Range Majority in Constant Time and Linear Space[☆]

Stephane Durocher[a], Meng He[b], J. Ian Munro[c], Patrick K. Nicholson[c], Matthew Skala[a]

[a]*Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada*
[b]*Faculty of Computer Science, Dalhousie University, Halifax, Nova Scotia, Canada*
[c]*Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada*

Abstract

Given an array A of size n, we consider the problem of answering range majority queries: given a query range $[i..j]$ where $1 \le i \le j \le n$, return the majority element of the subarray $A[i..j]$ if it exists. We describe a linear space data structure that answers range majority queries in constant time. We further generalize this problem by defining range α-majority queries: given a query range $[i..j]$, return all the elements in the subarray $A[i..j]$ with frequency greater than $\alpha(j-i+1)$. We prove an upper bound on the number of α-majorities that can exist in a subarray, assuming that query ranges are restricted to be larger than a given threshold. Using this upper bound, we generalize our range majority data structure to answer range α-majority queries in $O(\frac{1}{\alpha})$ time using $O(n \lg(\frac{1}{\alpha}+1))$ space, for any fixed $\alpha \in (0,1)$. This result is interesting since other similar range query problems based on frequency have nearly logarithmic lower bounds on query time when restricted to linear space.

1. Introduction

The *majority element*, or *majority*, of an array $A[1..n]$ is the element, if any, that occurs more than $\frac{n}{2}$ times in A. The *majority element problem* is to determine whether a given array has a majority element, and if so, to report that element. This problem is fundamental to data analysis and has been well studied. Linear time deterministic and randomized algorithms for this problem, such as the Boyer-Moore voting algorithm [8], are well known, and they are sometimes included in the curricula of introductory courses on algorithms.

In this paper, we consider the data structure counterpart to this problem. We are interested in designing a data structure that represents an array $A[1..n]$

[☆]This research was supported through NSERC of Canada and the Canada Research Chairs Program. A preliminary version of these results appeared at the 38th International Colloquium on Automata, Languages and Programming [12].

Email addresses: durocher@cs.umanitoba.ca (Stephane Durocher), mhe@cs.dal.ca (Meng He), imunro@cs.uwaterloo.ca (J. Ian Munro), p3nichol@cs.uwaterloo.ca (Patrick K. Nicholson), mskala@cs.umanitoba.ca (Matthew Skala)

to answer *range majority queries*: given a query range $[i..j]$ where $1 \leq i \leq j \leq n$, return the majority element of the subarray $A[i..j]$ if it exists, and ∞ otherwise. Here we define the majority of a subarray $A[i..j]$ as the element whose *frequency* in $A[i..j]$, i.e., the number of occurrences of the element in $A[i..j]$, is more than half of the size of the interval $[i..j]$.

We further generalize this problem by defining the *α-majorities* of a subarray $A[i..j]$ to be the elements whose frequencies are more than $\alpha(j-i+1)$, i.e., α times the size of the range $[i..j]$, for $0 < \alpha < 1$. Thus an *α-majority query* on array $A[1..n]$ can be defined as: given a query range $[i..j]$ where $1 \leq i \leq j \leq n$, return the α-majorities of the subarray $A[i..j]$ if they exist, and ∞ otherwise. A range α-majority query becomes a range majority query when $\alpha = \frac{1}{2}$.

For the case of range majority, we describe a linear space data structure that answers queries in constant time. We generalize this data structure to the case of range α-majority, yielding an $O(n \lg(\frac{1}{\alpha}+1))$ space[1] data structure that answers queries in $O(\frac{1}{\alpha})$ time, for any fixed $\alpha \in (0,1)$. Similar range query problems based on frequency are the range mode and k-frequency problems [14]. A *range mode query* for range $[i..j]$ returns an element in $A[i..j]$ that occurs at least as frequently as any other element. A *k-frequency query* for range $[i..j]$ determines whether any element in $A[i..j]$ occurs with frequency exactly k. Both of these problems have a lower bound that requires $\Omega(\frac{\lg n}{\lg\lg n})$ query time for any linear space data structure [14]. In light of this lower bound, it is interesting that a linear space data structure can answer range α-majority queries in constant time for fixed constant values of α.

1.1. Related Work

Computing the Mode, Majority, and Plurality of a Multiset. The mode of a multiset S of n items can be found in $O(n \lg n)$ time by sorting S and counting the frequency of each element. The decision problem of determining whether the frequency m of the mode exceeds one reduces to the element uniqueness problem, resulting in a lower bound of $\Omega(n \lg n)$ time in the algebraic decision tree model [5]. Better bounds have been obtained by parameterizing in terms of m: Munro and Spira [21] and Dobkin and Munro [11] described an $O(n \lg(\frac{n}{m}))$ time algorithm and corresponding lower bound of $\Omega(n \lg(\frac{n}{m}))$ time. Misra and Gries [20] gave $O(n)$ and $O(n \lg(\frac{1}{\alpha}))$ time algorithms for computing an α-majority when $\alpha \geq \frac{1}{2}$ and $\alpha < \frac{1}{2}$, respectively. The problem of computing α-majorities has also recently been studied in the approximate setting, using the term *heavy hitters* instead of α-majorities [10].

The *plurality* of a multiset S is a unique mode of S. That is, every multiset has a mode, but it might not have a plurality. The mode algorithms mentioned above can verify the uniqueness of the mode without any asymptotic increase in time. Numerous results established bounds on the number of comparisons required for computing a majority, α-majority, mode, or plurality (e.g., [1, 2, 11, 21]).

[1]In this paper $\lg n$ denotes $\log_2 n$.

Range Mode, Frequency, and Majority Queries. Krizanc et al. [19] described data structures that provide constant time range mode queries using $O(\frac{n^2 \lg\lg n}{\lg n})$ space, and $O(n^\epsilon \lg n)$ time queries using $O(n^{2-2\epsilon})$ space, for any fixed $\epsilon \in (0, \frac{1}{2}]$. Petersen and Grabowski [23] improved the first bound to constant time and $O(\frac{n^2 \lg\lg n}{\lg^2 n})$ space. Petersen [22] and Durocher and Morrison [13] improved the second bound to $O(n^\epsilon)$ time and $O(n^{2-2\epsilon})$ space, for any fixed $\epsilon \in (0, \frac{1}{2}]$. Durocher and Morrison [13] described four $O(n)$ space data structures that return the mode of a query range $[i..j]$ in $O(\sqrt{n})$, $O(k)$, $O(m)$, and $O(|j-i|)$ time, respectively, where k denotes the number of distinct elements. Greve et al. [14] proved a lower bound of $\Omega(\frac{\lg n}{\lg(sw/n)})$ query time for any range mode query data structure that uses s memory cells of w bits. Finally, various data structures have been designed to support approximate range mode queries, in which the objective is to return an element whose frequency is at least ε times the frequency of the mode, for a fixed $\varepsilon \in (0,1)$ (e.g., [7, 14]).

Greve et al. [14] examined the range k-frequency problem, in which the objective is to determine whether any element in the query range has frequency exactly k, where k is either fixed or given at query time. They noted that when k is fixed a straightforward linear space data structure exists for determining whether any element has frequency at least k in constant time; determining whether any element has frequency *exactly* k requires a different approach. For any fixed $k > 1$, they described how to support range k-frequency queries in $O(\frac{\lg n}{\lg\lg n})$ optimal time. When k is given at query time, Greve et al. showed a lower bound of $\Omega(\frac{\lg n}{\lg\lg n})$ time applies to either query: exactly k or at least k.

The current best result applicable to the range α-majority problem is that of Karpinski and Nekrich [18, Theorem 2]. They studied the problem in a geometric setting, in which points on the real line are assigned colours, and the goal is to find *τ-dominating colours*: that is, given a range Q, return all the colours that are assigned to at least a τ fraction of the points in Q. If we treat each entry of an array $A[1..n]$ as a point in a bounded universe $[1, n]$, their data structure can be used to represent A in $O(\frac{n}{\alpha})$ space to support range α-majority queries in $O(\frac{(\lg\lg n)^2}{\alpha})$ time.

1.2. Our Results

Our results can be summarized as follows.

- In Section 2 we present a data structure for answering range majority queries under the word-RAM model, with word size $\Omega(\lg n)$. It uses $O(n)$ words and answers range majority queries in constant time. The data structure is conceptually simple and based on the idea that, for query ranges above a certain size threshold, only a small set of *candidate* elements need be considered in order to determine the majority. In order to verify the frequency of these elements efficiently, we present a novel decomposition technique that uses wavelet trees [15].

- In Section 3 we generalize our data structure to answer range α-majority queries, for any fixed $\alpha \in (0, 1)$. Note that although α is fixed, it is not necessarily a constant. For example, setting $\alpha = \frac{1}{\lg n}$ is permitted. Our structure uses $O(n \lg(\frac{1}{\alpha} + 1))$ words and answers range α-majority queries in $O(\frac{1}{\alpha})$ time. In order to generalize our data structure when $\frac{1}{\alpha}$ is large, i.e., when $\frac{1}{\alpha} = \omega(1)$, we make use of *batched* queries over wavelet trees.

- In Section 4 we discuss the applications of our data structure for range α-majority queries to the coloured range searching problems defined by Karpinski and Nekrich [18], and present improved data structures for these problems.

- In Section 5 we improve the upper bound on the number of candidate elements that need be stored by our data structure. These bounds are independently interesting, and are tight for the case of $\alpha = \frac{1}{2}$. Although these bounds are not required to give an asymptotic analysis of the space used by our data structure, they may be of practical interest as they demonstrate that the constant factors are indeed smaller than the loose bounds given in Sections 2 and 3.

2. Range Majority Data Structure

In this section we describe a linear space data structure that supports range majority queries in constant time. To provide some intuition, suppose we partition the input array $A[1..n]$ into four contiguous equally sized blocks. If we are given a query range that contains one of these four blocks, then it is clear that a majority element for this query must have frequency greater than $\frac{n}{8}$ in A. Thus, at most seven elements need be considered when computing the majority for queries that contain an entire block.

Of course, not all queries contain one of these four blocks. Therefore, we decompose the array into multiple levels in order to support arbitrary queries (Sections 2.1 and 2.2). Using this decomposition in conjunction with succinct data structures [16], we design a linear space data structure that answers range majority queries in constant time (Section 2.3). The data structure works by counting the frequency of a constant number of *candidate* elements in order to determine the majority element for a given query. While a loose bound on the number of candidates that need be considered suffices to show that our data structures occupy linear space, it is more challenging to prove a tighter bound, such as that of Section 5.

From this point on we make the assumption that the elements stored in the input array A are drawn from the alphabet $\{1, ..., \sigma\}$, where $\sigma \leq n$ is the number of distinct elements in A. If this is not the case, then we can apply the well known technique of rank space reduction [9, 3] as a preprocessing step, and store an auxiliary array of size σ to invert an element in $\{1, ..., \sigma\}$ to its original value. This preprocessing step takes $O(n \lg \sigma)$ time using a balanced binary search tree.

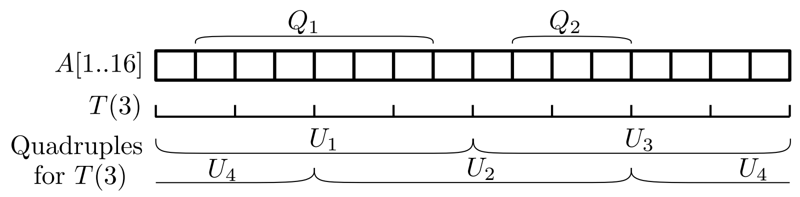

Figure 1: An example where $n = 16$. Blocks in $T(3)$ have size 2, and each of the 4 quadruples contain 4 blocks. Query ranges Q_1 and Q_2 are associated with quadruples U_1 and U_3 respectively.

2.1. Quadruple Decomposition

The first stage of our decomposition is to construct a notional complete binary tree T over the range $[1..n]$, in which each node represents a subrange of $[1..n]$. Let the root of T represent the entire range $[1..n]$. For a node corresponding to range $[a..b]$, its left child represents the left half of its range, i.e., the range $[a..\lfloor\frac{(a+b)}{2}\rfloor]$, and its right child represents the right half, i.e., the range $[\lfloor\frac{(a+b)}{2}\rfloor + 1..b]$. For simplicity, we assume that n is a power of 2. Each leaf of the tree represents a range of size 1, which corresponds to a single index of the array A. We refer to ranges represented by the nodes of T as *blocks*. Note that the tree T is for illustrative purposes only, so we need not store it explicitly.

The tree T has $\lg n + 1$ levels, which are numbered 0 through $\lg n$ from top to bottom. For each level ℓ, T partitions A into 2^ℓ blocks of size $\frac{n}{2^\ell}$. Let $T(\ell)$ denote the set of blocks at level ℓ in T.

The second stage of our decomposition consists of arranging adjacent blocks within each level $T(\ell)$, $2 \leq \ell \leq \lg n$, into groups. Each group consists of four blocks and is called a *quadruple*. Formally, we define a quadruple U_q to be a range $[a..b]$ at level $\ell \geq 2$ of size $\frac{4n}{2^\ell}$, where $a = \frac{2(q-1)n}{2^\ell} + 1$ and $b = \frac{2(q-1)n}{2^\ell} + \frac{4n}{2^\ell}$, for $1 \leq q \leq 2^{\ell-1} - 1$. In other words, each quadruple at level ℓ contains exactly 4 consecutive blocks, and its starting position is separated from the starting position of the previous quadruple by 2 blocks. To handle border cases, we also define an extra quadruple $U_{2^{\ell-1}}$ which contains both the first two and last two blocks in $T(\ell)$. Thus, at level ℓ there are $2^{\ell-1}$ quadruples, and each block in $T(\ell)$ is contained in two quadruples. These definitions are summarized in Figure 1.

2.2. Candidates

Based on the decomposition from the previous section, we observe the following:

Observation 1. *For every query range Q there exists a unique level ℓ such that Q contains at least one and at most two consecutive blocks in $T(\ell)$, and, if Q contains two blocks, then the nodes representing these blocks are not siblings in the tree T.*

Let U be a quadruple consisting of four consecutive blocks, B_1 through B_4 from $T(\ell)$, where ℓ is the level referred to in the previous observation. We *associate* Q with U if Q contains B_2 or B_3; for convenience we also say that Q is associated with level ℓ. Note that Q may contain both B_2 and B_3; see Q_1 in Figure 1. The following lemma can be proved by an argument analogous to that described at the beginning of Section 2:

Lemma 1. *There exists a set C of at most 7 elements such that, for any query range Q associated with quadruple U, the majority element for Q is in C.*

For a quadruple U, we define the set of *candidates* for U to be the elements in C. In Section 5.2 we improve the upper bound on $|C|$ from 7 to 5, which, as illustrated by the following example, is tight.

Example 1. Let U be a quadruple containing 4 blocks, each of size 32, and $(e)^y$ denote a sequence of y occurrences of the element e. In ascending order of starting position, the first block begins with an arbitrary element and is followed by $(e_1)^{28}$ and $(e_2)^3$. The second block contains $(e_2)^{15}$, and $(e_3)^{17}$. The third block contains $(e_1)^8$, $(e_4)^{17}$, and $(e_5)^7$. The final block contains $(e_5)^{19}$, followed by any arbitrary sequence of elements. Assume the range contained by the quadruple is [1..128]. The queries [2..72], [30..64], [33..64], [65..96] and [65..115] are all associated with U, and have e_1 through e_5 as majority elements respectively.

Next, we describe how the sets of candidates can be computed efficiently.

Lemma 2. *The sets of candidates for all the quadruples can be identified in $O(n \lg n)$ time.*

Proof. Recall that the elements in A are drawn from the alphabet $\{1, ..., \sigma\}$, where $\sigma \leq n$. We can count the frequencies of all the elements in quadruple U in $O(|U|)$ time, in a single pass over the elements in U, using an auxiliary array of size σ. When the count of an element exceeds $\frac{|U|}{8}$, we add it to the set of candidates for U. This implies that the sets of candidates for all the quadruples in all of the $\lg n + 1$ levels of T can be found in $O(n \lg n)$ time.

2.3. Data Structures for Counting

We now describe the data structures stored for each level ℓ of the tree T, for $2 \leq \ell \leq \lg n$. Given a quadruple U_q in level ℓ, for $1 \leq q \leq 2^{\ell-1}$ we store the set of candidates for U_q in an array F_q. Let Y_q be a string of length $|U_q|$, where the i-th symbol in Y_q is f if the i-th symbol in U_q is $F_q[f]$, and a unique symbol otherwise. Let Y be the concatenation of the strings Y_1 through $Y_{2^{\ell-1}}$. We use a *wavelet tree* [15] to represent Y, which has alphabet size $\sigma' = |F_q| + 1 \leq 6$. This representation uses $n \lg \sigma'(1 + o(1))$ bits to provide constant time support for the operation $\texttt{rank}_c(Y, i)$, which returns the number of occurrences of the character c in $Y[1..i]$.

Theorem 1. *Given an array $A[1..n]$, there exists an $O(n)$ word data structure that supports range majority queries on A in $O(1)$ time, and can be constructed in $O(n \lg n)$ time.*

Proof. Given a query $Q = [a..b]$, we first want to find the level ℓ and the index q of the quadruple U_q with which Q is associated. This can be reduced to finding the length of the longest common prefix of the $(\lg n)$-bit binary representations of a and b, which can be done in constant time in the following way. Let $z = \mathtt{MSB}(\frac{n}{b-a+1})$, where $\mathtt{MSB}(x)$ returns the most significant bit of x. If $(a-1)2^z \mod n = 0$, or $(b-1)2^z \mod n = 0$, or $\lceil \frac{(a-1)2^z}{n} \rceil \neq \lfloor \frac{(b-1)2^z}{n} \rfloor$, then $\ell = z$; otherwise, $\ell = z + 1$. Let $q' = \lfloor \frac{(a-1)2^{\ell-1}}{n} \rfloor$. The quadruple U_q associated with Q is either $q = \frac{q'}{2}$ or $q = \frac{q'-1}{2}$ depending on whether the size of the blocks at level ℓ divide into the starting position $a-1$. Note that we interpret U_0 to mean $U_{2^{\ell-1}}$. Since we can support the $\mathtt{MSB}$ operation in $O(1)$ time using a table of size $o(n)$ bits, we can compute both ℓ and q in $O(1)$ time.

Next, we show how to answer queries associated with a quadruple at levels ℓ, for $2 \leq \ell \leq \lg n$; the case in which $0 \leq \ell \leq 1$ can be handled similarly. The representation of quadruple U_q in Y begins at $s = \frac{4(q-1)n}{2^\ell} + 1$. We also must normalize the values a and b to the starting position s, so let $t = \frac{2(q-1)n}{2^\ell} + 1$. For each f in $[1..|F_q|]$, we count the frequency of $F_q[f]$ in $[a..b]$ using $\mathtt{rank}_f(Y, s+b-t) - \mathtt{rank}_f(Y, s+a-1-t)$, or, equivalently, $\mathtt{rank}_f(Y, t+b) - \mathtt{rank}_f(Y, t+a-1)$. We then report $F_q[f]$ if it is a majority. Since Y has a constant sized alphabet, this process takes $O(1)$ time.

In addition to the input array, we must store the arrays F_q for each of the $O(n)$ quadruples, and each array requires a constant number of words. For each of the $\lg n + 1$ levels in T we store a wavelet tree on an alphabet of size $\sigma' \leq 6$, requiring $O(n \lg n)$ bits. To answer queries in constant time, we require $o(n)$ bits of additional space for a lookup table to determine ℓ and q. Thus, the additional space requirements beyond the input array are $O(n)$ words. In terms of construction time, we can build the lists of candidates in $O(n \lg n)$ time by Lemma 2, and the wavelet trees require $O(n)$ time to construct per level. Thus, the overall construction time is $O(n \lg n)$. □

Remark 1. In most cases, the upper bound on the number of candidates stored for each quadruple is likely significantly greater than the number of candidates actually stored by the data structure. Therefore, we expect that the constant factor in the $O(n)$ space term is not very large in practice.

3. Generalization to Range α-Majority Queries

In this section we generalize the data structure from Theorem 1 to report α-majorities, for some fixed $\alpha \in (0, 1)$, supplied at construction time. Using the same arguments presented in the beginning of Section 2, it is clear that if an element is an α-majority for any query associated with quadruple U, then it must appear more than $\frac{\alpha|U|}{4}$ times in U. This implies that the set of candidate

α-majorities has size less than $\frac{4}{\alpha}$. Although this loose bound will suffice to bound the asymptotic behaviour of our data structure, we prove tighter bounds in Section 5.

3.1. Handling Large Alphabets

For a given α, if the number of candidates, $|C|$, is $\omega(1)$, then we require the following lemma about executing batched rank queries on a wavelet tree.

Lemma 3. *A string $S[1..n]$ over alphabet $[m]$, where $m \leq n$, can be represented using a wavelet tree such that given an index i, the results of $\texttt{rank}_f(S,i)$ for all $f = 1, 2, ..., m$ can be computed in $O(m)$ time.*

Proof. If we use the original wavelet tree algorithm to perform $\texttt{rank}_f(S,i)$ for a particular alphabet symbol, f, the algorithm consists of two steps. First, in the bit vector representing the leaf, u, corresponding to f (and one other alphabet symbol), we locate the bit that corresponds to $S[i]$. This step is done by performing constant-time rank queries on bit vectors, representing internal nodes of the wavelet tree, on the path from the root to the leaf u. The second step is to perform a rank query in constant time on the bit vector representing u, to compute the number of 1s up to and including the bit corresponding to $S[i]$. By performing batch processing of the queries $\texttt{rank}_f(S,i)$ for f in $1, 2, ..., \sigma$, as there are $m-1$ nodes in a wavelet tree, the first step for all f requires $O(m)$ constant-time rank queries on bit vectors stored in internal nodes. Therefore, with careful tuning, we can perform the first step of all the $\texttt{rank}_f(S,i)$ queries in $O(m)$ time. The second step requires a $O(1)$ time rank query for each leaf, so it uses $O(m)$ time in total. $\square$

With the above observation we present the following theorem:

Theorem 2. *Given an array $A[1..n]$ and any fixed $\alpha \in (0,1)$, there exists an $O(n \lg(\frac{1}{\alpha}+1))$ word data structure that supports range α-majority queries on A in $O(\frac{1}{\alpha})$ time, and can be constructed in $O(n \lg n \lg(\frac{1}{\alpha}+1))$ time.*

Proof. From Theorem 1 and Lemma 3 the query time follows, so we focus on analyzing the space. We observe that if $\alpha < \frac{1}{4}$, then we need not keep data structures at level $\lg n$ in T, since every distinct element contained in a query range, Q, associated with this level is a $(\frac{1}{4} - \varepsilon)$-majority for Q, for $0 < \varepsilon < \frac{1}{4}$. Instead, we perform a linear scan of the query range in $O(\frac{1}{\alpha})$ time, returning all the distinct elements. Continuing this argument, we observe that we only require the array F_q, for quadruple q, if q represents a range of size larger than $O(\frac{1}{\alpha})$. Since there are $O(n\alpha)$ quadruples of this size, the arrays require $O(n\alpha \times \frac{1}{\alpha} \lg n) = O(n \lg n)$ bits in total. The overall space required for the wavelet tree data structures is $O(n \lg(\frac{1}{\alpha}+1) \times \lg n)$ bits, and this term dominates the overall space requirements.

We can construct the sets of candidates for all quadruples of size larger than $O(\frac{1}{\alpha})$ in $O(n \lg n)$ time using the same technique described in Lemma 2. To construct the wavelet trees requires $O(n \lg(\frac{1}{\alpha}+1))$ time per level, for an overall time bound of $O(n \lg n \lg(\frac{1}{\alpha}+1))$. Thus, the construction time is dominated by the wavelet tree construction, and requires $O(n \lg n \lg(\frac{1}{\alpha}+1))$ time overall. □

4. Applications

In this section we describe some consequences of Theorem 2 to the more geometric versions of the range α-majority problem described by Karpinski and Nekrich [18]. These problems have practical applications to range searching problems in databases, where we would like to identify attributes that are frequently associated with points in a query range [18]. In the next subsections we describe these geometric problems, which deal with coloured points instead of elements in arrays.

4.1. Range Majority for Coloured Points in One Dimension

We are given a set, P, of points in one dimension, where each point $p \in P$ is assigned a colour c from a set, C, of colours. Let $\mathrm{col}(p) = c$ denote the colour of p. We are also given a fixed parameter $\alpha \in (0,1)$, which defines the threshold for determining whether a colour is to be considered frequent. Let $P(Q)$ be the set $\{p \mid p \in Q, p \in P\}$, and $P(Q,c)$ be the set $\{p \mid p \in P(Q), \mathrm{col}(p) = c\}$. Our goal is to design a data structure that, given query range Q, can return the set of colours M such that for each colour $c \in M$, the size of the set $|P(Q,c)| > \alpha|P(Q)|$. To be consistent with our original formulation of the problem, we refer to a colour $c \in M$ as an *α-majority* for Q, and the query Q as an *α-majority query*, though they are also referred to as *α-dominating colours* [18].

To solve this problem, we apply the reduction to rank space technique [9, 3] to the coordinates of the points, and store the sequence of colours in the data structure from Theorem 2. We store the original coordinates of the points in any linear space data structure that supports predecessor queries. Given a query, we can use the predecessor search data structure to map the query to a rank based query on the range α-majority data structure. We present the following theorem:

Theorem 3. *Given a set P of n points in one dimension and a fixed $\alpha \in (0,1)$, there is an $O(n \lg(\frac{1}{\alpha}+1))$ space data structure that supports range α-majority queries on P in $O(\texttt{pred}(P) + \frac{1}{\alpha})$ time, where $\texttt{pred}(P)$ is the time required to do a one-dimensional predecessor search on the coordinates of the points in P.*

Remark 2. The previous theorem implies that if we only assume the points can be compared in constant time, then we can answer range α-majority queries in $O(\lg n + \frac{1}{\alpha})$ time by storing the coordinates of the points in a balanced binary search tree. If the points have integer coordinates, then we can store their coordinates in an exponential search tree [4] and answer range α-majority queries in $O(\sqrt{\lg n / \lg\lg n} + \frac{1}{\alpha})$ time. Alternatively, we can store their coordinates in a y-fast trie [24], which yields a query time of $O(\lg\lg U + \frac{1}{\alpha})$, where U is the value of the largest coordinate in P.

4.2. Range Majority in Higher Dimensions

In the same manner as Karpinski and Nekrich [18], we can extend Theorem 3 to higher dimensions using the well-known range tree technique of Bentley [6]. The main problem with moving to higher dimensions is that we cannot use the wavelet tree to verify the frequency of candidates when $d \geq 2$. However, we can use the small list of candidates generated by the data structure in conjunction with any d-dimensional range counting data structure, such as that of Chazelle [9]. Furthermore, by removing the wavelet trees from the data structure of Theorem 2, we remove the $\lg(\frac{1}{\alpha}+1)$ term from the space bound. Thus, this stripped down data structure uses $O(n)$ space, and can return, for any range α-majority query Q on A, a list of $O(\frac{1}{\alpha})$ elements that contains all the α-majorities for Q in $O(\frac{1}{\alpha})$ time. By combining this discussion with the analysis presented by Karpinski and Nekrich [18, Section 4], we get the following theorem that improves the space bound of their d-dimensional structure by a factor of $O(\frac{1}{\alpha})$:

Theorem 4. *Given a set P of n points in d-dimensions, for any constant $d \geq 2$, and a fixed $\alpha \in (0,1)$, there is an $O(n \lg^{d-1} n)$ space data structure that supports range α-majority queries on P in $O(\frac{\lg^d n}{\alpha})$ time.*

Remark 3. As in the one-dimensional case, we can exploit word-level parallelism to improve the time complexity of queries in the case where the points in P have integer coordinates. Using the data structure of JaJa et al. [17], the query time improves to $O(\frac{\lg^d n}{\alpha \lg\lg n})$.

5. Tighter Bounds on the Number of α-Majorities

In this section we provide a tighter upper bound on the number of candidates we need store from each quadruple to support α-majority queries. Specifically, we show that the size of the set of candidates that need be stored from each quadruple is less than:

$$2\left\lceil \frac{1}{\alpha} \right\rceil + \frac{2}{\lg \frac{1}{1-\alpha}} \; . \tag{1}$$

Thus, for the case when $\alpha = \frac{1}{2}$, the bound on the number of elements improves from 7 to 5, which is tight. Furthermore, since the proof is constructive, the bound is tight for unit fraction values of α.

5.1. Definitions

Before discussing the proof of the above bound, we require the following definitions. We refer to the range $[a..b']$, where $a \leq b' \leq b$, as a *prefix* of the range $[a..b]$. Similarly, the range $[a'..b]$, where $a \leq a' \leq b$, is a *suffix* of $[a..b]$. For a block $L \in T(\ell)$, we refer to the *successor* of L, which is the block $L_s \in T(\ell)$ such that the range represented by L_s immediately follows the range represented by L. The *predecessor* is defined analogously.

Consider a query $[a..b']$ that contains block $L = [a..b] \in T(\ell)$, where $b \leq b' < b + |L|$. Thus, $[a..b']$ contains L and a prefix of the successor of L. We refer to a query of this form as a *prefix query.* We refer to the symmetric case, where a query $[a'..b]$ contains L and $a - |L| < a' \leq a$ as a *suffix query.* Finally, let $|A[i..j]|_t$ denote the frequency of an element t in $A[i...j]$.

5.2. Relaxed Triples

Suppose we are given a block L, where L_p and L_s are the predecessor and successor of L respectively; we call $L_p \cup L \cup L_s$ a *triple.* In order to prove an upper bound on the number of candidates in a quadruple, we generalize a triple in two ways, and then prove several properties of these objects. First, we relax the restriction that blocks in the triple have equal size, and only require that $|L_p| + |L_s| \leq 2|L|$. Second, we relax the restriction that blocks and occurrences of elements are of integer size; i.e., the ranges described in this section may start and end at arbitrary real numbers. Although the ranges are real-valued, we still refer to "occurrences" of elements. Thus, in the continuous setting described in this section, an occurrence of an element may contain an arbitrary fraction of a block; for example, inside a block there may be a contiguous range of occurrences of element e that has length 5.22. We refer to these generalized triples as *relaxed triples.*

Let $e_1, ..., e_m$ denote the m distinct α-majorities that exist for a query Q where $L \subseteq Q \subset (L_p \cup L \cup L_s)$; i.e., Q is a query contained in the relaxed triple and Q contains L. For brevity, whenever we refer to a query in the context of a relaxed triple, it is assumed to have this form. Let $\mathcal{Q} = \{Q_1, ..., Q_m\}$ be a set of queries within a relaxed triple such that Q_i is the smallest query for which e_i is an α-majority, breaking ties by taking the query with smallest starting position. We refer to $\mathcal{Q}$ as the *canonical query set* for the relaxed triple. If query Q_i is a prefix query or a suffix query we refer to it as *one-sided.* If Q_i is not one-sided, then it is *two-sided.* Note that the query $Q_i = L$ is one-sided, since it is both a suffix *and* a prefix query.

For two-sided canonical queries $Q_i \in \mathcal{Q}$, the element at both the starting position and ending position of Q_i must be e_i; otherwise we could reduce the size of Q_i. Thus, for all two-sided canonical queries $Q_i \in \mathcal{Q}$, no $Q_j \in \mathcal{Q}$ $(j \neq i)$ exists having the same starting or ending position as Q_i. However, there may be several occurrences of the query L in $\mathcal{Q}$, since many elements can share that particular range as a canonical query. From this point on we only consider relaxed triples where element e_i occurs only within the range Q_i for $1 \leq i \leq m$. Since the goal of this section is to find an upper bound on m, occurrences of e_i outside range Q_i can be removed without decreasing m.

Lemma 4. *Given a relaxed triple and its canonical query set $\mathcal{Q} = \{Q_1, ..., Q_m\}$, the elements $\{e_1, ..., e_m\}$ corresponding to the queries in $\mathcal{Q}$ can be rearranged such that they each appear in at most two contiguous ranges in the relaxed triple. This reordering induces a new canonical query set $\mathcal{Q}' = \{Q'_1, ..., Q'_m\}$, such that $|Q'_j| \leq |Q_j|$ for all $1 \leq j \leq m$.*

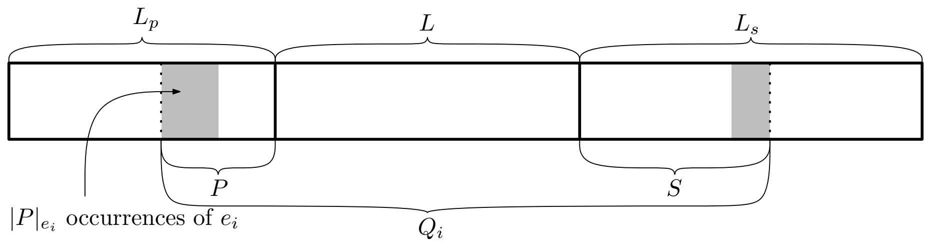

Figure 2: Illustration of the relaxed triple using notation from Step 1 in Lemma 5.

Proof. First, we describe a procedure for reordering the elements in L_p. Let $L'_p = L_p$, $\mathcal{Q}' = \mathcal{Q}$, and $Q_b \in \mathcal{Q}'$ be the query with the smallest starting position in L'_p. Then Q_b contains a non-empty suffix of L'_p; if no such query exists, then L'_p is empty and we are done. Let e_b be the element corresponding to Q_b. We swap the positions of all the occurrences of e_b in L'_p such that they occupy a prefix P of Q_b. All elements that were in P are shifted toward L. Thus, it may be possible to reduce the size of a query $Q_i \in \mathcal{Q}'$ that originally had a starting position in P, and we recompute $\mathcal{Q}'$. Let L'_p be the largest suffix of L_p that does not contain any occurrences of e_b. At this point we recurse and compute the next Q_b.

After we have finished moving e_b, at no point later in the procedure will an occurrence of e_b in L_p be touched. At the end of the procedure each element in L_p that corresponds to a canonical query will occupy a contiguous block. Furthermore, $|\mathcal{Q}'| = |\mathcal{Q}|$, since moving elements in P closer to the ending position of L_p will not decrease the ratio of their frequency to canonical query size. The procedure for reordering L_s is identical, though we process the elements in decreasing order by ending position.

After executing the procedure on L_p and L_s, consider an element e_i corresponding to Q_i. We can delete all k occurrences of e_i in L and insert k copies of e_i immediately before the first occurrence of e_i in L_s. This does not change the relative order of any other elements in the relaxed triple, and shifts all other elements in L_s in positions before the new first occurrence of e_i closer to L. Thus, each element appears in at most two contiguous ranges. $\square$

Remark 4. The procedure described in Lemma 4 implies that if Q_i is one-sided, then the element corresponding to Q_i can be reordered to appear in a single contiguous range. For an arbitrary query $Q_i \in \mathcal{Q}$, one of the contiguous ranges in which e_i appears may contain a suffix of L.

Lemma 5. *Given a relaxed triple and its canonical query set $\mathcal{Q} = \{Q_1, ..., Q_m\}$, its elements can be rearranged, creating a new relaxed triple that has a canonical query set $\mathcal{Q}' = \{Q'_1, ..., Q'_m\}$ such that Q'_i is one-sided for $1 \le i \le m$.*

Proof. We describe a procedure for rearranging the elements in the relaxed triple.

Step 1: We choose an arbitrary two-sided query, $Q_i \in \mathcal{Q}$. We apply Lemma 4 to the triple, such that all occurrences of e_i appear in the prefix and suffix of Q_i. Let P represent the prefix of Q_i contained in L_p, and S the suffix of Q_i contained in L_s. P is contained in $c \geq 0$ queries in $\mathcal{Q}$, distinct from Q_i, and S is contained in $d \geq 0$ queries in $\mathcal{Q}$, distinct from Q_i. Without loss of generality, assume $c \geq d$. Note that $|L|_{e_i} = \alpha|L| - \Delta_L$, for some value $\Delta_L \geq 0$; otherwise L would be the canonical query for e_i. Since $|Q_i|_{e_i} > \alpha(|L| + |P| + |S|)$, we have $|P|_{e_i} = \alpha|P| + \Delta_P$, and $|S|_{e_i} = \alpha|S| + \Delta_S$, for some values Δ_P and Δ_S, where $\Delta_P + \Delta_S > \Delta_L$. Note that $\Delta_P > 0$ and $\Delta_S > 0$; if $\Delta_P \leq 0$, then $S \cup L$ would be the canonical query for e_i, and the same argument applies to Δ_S. This implies that $|P|_{e_i} \geq \frac{\Delta_P}{1-\alpha}$ and $|S|_{e_i} \geq \frac{\Delta_S}{1-\alpha}$. See Figure 2.

Step 2: We remove all $|P|_{e_i} = \alpha|P| + \Delta_P \geq \frac{\Delta_P}{1-\alpha}$ occurrences of e_i from L_p. This shifts the starting position of c queries in $\mathcal{Q}$ closer to L. Let Q_j be the innermost of the c queries, i.e., Q_j has the largest starting position of the c queries. Since there were no occurrences of e_j in the removed block, in order for e_j to be an α-majority for Q_j, there must have been at least f occurrences of e_j to pay for the removed block, where $f = \alpha(|P|_{e_i} + f)$. This implies $f = |P|_{e_i}\frac{\alpha}{1-\alpha}$. Generalizing this formula to consider the number of occurrences of the c different elements required to pay for the removed block, as well as the payments made by the innermost queries, we get a recurrence relation. Let f_i be the savings of the i-th innermost of the c queries. It follows that $f_i = \frac{\alpha}{1-\alpha}(\delta_p + \sum_{j=1}^{i-1} f_j)$, for $1 \leq i \leq c$. Thus, we have reduced the size of L_p by the total sum $\delta_p + \sum_{i=1}^{c} f_c$.

Step 3: We insert $\frac{\Delta_P}{1-\alpha} \leq |P|_{e_i}$ elements immediately after the last occurrence of e_i in S. After this, there exists a prefix query on the relaxed triple which returns e_i as a majority. This insertion causes the ending positions of d queries in $\mathcal{Q}$ to be shifted farther from L. By the same argument as before, we must insert at most $\sum_{i=1}^{d} f_d$ elements in order to correct for this shift. Since $c \geq d$, our new arrangement satisfies the constraint $|L_p| + |L_s| \leq 2|L|$, and is therefore a relaxed triple.

Step 4: We reorder the elements according to Lemma 4 and recompute the canonical query set. The procedure described in the proof of Lemma 4 does not increase the number of two-sided queries. If any two-sided queries remain, then go to step 1.

After rearranging element e_i, Q_i will remain one-sided in any future iteration of the procedure; no occurrence of e_i will subsequently be moved back to L_p. Each iteration guarantees that e_i will be an α-majority for a one-sided query, and that the size of the canonical set remains unchanged. □

Remark 5. We note that Lemma 5 only holds in the continuous setting where we can manipulate fractional parts of elements. For an example where Lemma 5 does not hold in the discrete setting, consider the case where $|L| = |L_p| = |L_s| = 3$, and $L_p = \{e_5, e_5, e_4\}$, $L = \{e_1, e_2, e_3\}$, $L_s = \{e_4, e_6, e_6\}$, and $\frac{2}{7} < \alpha < \frac{1}{3}$. In this example, we cannot rearrange the triple such that Q_4 is one-sided, without decreasing the size of the canonical query set.

We have shown that to give an upper bound on the number of candidates in a relaxed triple, it suffices to examine the worst case restricted to prefix and suffix queries in the successor and predecessor of L, respectively. Without loss of generality, we consider the successor, then prove an upper bound on the size of the canonical query set in a relaxed triple. First, we require the following recurrence relation:

Lemma 6. *Consider the recurrence relation*

$$d_j = \frac{\alpha}{1-\alpha}\left(1 + \sum_{i=1}^{j-1}(1+d_i)\right) \quad ,$$

for $j > 1$, and $d_1 = \frac{\alpha}{1-\alpha}$. For $j \geq 1$,

$$d_j = \frac{1}{(1-\alpha)^j} - 1 \quad .$$

Proof. Proof by induction. Base case:

$$d_1 = \frac{\alpha}{1-\alpha} = \frac{1}{1-\alpha} - 1 \quad .$$

Inductive step: assume the statement is true for $1 \leq j' < j$. Thus,

$$d_j = \frac{\alpha}{1-\alpha} + \sum_{i=1}^{j-1} \frac{\alpha}{1-\alpha}\left(1 + \frac{1}{(1-\alpha)^i} - 1\right) \quad .$$

Simplifying this we get a geometric series,

$$d_j = \sum_{i=1}^{j} \frac{\alpha}{(1-\alpha)^i} \quad .$$

Finally, summing the terms of this series we get $d_j = \frac{1}{(1-\alpha)^j} - 1$, completing the inductive step. □

Next, we bound the number of candidates for prefix queries over a relaxed triple.

Lemma 7. *Let L be a block and L_s its successor in a relaxed triple. There exists a set of elements C, of size less than*

$$\left\lceil \frac{1}{\alpha} \right\rceil + \frac{\lg(1 + \frac{|L_s|}{|L|})}{\lg \frac{1}{1-\alpha}} \quad ,$$

such that for all prefix queries Q containing L, all α-majorities for Q are contained in C.

Proof. We keep the set $F = \{f_1, ..., f_h\}$ of the $h = \lceil \frac{1}{\alpha} \rceil$ most frequently occurring elements from the block $L = [a..b]$. Let prefix query $Q_1 = [a..b_1]$, where $b_1 = b$ initially, and increase b_1 until a new element $e_1 \notin F$ becomes an α-majority for Q_1. We continue this process k times, where k is a value determined later: for $1 \le i \le k$, define $Q_i = [a..b_i]$, where $b_i = b_{i-1}$ initially, and b_i is increased until a new element $e_i \notin F \cup \{e_1, ..., e_{i-1}\}$ becomes an α-majority. Let R_i be the largest prefix of L_s contained in Q_i, and $d_i = |R_i| = \frac{b_i - b}{|L|}$ for $1 \le i \le k$. In order for Q_i to be a prefix query, $0 < d_i < \frac{|L_s|}{|L|}$ must hold for each $1 \le i \le k$. We want to determine the maximum value of k for which $d_k < \frac{|L_s|}{|L|}$ for the specific value of α. The value $h + k$ provides an upper bound on the number of elements we need examine to determine the α-majorities for any prefix query.

To maximize k, assume that all elements in F are α-majorities for the query $Q' = L$. Applying Lemma 4, each element f_i appears in a contiguous block within L. Note that any extra occurrences of f_i can be removed without decreasing k.

With the exception of at most one element e_{k+1}, we can assume that $L \cup L_s$ only contains elements e' for which there exists some prefix query that returns e' as an α-majority; otherwise, we could replace all occurrences of these elements with e_{k+1}. We have filled L entirely with elements in F, and each element e_i can only occur in a single contiguous block in L_s, for $1 \le i \le k$, by Lemmas 4 and 5. Thus, each e_i appears in a contiguous block immediately following e_{i-1}.

Now we have an upper bound,

$$|R_j|_{e_j} \le d_j |L| - \sum_{i=1}^{j-1} |R_i|_{e_i} \ ,$$

and a lower bound,

$$|R_j|_{e_j} > \alpha(1 + d_j)|L| - |L|_{e_j} \ ,$$

for $1 \le j \le k$. By our construction, $|L|_{e_j} = 0$ for all $1 \le j \le k$. Rearranging the upper and lower bounds, we get that

$$d_k > \frac{\alpha}{1-\alpha} + \sum_{i=1}^{k-1} \frac{|R_i|_{e_i}}{|L|(1-\alpha)} \ ,$$

which implies

$$d_k > \frac{\alpha}{1-\alpha} + \sum_{i=1}^{k-1} \frac{\alpha(1+d_i)}{(1-\alpha)} \ .$$

Applying Lemma 6, we get $d_k > \frac{1}{(1-\alpha)^k} - 1$. Since $\frac{|L_s|}{|L|} > d_k$, this statement implies

$$1 + \frac{|L_s|}{|L|} > \frac{1}{(1-\alpha)^k} \ .$$

After isolating k we get that

$$h + k < \left\lceil \frac{1}{\alpha} \right\rceil + \frac{\lg\left(1 + \frac{|L_s|}{|L|}\right)}{\lg \frac{1}{1-\alpha}} \quad ,$$

completing the proof. □

Extending the above lemma to arbitrary queries on relaxed triples yields the following lemma:

Lemma 8. *The canonical query set $\mathcal{Q}$ of any relaxed triple has size less than*

$$\left\lceil \frac{1}{\alpha} \right\rceil + \frac{2}{\lg \frac{1}{1-\alpha}} \quad .$$

Proof. We consider the worst case in both predecessor, L_p, and successor, L_s, of L, noting that the contents of L are shared. We apply Lemmas 5 and 7 to L_p and L_s. Recall the constraint $|L_p| + |L_s| \leq 2|L|$, and note that the expression

$$\left\lceil \frac{1}{\alpha} \right\rceil + \frac{\lg\left(1 + \frac{|L_s|}{|L|}\right) + \lg\left(1 + \frac{|L_p|}{|L|}\right)}{\lg \frac{1}{1-\alpha}} \quad ,$$

is maximized when $|L_s| = |L_p| = |L|$. □

We now extend Lemma 8 to the case of quadruples.

Theorem 5. *For any quadruple U there exists a set C such that*

$$|C| < 2\left\lceil \frac{1}{\alpha} \right\rceil + \frac{2}{\lg \frac{1}{1-\alpha}} \quad ,$$

and for any Q associated with U, all α-majorities for Q are in C.

Proof. Let B_l and its successor, B_r, denote the two middle blocks of U. Let L_p and L_s denote the predecessor of B_l and successor of B_r, respectively. We decompose U into two relaxed triples, $R_l = L_p \cup B_l \cup B_r$ and $R_r = B_l \cup B_r \cup L_s$. The key observation is that in order to maximize the size of the canonical query sets of these triples — and therefore the number of candidates in the quadruple — the sets of elements corresponding to the canonical query sets of the two relaxed triples should be disjoint. Thus, since $\frac{1}{\alpha}$ grows faster than $1/\left(\lg \frac{1}{1-\alpha}\right)$, we can apply Lemma 7 to both R_l and R_r, achieving the desired bound. □

At this point, we have not explained how to compute the canonical query set of a triple, or the set of candidates for a quadruple. We present the following lemma, which borrows notation from Lemma 5:

Lemma 9. *Given a triple $L_p \cup L \cup L_s$, the set of elements corresponding to its canonical query set can be found in $O(|L|)$ time, provided we have access to four arrays of size σ in which all the entries have been initialized.*

PROOF. Notice that we are proving this bound for a triple, where each block in the triple is an array of length $|L|$, rather than a relaxed triple. Let F be the set of elements in $L_p \cup L \cup L_s$ that appear more than $\alpha|L|$ times. We can compute F in $O(|L|)$ time using one array of size σ, and after computing it, return the array to its initial state in $O(|L|)$ time. We also compute, for each $f \in F$, the value $D_L[f]$, which is the value Δ_L for f. Thus, $|F| < \frac{3}{\alpha}$, and the next step is to remove elements in F that do not correspond to a query in the canonical query set.

We use an array C, that will be used to store frequency counts, and an array D_S, that will be used to compute the maximum value of Δ_S for each element in F. Assume the arrays C and D_S have been initialized such that for each $f \in F$, $C[f] = 0$, and $D_S[f] = -\infty$. We perform a scan of L_s, and when we encounter element $f \in F$ at position i, we increment $C[f]$, and set $D_S[f] = \max\{D_S[f], C[f] - \alpha \times i\}$. Similarly, we also perform a scan of L_p, reusing the array C and another auxiliary array D_P, in order to compute the maximum value of Δ_P for each element in F. For a given element $f \in F$, the value $D_S[f]+D_L[f]$ or $D_P[f]+D_L[f]$ or $D_S[f]+D_P[f]+D_L[f]$ is positive if and only if f an element associated with a canonical query. This procedure requires two scans of the triple, and therefore runs in $O(|L|)$ time. After computing the canonical query set, we can return the entries in the four arrays to their initial value in $O(|F|)$ time. □

We can extend Lemma 9 to the case of a quadruple, U, to show that computing the set of candidates for U can be done in $O(|U|)$ time. The idea is to decompose the quadruple into two triples, as in Theorem 5. We then run the algorithm from Lemma 9 on both of these triples. The distinct set of elements in the union of these canonical query sets is the set of candidates for U.

6. Conclusions

We have presented an $O(n)$ word data structure that answers range majority queries in constant time, and an $O(n \lg(\frac{1}{\alpha}+1))$ word data structure that answers range α-majority queries in $O(\frac{1}{\alpha})$ time, for any fixed $\alpha \in (0,1)$. This result is interesting in light of the nearly logarithmic cell-probe lower bounds of Greve et al. for the closely related problems of range mode and range k-frequency [14].

Our data structure is based on an interesting tree decomposition method used to preprocess an array such that each query is associated with a short list of candidate α-majorities. Then, using techniques from the area of succinct data structures, each element in this short list is efficiently checked in order to determine whether it is an α-majority for the given query.

One open problem would be to determine if the space bound of $O(n \lg(\frac{1}{\alpha}+1))$ words can be improved, while maintaining the $O(\frac{1}{\alpha})$ query time. Another would be to determine if the data structure can be made output sensitive, i.e., whether there is a data structure that outputs the k α-majorities in a query range in $O(k)$ time.

References

[1] Aigner, M.: Variants of the majority problem. Discrete Applied Mathematics 137, 3–25 (2004)

[2] Alonso, L., Reingold, E.M.: Determining plurality. ACM Transactions on Algorithms 4(3), 26:1–26:19 (2008)

[3] Alstrup, S., Brodal, S., Rauhe, T.: New data structures for orthogonal range searching. In: Proc. of the 41st Annual Symposium on Foundations of Computer Science. pp. 198–207. IEEE (2000)

[4] Andersson, A., Thorup, M.: Dynamic ordered sets with exponential search trees. J. ACM 54 (June 2007)

[5] Ben-Or, M.: Lower bounds for algebraic computation trees. In: Proc. of ACM Symposium on Theory of Computing. pp. 80–86. STOC '83, ACM, New York, NY, USA (1983)

[6] Bentley, J.: Multidimensional divide-and-conquer. Communications of the ACM 23(4), 214–229 (1980)

[7] Bose, P., Kranakis, E., Morin, P., Tang, Y.: Approximate range mode and range median queries. In: Proc. of the International Symposium on Theoretical Aspects of Computer Science. LNCS, vol. 3404, pp. 377–388. Springer (2005)

[8] Boyer, R.S., Moore, J.S.: MJRTY - A fast majority vote algorithm. In: Boyer, R.S. (ed.) Automated Reasoning: Essays in Honor of Woody Bledsoe, pp. 105–117. Automated Reasoning Series, Kluwer, Dordrecht, The Netherlands (1991)

[9] Chazelle, B.: Functional approach to data structures and its use in multidimensional searching. SIAM Journal on Computing 17(3), 427–462 (1988)

[10] Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. Journal of Algorithms 55(1), 58–75 (2005)

[11] Dobkin, D., Munro, J.I.: Determining the mode. Theoretical Computer Science 12(3), 255–263 (1980)

[12] Durocher, S., He, M., Munro, J.I., Nicholson, P.K., Skala, M.: Range majority in constant time and linear space. In: Proc. of the International Colloquium on Automata, Languages, and Programming. LNCS, vol. 6755, pp. 244–255 (2011)

[13] Durocher, S., Morrison, J.: Linear-space data structures for range mode query in arrays (2011), arXiv:1101.4068v1

[14] Greve, M., Jørgensen, A.G., Larsen, K.D., Truelsen, J.: Cell probe lower bounds and approximations for range mode. In: Proc. of the International Colloquium on Automata, Languages, and Programming. LNCS, vol. 6198, pp. 605–616. Springer (2010)

[15] Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. of the 14th Symposium on Discrete Algorithms. pp. 841–850 (2003)

[16] Jacobson, G.: Space-efficient static trees and graphs. 30th Annual IEEE Symposium on Foundations of Computer Science pp. 549–554 (1989)

[17] JaJa, J., Mortensen, C., Shi, Q.: Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In: Proc. of the International Symposium on Algorithms and Computation. LNCS, vol. 3341, pp. 1755–1756 (2005)

[18] Karpinski, M., Nekrich, Y.: Searching for frequent colors in rectangles. In: Proc. of the 20th Canadian Conference on Computational Geometry. pp. 11–14 (2008)

[19] Krizanc, D., Morin, P., Smid, M.: Range mode and range median queries on lists and trees. Nordic Journal of Computing 12, 1–17 (2005)

[20] Misra, J., Gries, D.: Finding repeated elements. Science of Computer Programming 2(2), 143–152 (1982)

[21] Munro, J.I., Spira, M.: Sorting and searching in multisets. SIAM Journal on Computing 5(1), 1–8 (1976)

[22] Petersen, H.: Improved bounds for range mode and range median queries. In: Proc. of the Conference on Current Trends in Theory and Practice of Computer Science. LNCS, vol. 4910, pp. 418–423. Springer (2008)

[23] Petersen, H., Grabowski, S.: Range mode and range median queries in constant time and sub-quadratic space. Information Processing Letters 109, 225–228 (2009)

[24] Willard, D.: Log-logarithmic worst-case range queries are possible in space $\theta(n)$. Information Processing Letters 17(2), 81–84 (1983)