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

Stephane Durocher[1], Rahul Shah[2],
Matthew Skala[1], and Sharma V. Thankachan[2]

[1] University of Manitoba, Winnipeg, Canada, {durocher,mskala}@cs.umanitoba.ca
[2] Louisiana State University, Baton Rouge, USA, {rahul,thanks}@csc.lsu.edu

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, or α-minority 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 range 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). Finally, we describe a data structure that supports range α-minority queries on trees in $O(\alpha^{-1}\log\log n)$ time, where $\alpha \in [0, 1]$ can be specified at query time.

1 Introduction

The *frequency*, denoted $\text{freq}_{A[i:j]}(x)$, of an element x in a multiset stored as an array $A[i : j]$ is the number of occurrences 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 $c \in A[i : j]$, $\text{freq}_{A[i:j]}(a) \geq \text{freq}_{A[i:j]}(c) \geq \text{freq}_{A[i:j]}(b)$. Finally, given $\alpha \in [0, 1]$, an *α-minority* of $A[i : j]$ is an element $d \in A[i : j]$ such that $1 \leq \text{freq}_{A[i:j]}(d) \leq \alpha|j-i+1|$. Conversely, d is an *α-majority* of $A[i : j]$ if $\text{freq}_{A[i:j]}(d) > \alpha|j-i+1|$.

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), for which a mode, least frequent element, or α-minority of $A[i : j]$ must be returned. Range queries generalize to trees, where they are called *path queries*: given a tree T 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.

* Work supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the National Science Foundation (NSF).

Krizanc et al. [12] 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. [3, 4] 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, where $w \in \Omega(\log n)$ is the word size in bits.

For range least frequent elements, Chan et al. [5] presented an $O(n)$-space data structure that supports queries in $O(\sqrt{n})$ time on arrays. Range mode and range least frequent queries on arrays appear to require significantly longer times than either range minimum or range selection queries; respective reductions from boolean matrix multiplication show that query times significantly lower than $\sqrt{n}$ are unlikely for either problem with linear space [3, 5]. Whereas an $O(n)$-space data structure that supports range mode queries on arrays in $o(\sqrt{n})$ time is known [3], the space reduction techniques applied to achieve the time improvement are not directly applicable to the setting of least frequent elements. Chan et al. [5] ask whether $o(\sqrt{n})$ query time is possible in an $O(n)$-space linear 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" [5, p. 11]. By applying different techniques, this paper presents 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.

Finally, the range α-majority query problem was introduced by Durocher et al. [7, 8], who 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. [9] and Chan et al. [5] 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. [5] described an $O(n)$-space data structure that supports queries in $O(\alpha^{-1})$ time, where α is specified at query time.

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 queries on arrays, supporting queries in $O(\sqrt{n/w})$ time. We then extend this data structure to the setting of trees. 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 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 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).

range query	input	previous best	new (this paper)
least frequent element	array	$O(\sqrt{n})$ [5]	$O(\sqrt{n/w})$
	tree	no previous result	$O(\log\log n\sqrt{n/w})$
mode	array	$O(\sqrt{n/w})$ [3, 4]	
	tree	$O(\sqrt{n}\log n)$ [11, 12]	$O(\log\log n\sqrt{n/w})$
α-minority	array	$O(\alpha^{-1})$ [5]	
	tree	no previous result	$O(\alpha^{-1}\log\log n)$

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

Finally, in Section 5 we present an $O(n)$-space data structure that supports path α-minority query on trees in $O(\alpha^{-1}\log\log n)$ time, where α is given at query time. Our contributions are summarized in Table 1.

We assume the Word RAM model of computation using words of size $w \in \Omega(\log n)$ bits, where n denotes the number of elements stored in the input 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 $(\log n)^2$ instead of $\log^2 n$.

2 Chan et al.'s Framework for Range Least Frequent Element Query 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. [5]. 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 [3, 5]. Chan et al. [5] prove the following result.

Theorem 1 (Chan et al. [5]). *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. [5], 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. [5]). *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 [5]. 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. [3]:

Lemma 2 (Chan et al. [3]). *Given an array $A[0 : n-1]$, there exists an $O(n)$-space data structure that returns the index of the q-th instance of $A[i]$ in $A[i : n-1]$ in $O(1)$ time for any $0 \leq i \leq n-1$ and any q.*

3 Faster Range Least Frequent Element Query on Arrays

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.

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 3.1). 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*

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

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

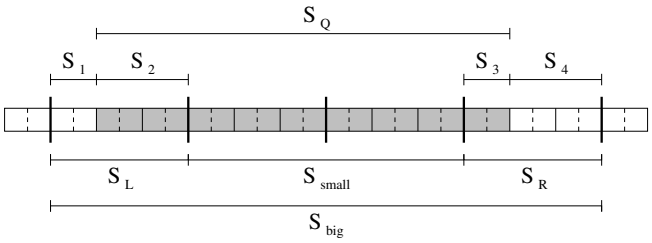

Fig. 1. illustration in support of Lemma 3

Following Chan et al. [3, 5], we call a consecutive sequence of blocks in A a *span*. For any span S_Q among the $\Theta((n/t')^2)$ possible spans of small blocks, we define $\mathcal{S}_{\text{big}}$, $\mathcal{S}_{\text{small}}$, $\mathcal{S}_L$, and $\mathcal{S}_R$, as follows (see Figure 1):

- $\mathcal{S}_{\text{big}}$: the unique minimal span of big blocks containing S_Q,
- $\mathcal{S}_L$: the leftmost big block in $\mathcal{S}_{\text{big}}$,
- $\mathcal{S}_R$: the rightmost big block in $\mathcal{S}_{\text{big}}$, and
- $\mathcal{S}_{\text{small}}$: the span of big blocks obtained by removing $\mathcal{S}_L$ and $\mathcal{S}_R$ from $\mathcal{S}_{\text{big}}$.
- $\mathcal{S}_L$ is divided into $\mathcal{S}_1$ (outside S_Q) and $\mathcal{S}_2$ (inside S_Q).
- $\mathcal{S}_R$ is divided into $\mathcal{S}_3$ (inside S_Q) and $\mathcal{S}_4$ (outside S_Q).

Let $S_{big} = A[i : j]$, hence $S_{small} = A[i + t : j - t]$ and $S_Q = A[i_Q : j_Q]$. In Sections 3.1 and 3.2 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.

3.1 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 2 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 $T(t) + O(1)$.

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.*

3.2 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 [14], Hagerup and Tholey [10], 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, π 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.

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:

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

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

$$\begin{aligned} S(\sqrt{n/w}) &\in O\left(nw \sum_{h=1}^{\log^* n} \frac{1}{\log^{(h)} n}\right) \qquad = O(nw) \\ T(\sqrt{n/w}) &\in O\left(\sqrt{n/w} \sum_{h=1}^{\log^* n} \frac{1}{\log^{(h)} n}\right) = O(\sqrt{n/w})\,. \end{aligned}$$

□

4 Path Frequency Queries on Trees

In this section, we generalize the range frequency query data structures to apply to trees (path mode query). The linear time bound of Chan et al. [5] for range mode queries on arrays depends on the ability to answer a query of the form "is the frequency of element x 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 the following lemma for an exact calculation of path frequency (not just whether it is greater than k). The proof is omitted due to space constraints.

Lemma 6. *Given any tree T of n nodes, there exists an $O(n)$-word data structure that can compute the number of occurrences of x on the path from i to j in $O(\log \log n)$ time for any nodes i and j in T and any element x.*

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. The proof is omitted due to space constraints.

Lemma 7. *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.*

4.1 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 6 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 8. *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. [5], 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 7. 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 6. 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.

4.2 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 9. *For any blocking factor t, given a data structure that can answer path mode queries between marked nodes in time $T(t)$ with a space requirement of $S(t)$ bits, there exists a data structure answering path mode queries between marked nodes for blocking factor t' in time $T(t') = T(t) + O(t'' \log\log n)$ with a space requirement $S(t') = S(t) + O(n + (n/t')^2 \log(t/t''))$ bits, where $t > t' > t''$.*

Proof. (Sketch) Assume the nodes in T are marked based on a blocking factor t using Lemma 7, 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 [13]. 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*. 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 [13]) 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 6 in $O(t'' \log\log n)$ overall time. □

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

Theorem 3. *There exists a linear-space (in words; that is, $O(n \log n)$ bits) data structure that answers path mode queries on trees 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 9 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 8 to $\log^* n$ levels of the recursion gives $t_{\log^* n} = \sqrt{n/w}$ and

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

Similar techniques lead to a data structure for tree path least frequent element queries; we defer the proof to the full version due to space constraints.

Theorem 4. *There exists a linear-space data structure that answers path least frequent element queries on trees in $O(\log\log n\sqrt{n/w})$ time.*

5 Path α-Minority Query 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 [5]; 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 section, we can compute path frequencies in $O(\log \log n)$ time (Lemma 6); 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 [5], 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 [6] 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. The proof is omitted due to space constraints.

Lemma 10. *There exists a linear-space data structure that answers 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 can be chosen interactively.*

Lemmas 6 and 10 give the following theorem.

Theorem 5. *There exists a linear-space data structure that answers path α-minority queries on trees in $O(\alpha^{-1} \log \log n)$ time (where α and the path's endpoints are specified at query time).*

Proof. We construct the data structures of Lemma 6 and Lemma 10, 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. □

6 Discussion and Directions for Future Research

Our data structures for path queries refer to Lemma 6. Consequently, each has query time $O(\log \log n)$ times greater than the corresponding time on arrays.

For arrays, Chan et al. [3] use $O(1)$-time range frequency queries for the case in which the element whose frequency is being measured is at an endpoint of query range. Generalizing this technique to path queries on trees should allow each data structure's query time to be decreased accordingly.

Acknowledgements

The authors thank the anonymous reviewers for their helpful suggestions.

References

1. D. Belazzougui, F. C. Botelho, and M. Dietzfelbinger. Hash, displace, and compress. In *Proc. ESA*, volume 5757 of *LNCS*, pages 682–693. Springer, 2009.
2. D. Belazzougui, T. Gagie, and G. Navarro. Better space bounds for parameterized range majority and minority. In *Proc. WADS*, volume 8037 of *LNCS*. Springer, 2013.
3. T. M. Chan, S. Durocher, K. G. Larsen, J. Morrison, and B. T. Wilkinson. Linear-space data structures for range mode query in arrays. In *Proc. STACS*, volume 14, pages 291–301, 2012.
4. T. M. Chan, S. Durocher, K. G. Larsen, J. Morrison, and B. T. Wilkinson. Linear-space data structures for range mode query in arrays. *Theory Comp. Sys.*, 2013.
5. T. M. Chan, S. Durocher, M. Skala, and B. T. Wilkinson. Linear-space data structures for range minority query in arrays. In *Proc. SWAT*, volume 7357 of *LNCS*, pages 295–306. Springer, 2012.
6. B. Chazelle. Filtering search: A new approach to query-answering. *SIAM J. Comp.*, 15(3):703–724, 1986.
7. S. Durocher, M. He, J. I. Munro, P. K. Nicholson, and M. Skala. Range majority in constant time and linear space. In *Proc. ICALP*, volume 6755 of *LNCS*, pages 244–255. Springer, 2011.
8. S. Durocher, M. He, J. I. Munro, P. K. Nicholson, and M. Skala. Range majority in constant time and linear space. *Inf. & Comp.*, 222:169–179, 2013.
9. T. Gagie, M. He, J. I. Munro, and P. Nicholson. Finding frequent elements in compressed 2D arrays and strings. In *Proc. SPIRE*, volume 7024 of *LNCS*, pages 295–300. Springer, 2011.
10. T. Hagerup and T. Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Proc. STACS*, volume 2010 of *LNCS*, pages 317–326. Springer, 2001.
11. D. Krizanc, P. Morin, and M. Smid. Range mode and range median queries on lists and trees. In *Proc. ISAAC*, volume 2906 of *LNCS*, pages 517–526. Springer, 2003.
12. D. Krizanc, P. Morin, and M. Smid. Range mode and range median queries on lists and trees. *Nordic J. Computing*, 12:1–17, 2005.
13. K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. ACM-SIAM SODA*, pages 134–149, 2010.
14. J. P. Schmidt and A. Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM J. Comput.*, 19(5):775–786, 1990.