Noname manuscript No.
(will be inserted by the editor)

Linear-Space Data Structures for Range Frequency Queries on Arrays and Trees

Stephane Durocher · **Rahul Shah** ·
Matthew Skala · **Sharma V. Thankachan**

the date of receipt and acceptance should be inserted later

Abstract We present $O(n)$-space data structures to support various range frequency queries on a given array $A[0 : n-1]$ or tree T with n nodes. Given a query consisting of an arbitrary pair of pre-order rank indices (i, j), our data structures return a least frequent element, mode, α-minority, or top-k colors (values) of the multiset of elements in the unique path with endpoints at indices i and j in A or T. We describe a data structure that supports range least frequent element queries on arrays in $O(\sqrt{n/w})$ time, improving the $\Theta(\sqrt{n})$ worst-case time required by the data structure of Chan et al. (SWAT 2012), where $w \in \Omega(\log n)$ is the word size in bits. We describe a data structure that supports path mode queries on trees in $O(\log\log n\sqrt{n/w})$ time, improving the $\Theta(\sqrt{n}\log n)$ worst-case time required by the data structure of Krizanc et al. (ISAAC 2003). We describe the first data structures to support path least frequent element queries, path α-minority queries, and path top-k color queries on trees in $O(\log\log n\sqrt{n/w})$, $O(\alpha^{-1}\log\log n)$, and $O(k)$ time, respectively, where $\alpha \in [0,1]$ and $k \in \{1,\ldots,n\}$ are specified at query time.

Work supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and National Science Foundation (NSF) Grants CCF–1017623 (R. Shah) and CCF–1218904 (R. Shah). Preliminary versions of some of these results appeared at the International Symposium on Mathematical Foundations of Computer Science (MFCS) [18] and the String Processing and Information Retrieval Symposium (SPIRE) [19].

S. Durocher · M. Skala
University of Manitoba, Winnipeg, Canada
E-mail: {durocher,mskala}@cs.umanitoba.ca

R. Shah
Louisiana State University, Baton Rouge, USA
E-mail: rahul@csc.lsu.edu

S.V. Thankachan
Georgia Institute of Technology, Atlanta, USA
E-mail: sharma.thankachan@gmail.com

1 Introduction

1.1 Definitions

Given an array $A[0:n-1]$ of integer values drawn from the alphabet $\{0,\ldots,\sigma-1\}$, the *frequency*, denoted $\mathrm{freq}_{A[i:j]}(x)$, of an element x in the multiset stored in the subarray $A[i:j]$ is the number of occurrences (i.e., the multiplicity) of x in $A[i:j]$. Elements a and b in $A[i:j]$ are respectively a *mode* and a *least frequent element* of $A[i:j]$ if for all $c \in A[i:j]$, $\mathrm{freq}_{A[i:j]}(a) \geq \mathrm{freq}_{A[i:j]}(c) \geq \mathrm{freq}_{A[i:j]}(b)$. Given $\alpha \in [0,1]$, an α*-minority* of $A[i:j]$ is an element $d \in A[i:j]$ such that $1 \leq \mathrm{freq}_{A[i:j]}(d) \leq \alpha|j-i+1|$. Conversely, d is an α*-majority* of $A[i:j]$ if $\mathrm{freq}_{A[i:j]}(d) > \alpha|j-i+1|$. The *top-k colors* in $A[i:j]$ are the k distinct highest values that occur in the multiset $A[i:j]$, reported in descending order. Range top-k color queries generalize range minimum/maximum queries; when $k=1$, a range top-1 query is a range maximum query. Modes, least frequent elements, α-majorities, α-minorities, and top-k colors of a multiset are not necessarily unique.

We study the problem of indexing a given array $A[0:n-1]$ to construct data structures that can be stored using $O(n)$ words of space and support efficient range frequency queries. Each query consists of a pair of input indices (i,j) (along with a value $\alpha \in [0,1]$ for α-minority queries or $k \in \{1,\ldots,n\}$ for top-k color queries), for which a mode, least frequent element, α-minority, or top-k colors of $A[i:j]$ must be returned. Any two nodes in a tree define a unique path between them, just as two elements of an array define a unique range. When the entire tree is a path, tree paths reduce to array ranges. Thus, range queries generalize to trees, where they are called *path queries*: given a tree T on n nodes and a pair of indices (i,j), a query is applied to the multiset of elements stored at nodes along the unique path in T whose endpoints are the two nodes with pre-order traversal ranks i and j.

We assume the Word RAM model of computation using words of size $w \in \Omega(\log n + \log\sigma)$ bits, where n denotes the number of elements stored in the input array/tree and σ denotes an upper bound on the magnitude of integers stored in the array/tree. Unless explicitly specified otherwise, space requirements are expressed in multiples of words. We use the notation $\log^{(k)}$ to represent logarithm iterated k times; that is, $\log^{(1)} n = \log n$ and $\log^{(k)} n = \log\log^{(k-1)} n$ for any integer $k > 1$. To avoid ambiguity, we use the notation $(\log n)^k$ instead of $\log^k n$ to represent the kth power of $\log n$.

1.2 Related Work

Krizanc et al. [34] presented $O(n)$-space data structures that support range mode queries in $O(\sqrt{n}\log\log n)$ time on arrays and $O(\sqrt{n}\log n)$ time on trees. Chan et al. [8,9] achieved $o(\sqrt{n})$ query time with an $O(n)$-space data structure that supports queries in $O(\sqrt{n/w}) \subseteq O(\sqrt{n/\log n})$ time on arrays (also see [17]). In this paper we present a new $O(n)$-space data structure that improves the path mode query time on trees to $O(\sqrt{n/w}\log\log n)$.

Chan et al. [10] presented an $O(n)$-space data structure that supports range least frequent element queries on arrays in $O(\sqrt{n})$ time. Range mode and range least fre-

quent element queries on arrays appear to require significantly longer times than either range minimum or range selection queries; reductions from boolean matrix multiplication show that query times significantly lower than $\sqrt{n}$ are unlikely for either problem with $O(n)$-space data structures [8,10]. Whereas an $O(n)$-space data structure that supports range mode queries on arrays in $o(\sqrt{n})$ time is known [8], the space reduction techniques applied to achieve the time improvement are not directly applicable to the setting of least frequent elements. Chan et al. [10] ask whether $o(\sqrt{n})$ query time is possible in an $O(n)$-space data structure, observing that "unlike the frequency of the mode, the frequency of the least frequent element does not vary monotonically over a sequence of elements. Furthermore, unlike the mode, when the least frequent element changes [in a sequence], the new element of minimum frequency is not necessarily located in the block in which the change occurs" [10, p. 11]. By applying different techniques, in this paper we present the first $O(n)$-space data structure that supports range least frequent element queries on arrays in $o(\sqrt{n})$ time; specifically, we achieve $O(\sqrt{n/w}) \subseteq O(\sqrt{n/\log n})$ query time, matching the time achieved by Chan et al. [8] for range mode queries and answering the question posed by Chan et al. [10]. We also generalize our techniques to the setting of trees to support path least frequent element queries in $O(\log\log n\sqrt{n/w})$ time.

Karpinski and Nekrich [32] gave an $O(n\log\sigma)$-bit data structure that supports range top-k queries on arrays in $O(k)$ time, which is asymptotically optimal. Other related problems include reporting all distinct colors, counting the number of distinct colors, and finding a most frequent, least frequent, majority or minority color in the query range. Efficient data structures offering various space-time trade-offs for such range query problems are known [4,8,16,23–26,33]. Such problems arise often in information retrieval and computational geometry. In this paper we present the first $O(n)$-space data structure for path top-k color queries on trees, supporting queries in $O(k)$ time, which is asymptotically optimal.

The α-majority range query problem on arrays was introduced by Karpinski and Nekrich [32] and Durocher et al. [15,16]; the latter presented an $O(n\log(\alpha^{-1}))$-space data structure that supports queries in $O(\alpha^{-1})$ time for any $\alpha \in (0,1)$ fixed during preprocessing. When α is specified at query time, Gagie et al. [23] and Chan et al. [10] presented $O(n\log n)$-space data structures that support queries in $O(\alpha^{-1})$ time, and Belazzougui et al. [2] presented an $O(n)$-space data structure that supports queries in $O(\alpha^{-1}\log\log(\alpha^{-1}))$ time. For range α-minority queries, Chan et al. [10] described an $O(n)$-space data structure that supports queries in $O(\alpha^{-1})$ time, where α is specified at query time. In this paper we present the first $O(n)$-space data structure for path α-minority queries on trees, achieving $O(\alpha^{-1}\log\log n)$ query time.

A variety of other range query problems have been examined on arrays and trees, including range minimum/extrema, for which optimal data structures exist requiring $O(n)$ space and $O(1)$ query time (e.g., on arrays [5,12–14,21] and on trees [13]) and range selection/median, for which optimal data structures exist requiring $O(n)$ space and $O(\log n/\log\log n)$ query time (e.g., on arrays [7,25,26,31,34] and on trees [28,29,37]). Jørgensen and Larsen [31] proved a matching lower bound of $\Omega(\log n/\log\log n)$ on the worst-case time required for range median query on arrays by any $O(n)$-space data structure. See Skala [40] for a detailed survey of array range queries.

range/path query	input	previous best	new (this paper)
mode	array	$O(\sqrt{n/w})$ [8,9]	
	tree	$O(\sqrt{n}\log n)$ [33,34]	$O(\log\log n\sqrt{n/w})$
least frequent element	array	$O(\sqrt{n})$ [10]	$O(\sqrt{n/w})$
	tree	no previous result	$O(\log\log n\sqrt{n/w})$
α-minority	array	$O(\alpha^{-1})$ [10]	
	tree	no previous result	$O(\alpha^{-1}\log\log n)$
top-k	array	$O(k)$ [32]	
	tree	no previous result	$O(k)$

Table 1 Worst-case query times of previous best and new $O(n)$-space data structures

1.3 Overview of Contributions

In this paper we present new $O(n)$-space data structures for range least frequent element query on arrays and path mode query on trees that achieve faster query times than the previous best data structures, and the first $O(n)$-space data structures for least frequent, α-minority, and top-k color path queries on trees. After revisiting some necessary previous work in Section 2, in Section 3 we describe the first $O(n)$-space data structure that achieves $o(\sqrt{n})$ time for range least frequent element queries on arrays, supporting queries in $O(\sqrt{n/w})$ time. We then extend this data structure to the setting of trees in Section 5, where we achieve $O(\log\log n\sqrt{n/w})$ query time. In Section 4 we present an $O(n)$-space data structure that supports path mode queries on trees in $O(\log\log n\sqrt{n/w})$ time. To do so, we construct $O(n)$-space data structures that support colored nearest ancestor path queries on trees in $O(\log\log n)$ time (find the nearest ancestor with value k of node i, where i and k are given at query time); path frequency queries on trees in $O(\log\log n)$ time (count the number of instances of k on the path between nodes i and j, where i, j, and k are given at query time); and k-nearest distinct ancestor path queries on trees in $O(k)$ time (return k ancestors of node i such that each ancestor stores a distinct value and the distance to the furthest ancestor from i is minimized, where i and k are given at query time). In Section 6 we present an $O(n)$-space data structure that supports path α-minority queries on trees in $O(\alpha^{-1}\log\log n)$ time, where α is given at query time. Finally, in Section 7 we present an $O(n)$-space data structure that supports top-k color path queries on trees in $O(k)$ time, where k is given at query time. Our contributions are summarized in Table 1.

2 A Framework for Range Least Frequent Element Queries on Arrays

Our data structure for range least frequent element queries on an arbitrary given input array $A[0:n-1]$ uses a technique introduced by Chan et al. [10]. Upon applying a rank space reduction to A, all elements in A are in the range $\{0,\ldots,\Delta-1\}$, where Δ denotes the number of distinct elements in the original array A. Before returning the result of a range query computation, the corresponding element in the rank-reduced array is mapped to its original value in constant time by a table lookup [8,10]. Chan et al. [10] prove the following result.

Theorem 1 (Chan et al. [10]) *Given any array $A[0:n-1]$ and any fixed $s \in [1,n]$, there exists an $O(n+s^2)$-word space data structure that supports range least frequent element query on A in $O(n/s)$ time and requires $O(n \cdot s)$ preprocessing time.*

The data structure of Chan et al. includes index data that occupy a linear number of words, and two tables D_t and E_t whose sizes ($O(s^2)$ words each) depend on the parameter s. Let t be an integer blocking factor. Partition $A[0:n-1]$ into $s = \lceil n/t \rceil$ blocks of size t (except possibly the last block which has size $1+[(n-1) \bmod t]$). For every pair (i,j), where $0 \le i < j \le s-1$, the contents of the tables D_t and E_t are as follows:

- $D_t(i,j)$ stores a least frequent element in $A[i \cdot t : j \cdot t - 1]$, and
- $E_t(i,j)$ stores an element which is least frequent in the multiset of elements that are in $A[i \cdot t : j \cdot t - 1]$ but not in $A[i \cdot t : (i+1)t-1] \cup A[(j-1)t : j \cdot t - 1]$.

In the data structure of Chan et al. [10], the tables D_t and E_t are the only components whose space bound depends on s. The cost of storing and accessing the tables can be computed separately from the costs incurred by the rest of the data structure. The proof for Theorem 1 given by Chan et al. implies the following result.

Lemma 1 (Chan et al. [10]) *If the tables D_t and E_t can be stored using $S(t)$ bits of space to support lookup queries in $T(t)$ time, then, for any $\{i,j\} \subseteq \{0,\ldots,n-1\}$, a least frequent element in $A[i:j]$ can be computed in $O(T(t)+t)$ time using an $O(S(t)+n\log n)$-bit data structure.*

When $t \in \Theta(\sqrt{n})$, the tables D_t and E_t can be stored explicitly in linear space. In that case, $S(t) \in O((n/\sqrt{n})^2 \log n) = O(n \log n)$ bits and $T(t) \in O(1)$, resulting in an $O(n \log n)$-bit ($O(n)$-word) space data structure that supports $O(\sqrt{n})$-time queries [10]. In the present work, we describe how to encode the tables using fewer bits per entry, allowing them to contain more entries, and therefore allowing a smaller value for t and lower query time.

We also refer to the following lemma by Chan et al. [8]:

Lemma 2 (Chan et al. [8]) *Given an array $A[0:n-1]$, there exists an $O(n)$-space data structure that returns the index of the qth instance of $A[i]$ in $A[i:n-1]$ in $O(1)$ time for any $0 \le i \le n-1$ and any q.*

3 Range Least Frequent Element Queries on Arrays

In this section, we improve on the data structure of Chan et al. [10] for array range least frequent element queries to return results in $O(\sqrt{n/w})$ time. We first describe how to calculate the table entries for a smaller block size using lookups on a similar pair of tables for a larger block size and some index data that fits in linear space. Then, starting from the $t = \sqrt{n}$ tables which we can store explicitly, we apply that block-shrinking operation $\log^* n$ times, ending with blocks of size $O(\sqrt{n/w})$, which gives the desired lookup time.

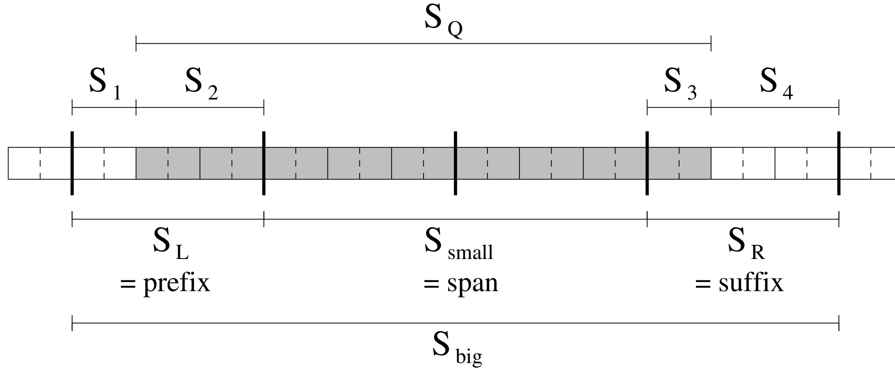

Fig. 1 Illustration in support of Lemma 3

3.1 Data Structure Overview

At each level of the construction, we partition the array into three levels of blocks whose sizes are t (*big blocks*), t' (*small blocks*), and t'' (*micro blocks*), where $1 \leq t'' \leq t' \leq t \leq n$. We will compute table entries for the small blocks, $D_{t'}$ and $E_{t'}$, assuming access to table entries for the big blocks, D_t and E_t. The micro block size t'' is a parameter of the construction but does not directly determine which queries the data structure can answer. Lemma 3 follows from Lemmas 4 and 5 (see Section 3.2). The bounds in Lemma 3 express only the cost of computing small block table entries $D_{t'}$ and $E_{t'}$, not for answering a range least frequent element query at the level of individual elements.

Lemma 3 *Given block sizes $1 \leq t'' \leq t' \leq t \leq n$, if the tables D_t and E_t can be stored using $S(t)$ bits of space to support lookup queries in $T(t)$ time, then the tables $D_{t'}$ and $E_{t'}$ can be stored using $S(t')$ bits of space to support lookup queries in $T(t')$ time, where*

$$S(t') = S(t) + O(n + (n/t')^2 \log(t/t'')) \, , \text{ and} \tag{1}$$

$$T(t') = T(t) + O(t'') \, . \tag{2}$$

Following Chan et al. [8, 10], we refer to a consecutive sequence of blocks in A as a *span*. When it does not coincide with the start (respectively, end) of a query range, we refer to the first (respectively, last) block that intersects the query range as the *prefix* (respectively, *suffix*). For any span S_Q among the $\Theta((n/t')^2)$ possible spans of small blocks, we define S_{big}, S_{small}, S_L, and S_R, as follows (see Figure 1):

- S_{big}: the unique minimal span of big blocks containing S_Q,
- S_L (prefix): the leftmost big block in S_{big},
- S_R (suffix): the rightmost big block in S_{big}, and
- S_{small} (span): the span of big blocks obtained by removing S_L and S_R from S_{big}.
- S_L is divided into S_1 (outside S_Q) and S_2 (inside S_Q).
- S_R is divided into S_3 (inside S_Q) and S_4 (outside S_Q).

Suppose $S_{big} = A[i : j]$; hence $S_{small} = A[i+t : j-t]$ and $S_Q = A[i_Q : j_Q]$. In Sections 3.2 and 3.3 we show how to encode the entries in $D_{t'}(\cdot,\cdot)$ and $E_{t'}(\cdot,\cdot)$ in $O(\log(t/t''))$ bits. In brief, we store an *approximate index* and *approximate frequency* for each entry and decode the exact values at query time.

3.2 Encoding and Decoding of $D_{t'}(\cdot,\cdot)$

We denote the least frequent element in S_Q by π and its frequency in S_Q by f_π. We consider three cases based on the indices at which π occurs in S_{big} as follows. The case that applies to any particular span can be indicated by two bits, hence $O(2(n/t')^2)$ bits in total. We use the same notation for representing a span as for the set of distinct elements within it.

Case 1: π is present in $S_L \cup S_R$ but not in S_{small}. As explicit storage of π is costly, we store the approximate index at which π occurs in $S_L \cup S_R$, and the approximate value of f_π, in $O(\log(t/t''))$ bits. Later we show how to decode π and f_π in $O(t'')$ time using the stored values.

The approximate value of f_π can be encoded using the following observations. We have $|S_L \cup S_R| \leq 2t$. Therefore $f_\pi \in [1, 2t]$. Explicitly storing f_π requires $\log(2t)$ bits. However, an approximate value of f_π (with an additive error at most of t'') can be encoded in fewer bits. Observe that $t''\lfloor f_\pi/t''\rfloor \leq f_\pi < t''\lfloor f_\pi/t''\rfloor + t''$. Therefore the value $\lfloor f_\pi/t''\rfloor \in [0, 2t/t'')$ can be stored using $O(\log(t/t''))$ bits and accessed in $O(1)$ time. The approximate location of π is a reference to a micro block within $S_L \cup S_R$ (among $2t/t''$ micro blocks) which contains π and whose index can be encoded in $O(\log(t/t''))$ bits. There can be many such micro blocks, but we choose one carefully from among the following possibilities:

- the rightmost micro block in S_1 which contains π,
- the leftmost micro block in S_2 which contains π,
- the rightmost micro block in S_3 which contains π, and
- the leftmost micro block in S_4 which contains π.

Next we show how to decode the exact values of π and f_π. Consider the case when the micro block (say B_m) containing π is in S_1. First initialize π' to any arbitrary element and f'_π to τ (an approximate value of f_π), such that $\tau - t'' \leq f_\pi < \tau$. Upon terminating the following algorithm, we obtain the exact values of π and f_π as π' and f'_π respectively. Scan the elements in B_m from left to right and let k denote the current index.

While k is an index in B_m, do:

1. If the second occurrence of $A[k]$ in $A[k : n-1]$ is in S_1, then go to Step 1 with $k \leftarrow k+1$.
2. If the $(f'_\pi + 1)$st occurrence of $A[k]$ in $A[k : n-1]$ is in S_Q, then go to Step 1 with $k \leftarrow k+1$.
3. Set $f'_\pi \leftarrow f'_\pi - 1$, $\pi' \leftarrow A[k]$, and go Step 2.

This algorithm finds the rightmost occurrence of π within B_m, i.e., the rightmost occurrence of π before the index i_Q. Correctness can be proved via induction as follows: after initializing π' and f'_π, at each step we check whether the element $A[k]$ is a least frequent element in S_Q among all the elements in B_m which we have seen so far. Step 1 discards the position k if the rightmost occurrence of $A[k]$ in B_m is not at k, because we will see the same element eventually. Note that if the rightmost occurrence of $A[k]$ in B_m is at the position k, then the frequency of the element $A[k]$ in $S_Q = A[i_Q : j_Q]$ is exactly one less than its frequency in $A[k : j_Q]$. Using this property, we can check in $O(1)$ time whether the frequency of $A[k]$ in S_Q is less than f'_π (Step 2). If so, we update the current best answer π' by $A[k]$ and compute the exact frequency of $A[k]$ in S_Q in Step 3. We scan all elements in B_m and on completion the value stored at π' represents the least frequent element in S_Q among all elements present in B_m. Since π is present in B_m, π is the same as π', and $f_\pi = f'_\pi$. By Lemma 2, each step takes constant time. Since $\tau - f_\pi \leq t''$, the total time is proportional to $|B_m| = t''$, i.e., $O(t'')$ time.

The remaining three cases, in which B_m is within S_2, S_3, and S_4, respectively, can be analyzed similarly.

Case 2: π is present in $S_L \cup S_R$ and in S_{small}. The approximate position of π is encoded as in Case 1. In this case, however, f_π can be much larger than $2t$. Observe that $\alpha \leq f_\pi \leq \alpha + 2t$, where α is the frequency of the least frequent element in S_{small}, which is already stored and can be retrieved in $T(t)$ time. Therefore, an approximate value $f_\pi - \alpha$ (with an additive error of at most t'') can be stored using $O(\log(t/t''))$ bits and decoded in $T(t) + O(1)$ time. The approximate location of π among the four possibilities as described in Case 1 is also maintained. By the algorithm above we can decode π and f_π in $T(t) + O(t'')$ time.

Case 3: π is present in S_{small} but in neither S_L nor S_R. Since π is the least frequent element in S_Q, and does not appear in $S_L \cup S_R$, it is the least frequent element in S_{small} that does not appear in $S_L \cup S_R$. This implies π is the least frequent element in S_{big} that does not appear in $S_L \cup S_R$ (which is precomputed as stored). Therefore the time required for decoding the values of π and f_π is $T(t) + O(1)$. That completes the proof of the following lemma.

Lemma 4 *The table $D_{t'}(\cdot,\cdot)$ can be stored using $O((n/t')^2 \log(t/t''))$ bits in addition to $S(t)$ and any value within it can be decoded in $T(t) + O(t'')$ time.*

3.3 Encoding and Decoding of $E_{t'}(\cdot,\cdot)$

Let ϕ denote the least frequent element in S_Q that does not appear in the leftmost and rightmost small blocks in S_Q and let f_ϕ denote its frequency in S_Q. As before, we consider three cases for the indices at which ϕ occurs in S_{big}. The case that applies to any particular span can be indicated by 2 bits, hence $O((n/t')^2 \times 2)$ bits in total for any single given value of t'.

For each small block (of size t') we maintain a hash table that can answer whether a given element is present within the small block in $O(1)$ time. We can maintain each

hash table in $O(t')$ bits for an overall space requirement of $O(n)$ bits for any single given value of t', using perfect hash techniques such as those of Schmidt and Siegel [39], Hagerup and Tholey [27], or Belazzougui et al. [1].

Case 1: ϕ is present in $S_L \cup S_R$ but not in S_{small}. In this case, $f_\phi \in [1, 2t]$, and its approximate value and approximate position (i.e., the relative position of a small block) can be encoded in $O(\log(t/t''))$ bits. Encoding is the same as the encoding of π in Case 1 of $D_{t'}(\cdot,\cdot)$. For decoding we modify the algorithm for $D_{t'}(\cdot,\cdot)$ to use the hash table for checking that $A[k]$ is not present in the first and last small blocks of S_Q. The decoding time can be bounded by $O(t'')$.

Case 2: ϕ is present in $S_L \cup S_R$ and in S_{small}. The approximate position of ϕ is stored as in Case 1. The encoding of f_ϕ is more challenging. Let α denote the frequency of the least frequent element in S_{small}, which is already stored and can be retrieved in $T(t)$ time. If $f_\phi > \alpha + 2t$, the element ϕ cannot be the least frequent element of any span S, where S contains S_{small} and is within S_{big}. In other words, ϕ is useful if and only if $f_\phi \leq \alpha + 2t$. Moreover, $f_\phi \geq \alpha$. Therefore we store the approximate value of f_ϕ if and only if it is useful information, and in such cases we can do it using only $O(\log(t/t''))$ bits. Using similar arguments to those used before, the decoding time can be bounded by $T(t) + O(t'')$.

Case 3: ϕ is present in S_{small} but in neither S_L nor S_R. Since ϕ is the least frequent element in S_Q that does not appear in the leftmost and rightmost small blocks in S_Q, and does not appear in $S_L \cup S_R$, it is the least frequent element in S_Q that does not appear in $S_L \cup S_R$. Therefore, ϕ it is the least frequent element in S_{small} (as well as S_{big}) that does not appear in $S_L \cup S_R$ (which is precomputed as stored). Hence ϕ and f_ϕ can be retrieved in $T(t) + O(1)$ time. That completes the proof of the following lemma.

Lemma 5 *The table $E_{t'}(\cdot,\cdot)$ can be encoded in $O(n + (n/t')^2 \log(t/t''))$ bits in addition to $S(t)$ and any value within it can be decoded in $T(t) + O(t'')$ time.*

By applying Lemma 3 with carefully chosen block sizes, followed by Lemma 1 for the final query on a range of individual elements, we show the following result.

Theorem 2 *Given any array $A[0:n-1]$, there exists an $O(n)$-word space data structure that supports range least frequent element queries on A in $O(\sqrt{n/w})$ time, where $w = \Omega(\log n)$ is the word size.*

Proof Let $t_h = \log^{(h)} n \sqrt{n/w}$ and $t''_h = \sqrt{n/w}/\log^{(h+1)} n$, where $h \geq 1$. Then by applying Lemma 3 with $t = t_h$, $t' = t_{h+1}$, and $t'' = t''_h$, we obtain the following:

$$S(t_{h+1}) = S(t_h) + O\big(n + (n/t_{h+1})^2 \log(t_h/t''_h)\big) \in S(t_h) + O(nw/\log^{(h+1)} n)$$
$$T(t_{h+1}) = T(t_h) + O(t''_h) \in T(t_h) + O(\sqrt{n/w}/\log^{(h+1)} n)\,.$$

By storing D_{t_1} and E_{t_1} explicitly, we have $S(t_1) \in O(n)$ bits and $T(t_1) \in O(1)$. Applying Lemma 1 to $\log^* n$ levels of the recursion gives $t_{\log^* n} = \sqrt{n/w}$ and

$$S(\sqrt{n/w}) \in O\left(nw \sum_{h=1}^{\log^* n} \frac{1}{\log^{(h)} n}\right) = O(nw)$$
$$T(\sqrt{n/w}) \in O\left(\sqrt{n/w} \sum_{h=1}^{\log^* n} \frac{1}{\log^{(h)} n}\right) = O(\sqrt{n/w}). \quad \square$$

4 Path Mode Queries on Trees

In this section, we generalize the range frequency query data structures to apply to trees (path mode query).

4.1 Preliminaries

The time bound of Chan et al. [10] for range mode queries on arrays depends on the ability to answer a query of the form "is the frequency of element $A[i]$ in the range $A[i:j]$ greater than k?" in constant time. There is no obvious way to generalize the data structure for such queries on arrays to apply to trees. Instead, we use an exact calculation of path frequency, not just whether it is greater than k.

Consider two nodes i and j in a tree, with common ancestor a, as shown in Figure 2A. The frequency of an element x in the path from i to j is the frequency from i to a plus the frequency from j to a. If at every node we store the frequency of that node's value on the path to the root, then we can compute the frequency in the path from any i to any j by the following procedure:

- find the nearest common ancestor a of i and j;
- find nearest ancestors of i, j, and a that have value x;
- find the frequencies of x from all those ancestors to the root;
- the frequency of x on the path from i to j is the sum of its frequencies from i and j to the root, minus twice its frequency from a to the root, plus b, where $b = 1$ if the value stored at a is x, and $b = 0$ otherwise.

We will perform these operations in time $O(\log\log n)$ with an $O(n)$-space data structure.

Finding the nearest common ancestor is a constant-time operation with an $O(n)$-space data structure, by well-known results such as those of Bender et al. [6]. The next step is to solve the *colored nearest ancestor* problem: in a tree where each node is assigned a color, given a color x and a node i find the nearest ancestor of i that has color x.

Lemma 6 *There exists an $O(n)$-space data structure that supports colored nearest ancestor queries on trees in $O(\log\log n)$ time.*

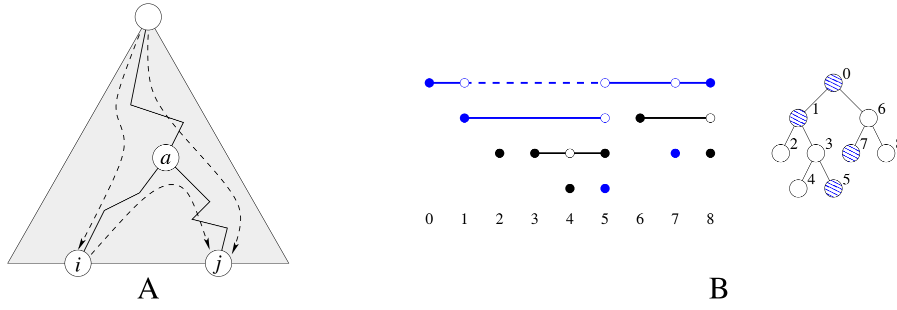

Fig. 2 **A.** The frequency from i to j is the sum of the frequencies from i to the root and j to the root, minus twice that from a to the root. **B.** A collection of segments representing nearest distinct ancestors.

Proof Suppose we wish to find the nearest ancestor i' of color x for a node i. If the nodes are indexed according to a pre-order traversal, then the descendants of a given node are exactly the nodes in one interval of indices. As we increase i from 1 to n, i' can only change at an endpoint of the interval corresponding to an x-colored node. There are twice as many such endpoints as there are x-colored nodes. For fixed x, finding the nearest x-colored ancestor of i reduces to finding the last endpoint before i. That is the predecessor problem on integers in the range $\{1,\ldots,n\}$. We can answer predecessor queries by well-known methods such as those of van Emde Boas [20] in $O(\log\log n)$ time, using a data structure of size linear in the number of nodes of color x. Storing one predecessor structure for each distinct color gives an overall space bound of $O(n)$. □

An improved result on colored nearest ancestor queries can be obtained as follows:

Lemma 7 *There exists an $(n(H+4)+o(nH+n))$-bit data structure that supports colored nearest ancestor queries on a tree T in $O(\log\log_w \sigma)$ time, where H is the entropy of the distribution of colors in T, σ is the number of distinct colors in T, and w is the word size.*

Proof The data structure consists of the following components:

1. a balanced-parenthesis representation of the tree T;
2. a sequence $S[1:n]$, where $S[i]$ is the color of node i (node 1 is the root); and
3. a balanced-parenthesis representation of T_c for every color c, where T_c is a tree obtained by retaining only those nodes in T whose color is c, and the parent of any node i in T_c is same as the nearest ancestor of i in T with color c. While constructing T_c, we assume the color of the root of T is c.

We use a constant-time parent operation on balanced-parenthesis representations, as described by Sadakane [38]. Also, we represent S in $nH+o(nH+n)$ bits, such that rank, select, and access on S can be supported in $O(\log\log_w \sigma)$ time [3]. To find the nearest ancestor i' of color c for a node i, if such an ancestor exists, let j be the index of the last occurrence of c in S before i; that is, the greatest integer j such that $1 \leq j < i$ and $S[j]=c$. Observe that the c-colored nearest ancestor i' of i is either j, or

the c-colored nearest ancestor of j. The integer j can be computed using a constant number of rank and select operations on S. If j is an ancestor of i, then j is the answer to the query. Otherwise, we can find the node in T_c corresponding to j in T using the balanced-parenthesis representation; find its parent; and then find the corresponding node j' in T using an access operation on S. If the color of j' is c, then j' is the answer to the query. The only remaining case is the j' is the root and not of color c, in which case i has no ancestor of color c. □

Given colored nearest ancestor queries, it is then straightforward to compute path frequencies.

Lemma 8 *There exists an $O(n)$-space data structure that supports path frequency queries on trees in $O(\log\log n)$ time.*

Proof Given node indices i and j and an element value x, we first use an $O(n)$-space nearest common ancestor data structure to find the nearest common ancestor a of i and j. That is a constant-time operation. Then we use the data structure of Lemma 6 to find i', j', and a', respectively the nearest x-colored ancestors of i, j, and a. At every node we store the frequency of that node's value on the path from it to the root. The desired path frequency is then the sum of these frequency values stored at i' and j', minus twice that stored at a', using zero for any x-colored ancestors found not to exist and adding one more if a' itself has color x. The time bound is $O(\log\log n)$ because of the colored nearest ancestor queries; the other calculations are constant time. □

We must also generalize to trees the idea of splitting an array into blocks. The following lemma describes a scheme for selecting some nodes in T as marked nodes, which split the tree into blocks over which we can apply the same kinds of block-based techniques that were effective in the array versions of the problems.

Lemma 9 *Given a tree T with n nodes and an integer $t < n$ which we call the blocking factor, we can choose a subset of the nodes, called the* marked *nodes, such that:*

- *at most $O(n/t)$ nodes are marked;*
- *the lowest common ancestor of any two marked nodes is marked; and*
- *the path between any two nodes contains $\leq t$ consecutive unmarked nodes.*

Proof For any node u, let $\mathrm{size}(u)$ represent the number of nodes in the subtree rooted at u. We mark nodes according to the following rules:

- mark any node u where $\mathrm{size}(u) \geq t$ and $\mathrm{size}(v) < t$ for every child v of u;
- mark the lowest common ancestor of any two marked nodes; and
- as long as there is an unmarked node v with a descendant u such that $\mathrm{size}(v) - \mathrm{size}(u) \geq t$, and v is the lowest such node in the tree, mark v.

There are $O(n/t)$ marked nodes. The first rule marks at most n/t nodes because there must be t descendants (including the marked node itself) for each marked node. Suppose after applying the first rule we remove all unmarked leaves, recursively until none remains. The remaining tree has only marked leaves, and all the nodes marked by the second rule must correspond to non-unary nodes in this remaining tree. There

can be no more of those than there are leaves, which correspond to the n/t nodes marked by the first rule.

The third rule may mark $O(n/t)$ additional nodes because each time it marks one, there are at least t possible values for u eliminated from further consideration. No more nodes need be marked under a re-application of the second rule, because if v has a descendant marked under the first rule, then any lowest common ancestors between v and other marked nodes would already be marked because of v's descendant, and if v does not have a descendant marked under the first rule, then v cannot meet the minimum size subtree condition to be chosen by the third rule. Therefore at most $O(n/t)$ nodes are marked overall, and the lowest common ancestor of two marked nodes must be marked by definition.

Given this set of marked nodes, suppose there were two nodes connected by a path containing a run of more than $2t$ consecutive unmarked nodes. Let u and v be the start and end of the run of unmarked nodes and let w be their lowest common ancestor. Then $\text{size}(w) - \text{size}(u) > t$ or $\text{size}(w) - \text{size}(v) > t$, and so the third marking rule would have forced one of the nodes on the path from u to w or from w to v to be marked. By contradiction, the longest possible run of unmarked nodes is of length $O(t)$. □

4.2 A Simple Data Structure for Path Mode Query

A simple path mode data structure follows naturally: we store the answers explicitly for all pairs of marked nodes, then use the data structure of Lemma 8 to compute exact frequencies for a short list of candidate modes. We let the blocking factor be a parameter, to support later use of this as part of a more efficient data structure.

Lemma 10 *For any blocking factor t, if we can answer path mode queries between marked nodes in time $T(t)$ with a data structure of $S(t)$ bits, then we can answer path mode queries between any nodes in time $T(t) + O(t \log\log n)$ with a data structure of $S(t) + O(n \log n)$ bits.*

Proof As in the array case considered by Chan et al. [10], we can split the query path into a prefix of size $O(t)$, a span with both endpoints marked, and a suffix of size $O(t)$ using Lemma 9 (see Figure 1). The mode of the query must either be the mode of the span, or it must occur within the prefix or the suffix. We find the mode of the span in $T(t)$ time by assumption, and compute its frequency in $O(\log\log n)$ time using the data structure of Lemma 8. Then we also compute the frequencies of all elements in the prefix and suffix, for a further time cost of $O(t \log\log n)$. The result follows. □

Setting $t = \sqrt{n}$ and using a simple lookup table for the marked-node queries gives $O(\sqrt{n} \log\log n)$ query time with $O(n)$ words of space.

4.3 A Faster Data Structure for Path Mode Query

To improve the time bound by an additional factor of $\sqrt{w}$, we derive the following lemma and apply it recursively.

Lemma 11 *For any blocking factor t, given an $S(t)$-bit data structure that supports path mode queries between marked nodes in time $T(t)$, there exists an $S(t')$-bit data structure that supports path mode queries between marked nodes for blocking factor t' in time $T(t') = T(t) + O(t'' \log\log n)$, where $S(t') = S(t) + O(n + (n/t')^2 \log(t/t''))$ and $t > t' > t''$.*

Proof Assume the nodes in T are marked based on a blocking factor t using Lemma 9, and the mode between any two marked nodes can be retrieved in $T(t)$ time using an $S(t)$-bit structure. Now we are interested in encoding the mode corresponding to the path between any two nodes i' and j', which are marked based on a smaller blocking factor t'. Note that there are $O((n/t')^2)$ such pairs. The tree structure along with this new marking information can be maintained in $O(n)$ bits using succinct data structures [38]. Where i and j are the first and last nodes in the path from i' to j', marked using t as the blocking factor, the path between i' and j' can be partitioned as follows: the path from i' to i, which we call the *path prefix*; the path from i to j; and the path from j to j', which we call the *path suffix* (see Figure 1). The mode in the path from i' to j' must be either (i) the mode of i to j path or (ii) an element in the path prefix or path suffix.

In Case (i), the answer is already stored using $S(t)$ bits and can be retrieved in $T(t)$ time. Case (ii) is more time-consuming. Note that the number of nodes in the path prefix and path suffix is $O(t)$. In Case (ii) our answer must be stored in a node in the path prefix which is $k < t$ nodes away from i', or in a node in the path suffix which is $k < t$ nodes away from j'. Hence an approximate value of k (call it k', with $k < k' \leq k + t''$) can be maintained in $O(\log(t/t''))$ bits. In order to obtain a candidate list, we first retrieve the node corresponding to k' using a constant number of level ancestor queries (each taking $O(1)$ time [38]) and its $O(t'')$ neighboring nodes in the i' to j' path. The final answer can be computed by evaluating the frequencies of these $O(t'')$ candidates using Lemma 8 in $O(t'' \log\log n)$ overall time. □

The following theorem is our main result on path mode query.

Theorem 3 *Given any tree T with n nodes, there exists an $O(n)$-word space data structure that supports path mode queries on T in $O(\log\log n \sqrt{n/w})$ time.*

Proof Let $t_h = \log^{(h)} n \sqrt{n/w}$ and $t''_h = \sqrt{n/w} / \log^{(h+1)} n$, where $h \geq 1$. Then by applying Lemma 11 with $t = t_h$, $t' = t_{h+1}$, and $t'' = t''_h$, we obtain the following:

$$S(t_{h+1}) = S(t_h) + O\big(n + (n/t_{h+1})^2 \log(t_h/t''_h)\big) \in S(t_h) + O(nw/\log^{(h+1)} n)$$
$$T(t_{h+1}) = T(t_h) + O(t''_h \log\log n) \in T(t_h) + O(\log\log n \sqrt{n/w}/\log^{(h+1)} n)\,.$$

By storing D_{t_1} and E_{t_1} explicitly, we have $S(t_1) \in O(n)$ bits and $T(t_1) \in O(1)$. Applying Lemma 10 to $\log^* n$ levels of the recursion gives $t_{\log^* n} = \sqrt{n/w}$ and

$$S(\sqrt{n/w}) \in O\left(nw \sum_{h=1}^{\log^* n} \frac{1}{\log^{(h)} n} \right) = O(nw)$$
$$T(\sqrt{n/w}) \in O\left(\log\log n \sqrt{n/w} \sum_{h=1}^{\log^* n} \frac{1}{\log^{(h)} n} \right) = O(\log\log n \sqrt{n/w})\,. \quad □$$

5 Path Least Frequent Element Queries on Trees

In this section we consider the problem of finding path least frequent elements on trees, for which we present $O(n)$-space data structures that support queries in $O(\log\log n\sqrt{n/w})$ time.

Like our data structure for range least frequent element queries on arrays described in Section 3, this data structure divides the tree into blocks, storing results for queries on block boundaries, and then doing some work at query time to compute the query on individual nodes based on the stored result for nearby block boundaries. See Lemma 9.

The marked nodes provide the boundaries for splitting the tree into blocks, at which point we can apply similar techniques to those used for the case of arrays.

Lemma 12 *Given a tree T with n nodes and an integer $t < n$ and some nodes marked by the procedure of Lemma 9, we can partition the nodes of T into $O(n/t)$* blocks *each of size $O(t)$, such that for any nodes u and v whose lowest marked ancestors are u' and v', the path from u to v consists of u, $O(t)$ unmarked nodes in the block of u, u', the path from u' to v', v', $O(t)$ unmarked nodes in the block of v, and finally v; or possibly just $O(t)$ nodes in the block shared by u and v if that is the same block.*

Proof From a marked node u, consider the nodes reachable by paths that exit u through the link to its parent and contain no marked nodes other than u. The set of all nodes reachable by such paths from u (including u itself) is of size $O(t)$, because if it were larger, the marking rules would force another node to be marked within the set. Any node is in at most one such set, because otherwise there would be two marked nodes whose least common ancestor is unmarked, again contradicting the marking rules. For any node v reachable from some marked node u by a path that exits u through the link to its parent and contains no other marked node, we assign v to a *block* B_u.

Any node v not reachable in this way from any u must be near the bottom of the tree, with a marked ancestor that has no marked descendants. We let w be the lowest marked ancestor of such a v and assign v to a block B'_w. Again by the marking rules, there can be at most $O(t)$ nodes assigned to each B'_w. And since at most two blocks are associated with each marked node, we have $O(n/t)$ blocks. □

The query time achieved by the data structure of Chan et al. [10] for range least frequent element queries on arrays depends on the ability to answer a query of the form "is the frequency of element $A[i]$ in the range $A[i:j]$ greater than k?" in constant time (see Lemma 2). There is no obvious way to generalize the data structure for such queries on arrays to apply to trees. Instead, we use an exact calculation of path frequency (not just whether it is greater than k), which takes $O(\log\log n)$ time per query by Lemma 8.

Theorem 4 *Given any tree T with n nodes, there exists an $O(n)$-word space data structure that supports path least frequent element queries on T in $O(\sqrt{n}\log\log n)$ time.*

Proof Let $t = \sqrt{n}$ and partitions the tree into blocks according to Lemma 12. For each pair of blocks, there is a unique path between the blocks that is contained in every path from any element of the first block to any element of the second block. (This path is possibly empty if the blocks are not distinct.) We store a table that records, for each pair of blocks, the element which is least frequent in the path between two blocks, and is absent in two blocks. This table requires $O(n)$ words of space.

Then for a path from a node i to a node j, which are contained in blocks B_i and B_j, respectively, the least frequent element on the path between i and j must either be the one recorded in the table for this pair of blocks, or be an element of B_i or B_j. That gives $O(\sqrt{n})$ candidates; by computing the exact frequency of each one in $O(\log\log n)$ time each using Lemma 8, we can find the least frequent in $O(\sqrt{n}\log\log n)$ time. □

The query time can be improved by a factor of $\sqrt{w}$ using our block-shrinking technique, and the following result can be obtained.

Theorem 5 *Given any tree T with n nodes, there exists an $O(n)$-word space data structure that supports path least frequent element queries on T in $O(\log\log n\sqrt{n/w})$ time.*

6 Path α-Minority Queries on Trees

An *α-minority* in a multiset A, for some $\alpha \in [0,1]$, is an element that occurs at least once and as no more than α proportion of A. If there are n elements in A, then the number of occurrences of the α-minority in A can be at most αn. Elements in A that are not α-minorities are called *α-majorities*. Chan et al. studied α-minority range queries in arrays [10]; here, we generalize the problem to path queries on trees. In general, an α-minority is not necessarily unique; given a query consisting of a pair of tree node indices and a value $\alpha \in [0,1]$ (specified at query time), our data structure returns one α-minority, if at least one exists, regardless of the number of distinct α-minorities. As in the previous sections, we can compute path frequencies in $O(\log\log n)$ time (Lemma 8); then a data structure similar to the one for arrays gives us distinct elements within a path in constant time per distinct element. Combining the two gives a bound of $O(\alpha^{-1}\log\log n)$ time for α-minority queries.

As discussed by Chan et al. for the case of arrays [10], examining α^{-1} distinct elements in a query range allows us to guarantee either that we have examined an α-minority, or that no α-minority exists. So we construct a data structure based on the hive graph of Chazelle [11] for the k-nearest distinct ancestor problem: given a node i, find a sequence $a_1, a_2, \ldots$ of ancestors of i such that $a_1 = i$, a_2 is the nearest ancestor of i distinct from a_1, a_3 is the nearest ancestor of i distinct from a_1 and a_2, and so on. Queries on the data structure return the distinct ancestors in order and in constant time each.

Lemma 13 *There exists an $O(n)$-space data structure that supports k-nearest distinct ancestor queries on trees in $O(k)$ time, returning them in nearest-to-furthest order in $O(1)$ time each, so that k need not be specified.*

Proof Suppose the nodes of the tree are indexed according to a pre-order traversal. Any node with index i is an ancestor of exactly those nodes whose indices are in the interval $[i, j]$ for some index j. Let d be the depth of i. For each node i in the tree, consider the line segment on the Euclidean plane from coordinates $(i, -d)$ to $(j, -d)$. Figure 2B shows an example of such segments. A vertical ray directed upward through the collection of segments at the x-coordinate corresponding to the node's index will intersect the segments for that node's ancestors, in order from the node to the root. However, such a query will find all the ancestors, including any with duplicate colors.

Suppose that as shown by the dashed lines and open circles in the figure, we delete any part of a segment that is covered by some other segment of the same color below it on the plane. The case of deleting a single point can be handled by opening up a small gap, bearing in mind that only queries at integer coordinates are relevant. This deletion procedure increases the number of segments in the collection, but only by a factor of at most two, because each endpoint of a deeper segment creates at most one new endpoint inside a shallower segment, in the segment corresponding to its nearest same-color ancestor. Therefore the number of segments remains linear to the size of the tree. Vertical ray queries on the resulting collection of segments return k-nearest distinct ancestor results.

We can build a hive graph on these segments, as described by Chazelle [11], and answer queries in the required time. The query procedure requires an initial point location query, which might be more than constant time in general, but because we only have n distinct query rays, we can precompute the point-location queries and store them in linear space for constant-time lookup. The remainder of the query process is constant time per ancestor returned. □

Lemmas 8 and 13 give the following theorem.

Theorem 6 *Given any tree T with n nodes, there exists an $O(n)$-word space data structure that supports path α-minority queries on T in $O(\alpha^{-1} \log\log n)$ time, where α is specified at query time.*

Proof We construct the data structures of Lemma 8 and Lemma 13, both of which use linear space. To answer a path α-minority query between two nodes i and j, we find the α^{-1} nearest distinct ancestors (or as many as exist, if that is fewer) above each of i and j. That takes α^{-1} time. If an α-minority exists between i and j, then one of these candidates must be an α-minority. We can test each one in $O(\log\log n)$ using the path frequency data structure, and the result follows. □

7 Path Top-k Queries on Trees

Suppose each node in a tree T of n nodes is assigned a color from the alphabet $\{0, \ldots, \sigma - 1\}$. Let $T[a : b]$ denote the unique path connecting the nodes with preorder ranks a and b in T. Then the *tree path top-k color query problem* is to report, given indices a and b and a count k, the k greatest distinct color values to occur in $T[a : b]$, in descending order. We will prove the following result.

Theorem 7 *Given any tree T with n nodes, there exists an $O(n)$-word space data structure that supports path top-k color queries on T in $O(k)$ time, where k is specified at query time.*

We begin by defining some useful concepts. Let T be a tree with n nodes. We define the *size* of a node v to be the number of leaves in the subtree rooted at v. A *heavy path* of a subtree $T' \subseteq T$ is a root-to-leaf path in T', in which each node v on the path has size at least as large as that of its largest sibling. Let the root of a heavy path be its highest node, that is, closest to the root of T. A *heavy path decomposition* of the tree T is a partition of the edges of T induced by recursively constructing the heavy path of each subtree that branches off the heavy path; thus, edges in T are partitioned into disjoint heavy paths. Each leaf node belongs to a unique path in the heavy path decomposition, and each heavy path contains exactly one leaf. Therefore the number of heavy paths is equal to the number of leaves. This decomposition will allow us to subdivide query paths into a small number of components by the following lemma.

Lemma 14 (Sleator and Tarjan [41]) *Any path from the root to a leaf in T intersects at most* $\log n$ *paths of the heavy path decomposition.*

We will also use a data structure of Navarro and Nekrich for three-sided two-dimensional top-k queries [36]. Given a set of n points on an $n \times n$ grid, each having a weight, this data structure can report the k points of greatest weight in a query region of the form $[a,b] \times [0,h]$, in $O(h+k)$ time, where a, b, h, and k are specified at query time. Furthermore, it reports the points in order of decreasing weight, and it reports them online, that is, $O(h)$ time to report the first point and $O(1)$ time per point for subsequent points.

Any tree path $T[a:b]$ can be divided into two overlapping paths $T[a:z]$ and $T[z:b]$, where z is the lowest common ancestor (LCA) of a and b. Since $O(n)$-space data structures exist to support $O(1)$-time LCA queries (e.g., [5]), if we can answer tree path top-k color queries on these two paths in $O(k)$ time, then we can merge the answers to answer such queries on arbitrary paths in the tree in the same time. Therefore, it suffices to solve the following restricted problem.

Problem 1 Preprocess T to efficiently answer tree path top-k color queries on paths of the form $T[a:z]$ where z is an ancestor of a.

We begin with a data structure providing $O(k \log n)$ query time, which is improved to $O(k + \log^{O(1)} n)$ time, and finally to optimal $O(k)$ query time. Each of these data structures can be stored in $O(n)$ space.

A tree path top-k query involves three constraints. Each element returned must be (1) on the query path; (2) among the top k; and (3) of a distinct color. In other words, each color must be reported, and counted towards the top k, only once, even if multiple instances of that color appear on the query path. We eliminate duplicates using an adaptation of Muthukrishnan's chaining approach to reporting distinct colors in array ranges [35].

Let $depth(i)$ denote the number of nodes on the path from the root to a node i. Let $chain(i)$ denote the depth of the lowest ancestor of i that has the same color as i, with

$chain(i) = 0$ if there is no such ancestor. If there exists at least one node with color c in $T[a : z]$, z being an ancestor of a, then there exists a unique node i in $T[a : z]$ with color c and $chain(i) < depth(z)$. Therefore, Problem 1 can be restated as follows:

Problem 2 Preprocess T to efficiently find, given a node a, one of its ancestors z, and a count k, the top k colors in decreasing order among the nodes $\{i \in T[a : z] \mid chain(i) < depth(z)\}$.

Given any node labelled by its preorder rank i, let $\phi(i)$ denote the root of the path containing i in the heavy path decomposition of T. Let ℓ_j denote the preorder rank of the jth leftmost leaf in T. We can transform T to another tree T', which is actually a path, by concatenating the paths $T[\ell_i, \phi(\ell_i)]$ for each i up to the number of leaves. Then we can define an array $A[1 : n]$ containing the colors of nodes in T', in order along the path starting from the root. The following property is ensured: any subpath of a path in the heavy path decomposition must correspond to a contiguous range of A.

We build the optimal array range top-k color query data structure of Karpinski and Nekrich on A, and store it using $O(n \log \sigma)$ bits [32]. From each node i in T, we store the index of the corresponding entry in A. The total space used is $O(n)$ words, assuming $\sigma \in O(n)$. Because heavy paths are contiguous in A, the special case in which both a and z are on the same heavy path can be handled optimally by first finding the corresponding range in A, and then performing a top-k color query on the array range top-k color query data structure.

For the case in which a and z are not on the same path of the decomposition, we map each node i in T to a weighted two-dimensional point (x_i, y_i) with weight w_i, letting w_i be the color of node i, x_i be the index in A corresponding to that node, and y_i be the number of paths of the heavy path decomposition intersected by the path from the root to $chain(i)$. We build the data structure of Navarro and Nekrich for three-sided two-dimensional top-k queries on these weighted points [36, Theorem 2.1]. Because $y_i \leq \log n$, the query time to return k points from this data structure is $O(\log n + k)$.

To answer a general tree path top-k color query, we first find the lowest common ancestor of the endpoints and split the query path into two queries of the form $T[a : z]$ with z an ancestor of a. Then for each of the two, we partition the query path $T[a : z]$ into at most $\log n$ disjoint subpaths $T[a_1 : \phi(a_1)]$, $T[a_2 : \phi(a_2)]$, $\ldots$, $T[a_r : z]$, where $a_1 = a$, a_i is the parent of $\phi(a_{i-1})$ for $i = 2, \ldots, r$, and $r \leq \log n$ is such that z is on the subpath $T[a_r : \phi(a_r)]$. This step takes only $O(r) \subseteq O(\log n)$ time, by consulting a stored copy of the heavy path decomposition.

The query $T[a_r : z]$ corresponds to a contiguous range in A, so we can find its top k colors in $O(k)$ time and merge them in $O(k)$ time with the top k colors for $T[a_1 : \phi(a_{r-1})]$. It only remains to query $T[a_1 : \phi(a_{r-1})]$ efficiently.

From the definition in Problem 2, we have that if a node i is included in the result for $T[a_1 : \phi(a_{r-1})]$ then $chain(i) < depth(\phi(a_{r-1}))$. In fact, it suffices to check that the number of heavy paths intersected by the path from the root of T to $chain(i)$ is less than the number intersected from the root to $\phi(a_{r-1})$. Since $\phi(a_{r-1})$ and its parent cannot be on the same path of the decomposition, any ancestor of it must be in a path nearer to the root.

Let h be the number of paths of the heavy path decomposition that are intersected by the path from $\phi(a_{r-1})$ to the root, and let $A[s_i : e_i]$ be the range of A associated with $T[a_i : \phi(a_i)]$. Then for each $i \in \{1,\ldots,r-1\}$, the weighted points $(x_j, y_j) \in [s_i, e_i] \times [0,h]$ correspond to nodes with distinct colors on the path $T[a_1 : \phi(a_{r-1})]$. The union of those lists would include the top k colors in $T[a_1 : \phi(a_{r-1})]$, but we must still merge the lists.

We issue $r-1$ simultaneous queries to the top-k geometric data structure, corresponding to the query regions $[s_i, e_i] \times [0,h]$ for $i = 1,\ldots,r-1$. The answers can be merged using a max-heap H with its size limited to at most $r-1 \in O(\log n)$ points. We insert the first point returned from each R_i into H. Then we repeat the following steps until we have reported k colors:

1. Extract the highest weighted point in H and report it.
2. If the reported point was from the query box R_i, then fetch the next highest weighted point from R_i and insert it into H.

Since the size of H is always $\log^{O(1)} n$, we can use an atomic heap, which can perform all heap operations in constant time in the Word RAM model [22]. Therefore, the number of heap operations, and the required time, can be bounded by $O(k+\log n)$. Each three-sided two-dimensional top-k query takes $O(\log n)$ time in addition to the number of points it returns. Therefore the total time for query is $O(k+\log^2 n)$, giving the following result.

Lemma 15 *There exists an $O(n)$-space data structure that supports tree path top-k color queries in $O(k+\log^2 n)$ time.*

For $k \in \Omega(\log^2 n)$, the query time above is optimal. We apply different techniques for the case $k \in o(\log^2 n)$. First, we identify a subset of the nodes in T, called the *marked* nodes, as follows. Let g denote an integer (whose value we specify later) called the *grouping factor*, and mark every node i in T such that $i \equiv 0 \pmod g$. Also mark the lowest common ancestor of any pair of marked nodes. This is a simplified version of the scheme introduced by Hon et al. [30] for identifying marked nodes in a suffix tree. It has the properties given in Lemma 9.

We mark nodes in T using $g = \log^3 n$. For every marked node i and every j such that j is an ancestor of i and j is the root of a path in the heavy path decomposition, we store explicitly a precomputed sorted list of the top $O(\log^2 n)$ colors on the path $T[i : j]$. The space is bounded by $O((n/\log^3 n)\log^2 n \log n) = O(n)$ words. Using these precomputed lists we can find the top k colors in $T[a : z]$, where a is a marked node and z is one of its ancestors, by splitting the query, as before, into $T[a_1 : \phi(a_{r-1})]$ and $T[a_r : z]$. The former is precomputed and the latter corresponds to a contiguous range of A. We can find the top k colors in both paths and merge the lists in $O(k)$ time, giving the following result.

Lemma 16 *There exists an $O(n)$-space data structure that supports tree path top-k color queries of the form $T[a : z]$ in $O(k)$ time, where a is a marked node and z is one of its ancestors.*

Next we must handle the case in which a is not a marked node. For any node i such that i is not marked but its parent is marked, define the *mini-tree* T^i to be the subtree rooted at i but excluding any descendants of the highest marked descendant of i, if any. By Lemma 9, T^i is of size $O(g)$. We choose a grouping factor $g' = \log^3 g \in \Theta(\log^3 \log n)$ which we use to mark nodes within each mini-tree, and build the data structure of Lemma 16 for each mini-tree. The total space is bounded by $O(n)$ because each node in T belongs to exactly one mini-tree. By querying from a to the root of its mini-tree, and from the parent of that root to z, and merging the results, we can answer top-k color queries of the form $T[a:z]$ in $O(k)$ time when a is marked in its mini-tree and z is its ancestor, even if a is not marked in T.

Finally, we generalize the optimal-time solution to arbitrary a. For any i not marked in T nor in the mini-tree that contains i, let j be the lowest marked node above i in the same mini-tree. The node j is at most $g' \in O(\log^3 \log n)$ nodes above i. Therefore the top k colors on the path $T[i:j]$, for all choices of i, can be stored in $O(n(\log^3 \log n)^2 \log\log\log n)$ bits: there are n choices of i, $O(\log^3 \log n)$ lists, each of length at most $O(\log^3 \log n)$, and only $O(\log\log\log n)$ bits are needed (as indices into the mini-block) to store the entries in the lists. That is a total of $o(n)$ words.

Then to answer an arbitrary query $T[a:b]$, we first split into two queries $T[a:z]$ and $T[b:z]$ with z the lowest common ancestor of a and b. We can find the top k colors from a to its lowest marked ancestor within its mini-tree, and from there to the root of the mini-tree, using the precomputed lists. From the lowest marked ancestor of a in T to the highest marked ancestor of a below z, we use Lemma 16; and from there to z we do an array range query in A. We do the same with the query $T[b:z]$. All these queries can be performed, and their results merged, in $O(k)$ time. This completes the proof of Theorem 7.

8 Discussion and Directions for Future Research

The asymptotic difference in query times for a range query on arrays and the corresponding path query on trees can often be eliminated. This is the case for range/path minimum [13], range/path selection [28,29,37], and range/path top-k queries (this paper, Section 7). At present, the query times for mode, least frequent element, and α-minority path queries on trees remain slower by a factor of $O(\log\log n)$. Can these gaps be closed? These last three data structures for path queries refer to Lemma 8, resulting in query time $O(\log\log n)$ times greater than the corresponding time on arrays. For arrays, Chan et al. [8] use $O(1)$-time range frequency queries for the case in which the element whose frequency is being measured is at an endpoint of the query range. Generalizing this technique to path queries on trees should allow each data structure's query time to be decreased accordingly.

Additional possible directions for future research include making these data structures succinct or dynamic. The path top-k query data structure presented in Section 7 requires $O(n)$ words of space. The range top-k query data structure of Karpinski and Nekrich [32] requires only $O(n \log \sigma)$ bits of space. It seems feasible that the space requirement for the path query data structure could be reduced to $O(n \log \sigma)$ bits, but this remains to be determined.

Acknowledgements

The authors thank the anonymous reviewers as well as Djamal Belazzougui for their helpful suggestions. Part of this work was done while the fourth author was visiting the University of Manitoba in July 2012 and February 2013.

References

1. Belazzougui, D., Botelho, F.C., Dietzfelbinger, M.: Hash, displace, and compress. In: Proc. ESA, *LNCS*, vol. 5757, pp. 682–693. Springer (2009)
2. Belazzougui, D., Gagie, T., Navarro, G.: Better space bounds for parameterized range majority and minority. In: Proc. WADS, *LNCS*, vol. 8037. Springer (2013)
3. Belazzougui, D., Navarro, G.: New lower and upper bounds for representing sequences. In: Proc. ESA, *LNCS*, vol. 7501, pp. 181–192. Springer (2012)
4. Belazzougui, D., Navarro, G., Valenzuela, D.: Improved compressed indexes for full-text document retrieval. J. Disc. Alg. **18**, 3–13 (2013)
5. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Proc. LATIN, *LNCS*, vol. 1776, pp. 88–94. Springer (2000)
6. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. J. Alg. **57**(2), 75–94 (2005)
7. Brodal, G.S., Gfeller, B., Jørgensen, A.G., Sanders, P.: Towards optimal range medians. Theor. Comp. Sci. **412**(24), 2588–2601 (2011)
8. Chan, T.M., Durocher, S., Larsen, K.G., Morrison, J., Wilkinson, B.T.: Linear-space data structures for range mode query in arrays. In: Proc. STACS, vol. 14, pp. 291–301 (2012)
9. Chan, T.M., Durocher, S., Larsen, K.G., Morrison, J., Wilkinson, B.T.: Linear-space data structures for range mode query in arrays. Theory Comp. Sys. (2013). To appear
10. Chan, T.M., Durocher, S., Skala, M., Wilkinson, B.T.: Linear-space data structures for range minority query in arrays. In: Proc. SWAT, *LNCS*, vol. 7357, pp. 295–306. Springer (2012)
11. Chazelle, B.: Filtering search: A new approach to query-answering. SIAM J. Comp. **15**(3), 703–724 (1986)
12. Davoodi, P., Raman, R., Rao, S.S.: Succinct representations of binary trees for range minimum queries. In: Proc. COCOON, *LNCS*, vol. 7434, pp. 396–407. Springer (2012)
13. Demaine, E.D., Landau, G.M., Weimann, O.: On Cartesian trees and range minimum queries. In: Proc. ICALP, *LNCS*, vol. 5555, pp. 341–353. Springer (2009)
14. Durocher, S.: A simple linear-space data structure for constant-time range minimum query. In: Proc. Conference on Space Efficient Data Structures, Streams and Algorithms (Munro Festschrift), vol. 8066, pp. 48–60 (2013)
15. Durocher, S., He, M., Munro, J.I., Nicholson, P.K., Skala, M.: Range majority in constant time and linear space. In: Proc. ICALP, *LNCS*, vol. 6755, pp. 244–255. Springer (2011)
16. Durocher, S., He, M., Munro, J.I., Nicholson, P.K., Skala, M.: Range majority in constant time and linear space. Inf. & Comp. **222**, 169–179 (2013)
17. Durocher, S., Munro, J.I., El-Zein, H., Thankachan, S.V.: Low space data structures for geometric range mode query. In: Proc. CCCG (2014)
18. Durocher, S., Shah, R., Skala, M., Thankachan, S.: Linear-space data structures for range frequency queries on arrays and trees. In: Proc. MFCS, *LNCS*, vol. 8087, pp. 325–336. Springer (2013)
19. Durocher, S., Shah, R., Skala, M., Thankachan, S.: Top-k color queries on tree paths. In: Proc. SPIRE, *LNCS*, vol. 8214, pp. 109–115. Springer (2013)
20. Emde Boas, P.v.: Preserving order in a forest in less than logarithmic time and linear space. Inf. Proc. Let. **6**(3), 80–82 (1977)
21. Fischer, J.: Optimal succinctness for range minimum queries. In: Proc. LATIN, *LNCS*, vol. 6034, pp. 158–169. Springer (2010)
22. Fredman, M.L., Willard, D.E.: Trans-dichotomous algorithms for minimum spanning trees and shortest paths. J. Comput. Syst. Sci. **48**(3), 533–551 (1994)
23. Gagie, T., He, M., Munro, J.I., Nicholson, P.: Finding frequent elements in compressed 2D arrays and strings. In: Proc. SPIRE, *LNCS*, vol. 7024, pp. 295–300. Springer (2011)

24. Gagie, T., Kärkkäinen, J., Navarro, G., Puglisi, S.J.: Colored range queries and document retrieval. Theor. Comput. Sci. **483**, 36–50 (2013)
25. Gagie, T., Puglisi, S.J., Turpin, A.: Range quantile queries: Another virtue of wavelet trees. In: Proc. SPIRE, *LNCS*, vol. 5721, pp. 1–6. Springer (2009)
26. Gfeller, B., Sanders, P.: Towards optimal range medians. In: Proc. ICALP, *LNCS*, vol. 5555, pp. 475–486. Springer (2009)
27. Hagerup, T., Tholey, T.: Efficient minimal perfect hashing in nearly minimal space. In: Proc. STACS, *LNCS*, vol. 2010, pp. 317–326. Springer (2001)
28. He, M., Munro, J.I., Zhou, G.: Path queries in weighted trees. In: Proc. ISAAC, pp. 140–149 (2011)
29. He, M., Munro, J.I., Zhou, G.: Succinct data structures for path queries. In: Proc. ESA, pp. 575–586 (2012)
30. Hon, W.K., Shah, R., Vitter, J.S.: Space-efficient framework for top-k string retrieval problems. In: Proc. FOCS, pp. 713–722 (2009)
31. Jørgensen, A.G., Larsen, K.D.: Range selection and median: Tight cell probe lower bounds and adaptive data structures. In: Proc. SODA, pp. 805–813 (2011)
32. Karpinski, M., Nekrich, Y.: Top-k color queries for document retrieval. In: Proc. SODA, pp. 401–411 (2011)
33. Krizanc, D., Morin, P., Smid, M.: Range mode and range median queries on lists and trees. In: Proc. ISAAC, *LNCS*, vol. 2906, pp. 517–526. Springer (2003)
34. Krizanc, D., Morin, P., Smid, M.: Range mode and range median queries on lists and trees. Nordic J. Comp. **12**, 1–17 (2005)
35. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: Proc. SODA, pp. 657–666 (2002)
36. Navarro, G., Nekrich, Y.: Top-k document retrieval in optimal time and linear space. In: Proc. SODA, pp. 1066–1077 (2012)
37. Patil, M., Shah, R., Thankachan, S.V.: Succinct representations of weighted trees supporting path queries. J. Disc. Alg. **17**, 103–108 (2012)
38. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Proc. SODA, pp. 134–149 (2010)
39. Schmidt, J.P., Siegel, A.: The spatial complexity of oblivious k-probe hash functions. SIAM J. Comput. **19**(5), 775–786 (1990)
40. Skala, M.: Array range queries. In: Proc. Conference on Space Efficient Data Structures, Streams and Algorithms (Munro Festschrift), vol. 8066, pp. 333–350 (2013)
41. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. J. Comp. Sys. Sci. **26**(3), 362–391 (1983)