d)) = \Omega(\log_{1+k} n + \frac{d}{k} \log_{1+d} n)$. This proves the second bullet of Theorem 1.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a new algorithm for the POMS (partial order multiway search) problem, which performs $O(\log_{1+k} n + \frac{d}{k} \log_{1+d} n)$ probes, where n is the number of vertices in the input (acyclic) graph G, d is the maximum vertex out-degree in G, and k is the number of nodes that can be queried in one probe. The probing complexity is asymptotically optimal. We leave it as a (challenging) open problem to minimize the hidden constants in our probing complexity. It will also be interesting to study the empirical efficiency of our algorithms, e.g., in the scenarios experimented in [29].

Another promising direction for future work is to study other variants of POMS arising in real-world applications. One, for example, is to consider the scenario where the oracle selects multiple — say ℓ — targets, and an algorithm must discover all of them. One needs to be careful in defining the outcome of a *probe*. Consider, for simplicity, $k = 1$. If, given a node u, an oracle is required to provide ℓ reachability answers (one for each target), our algorithm in this paper is readily applicable. The problem, however, becomes non-trivial when the oracle is required to provide only one answer: whether or not u can reach *at least one* target (i.e., the oracle answers "no" if and only if u can reach no targets at all).

6. REFERENCES

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

[2] E. M. Arkin, H. Meijer, J. S. B. Mitchell, D. Rappaport, and S. Skiena. Decision trees for geometric models. *International Journal of Computational Geometry and Applications*, 8(3):343–364, 1998.

[3] Y. Ben-Asher and E. Farchi. The cost of searching in general trees versus complete binary trees. Technical report, 1997.

[4] Y. Ben-Asher, E. Farchi, and I. Newman. Optimal search in trees. *SIAM J. of Comp.*, 28(6):2090–2102, 1999.

[5] R. Carmo, J. Donadelli, Y. Kohayakawa, and E. S. Laber. Searching in random partially ordered sets. *Theo. Comp. Sci.*, 321(1):41–57, 2004.

[6] V. T. Chakaravarthy, V. Pandit, S. Roy, P. Awasthi, and M. K. Mohania. Decision trees for entity identification: Approximation algorithms and hardness results. *ACM Transactions on Algorithms*, 7(2):15:1–15:22, 2011.

[7] V. T. Chakaravarthy, V. Pandit, S. Roy, and Y. Sabharwal. Approximating decision trees with multiway branches. In *ICALP*, pages 210–221, 2009.

[8] F. Cicalese, T. Jacobs, E. S. Laber, and M. Molinaro. On greedy algorithms for decision trees. In *ISAAC*, pages 206–217, 2010.

[9] F. Cicalese, T. Jacobs, E. S. Laber, and M. Molinaro. On the complexity of searching in trees and partially ordered structures. *Theoretical Computer Science*, 412(50):6879–6896, 2011.

[10] F. Cicalese, T. Jacobs, E. S. Laber, and M. Molinaro. Improved approximation algorithms for the average-case tree searching problem. *Algorithmica*, 68(4):1045–1074, 2014.

[11] F. Cicalese, T. Jacobs, E. S. Laber, and C. D. Valentim. The binary identification problem for weighted trees. *Theoretical Computer Science*, 459:100–112, 2012.

[12] F. Cicalese, B. Keszegh, B. Lidický, D. Pálvölgyi, and

T. Valla. On the tree search problem with non-uniform costs. *Theoretical Computer Science*, 647:22–32, 2016.

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

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

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

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

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

[18] E. Emamjomeh-Zadeh, D. Kempe, and V. Singhal. Deterministic and probabilistic binary search in graphs. In *STOC*, pages 519–532, 2016.

[19] A. V. Iyer, H. D. Ratliff, and G. Vijayan. On an edge ranking problem of trees and graphs. *Discrete Applied Mathematics*, 30(1):43–52, 1991.

[20] T. Jacobs, F. Cicalese, E. S. Laber, and M. Molinaro. On the complexity of searching in trees: Average-case minimization. In *ICALP*, pages 527–539, 2010.

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

[22] S. R. Kosaraju, T. M. Przytycka, and R. S. Borgstrom. On an optimal split tree problem. In *WADS*, pages 157–168, 1999.

[23] E. S. Laber and M. Molinaro. An approximation algorithm for binary searching in trees. *Algorithmica*, 59(4):601–620, 2011.

[24] E. S. Laber and L. T. Nogueira. Fast searching in trees. *Electronic Notes in Discrete Mathematics*, 7:90–93, 2001.

[25] T. W. Lam and F. L. Yue. Optimal edge ranking of trees in linear time. *Algorithmica*, 30(1):12–33, 2001.

[26] S. Lu, W. Martens, M. Niewerth, and Y. Tao. Optimal algorithms for multiway search on partial orders. In *PODS*, pages 175–187, 2022.

[27] S. Mozes, K. Onak, and O. Weimann. Finding an optimal tree searching strategy in linear time. In *SODA*, pages 1096–1105, 2008.

[28] K. Onak and P. Parys. Generalization of binary search: Searching in trees and forest-like partial orders. In *FOCS*, pages 379–388, 2006.

[29] Y. Tao, Y. Li, and G. Li. Interactive graph search. In *SIGMOD*, pages 1393–1410, 2019.