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

Range Majorities and Minorities in Arrays

Djamal Belazzougui · Travis Gagie ·
J. Ian Munro · Gonzalo Navarro ·
Yakov Nekrich

Abstract The problem of parameterized range majority asks us to preprocess a string of length n such that, given the endpoints of a range, one can quickly find all the distinct elements whose relative frequencies in that range are more than a threshold τ. This is a more tractable version of the classical problem of finding the range mode, which is unlikely to be solvable in polylogarithmic time and linear space. In this paper we give the first linear-space solution with optimal $\mathcal{O}(1/\tau)$ query time, even when τ can be specified with the query.

We then consider data structures whose space is bounded by the entropy of the distribution of the symbols in the sequence. For the case when the alphabet size σ is polynomial on the computer word size, we retain the optimal time within optimally compressed space (i.e., with sublinear redundancy). Otherwise, either the compressed space is increased by an arbitrarily small constant

Funded in part by Academy of Finland grant 268324, by NSERC grant RGPIN-07185-2020, by ANID – Millennium Science Initiative Program – Code ICN17_002, and by Fondecyt Grant 1-200038, Chile. An early partial version of this article appeared in *Proc. WADS 2013*.

Djamal Belazzougui
Research Center on Technical and Scientific Information (CERIST), Algeria. E-mail: djamal.belazzougui@gmail.com

Travis Gagie
Faculty of Computer Science, Dalhousie University, Halifax, Canada. E-mail: travis.gagie@gmail.com

J. Ian Munro
David Cheriton School of Computer Science, University of Waterloo, Canada. E-mail: imunro@uwaterloo.ca

Gonzalo Navarro
Millennium Institute for Foundational Research on Data, Department of Computer Science, University of Chile, Chile. E-mail: gnavarro@dcc.uchile.cl

Yakov Nekrich
Department of Computer Science, Michigan Technological University, Houghton, USA. E-mail: yakov.nekrich@googlemail.com

factor or the time rises to any function in $(1/\tau) \cdot \omega(1)$. We obtain the same results on the complementary problem of parameterized range minority.

Keywords Compressed data structures · Range majority and minority

1 Introduction

Finding frequent elements in a dataset is a fundamental operation in data mining. Consider, for example, spotting frequent terms tweeted during a certain event, or frequently visited web pages, or items frequently purchased in an online store during the weekend.

The most frequent elements can be difficult to spot when all the elements have nearly equal frequencies. It is more natural, however, to be interested in the most frequent elements only if they really are frequent. For example, Boyer and Moore [8] showed how we can scan a given string twice using $\mathcal{O}(1)$ space and find a majority element if one exists. Generalizing Boyer and Moore's result (see [11]), Misra and Gries [31] showed how, given a string and a threshold $0 < \tau \leq 1$, we can scan the string twice using $\mathcal{O}(1/\tau)$ space and find all the distinct elements whose relative frequencies exceed τ. These elements are called the τ-majorities of the string. If the element universe is $[1..\sigma]$, their algorithm can run in linear time and $\mathcal{O}(\sigma)$ space. Demaine et al. [13] rediscovered the algorithm and deamortized the cost per element; Karp et al. [26] rediscovered it again, obtaining $\mathcal{O}(1/\tau)$ space and linear randomized time. As Cormode and Muthukrishnan [12] put it, "papers on frequent items are a frequent item!".

Krizanc et al. [29] introduced the problem of preprocessing the string such that later, given the endpoints of a range, we can quickly return the mode of that range (i.e., the most frequent element). They gave two solutions, one of which takes $\mathcal{O}\left(n^{2-2\epsilon}\right)$ space for any fixed positive $\epsilon \leq 1/2$, and answers queries in $\mathcal{O}(n^\epsilon \lg\lg n)$ time; the other takes $\mathcal{O}\left(n^2 \lg\lg n / \lg n\right)$ space and answers queries in $\mathcal{O}(1)$ time. Petersen [35] reduced Krizanc et al.'s first time bound to $\mathcal{O}(n^\epsilon)$ for any fixed non-negative $\epsilon < 1/2$, and Petersen and Grabowski [36] reduced the second space bound to $\mathcal{O}\left(n^2 \lg\lg n / \lg^2 n\right)$. Chan et al. [9] gave an $\mathcal{O}(n)$ space solution that answers queries in $\mathcal{O}\left(\sqrt{n/\lg n}\right)$ time. They also gave evidence suggesting we cannot easily achieve query time substantially smaller than $\sqrt{n}$ using linear space; however, the best known lower bound [23] says only that we cannot achieve query time $o(\lg(n)/\lg(sw/n))$ using s words of w bits each. Because of the difficulty of supporting range mode queries, Bose et al. [7] and Greve et al. [23] considered the problem of approximate range mode, for which we are asked to return an element whose frequency is at least a constant fraction of the mode's frequency. Bose et al. showed, for example, how to 4-approximate the range mode in linear space with constant query time or $(1+\epsilon)$-approximate the range mode in $\mathcal{O}(n/\epsilon)$ space with $\mathcal{O}(\lg\lg n + \lg(1/\epsilon))$ query time. Among other results, Greve et al. improved Bose et al.'s second query time to $\mathcal{O}(\lg(1/\epsilon))$.

Karpinski and Nekrich [27] took a different direction, analogous to Misra and Gries' approach, when they introduced the problem of preprocessing the string such that later, given the endpoints of a range, we can quickly return the τ-majorities of that range. We refer to this problem as parameterized range majority. Note that, if a τ-majority exists, a random element in the range matches it with probability at least τ, so in expectation we find a τ-majority after $1/\tau$ attempts. The main challenges are how to obtain worst-case times to obtain all the τ-majorities by storing suitable sets of candidates, and how to efficiently calculate their frequency in the range.

Assuming τ is fixed when we are preprocessing the string, Karpinski and Nekrich [27] showed how we can store the string in $\mathcal{O}(n(1/\tau))$ space and answer queries in $\mathcal{O}\big((1/\tau)(\lg\lg n)^2\big)$ time. They also gave bounds for dynamic and higher-dimensional versions. Durocher et al. [14] independently posed the same problem and showed how we can store the string in $\mathcal{O}(n\lg(1/\tau+1))$ space and answer queries in $\mathcal{O}(1/\tau)$ time. Notice that, because there can be up to $1/\tau$ distinct elements to return, this time bound is worst-case optimal. Gagie et al. [20] showed how to store the string in compressed space — i.e., $\mathcal{O}(n(H+1))$ bits, where H is the entropy of the distribution of elements in the string — such that we can answer queries in $\mathcal{O}((1/\tau)\lg\lg n)$ time. Note that $H \leq \lg\sigma$, thus $\mathcal{O}(n(H+1))$ bits is $\mathcal{O}(n)$ space. They also showed how to handle a variable τ and still achieve optimal query time, at the cost of increasing the space bound by a $(\lg n)$-factor. That is, they gave a data structure that stores the string in $\mathcal{O}(n(H+1))$ words —of $w = \Omega(\lg n)$ bits each— such that later, given the endpoints of a range and τ, we can return the τ-majorities of that range in $\mathcal{O}(1/\tau)$ time. Chan et al. [10] gave another solution for variable τ, which also has $\mathcal{O}(1/\tau)$ query time and uses $\mathcal{O}(n\lg n)$ space. All these results are summarized in Table 1 together with our new results. Related work includes dynamic structures [27,16,21], approximate solutions [30,39], and encodings that do not access the string at query time [34,22].

Our contributions The central result of this paper is the first linear-space and optimal-time data structure for parameterized range majority, with variable τ. Our linear space is indeed $\mathcal{O}(n\lg\sigma)$ bits, stricter than other linear-space solutions using $\mathcal{O}(n\lg n)$ bits. This is summarized in the following theorem.

Theorem 1 *Let $S[1..n]$ be a string over alphabet $[1..\sigma]$. We can store S in $\mathcal{O}(n\lg\sigma)$ bits such that later, given the endpoints of a range and τ, we can return the τ-majorities for that range in time $\mathcal{O}(1/\tau)$. The structure is built within $\mathcal{O}(n\lg n)$ deterministic time and $\mathcal{O}(n\lg\sigma)$ bits.*

This theorem is proved along Sections 3 and 4. In the former, we consider the case where the alphabet of the string is polynomial on the computer word size, $\lg\sigma = \mathcal{O}(\lg w)$. We then give a linear-space structure with worst-case optimal $\mathcal{O}(1/\tau)$ query time, based on spotting a sufficient set of candidates and then checking each one in constant time. In Section 4 we extend this solution to the more challenging case of larger alphabets, $\lg\sigma = \omega(\lg w)$, where another

Table 1 Results for the problem of parameterized range majority on a string of length n over an alphabet $[1..\sigma]$ with $\sigma \leq n$ in which the distribution of the elements has entropy $H \leq \lg \sigma$. Note that all the spaces given in bits are in $\mathcal{O}(n)$ words.

source	space	time	τ is
Karpinski and Nekrich [27]	$\mathcal{O}(n(1/\tau))$ words	$\mathcal{O}\big((1/\tau)(\lg\lg n)^2\big)$	fixed
Durocher et al. [14]	$\mathcal{O}(n \lg(1/\tau))$ words	$\mathcal{O}(1/\tau)$	fixed
Gagie et al. [20]	$\mathcal{O}(n(H+1))$ bits	$\mathcal{O}((1/\tau)\lg\lg\sigma)$	fixed
Chan et al. [10]	$\mathcal{O}(n \lg n)$ words	$\mathcal{O}(1/\tau)$	variable
Gagie et al. [20]	$\mathcal{O}(n(H+1))$ words	$\mathcal{O}(1/\tau)$	variable
Theorem 5 ($\lg\sigma = \mathcal{O}(\lg w)$)	$nH + o(n)$ bits	$\mathcal{O}(1/\tau)$	variable
Theorem 6	$nH + o(n)(H+1)$ bits	Any $(1/\tau)\cdot\omega(1)$	variable
Theorem 7	$(1+\epsilon)nH + o(n)$ bits	$\mathcal{O}(1/\tau)$	variable

mechanism to check candidates in constant time is needed. Theorems 2 and 3, proved respectively in those sections, amount to Theorem 1.

The paper then enters in a more technical stage, where we show that the linear space can be reduced to n times the entropy of the distribution of the symbols in the string S (plus a sublinear redundancy); we call this "optimal compressed space". For small alphabets, Section 5 shows how to obtain optimal compressed space with no slowdown on the optimal query time. For large alphabets, instead, Section 6 obtains either optimal time on nearly-optimal compressed space, or nearly-optimal time on optimally compressed space. More precisely, we obtain the following results (see Table 1):

1. $\mathcal{O}(1/\tau)$ query time using $nH + o(n)$ bits, where $\lg\sigma = \mathcal{O}(\lg w)$;
2. $\mathcal{O}(1/\tau)$ query time using $(1+\epsilon)nH + o(n)$ bits, for any constant $\epsilon > 0$; and
3. any query time of the form $(1/\tau)\cdot\omega(1)$ using $nH + o(n)(H+1)$ bits.

In all cases, we preserve constant-time access to S within the compressed space we use, and our queries require $\mathcal{O}(1/\tau)$ extra working space. As a byproduct, we also show how to find the range mode in a time that depends on how frequent it actually is.

We finally consider the complementary problem of parameterized range minority, which was introduced by Chan et al. [10] (and then generalized to trees by Durocher et al. [15]). For this problem we are asked to preprocess the string such that later, given the endpoints of a range, we can return (if one exists) an element that occurs in that range but is not one of its τ-majorities. Such an element is called a τ-minority for the range. At first, finding a τ-minority might seem harder than finding a τ-majority because, for example, we are less likely to find a τ-minority by sampling. Nevertheless, Chan et al. gave an $\mathcal{O}(n)$ space solution with $\mathcal{O}(1/\tau)$ query time even for variable τ. In Section 7 we exploit the duality with τ-majorities in order to reuse the results obtained for the latter, so that the tradeoffs (1)–(3) are obtained for τ-minorities as well. Actually, a single data structure with the spaces given in points (1)–(3) solves at the same time τ-minority and τ-majority queries.

Our results dominate all the previous work [27,14,20,10]. An early partial version of this article [4] already included the optimal-time and linear-space result, but its compressed structures were not so efficient: their structure for optimal-time τ-majorities required $\mathcal{O}(n \lg\lg\sigma)$ bits of space instead of our near-nH bits space in (1–2); their structures using near-nH bits of space required $\mathcal{O}((1/\tau)\lg\lg\sigma)$ or $\mathcal{O}((1/\tau)\lg(1/\tau)/\lg\lg n)$ time, instead of our optimal or near-optimal time in (3); and their structure for optimal-time τ-minorities in optimal time required $\mathcal{O}(n)$ extra bits.

2 Preliminaries

We use the RAM model of computation with word size in bits $w = \Omega(\lg n)$, allowing multiplications. The input is an array $S[1..n]$ of symbols (or "elements") from $[1..\sigma]$, where for simplicity we assume $\sigma \le n$ (otherwise we could remap the alphabet so that every symbol actually appears in S, without changing the output of any τ-majority or τ-minority query).

2.1 Access, select, rank, and partial rank

Let $S[1..n]$ be a string over alphabet $[1..\sigma]$, for $\sigma \le n$, and let $H \le \lg\sigma$ be the entropy of the distribution of elements in S, also called the zero-order entropy of S; that is, $H = \sum_{a\in[1..\sigma]} \frac{n_a}{n} \lg \frac{n}{n_a}$, where each element a appears n_a times in S. An access query on S takes a position k and returns $S[k]$; a rank query takes an alphabet element a and a position k and returns $\mathrm{rank}_a(S,k)$, the number of occurrences of a in $S[1..k]$; a select query takes an element a and a rank r and returns $\mathrm{select}_a(S,r)$, the position of the rth occurrence of a in S (or $n+1$ if there are fewer than r occurrences of a in S). A partial rank query, $\mathrm{rank}_{S[k]}(S,k)$, is a rank query with the restriction that the element a must occur in the position k. These are among the most well-studied operations on strings, so we state here only the results most relevant to this paper.

For $\sigma = 2$ and any constant c, Pătraşcu [37] showed how we can store S in $nH + \mathcal{O}(n/\lg^c n)$ bits, supporting all the queries in time $\mathcal{O}(c)$. If S has m 1s, this space is $\mathcal{O}\big(m \lg \frac{n}{m} + n/\lg^c n\big)$ bits. For $\lg\sigma = \mathcal{O}(\lg\lg n)$, Ferragina et al. [17] showed how we can store S in $nH + o(n)$ bits and support all the queries in $\mathcal{O}(1)$ time. This result was later extended to the case $\lg\sigma = \mathcal{O}(\lg w)$ [6, Thm. 7]. Barbay et al. [1] showed how, for any positive constant ϵ, we can store S in $(1+\epsilon)nH + o(n)$ bits and support access and select in $\mathcal{O}(1)$ time and rank in $\mathcal{O}(\lg\lg\sigma)$ time. Alternatively, they can store S in $nH + o(n)(H+1)$ bits and support either access or select in time $\mathcal{O}(1)$, and the other operation, as well as rank, in time $\mathcal{O}(\lg\lg\sigma)$. Belazzougui and Navarro [6, Thm. 8] improved the time of rank to $\mathcal{O}(\lg\lg_w \sigma)$, which they proved optimal, and the time of the non-constant operation to any desired function in $\omega(1)$. Belazzougui and Navarro [5, Sec. 3] showed how to support $\mathcal{O}(1)$-time partial rank using $o(n)(H+1)$ further bits.

If $\sigma > n$, we can remap the distinct symbols that appear in S to $[1..\sigma']$, where $\sigma' \leq n$. A bitvector $B[1..\sigma]$ with 1s the σ' positions of the symbols that appear in S can be used to translate back from the mapped symbol a' to the original symbol $a = \mathrm{select}_1(B, a')$ in $\mathcal{O}(1)$ time and $\sigma' \lg \frac{\sigma}{\sigma'} + \mathcal{O}(\sigma') \leq n \lg \frac{\sigma}{n} + \mathcal{O}(n)$ additional bits. We omit this technical detail for simplicity in the article; we also use some simpler variants of the above results.

2.2 Successors in succinct space

The successor of x in a set $X = \{x_1, \ldots, x_m\}$, with $x_i < x_{i+1}$, is the smallest $x_i \geq x$. The succinct SB-tree [24, Lem 3.3] is a data structure for efficiently computing successors[1]. If the numbers x_i are integers in $[1..n]$, this data structure uses $\mathcal{O}(m \lg\lg n)$ bits and finds the successor of any x in time $\mathcal{O}(\lg m / \lg\lg n)$, requiring only one access to some x_i (the succinct SB-tree does not store the set X, so it needs a separate data structure providing constant-time access to any $x_i \in X$). The succinct SB-tree also needs a table of size $\mathcal{O}(n^\gamma)$ for any constant $0 < \gamma < 1$, which is independent of X and thus can be shared among all the structures on the same universe. The succinct SB-tree can be built in $\mathcal{O}(m)$ time [32, Sec. 4].

In this article we will need to store succinct SB-trees on universes $[1..n']$ for $n' < n$, retaining the $\mathcal{O}(\lg m / \lg\lg n)$ successor time but reducing the space to $\mathcal{O}(m \lg\lg n')$ bits. We show next that this is indeed possible.

Lemma 1 *We can store a set X of m integers in $[1..n']$ in a succinct SB-tree variant that requires $\mathcal{O}(m \lg\lg n') + \mathcal{O}(n^\gamma)$ bits of space, for any $n \geq n'$ and constant $0 < \gamma < 1$, and computes successors on X in time $\mathcal{O}(\lg m / \lg\lg n)$ plus access to one element of X. The $\mathcal{O}(n^\gamma)$ bits depend only on $\lceil \sqrt{\lg n} \rceil$ and $\lceil \lg\lg n' \rceil$.*

Proof The proof of Lemma 3.3 in Grossi et al. [24] can be used almost verbatim. We maintain the same succinct B-tree of arity $b = \lceil \sqrt{\lg n} \rceil$ and build, for each node, a Patricia tree on its b separator keys. The fact that the keys are shorter (i.e., of $\lg n'$ bits) shows up when storing the Patricia tree skips: the whole Patricia tree (without storing the explicit keys) can be encoded in $b \lceil \lg\lg n' \rceil$ bits, plus $\mathcal{O}(b)$ bits to describe the tree topology. This contributes the bulk of the space, $\mathcal{O}(m \lg\lg n')$ bits.

A table with $2^{\mathcal{O}(b \lg\lg n')} = \mathcal{O}(n^\gamma / \lg n)$ entries simulates any Patricia tree traversal in constant time; this requires $\mathcal{O}(n^\gamma)$ bits. Note that this table depends only on $b = \lceil \sqrt{\lg n} \rceil$ and on the number of bits used to encode the skips in the Patricia tree, $\lceil \lg\lg n' \rceil$. □

[1] In that paper they find the predecessor of x, which is the largest $x_i \leq x$, but the problem is analogous.

2.3 Characterizing range majorities and minorities

If we preprocess S to answer the operations described in Section 2.1, then we can easily determine whether a is a τ-majority or a τ-minority on $S[i..j]$ by comparing $\mathrm{rank}_a(S, j) - \mathrm{rank}_a(S, i-1)$ with $\tau(j-i+1)$. However, those rank operations cannot be supported in constant time on large alphabets. The next simple lemma, which we will use throughout, gives a characterization in terms of the other string operations, which can be supported in constant time.

Lemma 2 *Suppose we know the position of the leftmost occurrence of an element in a range. Then we can check whether that element is a τ-minority or a τ-majority using a partial rank query and a select query on S.*

Proof Let k be the position of the first occurrence of a in $S[i..j]$. If $S[k]$ is the rth occurrence of a in S, then a is a τ-minority for $S[i..j]$ if and only if the $(r + \lfloor \tau(j-i+1) \rfloor)$th occurrence of a in S is strictly after j; otherwise a is a τ-majority. That is, we can check whether a is a τ-minority for $S[i..j]$ by checking whether

$$\mathrm{select}_a(S, \mathrm{rank}_a(S, k) + \lfloor \tau(j-i+1) \rfloor) > j \,;$$

since $S[k] = a$, computing $\mathrm{rank}_a(S, k)$ is only a partial rank query. □

2.4 Colored range listing

Muthukrishnan [33] showed how we can store $S[1..n]$ such that, given the endpoints of a range, we can quickly list the distinct elements in that range and the positions of their leftmost occurrences therein. Let $C[1..n]$ be the array in which $C[k]$ is the position of the rightmost occurrence of the element $S[k]$ in $S[1..k-1]$ — i.e., the last occurrence before $S[k]$ itself — or 0 if there is no such occurrence. Notice $S[k]$ is the first occurrence of that distinct element in a range $S[i..j]$ if and only if $i \le k \le j$ and $C[k] < i$. We store C, implicitly or explicitly, and a data structure supporting $\mathcal{O}(1)$-time range-minimum queries on C: it returns the position of the leftmost occurrence of the minimum in any given range. Sadakane [38] and Fischer and Heun [19] gave $\mathcal{O}(n)$-bit data structures supporting $\mathcal{O}(1)$-time range-minimum queries.

To list the distinct elements in a range $S[i..j]$, we find the position m of the leftmost occurrence of the minimum in the range $C[i..j]$; check whether $C[m] < i$; and, if so, output $S[m]$ and m and recurse on $C[i..m-1]$ and $C[m+1..j]$. This procedure is online — i.e., we can stop it early if we want only a certain number of distinct elements — and the time it takes per distinct element is $\mathcal{O}(1)$ plus the time to access C.

Suppose we already have data structures supporting access, select and partial rank queries on S, all in $\mathcal{O}(t)$ time, for some t. Notice that $C[k] = \mathrm{select}_{S[k]}(S, \mathrm{rank}_{S[k]}(S, k) - 1)$, so we can also support access to C in $\mathcal{O}(t)$ time. Therefore, we can implement Muthukrishnan's solution using $\mathcal{O}(n)$ extra bits such that it takes $\mathcal{O}(t)$ time per distinct element listed.

2.5 Sparsifying Muthukrishnan's structure

To reduce the $\mathcal{O}(n)$ bits of extra space of Section 2.4 to $o(n)$, we can sparsify the range-minimum data structure. Such a result was sketched by Hon et al. [25, Thm. 7], but it lacks sufficient detail to ensure correctness. We give these details next.

Let $g(n) = \omega(1)$. We cut the sequence into blocks of length $g(n)$, choose the $n/g(n)$ minimum values of each block, and build the range-minimum data structure on the new array $C'[1..n/g(n)]$ (i.e., $C'[i]$ stores the minimum of $C[(i-1) \cdot g(n) + 1..i \cdot g(n)]$). This requires $\mathcal{O}(n/g(n)) = o(n)$ bits. Mutukrishnan's algorithm is then run over C' as follows. We map the range $C[i..j]$ to C' and find the minimum position in the mapped range of C'. We then recursively process its left interval, then process the minimum of C' by considering the $g(n)$ corresponding cells in C, and finally process the right part of the interval. The recursion stops when the interval becomes empty or when all the $g(n)$ elements in the block of C are already reported.

Lemma 3 *The procedure described identifies the leftmost positions of all the distinct elements in an interval $S[i..j]$, working over at most $g(n)$ cells per new element discovered.*

Proof Let us first consider the case of block-aligned intervals $S[i..j]$. We prove, by induction on the size of the current subinterval $[\ell..r]$ (which is always block-aligned), that if we have already reported all the distinct elements in $S[i..\ell-1]$ before the recursive call on $[\ell..r]$, then we have reported all the distinct elements in $S[i..r]$ after the call. This is clearly sufficient to establish the result.

Let k' be the position of the minimum in $C'[(\ell-1)/g(n)+1..r/g(n)]$ and let k be the position of the minimum in $C[(k'-1) \cdot g(n)+1..k' \cdot g(n)]$. Then $C[k]$ is the minimum in $C[\ell..r]$ and $S[k]$ is the leftmost occurrence in $S[\ell..r]$ of the element $a = S[k]$. If $C[k] \geq i$, then a already occurs in $S[i..\ell-1]$ and we have already reported it. Since the minimum of $C[\ell..r]$ is within the block k' of C', it is sufficient that $C[k] \geq i$ for all the positions k in that block to ensure that all the values in $S[\ell..r]$ have already been reported, in which case we can stop the procedure. The $g(n)$ scanned cells can be charged to the function that recursively invoked the interval $[\ell..r]$.

Otherwise, we recursively process the interval to the left of block k', $C[\ell..(k'-1) \cdot g(n)]$, which by inductive hypothesis reports the unique elements in that interval. Then we process the current block of size $g(n)$, finding at least the new occurrence of element $S[k]$ (which cannot have been found to the left of k'). Finally, we process the interval to the right of k', $C[k' \cdot g(n)+1..r]$, where the inductive hypothesis again holds.

Note that the method is also correct if, instead of checking whether all the elements in the block k' of C' are $\geq i$, we somehow check that all of them have already been reported. We will use this variant later in the paper.

For general ranges $S[i..j]$, we must include in the range of C' the two partially overlapped blocks on the extremes of the range. When it comes to process one of those blocks, we only consider the cells that are inside $[i..j]$; the condition to report an element is still that $C[k] < i$. $\square$

3 Parameterized Range Majority on Small Alphabets

In this section we consider the case $\lg \sigma = \mathcal{O}(\lg w)$, where rank queries on S can be supported in constant time. Our strategy is to find a set of $\mathcal{O}(1/\tau)$ candidates that contain all the possible τ-majorities and then check them one by one, counting their occurrences in $S[i..j]$ via rank queries on S. The time will be worst-case optimal, $\mathcal{O}(1/\tau)$, and our structures will use linear space, $\mathcal{O}(n \lg \sigma)$ bits. In Section 5 we will obtain compressed space.

First, note that if $\tau < 1/\sigma$, we can simply assume that all the σ symbols are candidates for majority, and check them one by one; therefore we care only about how to find $\mathcal{O}(1/\tau)$ candidates in the case $\tau \geq 1/\sigma$.

3.1 Structure

We store an instance of the structure of Belazzougui and Navarro [6, Thm. 5] supporting access, rank, and select on S in $\mathcal{O}(1)$ time, using $n \lg \sigma + o(n)$ bits. For every $0 \leq t \leq \lceil \lg \sigma \rceil$ and $t \leq b \leq \lfloor \lg n \rfloor$, we divide S into blocks of length 2^{b-1} and store a binary string $G_b^t[1..n]$ in which $G_b^t[k] = 1$ if (1) the element $S[k]$ occurs at least 2^{b-t} times in $S[k-2^{b+1}..k+2^{b+1}]$, and (2) k is the leftmost or rightmost position where $S[k]$ occurs in its block. We use $G_b^t[1..n]$ to answer queries with $\tau \geq 1/\sigma$ and $b \geq t$, where $b = \lfloor \lg(j-i+1) \rfloor$ and $t = \lceil \lg(1/\tau) \rceil$; we will say later how we deal with queries with $\tau < 1/\sigma$ or $b < t$.

Assuming $\tau < 1/\sigma$ and $b \geq t$, the following lemma shows that it is sufficient to consider the candidates $S[k]$ for $i \leq k \leq j$ where $G_b^t[k] = 1$.

Lemma 4 *For every τ-majority a of $S[i..j]$ there exists some $k \in [i..j]$ such that $S[k] = a$ and $G_b^t[k] = 1$.*

Proof Since $S[i..j]$ cannot be completely contained in a block of length 2^{b-1}, if $S[i..j]$ overlaps a block then it includes one of that block's endpoints. Therefore, if $S[i..j]$ contains an occurrence of an element a, then it includes the leftmost or rightmost occurrence of a in some block. Suppose a is a τ-majority in $S[i..j]$, and $b \geq t$. For all $i \leq k \leq j$, a occurs at least $\tau 2^b \geq 2^{b-t}$ times in $S[k-2^{b+1}..k+2^{b+1}]$, so since some occurrence of a in $S[i..j]$ is the leftmost or rightmost in its block, it is flagged by a 1 in $G_b^t[i..j]$. $\square$

The number of distinct elements that occur at least 2^{b-t} times in a range of size $\mathcal{O}(2^b)$ is $\mathcal{O}(2^t)$, so in each block there are $\mathcal{O}(2^t)$ positions flagged by 1s in G_b^t, for a total of $m = \mathcal{O}(n\, 2^{t-b})$ 1s. It follows that we can store an instance of the structure of Pătraşcu [37] (recall Section 2.1) supporting $\mathcal{O}(1)$-time access, rank and select on G_b^t in $\mathcal{O}(n 2^{t-b}(b-t) + n/\lg^3 n)$ bits in total. Summing over t from 0 to $\lceil \lg \sigma \rceil$ and over b from t to $\lfloor \lg n \rfloor$, calculation shows we use a total of $\mathcal{O}(n \lg \sigma)$ bits for the binary strings.

3.2 Queries

Given endpoints i and j and a threshold τ, if $\tau < 1/\sigma$, we simply report every element $a \in [1..\sigma]$ such that $\mathrm{rank}_a(S, j) - \mathrm{rank}_a(S, i-1) > \tau(j - i + 1)$, in total time $\mathcal{O}(\sigma) = \mathcal{O}(1/\tau)$.

Otherwise, we compute b and t as explained and, if $b < t$, we run a version of Misra and Gries' algorithm [31] that takes $\mathcal{O}(j - i) = \mathcal{O}(1/\tau)$ worst-case time on $S[i..j]$. To do this, we take advantage of the rank and select operations on S. We create a doubly-linked list with the positions i to j, plus an array $T[1..j - i + 1]$ that provides direct access to the list nodes, that is, $T[k]$ points to the list node representing $S[i + k - 1]$. We take the element $S[i] = a$ at the head of the list and know that it is a τ-majority in $S[i..j]$ if $\mathrm{rank}_a(S, j) - \mathrm{rank}_a(S, i - 1) > \tau(j - i + 1)$. If it is, we immediately report it. In either case, we remove all the occurrences of a from the doubly-linked list, that is, the list nodes $T[\mathrm{select}_a(S, \mathrm{rank}_a(S, i) + r)]$, $r = 0, 1, 2, \ldots$ until exhausting T. We proceed with the new header of the doubly-linked list, which points to a different element $S[i'] = a'$, and so on. It is clear that we perform $\mathcal{O}(j - i)$ constant-time rank and select operations on S, and that at the end we have found all the τ-majorities.

Finally, if $\tau \geq 1/\sigma$ and $b \geq t$, we use rank and select on G^t_b to find all the 1s in $G^t_b[i..j]$. Since $S[i..j]$ overlaps at most 5 blocks of length 2^{b-1}, it contains $\mathcal{O}(1/\tau)$ elements flagged by 1s in G^t_b; therefore, we have $\mathcal{O}(1/\tau)$ candidates to evaluate, and these include all the possible τ-majorities. Each candidate a is tested in constant time for the condition $\mathrm{rank}_a(S, j) - \mathrm{rank}_a(S, i - 1) > \tau(j - i + 1)$.

3.3 Construction

The construction time is dominated by the creation of the bitvectors G^t_b. For each value of b, we slide a window of length $2^{b+2}+1$ over S. This window covers 9 consecutive blocks of length 2^{b-1} in S. We store an array that, at any window position $S[k - 2^{b+1}..k + 2^{b+1}]$, will maintain for each symbol $a \in [1..\sigma]$ (1) the number of times a appears in the window, and (2) the first and last position of a in each of the 9 blocks covered by the window. It is easy to maintain this information in $\mathcal{O}(1)$ time as we slide the window to the next position in S. Then, for any window $S[k - 2^{b+1}..k + 2^{b+1}]$, if k is the first or the last position where $a = S[k]$ occurs in its block (which is currently covered by the window), then we set $G^t_b[k] = 1$ for all t such that $S[k]$ occurs $occ \geq 2^{b-t}$ times in the window, that is, for all $\lceil b - \lg(occ)\rceil \leq t \leq \min(b, \lceil \lg \sigma \rceil)$.

Sliding the window requires $\mathcal{O}(n)$ time for each value of b, and therefore it amounts to $\mathcal{O}(n \lg n)$ time in total. On the other hand, we work $\mathcal{O}(1)$ time for each bit set in some G^t_b, and we have shown that these amount to $\mathcal{O}(n \lg \sigma)$ in total. Therefore, we create all the bitvectors G^t_b in time $\mathcal{O}(n \lg n)$. As for the space, sliding the window requires just $\mathcal{O}(\sigma)$ space for the window and block data. We can traverse S once per value of b, and then compress all the resulting bitvectors G_b^t for all t, before considering the next value of b. Before compressing, the bitvectors G_b^t for fixed b amount to $\mathcal{O}(n \lg \sigma)$ bits.

The construction time of the compressed bitvector representations for G_b^t [37] is linear in their bit-space[2], thus it adds up to $\mathcal{O}(n \lg \sigma)$ over all b and t. Finally, the representation we use for S [6, Thm. 5] can be built within $\mathcal{O}(n \lg_w \sigma)$ time and $\mathcal{O}(n \lg \sigma)$ bits. The total construction cost is then $\mathcal{O}(n \lg n)$ time and $\mathcal{O}(\sigma \lg n + n \lg \sigma) = \mathcal{O}(n \lg \sigma)$ bits of space.

Theorem 2 *Let $S[1..n]$ be a string over alphabet $[1..\sigma]$, with $\lg \sigma = \mathcal{O}(\lg w)$. We can store S in $\mathcal{O}(n \lg \sigma)$ bits such that later, given the endpoints of a range and τ, we can return the τ-majorities for that range in time $\mathcal{O}(1/\tau)$. The structure is built within $\mathcal{O}(n \lg n)$ deterministic time and $\mathcal{O}(n \lg \sigma)$ bits.*

4 Parameterized Range Majority on Large Alphabets

Rank queries cannot be performed in constant time on large alphabets [6]. To obtain optimal query time in this case, where $\lg \sigma = \omega(\lg w)$, we resort to the use of Lemma 2 instead of performing rank queries on S. For this purpose, we must be able to find the leftmost occurrence of each τ-majority in a range. This is done by adding further structures on top of the bitvectors G_b^t used in Theorem 2. Those include a Muthukrishnan structure (Section 2.4) on a sampled set of positions plus succinct SB-trees (Section 2.2) to find successors between samples. The leftmost occurrence of any $a = S[k]$ with $G_b^t[k] = 1$ will then be found as the successor of i in the succinct SB-tree containing k (if there are no samples between i and k) or in the one containing the leftmost sampled occurrence of a in $S[i..j]$ (found with Muthukrishnan's algorithm).

The bitvectors G_b^t alone require $\mathcal{O}(n \lg \sigma)$ bits of space, whereas our further structures add another $\mathcal{O}(n \lg \sigma)$ bits. Within these $\mathcal{O}(n \lg \sigma)$ bits, we can also store a simple representation of S [1], which supports both access and select queries in constant time. We also add the structures to support partial rank in constant time, within $o(n \lg \sigma)$ further bits. Therefore we can apply Lemma 2 in constant time and solve τ-majority queries in time $\mathcal{O}(1/\tau)$. We consider compression in Section 6.

4.1 Structure

First, to cover the case $\tau < 1/\sigma$, we build the structure of Muthukrishnan on S, using $\mathcal{O}(n)$ extra bits as shown in Section 2.4, so that we can find the $\mathcal{O}(\sigma) = \mathcal{O}(1/\tau)$ leftmost occurrences of each distinct element in $S[i..j]$. On each leftmost occurrence we can then apply Lemma 2 in constant time. Now we focus on the case $\tau \geq 1/\sigma$.

In addition to each bitvector G_b^t of the previous section, we store a second bitvector J_b^t with a 1 marking $S[i] = a$ if $S[i]$ is the $(q \lg^4 n)$-th occurrence a

[2] M. Pătraşcu, personal communication, 2009.

in S, for some integer q, and some nearby occurrence $S[k]$ of a is marked by $G_b^t[k] = 1$, where "nearby" means $k - 2^{b+1} \leq i \leq k + 2^{b+1}$. More precisely, let $G_b^t[k] = 1$ mark $a = S[k]$, thus a occurs at least 2^{b-t} times in $S[k - 2^{b+1}..k + 2^{b+1}]$; let $i_1^a, i_2^a, \ldots, i_{n_a}^a$ be the positions of a in S, where $n_a = \mathrm{rank}_a(S, n) > \lg^4 n$. Then we mark in J_b^t the positions

$$\{i^a_{q \lg^4 n} \ : \ 1 \leq q \leq n_a / \lg^4 n \text{ and } k - 2^{b+1} \leq i^a_{q \lg^4 n} \leq k + 2^{b+1}\}.$$

For the subsequence S_b^t of elements of S marked in J_b^t, we build an instance of Muthukrishnan's structure. That is, we build the structure on the array C_b^t corresponding to the string S_b^t, where $S_b^t[k] = S[\mathrm{select}_1(J_b^t, k)]$. This string need not be stored explicitly whereas, in contrast, we do store C_b^t explicitly.

Furthermore, for each a and q, if it holds $J_b^t[i^a_{q \lg^4 n}] = 1$ for some b and t, we create a succinct SB-tree successor structure (Section 2.2) on the chunk of $\lg^4 n$ consecutive positions of a: $i^a_{1+(q-1)\lg^4 n}, \ldots, i^a_{q \lg^4 n}$. This structure is stored associated with the 1 at $J_b^t[i^a_{q \lg^4 n}]$ (all the 1s at the same position $i^a_{q \lg^4 n}$, for different b and t values, point to the same succinct SB-tree, as it does not depend on b or t). The SB-tree operates in time $\mathcal{O}\big(\lg(\lg^4 n)/\lg\lg n\big) = \mathcal{O}(1)$ and uses $\mathcal{O}\big(\lg^4 n \lg\lg n\big)$ bits. It needs constant-time access to the positions $i^a_{r+(q-1)\lg^4 n}$, as it does not store them. We provide those positions using $i_k^a = \mathrm{select}_a(S, k)$.

Added over all the symbols a, occurring n_a times in S, each bitvector J_b^t contains $\sum_a \lfloor n_a / \lg^4 n \rfloor = \mathcal{O}\big(n/\lg^4 n\big)$ 1s. Thus, added over every b and t, the bitvectors J_b^t, arrays C_b^t, and pointers to succinct SB-trees (using $\mathcal{O}(\lg n)$ bits per pointer), require $\mathcal{O}(n/\lg n) = o(n)$ bits. Each succinct SB-tree requires $\mathcal{O}\big(\lg^4 n \lg\lg n\big)$ bits, and they may be built for $\mathcal{O}\big(n/\lg^4 n\big)$ chunks, adding up to $\mathcal{O}(n \lg\lg n)$ bits. This is $\mathcal{O}(n \lg \sigma)$ since we are assuming $\lg \sigma = \Omega(\lg\lg n)$.

4.2 Queries

Given i and j, we compute $b = \lfloor \lg(j - i + 1) \rfloor$ and $t = \lceil \lg(1/\tau) \rceil$, and find the $\mathcal{O}(1)$ blocks of length 2^{b-1} overlapping $S[i..j]$. As in the previous section, every $G_b^t[k] = 1$ in $G_b^t[i..j]$ is a candidate to verify, but this time we need to find its leftmost occurrence in $S[i..j]$.

To find the leftmost position of $a = S[k]$, we see if the positions k and i are in the same chunk. That is, we compute the chunk index $q = \lceil \mathrm{rank}_a(S, k)/\lg^4 n \rceil$ of k (via a partial rank on S) and its limits $i_l = \mathrm{select}_a(S, (q-1)\lg^4 n)$ and $i_r = \mathrm{select}_a(S, q \lg^4 n)$. Then we see if $i_l < i \leq i_r$. In this case, we use the succinct SB-tree associated with $J_b^t[i_r] = 1$ to find the successor of i in time $\mathcal{O}(1)$. Then we use Lemma 2 from that position to determine in $\mathcal{O}(1)$ time if a is a τ-majority in $S[i..j]$.

If k is not in the same chunk of i, we disregard it because, in this case, there is an occurrence $S[i_l] = a$ in $S[i..j]$ that is marked in J_b^t. We will instead find separately the leftmost occurrence in $S[i..j]$ of any candidate a that is marked in $J_b^t[i..j]$, as follows. We apply Muthukrishnan's algorithm on the 1s of $J_b^t[i..j]$, to find the distinct elements of $S_b^t[\mathrm{rank}_1(J_b^t, i-1)+1..\mathrm{rank}_1(J_b^t, j)]$. Thus we obtain the leftmost sampled occurrences in $S[i..j]$ of all the τ-majorities, among other candidates. For each leftmost occurrence $S_b^t[k']$, it must be that $k = \mathrm{select}_1(J_b^t, k')$ is in the same chunk of i, and therefore we can find the successor of i using the corresponding succinct SB-tree in constant time, and then verify the candidate using Lemma 2.

It follows from the construction of J_b^t that the distinct elements sampled in any $S[i..j]$ must appear at least 2^{b-t} times in an interval of size $\mathcal{O}\big(2^b\big)$ containing $S[i..j]$, and so there can only be $\mathcal{O}(1/\tau)$ distinct sampled elements. Therefore, Muthukrishnan's algorithm on $J_b^t[i..j]$ gives us $\mathcal{O}(1/\tau)$ candidates to verify, in time $\mathcal{O}(1/\tau)$.

When $b < t$, we use our sequential algorithm of Section 3.2 with the only difference that, since we always find the leftmost occurrence of each candidate in $S[i..j]$, we can use Lemma 2 to verify the τ-majorities. Thus the algorithm uses only select and partial rank queries on S, and therefore it runs in time $\mathcal{O}(1/\tau)$ as well.

4.3 Construction

We enhance the construction of the bitvectors G_b^t of Section 3.3, which required $\mathcal{O}(n \lg n)$ time and $\mathcal{O}(n \lg \sigma)$ bits of space. For each $a \in [1..\sigma]$, we now also record (3) the number of times a has been seen so far, and (4) the doubly linked list of all the occurrence positions i of a that are within the current window and such that $\mathrm{rank}_a(S,i)$ is a multiple of $\lg^4 n$. This data is easily maintained in $\mathcal{O}(1)$ time each time we slide the window for a fixed value of b, and its extra space amounts to $\mathcal{O}\big(\sigma + n/\lg^4 n\big)$ words. Then, every time we mark some $G_b^t[k]=1$, we fetch the list of (sampled) positions associated with $a = S[k]$ and set all those positions in J_b^t.

The total number of 1s in all the bitvectors J_b^t is $\mathcal{O}\big(n/\lg^2 n\big)$, so this is the extra time we spend setting those bits. By building the bitvectors for each value of b separately (and for all the values of t together), the extra space stays within $\mathcal{O}(n \lg \sigma)$ bits, since we only maintain $\lg \sigma$ bitvectors G_b^t and J_b^t in plain form at the same time. Once those bitvectors G_b^t and J_b^t, for a fixed b, are completed, we compress them [37], which requires $\mathcal{O}(n \lg \sigma)$ time and space in total, as seen in Section 3.3.

Once all the bitvectors J_b^t are built and compressed, we traverse the 1s in each J_b^t and collect the corresponding symbols of S in S_b^t. From S_b^t we build C_b^t in time $\mathcal{O}(|S_b^t|)$ and $\mathcal{O}(\sigma + |S_b^t|)$ space. Finally, the range minimum query data structure on top of C_b^t is built in $\mathcal{O}(|C_b^t|)$ time [19]. Adding up over all the b and t values, this amounts to $\mathcal{O}\big(n/\lg^2 n\big)$ time and $\mathcal{O}\big(\sigma n + n/\lg^2 n\big)$ space.

Finally, we create a bitvector $J[1..n]$ that is the union of all the bits set in every J_b^t. For each $J[i]=1$, which corresponds to some $a = S[i]$, we create a succinct SB-tree with the occurrences $(r - \lg^4 n + 1)$th to rth of a in S, for $r = \mathrm{rank}_a(S,i)$. This set is collected using select queries on S, in total time $\mathcal{O}\big(\lg\lg\sigma + \lg^4 n\big)$, and then the succinct SB-tree is built in linear time,

$\mathcal{O}(\lg^4 n)$ (Section 2.2). Therefore, the total time to build the succinct SB-trees is $\mathcal{O}(n)$ and the extra space is negligible.

The remaining elements are the structure for S, Muthukrishnan's structure for the whole string S, and the partial rank data structures. The string representation [1] is easily built in $\mathcal{O}(n)$ time and $\mathcal{O}(n \lg \sigma)$ bits of space. Muthukrishnan's structure [33] is also easily built in $\mathcal{O}(n)$ time, but the intermediate representation of array C requires $n \lg n$ bits. Instead, we simulate access to C with constant-time select and partial rank queries on S, and therefore the space stays within $\mathcal{O}(n \lg \sigma)$ bits. Finally, the partial rank data structures we refer to [5, Sec. 3] require linear *randomized* time. However, if we can use any space in $\mathcal{O}(n \lg \sigma)$ bits for the final structure, then a simpler construction can be used [3, Lem. 3.5], that can be built in linear deterministic time. Overall, we retain the $\mathcal{O}(n \lg n)$ construction time within $\mathcal{O}(n \lg \sigma)$ bits of space.

Theorem 3 *Let $S[1..n]$ be a string over alphabet $[1..\sigma]$, with $\lg \sigma = \Omega(\lg \lg n)$. We can store S in $\mathcal{O}(n \lg \sigma)$ bits such that later, given the endpoints of a range and τ, we can return the τ-majorities for that range in time $\mathcal{O}(1/\tau)$. The structure is built within $\mathcal{O}(n \lg n)$ deterministic time and $\mathcal{O}(n \lg \sigma)$ bits.*

5 Compressing Space on Small Alphabets

In this section we show how to retain our optimal query time, $\mathcal{O}(1/\tau)$, while reducing the space to $nH+o(n)$ bits, whenever the alphabet size is small, $\lg \sigma = \mathcal{O}(\lg w)$. We build on the basic idea of Section 3. We first obtain succinct space, $n \lg \sigma + o(n)$ bits, by using bit-parallelism on a hierarchical representation of S. Then we change the plain representation of the small blocks in the hierarchy by an entropy-compressed representation.

5.1 Succinct space

To reduce the space we will open the structure we are using to represent S [6, Thm. 5]. This is a multiary wavelet tree: it cuts the alphabet range $[1..\sigma]$ into w^β contiguous subranges of about the same size, for some conveniently small constant $0 < \beta \leq 1/4$ such that w^β is a power of 2. The root node v of the wavelet tree stores the sequence $S_v[1..n]$ indicating the range to which each symbol of S belongs, $S_v[i] = \lceil S[i]/\lceil \sigma/w^\beta \rceil \rceil - 1$. This node has w^β children, where the $(p+1)$th child represents the subsequence of the symbols $S[i]$ such that $S_v[i] = p$. The alphabet of each child has been reduced to a range of size $\lceil \sigma/w^\beta \rceil$. This range is split again into w^β subranges, creating w^β children for each child, and so on. The process is repeated recursively until the alphabet range is of size at most w^β. The wavelet tree has height $\mathcal{O}(\lg_{w^\beta} \sigma) = \mathcal{O}\left(\frac{\lg \sigma}{\beta \lg w}\right) = \mathcal{O}(1)$, and at each level the strings S_v stored add up to $n \lg(w^\beta) = \beta n \lg w$ bits, for a total of $n \lg \sigma$ bits of space. The other

$o(n)$ bits are needed to provide constant-time rank and select support on the strings S_v.

We use this hierarchical structure to find the τ-majorities as follows. Assume we have the structures to find τ-majorities in any of the strings S_v associated with wavelet tree nodes v, in time $\mathcal{O}(1/\tau)$ (we show later how to provide those structures). Then, if a is a τ-majority in $S[i..j]$, the symbol $p = \lceil a/(\sigma/w^\beta)\rceil - 1$ is also a τ-majority in $S_v[i..j]$, where v is the wavelet tree root. Therefore, we find in time $\mathcal{O}(1/\tau)$ the τ-majorities p in $S_v[i..j]$. We verify if each symbol mapped to such a p is a τ-majority, by recursively looking for majorities in the $(p+1)$th child of v. In this child u, the range $S_v[i..j]$ is projected to $S_u[i_u..j_u] = S_u[\text{rank}_p(S_v, i-1)+1..\text{rank}_p(S_v, j)]$, and the corresponding threshold is $\tau_u = \tau(j-i+1)/(j_u-i_u+1) < 1$. This process continues recursively until we find the majorities in the leaf nodes, which correspond to actual symbols that can be reported as τ-majorities in $S[i..j]$.

The time to find the τ_u-majorities in each child u of the root v is $\mathcal{O}(1/\tau_u) = \mathcal{O}((j_u - i_u + 1)/((j-i+1)\tau))$. Added over all the children u, this gives a space per level of $\sum_u \mathcal{O}(1/\tau_u) = \sum_u \mathcal{O}((j_u - i_u + 1)/((j-i+1)\tau)) = \mathcal{O}(1/\tau)$. Adding this over all the levels, we obtain $\mathcal{O}\left(\frac{\lg \sigma}{\beta \lg w} \cdot (1/\tau)\right) = \mathcal{O}(1/\tau)$.

Finding τ'-majorities on tiny alphabets The remaining problem is how to find τ'-majorities on an alphabet $[0..\sigma'-1]$, where $\sigma' = w^\beta$, on each of the strings S_v of length n_v. Here is where we enhance the wavelet tree, by adding structures able to find those τ'-majorities on the strings S_v. We do almost as we did for Theorem 2, except that the range for b is slightly narrower: $\lfloor \lg(2^t \cdot w^\beta/4)\rfloor \le b \le \lfloor \lg n_v \rfloor$. Then calculation shows that the total space for the bitvectors G_b^t is $\mathcal{O}\left(\frac{n_v \lg \sigma' \lg w}{w^\beta} + \frac{n_v}{\lg n_v}\right) = o(n_v)$, so added over the whole wavelet tree is $o(n)$.

The price of using this higher lower bound for b is that it requires us to sequentially find τ'-majorities in time $\mathcal{O}(1/\tau')$ on ranges of length $\mathcal{O}\big((1/\tau')w^\beta\big)$. However, we can take advantage of the small alphabet. First, if $1/\tau' \ge \sigma'$, we just perform σ' pairs of constant-time rank queries on S_v. For $1/\tau' < \sigma'$, we will compute an array of σ' counters with the frequency of the symbols in the range, and then report those exceeding the threshold. The details are given in the Appendix.

5.2 Construction

The only difference with respect to Section 3.3 is that we build the τ-majority data structures for each level of the wavelet tree, thus the overall time is $\mathcal{O}(n \lg n \lg_w \sigma)$. However, since $\lg \sigma = \mathcal{O}(\lg w)$, the number of levels is constant and thus the construction time stays the same.

Theorem 4 *Let $S[1..n]$ be a string over alphabet $[1..\sigma]$, with $\lg \sigma = \mathcal{O}(\lg w)$. We can store S in $n \lg \sigma + o(n)$ bits such that later, given the endpoints of a range and τ, we can return the τ-majorities for that range in time $\mathcal{O}(1/\tau)$. The structure is built within $\mathcal{O}(n \lg n)$ deterministic time and $\mathcal{O}(n \lg \sigma)$ bits.*

5.3 Optimally compressed space

One choice to compress the space is to use a compressed representation of the strings S_v [17]. This takes chunks of $c = (\lg n)/2$ bits and assigns them a code formed by a header of $\lg c$ bits and a variable-length remainder of at most c bits. To decode a chunk in constant time, they use a directory of $\mathcal{O}(n_v \lg c/c)$ bits, plus a constant table of size $2^c = \mathcal{O}(\sqrt{n})$ that receives any encoded string and returns the original chunk. The compressed size of any string S_v with zero-order entropy $H(v)$ then becomes $n_v H(v) + \mathcal{O}(n_v \lg \sigma' \lg\lg n / \lg n)$ bits, which added over the whole wavelet tree is $nH + \mathcal{O}\left(\frac{n \lg \sigma \lg\lg n}{\lg n}\right)$ bits. This can be used in replacement of the direct representation of sequences S_v in Theorem 4, since we only change the way a chunk of $\Theta(\lg n)$ bits is read from any S_v. Note that we read chunks of w^β symbols from S_v, which could be $\omega(\lg n)$ if n is very small. To avoid this problem, we apply this method only when $\lg \sigma = \mathcal{O}(\lg\lg n)$, as in this case we can use computer words of $w = \lg n$ bits. The extra cost of creating the compressed representation is $\mathcal{O}(n)$.

Corollary 1 *Let $S[1..n]$ be a string whose distribution of symbols has entropy H, over alphabet $[1..\sigma]$, with $\lg \sigma = \mathcal{O}(\lg\lg n)$. We can store S in $nH + o(n)$ bits such that later, given the endpoints of a range and τ, we can return the τ-majorities for that range in time $\mathcal{O}(1/\tau)$. The structure is built within $\mathcal{O}(n \lg n)$ deterministic time and $\mathcal{O}(n \lg \sigma)$ bits.*

For the case where $\lg \sigma = \omega(\lg\lg n)$ but still $\lg \sigma = \mathcal{O}(\lg w)$, we use another technique. We represent S using the optimally compressed structure of Barbay et al. [1]. This structure separates the alphabet symbols into $\lg^2 n$ classes according to their frequencies. A sequence $K[1..n]$, where $K[i]$ is the class to which $S[i]$ is assigned, is represented using the structure of Corollary 1, which supports constant-time access, rank, and select (since the alphabet of K is of polylogarithmic size), and also τ-majority queries in time $\mathcal{O}(1/\tau)$. For each class c, a sequence $S_c[1..n_c]$ contains the subsequence of S of the symbols $S[i]$ where $K[i] = c$; the distinct symbols in S_c are mapped to a contiguous range $[1..\sigma_c]$. We will represent the subsequences S_c using Theorem 4. Then the structure for K takes $nH(K) + o(n)$ bits, where $H(K)$ is the entropy of the distribution of the symbols in K, and the structures for the strings S_c take $n_c \lg \sigma_c + o(n_c)$ bits. Barbay et al. show that these space bounds add up to $nH + o(n)$ bits and that one can support access, rank and select on S via access, rank and select on K and some S_c. The extra time to build this representation is $\mathcal{O}(n)$ and the space is $\mathcal{O}(n \lg \sigma)$ bits.

Our strategy to solve a τ-majority query on $S[i..j]$ resembles the one used to prove Theorem 4. We first run a τ-majority query on string K. This will yield the at most $1/\tau$ classes of symbols that, together, occur more than $\tau(j-i+1)$ times in $S[i..j]$. The classes excluded from this result cannot contain symbols that are τ-majorities. Now, for each included class c, we map the interval $S[i..j]$ to $S_c[i_c..j_c]$ in the subsequence of its class, where $i_c = \text{rank}_c(K, i-1) + 1$ and $j_c = \text{rank}_c(K, j)$, and then run a τ_c-majority query on $S_c[i_c..j_c]$, for $\tau_c =$

$\tau(j-i+1)/(j_c-i_c+1)$. The results obtained for each considered class c are reported as τ-majorities in $S[i..j]$. The query time, added over all the possible τ_c values, is $\sum_c \mathcal{O}(1/\tau_c) = \mathcal{O}(1/\tau)$ as before.

Theorem 5 *Let $S[1..n]$ be a string whose distribution of symbols has entropy H, over alphabet $[1..\sigma]$, with $\lg \sigma = \mathcal{O}(\lg w)$. We can store S in $nH + o(n)$ bits such that later, given the endpoints of a range and τ, we can return the τ-majorities for that range in time $\mathcal{O}(1/\tau)$. The structure is built within $\mathcal{O}(n \lg n)$ deterministic time and $\mathcal{O}(n \lg \sigma)$ bits.*

6 Compressing Space on Large Alphabets

In this section we compress the space of our solution of Section 4. This time, apparent limits on the operation times on strings [6] prevent us from reaching optimal query time and at the same time optimally compressed space, but we get close. We find several technical obstacles to obtain compressed space, in particular related to the succinct SB-trees used in Section 4. At the end, we give a simple application of our result to solve a variant of the range mode problem.

6.1 Compressed space

To reduce the space, we use the same strategy used to prove Theorem 5: we represent S using the optimally compressed structure of Barbay et al. [1]. This time, however, closer to the original article, we use different representations for the strings S_c with alphabets of size $\sigma_c \le w$ and of size $\sigma_c > w$. For the former, we use the representation of Theorem 4, which uses $n_c \lg \sigma_c + o(n_c)$ bits and answers τ_c-majority queries in time $\mathcal{O}(1/\tau_c)$. For the larger alphabets, we use a slight variant of Theorem 3: we use the same structures G_b^t, J_b^t, C_b^t, and pointers to succinct SB-trees, except that the lower bound for b will be $\lfloor \lg(2^t \cdot g(n,\sigma)) \rfloor$, for any function $g(n,\sigma) = \omega(1)$. The total space for the bitvectors G_b^t of string S_c is thus $\mathcal{O}\Big(\frac{n_c \lg \sigma_c \lg g(n,\sigma)}{g(n,\sigma)}\Big) = o(n_c \lg \sigma_c)$, whereas the other structures already used $o(n_c)$ bits (with a couple of exceptions we consider soon).

Then, representing S_c with the structure of Belazzougui and Navarro [6, Thm. 6], so that it supports select in time $\mathcal{O}(g(n,\sigma))$ and access in time $\mathcal{O}(1)$, the total space for S_c is $n_c \lg \sigma_c + o(n_c \lg \sigma_c)$, and the whole structure uses $nH + o(n)(H+1)$ bits. The structure is built in $\mathcal{O}(|S_c|)$ time and $\mathcal{O}(|S_c| \lg \sigma)$ bits of space. On the other hand, as noted before, the partial rank structures on S_c that use $o(|S_c| \lg \sigma)$ bits can be built in $\mathcal{O}(|S_c|)$ randomized time only.

The cases where $b \ge \lfloor \lg(2^t \cdot g(n,\sigma)) \rfloor$ are solved with $\mathcal{O}(1/\tau_c)$ applications of select on S, and therefore take time $\mathcal{O}((1/\tau_c)\, g(n,\sigma))$. Instead, the shorter ranges, of length $\mathcal{O}((1/\tau_c)\, g(n,\sigma))$, must be processed sequentially, as in Section 3.2. The space of the sequential algorithm can be maintained in

$\mathcal{O}(1/\tau_c) = \mathcal{O}(1/\tau)$ words as follows. We cut the interval $S_c[i_c..j_c]$ into chunks of $m = \lceil 1/\tau_c \rceil$ consecutive elements, and process each chunk in turn as in Section 3.2. The difference is that we maintain an array with the τ_c-majorities a we have reported and the last position p_a we have deleted in the lists. From the second chunk onwards, we remove all the positions of the known τ_c-majorities a before processing it, $\text{select}_a(S_c, \text{rank}_a(S_c, p_a) + r)$, for $r = 1, 2, \ldots$; note that $\text{rank}_a(S_c, p_a)$ is a partial rank query. Since select on S_c costs $\mathcal{O}(g(n, \sigma))$ and we perform $\mathcal{O}((1/\tau_c)\, g(n, \sigma))$ operations, the total time is $\mathcal{O}\big((1/\tau_c)\, g(n, \sigma)^2\big)$. Then we can retain the optimally compressed space and have any time of the form $\mathcal{O}((1/\tau)\, f(n, \sigma))$ by choosing $g(n, \sigma) = \sqrt{f(n, \sigma)}$.

There are, as anticipated, two final obstacles related to the space. The first are the $\mathcal{O}(n_c)$ bits of Muthukrishnan's structure associated with S_c to handle the case $\tau_c < 1/\sigma_c$. To reduce this space to $o(n_c)$, we sparsify the structure as in Section 2.5. The case of small τ_c is then handled in time $\mathcal{O}\big(\sigma_c\, g(n, \sigma)^2\big) = \mathcal{O}((1/\tau_c)\, f(n, \sigma))$ and the space for the sparsified structure is $\mathcal{O}(n_c / g(n, \sigma)) = o(n_c)$.

The second obstacle is the $\mathcal{O}(n_c \lg\lg n_c)$ bits used by the succinct SB-trees. We reduce their universe size as follows. We logically cut the string S_c into n_c/σ_c^2 pieces of length σ_c^2. For each symbol a we store a bitvector $B_a[1..n_c/\sigma_c^2]$ where $B_a[i] = 1$ if and only if a appears in the ith piece. These bitvectors require $\mathcal{O}\big(\sigma_c \cdot n_c/\sigma_c^2\big) = o(n_c)$ bits in total, including support for rank and select. The succinct SB-trees are now local to the pieces: a succinct SB-tree that spans several pieces is split into several succinct SB-trees, one covering the positions in each piece. The 1s corresponding to these pieces in bitvectors B_a point to the newly created succinct SB-trees. To find the successor of position i given that it is in the same chunk of $i_r > i$, with $J_b^t[i_r] = 1$, we first compute the piece $p = \lceil i/\sigma_c^2 \rceil$ of i and the piece $p_r = \lceil i_r/\sigma_c^2 \rceil$ of i_r, and see if i and i_r are in the same piece, that is, if $p = p_r$. If so, the answer is to be found in the succinct SB-tree associated with the 1 at $J_b^t[i_r]$. Otherwise, that original structure has been split into several structures, and the part that covers the piece of i is associated with the 1 at $B_a[p]$. It is possible, however, that there are no elements in the piece p, that is, $B_a[p] = 0$, or that there are elements but no one is after i, that is, the succinct SB-tree associated with piece p finds no successor of i. In this case, we find the next piece that follows p where a has occurrences, $p' = \text{select}_1(B_a, \text{rank}_1(B_a, p) + 1)$, and if $p' < p_r$ we query the succinct SB-tree associated with $B_a[p'] = 1$ for its first element (or the successor of the minimum of its universe). If, instead, $p' \geq p_r$, we query instead the succinct SB-tree associated with $J_b^t[i_r] = 1$, as its positions are to the left of those associated with $B_a[p'] = 1$. Since the $m = \lg^4 n_c$ elements of a chunk can fall in the same piece, and Lemma 1 lets us retain the time complexity of the full universe $[1..n]$, this is $\mathcal{O}(\lg m / \lg\lg n) = \mathcal{O}(1)$. The total number of elements stored in succinct SB-trees is still at most n_c, because no duplicate elements are stored, but now, again due to Lemma 1, each requires only $\mathcal{O}(\lg\lg \sigma_c)$ bits, for a total space of $\mathcal{O}(n_c \lg\lg \sigma_c) = o(n_c \lg \sigma_c)$ bits. There may be up to $\sigma_c \cdot (n_c/\sigma_c^2)$ pointers to succinct SB-trees from bitvectors B_a, each requiring $\mathcal{O}(\lg n_c)$ bits, for a total of $\mathcal{O}\big(\sigma_c(n_c \lg n_c)/\sigma_c^2\big) = \mathcal{O}(n_c)$ bits, since $\sigma_c > w$. Finally, the extra tables of $\mathcal{O}(n^\gamma)$ bits of Lemma 1 add up to $\mathcal{O}(n^\gamma \lg\lg\sigma) = o(n)$, because they depend only on $\lceil\sqrt{\lg n}\rceil$ and on $\lceil\lg\lg(\sigma_c^2)\rceil$, and thus we need at most $\lg\lg\sigma$ different tables.

Theorem 6 *Let $S[1..n]$ be a string whose distribution of symbols has entropy H, over alphabet $[1..\sigma]$. For any $f(n,\sigma) = \omega(1)$, we can store S in $nH + o(n)(H+1)$ bits such that later, given the endpoints of a range and τ, we can return the τ-majorities for that range in time $\mathcal{O}((1/\tau)\, f(n,\sigma))$. The structure is built within $\mathcal{O}(n\lg n)$ randomized time and $\mathcal{O}(n\lg\sigma)$ bits.*

Note that accessing a position in S still requires constant time with this representation. Further, we can obtain a version using nearly compressed space, $(1+\epsilon)nH+o(n)$ bits for any constant $\epsilon > 0$, with optimal query time, by setting $g(n,\sigma)$ to a constant value. First, use for S_c the structure of Barbay et al. [1] that needs $(1+\epsilon/3)nH+o(n)$ bits and solves access and select in constant time. Second, let κ be the constant associated with the $\mathcal{O}\Big(\frac{n_c\lg\sigma_c\lg g(n,\sigma)}{g(n,\sigma)}\Big)$ bits used by bitvectors G_b^t and the sparsified Muthukrishnan's structures. Then, choosing $g(n,\sigma) = \frac{6\kappa}{\epsilon}\lg\frac{6\kappa}{\epsilon}$ ensures that the space becomes $(\epsilon/3)n_c\lg\sigma_c$ bits, which add up to $(\epsilon/3)nH$. All the other terms of the form $o(nH)$ are smaller than another $(\epsilon/3)nH + o(n)$. Therefore the total space adds up to $(1+\epsilon)nH + o(n)$ bits. The time to sequentially solve a range of length $\mathcal{O}((1/\tau_c)\,g(n,\sigma))$ is $\mathcal{O}\big((1/\tau_c)\,g(n,\sigma)^2\big) = \mathcal{O}(1/\tau_c)$.

Theorem 7 *Let $S[1..n]$ be a string whose distribution of symbols has entropy H, over alphabet $[1..\sigma]$. For any constant $\epsilon > 0$, we can store S in $(1+\epsilon)nH + o(n)$ bits such that later, given the endpoints of a range and τ, we can return the τ-majorities for that range in time $\mathcal{O}(1/\tau)$. The structure is built within $\mathcal{O}(n\lg n)$ randomized time and $\mathcal{O}(n\lg\sigma)$ bits.*

6.2 Finding range modes

While finding range modes is a much harder problem in general, we note that we can use our data structure from Theorem 6 to find a range mode quickly when it is actually reasonably frequent. Suppose we want to find the mode of $S[i..j]$, where it occurs occ times (we do not know occ). We perform multiple range τ-majority queries on $S[i..j]$, starting with $\tau = 1/2$ and repeatedly reducing it by a factor of 2 until we find at least one τ-majority. This takes time

$$\mathcal{O}\Big(\Big(2+4+\ldots+2^{\lceil\lg\frac{j-i+1}{\mathrm{occ}}\rceil}\Big)\, f(n,\sigma)\Big) = \mathcal{O}\bigg(\frac{(j-i+1)f(n,\sigma)}{\mathrm{occ}}\bigg)$$

and returns a list of $\mathcal{O}\big(\frac{j-i+1}{\mathrm{occ}}\big)$ elements that includes all those that occur at least occ times in $S[i..j]$. We use rank queries to determine which of these elements is the mode. For the fastest possible time on those rank queries, we use for S the representation of Belazzougui and Navarro [6, Thm. 8], and also set

$f(n,\sigma) = \lg\lg_w \sigma$, the same time of rank. The cost is then $\mathcal{O}\left(\frac{(j-i+1)\lg\lg_w \sigma}{\mathrm{occ}}\right)$. The theorem holds for $\lg \sigma = \mathcal{O}(\lg w)$ too, as in this case we can use Theorem 5 with constant-time rank queries.

Theorem 8 *Let $S[1..n]$ be a string whose distribution of symbols has entropy H. We can store S in $nH + o(n)(H+1)$ bits such that later, given endpoints i and j, we can return the mode of $S[i..j]$ in $\mathcal{O}\left(\frac{(j-i+1)\lg\lg_w \sigma}{\mathrm{occ}}\right)$ time, where* occ *is the number of times the mode occurs in $S[i..j]$. The structure is built within $\mathcal{O}(n\lg n)$ randomized time and $\mathcal{O}(n\lg\sigma)$ bits.*

We note that, if we store an instance of Bose et al.'s [7] linear-space structure for 4-approximating the range mode with constant query time, or Greve et al.'s [23] linear-space structure for $(1+\epsilon)$-approximating the range mode with constant query time for positive constant ϵ, then we need not perform $\lceil \lg \frac{j-i+1}{\mathrm{occ}} \rceil$ rounds of τ-majority queries. We can instead find an approximate range mode using their structure; find its frequency with two rank queries; perform only $\mathcal{O}(1)$ rounds of τ-majority queries to obtain a list of elements that includes the true range mode; and use two rank queries per element to determine all their frequencies, which tells us the true range mode. This still takes $\mathcal{O}\left(\frac{(j-i+1)\lg\lg_w \sigma}{\mathrm{occ}}\right)$ time, however, and uses linear instead of compressed space. So far, we have not found a way to compare Theorem 8 to known results for approximating range modes, nor to combine them fruitfully.

7 Parameterized Range Minority

Chan et al. [10] gave a linear-space solution with $\mathcal{O}(1/\tau)$ query time for parameterized range minority, even for the case of variable τ (i.e., chosen at query time). They first build a list of $\lceil 1/\tau \rceil$ distinct elements that occur in the given range (or as many as there are, if fewer) and then check those elements' frequencies to see which are τ-minorities. There cannot be as many as $\lceil 1/\tau \rceil$ τ-majorities so, if there exists a τ-minority for that range, then at least one must be in the list. In the early version of this article [4] we used a simple approach to implement this idea using compressed space; here we obtain more refined results based on our new structures for τ-majorities.

The main idea is still that, if we test any $\lceil 1/\tau \rceil$ distinct elements, we must find a τ-minority because not all of those can occur more than $\tau(j-i+1)$ times in $S[i..j]$. Therefore, we can use mechanisms similar to those we designed to find $\mathcal{O}(1/\tau)$ distinct candidates to τ-majorities.

Let us first consider the bitvectors G_b^t defined in Section 3. We now define bitvectors I_b^t, where we flag the positions of the first 2^t and the last 2^t distinct values in each block (we may flag fewer positions if the block contains less than 2^t distinct values). Since we set $\mathcal{O}(2^t)$ bits per block, the bitvectors I_b^t use asymptotically the same space of the bitvectors G_b^t.

Note that I_b^t can be built by scanning S block by block, and computing an array of the first 2^t elements seen in each, by using a bitvector of size σ

where the already seen elements are marked. The last 2^t elements are obtained similarly, by scanning the block in reverse. These arrays require up to $\mathcal{O}(n \lg \sigma)$ bits. In $\mathcal{O}(n)$ time we can scan S for each value of b and the largest value $t^* = b - 1$, flagging in I_b^t the first 2^t of the 2^{t^*} elements collected, for each t. Thus we need $\mathcal{O}(n \lg \sigma)$ space to maintain all the bitvectors I_b^t, for a fixed b, in plain form, and later they can all be compressed just like the bitvectors G_b^t. Doing this for all the values of b, these bitvectors are also built in $\mathcal{O}(n \lg n)$ deterministic time and $\mathcal{O}(n \lg \sigma)$ bits of space.

Given a τ-minority query, we compute b and t as in Section 3 and use rank and select to find all the 1s in the range $I_b^t[i..j]$. Those positions contain a τ-minority in $S[i..j]$ if there is one, as shown next.

Lemma 5 *The positions flagged in $I_b^t[i..j]$ contain a τ-minority in $S[i..j]$, if there is one.*

Proof If $I_b^t[i..j]$ contains a part of a block with $2^t = \lceil 1/\tau \rceil$ distinct elements flagged, then one is for sure a τ-minority in $S[i..j]$. Otherwise, all the distinct block elements that fall inside $[i..j]$ are flagged. This is obvious if $[i..j]$ contains the whole block, and it also holds if $[i..j]$ intersects a prefix or a suffix of the block, since the block marks its 2^t first and last occurrences of distinct elements. Then, if $I_b^t[i..j]$ does not flag 2^t elements in any of the blocks it overlaps, it must flag all the distinct elements in $S[i..j]$, and thus it flags a τ-minority. $\square$

Just as for τ-majorities, we use I_b^t only if $1/\tau \leq \sigma$ and $t < b$, since otherwise we can test, one by one, all the alphabet elements or all the elements in $S[i..j]$, respectively. The test proceeds using rank on S if σ is small, or using Lemma 2 if σ is large. We describe precisely how we proceed.

7.1 Small alphabets

If $\lg \sigma = \mathcal{O}(\lg w)$, we use a multiary wavelet tree as in Section 3. This time, we do not run τ-majority queries on each wavelet tree node v to determine which of its children to explore, but rather we explore every child having some symbol in the range $S_v[i_v..j_v]$. To efficiently find the distinct symbols that appear in the range, we store a sparsified Muthukrishnan's structure similar to the one described in Section 2.5; here we will have no slowdown thanks to the small alphabet of S_v.

Let C_v be the array corresponding to string S_v. We cut S_v into blocks of w^β elements, and record in an array $C'_v[1..n_v/w^\beta]$ the minimum value in the corresponding block of C_v. Then, the leftmost occurrence $S[k] = p$ of each distinct symbol p in $S_v[i_v..j_v]$ has a value $C_v[k] < i_v$, and thus also for its corresponding block in C'_v, k', it holds that $C'_v[k'] < i_v$. We initialize a word $E \leftarrow 0$ containing flags for the $\sigma' = w^\beta$ symbols, separated as in the final state of the word A of Section 5.1. Each time the algorithm of Muthukrishnan on C'_v gives us a new block, we apply the algorithm of Section 5.1 to count in a word A the occurrences of the distinct symbols in that block, we isolate the counters reaching the threshold $y = 1$, and compare E with E OR A. If they are equal, then we stop the recursive algorithm, since all the symbols in the range had already appeared before (see the final comments on the proof of Lemma 3). Otherwise, we process the subrange to the left of the block, update $E \leftarrow E$ OR A, and process the subrange to the right. When we finish, E contains all the symbols that appear in $S_v[i_v..j_v]$. In the recursive process, we also stop when we have considered $\lceil 1/\tau_v \rceil$ blocks, since each block includes at least one new element and it is sufficient to explore $\lceil 1/\tau_v \rceil$ children to find a τ_v-minority (because each child contains at least one candidate). Finally, we extract the bits of E one by one as done in Section 5.1 with the use of D. For each extracted bit, we enter the corresponding child in the wavelet tree. The total time is thus $\mathcal{O}(1/\tau_v)$ and the bitvectors C_v add up to $\mathcal{O}\big(n/w^\beta\big) = o(n)$ bits in total.

The τ_u values to use in the children u of v are computed as in Section 5.1, so the analysis leading to $\mathcal{O}(1/\tau)$ total time applies. When we arrive at the leaves u of the wavelet tree, we obtain the distinct elements and compute using rank the number of times they occur in $S_u[i_u..j_u]$, so we can immediately report the first τ-minority we find.

We still have to describe how we handle the intervals that are smaller than the lower limit for b, $\lfloor 2^t \cdot w^\beta/4 \rfloor$. We do the counting exactly as in Section 5.1. We must then obtain the counters that are between 1 and $y-1$. On one hand, we use the bound $y' = 1$ and repeat their computation to obtain in $A_{\geq 1} \leftarrow A$ the counters that are at least 1. On the other, we compute $A \leftarrow A + (2^{2\ell} - y) \cdot (0^{k\ell+\ell-1}10^{(k-1)\ell})^{\sigma'}$ as before, and isolate the non-overflowed bits with $A_{<y} \leftarrow$ (NOT A) AND $(0^{(k-1)\ell-1}10^{(k+1)\ell})^{\sigma'}$. Then we extract the first of the bits marked in $A \leftarrow A_{\geq 1}$ AND $A_{<y}$ and report it.

To obtain compressed space, we use the alphabet partitioning technique of Section 5.3. Once again, we must identify at most $\lceil 1/\tau \rceil$ nonempty ranges $[i_c..j_c]$ from $K[i..j]$. Those are obtained in the same way as on the multiary wavelet tree, since K is represented in that way (albeit the strings S_v are compressed). We then look for τ_c-minorities in the strings $S_c[i_c..j_c]$ one by one, until we find one or we exhaust them. The total time is $\mathcal{O}(1/\tau)$.

Theorem 9 *Let $S[1..n]$ be a string whose distribution of symbols has entropy H, over alphabet $[1..\sigma]$, with $\lg \sigma = \mathcal{O}(\lg w)$. We can store S in $nH + o(n)$ bits such that later, given the endpoints of a range and τ, we can return a τ-minority for that range (if one exists) in time $\mathcal{O}(1/\tau)$. The structure is built within $\mathcal{O}(n \lg n)$ deterministic time and $\mathcal{O}(n \lg \sigma)$ bits.*

Note that we can use a single representation using $nH + o(n)$ bits solving both the τ-majority queries of Theorem 5 and the τ-minority queries of Theorem 9.

7.2 Large alphabets

For large alphabets we must use Lemma 2 to check for τ-minorities, and thus we must find the leftmost positions in $S[i..j]$ of the τ-minority candidates. We use the same bitvectors J_b^t of Section 4, so that they store sampled positions corresponding to the 1s in I_b^t, and proceed exactly as in that section, both if $\tau < 1/\sigma$ or if $\tau \geq 1/\sigma$.

To obtain compression, we also use alphabet partitioning. We use the multiary wavelet tree of K as described in Section 7.1, and then complete the queries with τ_c-minority queries on the strings S_c over small or large alphabets, as required, until we find one result or exhaust all the strings. The only novelty is that we must now find τ_c-minorities sequentially for the ranges that are shorter than $\lfloor \lg(2^t \cdot g(n,\sigma)) \rfloor = \mathcal{O}((1/\tau_c)\, g(n,\sigma))$. For this, we adapt the $\mathcal{O}\big((1/\tau)\, g(n,\sigma)^2\big)$-time sequential algorithm described in Section 6.1. The only difference is that we stop as soon as we test a candidate a that turns out not to be a τ_c-majority, then reporting the τ-minority a.

Depending on whether we use Theorem 6 or 7 to represent S and how we choose $f(n,\sigma)$, we obtain space or time optimality.

Theorem 10 *Let $S[1..n]$ be a string whose distribution of symbols has entropy H. For any function $f(n) = \omega(1)$, we can store S in $nH + o(n)(H+1)$ bits such that later, given the endpoints of a range and τ, we can return a τ-minority for that range (if one exists) in time $\mathcal{O}((1/\tau)\, f(n))$. The structure is built within $\mathcal{O}(n \lg n)$ randomized time and $\mathcal{O}(n \lg \sigma)$ bits.*

Theorem 11 *Let $S[1..n]$ be a string whose distribution of symbols has entropy H, over alphabet $[1..\sigma]$. For any constant $\epsilon > 0$, we can store S in $(1+\epsilon)nH + o(n)$ bits such that later, given the endpoints of a range and τ, we can return a τ-minority for that range (if one exists) in time $\mathcal{O}(1/\tau)$. The structure is built within $\mathcal{O}(n \lg n)$ randomized time and $\mathcal{O}(n \lg \sigma)$ bits.*

In both cases, we can share the same structures to find majorities and minorities.

8 Conclusions

We have given the first linear-space data structure for parameterized range majority with query time $\mathcal{O}(1/\tau)$, even in the more difficult case of τ specified at query time. This is worst-case optimal in terms of n and τ, since the output size may be up to $1/\tau$. Therefore, we have closed this problem in terms of asymptotic space and query time.

We have also aimed at using not only linear space, but optimal compressed space with respect to the entropy H of the distribution of the symbols in the sequence. We obtained optimal or near-optimal query time within optimal or near-optimal space, for both parameterized range majority and minority. In terms of compressed to this entropy space, the problem is also essentially closed.

Subsequent work Our original publication [4] triggered further research on other variants of the problem. For example, Navarro and Thankachan [34] considered the encoding version of the problem, where S cannot be stored. They proved that such an encoding requires $\Omega(n \lg(1/\tau))$ bits, and therefore the encoding problem is relevant only for $1/\tau < \sigma$ (fixed at indexing time); otherwise the allowed space is sufficient to store our current indexes. By using techniques similar to our blocks, among others, they obtained query times in $\mathcal{O}((1/\tau)\lg n)$ within this space. This was improved by Gawrychowski and Nicholson [22], who reached the optimal query time, $\mathcal{O}(1/\tau)$, within $\mathcal{O}(n \lg(1/\tau))$ bits of space.

Given that their result [22] is an encoding, they can couple their structure with any compressed representation of S that yields constant-time access, like the one of Ferragina and Venturini [18], to obtain a representation using $nH_k + o(n \lg \sigma) + \mathcal{O}(n \lg(1/\tau))$ bits, for any $k = o(\lg_\sigma n)$, and finding τ-majorities in optimal time. Here $H_k \leq H_0 = H$ is the kth order empirical entropy of S [28]. Note that this works only for fixed τ.

Gagie et al. [21] noticed that another tradeoff could be directly obtained from our results just by modifying our base compressed sequence representation: $nH_k + o(n \lg \sigma)$ bits (plus $2n$ bits for minorities) and $\mathcal{O}((1/\tau)\lg\lg_w \sigma)$ query time, for variable τ. It is unknown if we can reduce the space usage to just $nH_k + o(n \lg \sigma)$ while retaining $\mathcal{O}(1/\tau)$ time complexity.

Another interesting result of Gawrychowski and Nicholson [22] (see Appendix A in their extended paper) is that it is unlikely that we can improve the $\mathcal{O}(1/\tau)$ worst-case time to $\mathcal{O}(\mathrm{occ}+1)$ when returning $\mathrm{occ} = o(1/\tau)$ results in τ-majority queries.

Another variant is the dynamic version of the problem, where S can undergo insertions and deletions. Elmasry et al. [16] obtained $\mathcal{O}((1/\tau)\lg n/\lg\lg n)$ query time and linear space, with updates supported in $\mathcal{O}((1/\tau)\lg n)$ amortized time. Gagie et al. [21] retained those complexities while reducing the space to $nH_k + o(n \lg \sigma)$ bits, for any $k = o(\lg_\sigma n)$.

Acknowledgements

Many thanks to Patrick Nicholson for helpful comments.

References

1. J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, 69(1):232–268, 2014.
2. D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Theory and practice of monotone minimal perfect hashing. *ACM Journal of Experimental Algorithmics*, 16(3):article 2, 2011.
3. D. Belazzougui, F. Cunial, J. Kärkkäinen, and V. Mäkinen. Linear-time string indexing and analysis in small space. *ACM Transactions on Algorithms*, 16(2):17:1–17:54, 2020.
4. D. Belazzougui, T. Gagie, and G. Navarro. Better space bounds for parameterized range majority and minority. In *Proc. 12th Annual Workshop on Algorithms and Data Structures (WADS)*, LNCS 8037, pages 121–132, 2013.

5. D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10(4):article 23, 2014.
6. D. Belazzougui and G. Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms*, 11(4):article 31, 2015.
7. P. Bose, E. Kranakis, P. Morin, and Y. Tang. Approximate range mode and range median queries. In *Proc. 22nd Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 377–388, 2005.
8. R. S. Boyer and J. S. Moore. MJRTY–a fast majority vote algorithm. In *Automated Reasoning*, pages 105–117. Springer, 1991.
9. 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 of Computing Systems*, 55(4):719–741, 2014.
10. T. M. Chan, S. Durocher, M. Skala, and B. T Wilkinson. Linear-space data structures for range minority query in arrays. *Algorithmica*, 72(4):901–913, 2015.
11. G. Cormode. Misra-Gries summaries. In *Encyclopedia of Algorithms*, pages 1334–1337. Springer, 2016.
12. G. Cormode and S. Muthukrishnan. Data stream methods. `http://www.cs.rutgers.edu/~muthu/198-3.pdf`, 2003. Lecture 3 of Rutger's *198:671 Seminar on Processing Massive Data Sets.*
13. E. D. Demaine, A. López-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. In *Proc. 10th European Symposium on Algorithms (ESA)*, pages 348–360, 2002.
14. S. Durocher, M. He, J. I. Munro, P. K. Nicholson, and Matthew Skala. Range majority in constant time and linear space. *Information and Computation*, 222:169–179, 2013.
15. S. Durocher, R. Shah, M. Skala, and S. V. Thankachan. Linear-space data structures for range frequency queries on arrays and trees. *Algorithmica*, 74(1):344–366, 2016.
16. A. Elmasry, M. He, J. I. Munro, and P. K. Nicholson. Dynamic range majority data structures. *Theoretical Computer Science*, 647:59–73, 2016.
17. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2), 2007.
18. P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. *Theoretical Computer Science*, 371(1):115–121, 2007.
19. J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.
20. T. Gagie, M. He, J. I. Munro, and P. K. Nicholson. Finding frequent elements in compressed 2D arrays and strings. In *Proc. 18th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 295–300, 2011.
21. T. Gagie, M. He, and G. Navarro. Compressed dynamic range majority and minority data structures. *Algorithmica*, 82(7):2063–2086, 2020.
22. P. Gawrychowski and P. K. Nicholson. Optimal query time for encoding range majority. In *Proc. 15th International Symposium on Algorithms and Data Structures (WADS)*, pages 409–420, 2017. Extended version in *CoRR* abs/1704.06149.
23. M. Greve, A. G. Jørgensen, K. D. Larsen, and J. Truelsen. Cell probe lower bounds and approximations for range mode. In *Proc. 37th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 605–616, 2010.
24. R. Grossi, A. Orlandi, R. Raman, and S. Srinivasa Rao. More haste, less waste: Lowering the redundancy in fully indexable dictionaries. In *Proc. 26th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 517–528, 2009.
25. W.-K. Hon, R. Shah, and J. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. 50th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 713–722, 2009.
26. R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems*, 28(1):51–55, 2003.
27. M. Karpinski and Y. Nekrich. Searching for frequent colors in rectangles. In *Proc. 20th Canadian Conference on Computational Geometry (CCCG)*, pages 11–14, 2008.
28. R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 2000.

29. D. Krizanc, P. Morin, and M. H. M. Smid. Range mode and range median queries on lists and trees. *Nordic Journal of Computing*, 12(1):1–17, 2005.
30. Y. K. Lai, C. K. Poon, and B. Shi. Approximate colored range and point enclosure queries. *Journal of Discrete Algorithms*, 6(3):420–432, 2008.
31. J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2(2):143–152, 1982.
32. J. I. Munro, G. Navarro, and Y. Nekrich. Fast compressed self-indexes with deterministic linear-time construction. *Algorithmica*, 82(2):316–337, 2020.
33. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. 13th Symposium on Discrete Algorithms (SODA)*, pages 657–666, 2002.
34. G. Navarro and S. V. Thankachan. Optimal encodings for range majority queries. *Algorithmica*, 74(3):1082–1098, 2016.
35. H. Petersen. Improved bounds for range mode and range median queries. In *Proc. 34th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, pages 418–423, 2008.
36. H. Petersen and S. Grabowski. Range mode and range median queries in constant time and sub-quadratic space. *Information Processing Letters*, 109(4):225–228, 2009.
37. M. Pătraşcu. Succincter. In *Proc. 49th Symposium on Foundations of Computer Science (FOCS)*, pages 305–313, 2008.
38. K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.
39. Z. Wei and K. Yi. Beyond simple aggregates: indexing for summary queries. In *Proc. 30th Symposium on Principles of Database Systems (PODS)*, pages 117–128, 2011.

A Finding τ'-majorities on tiny alphabets

We show how to find τ'-majorities in time $\mathcal{O}(1/\tau')$ on ranges of length $\mathcal{O}\big((1/\tau')w^\beta\big)$, over alphabet $[0..\sigma'-1]$, with $\sigma' = w^\beta$, in the case $1/\tau' < \sigma'$. We will compute an array of σ' counters with the frequency of the symbols in the range, and then report those exceeding the threshold. The maximum size of the range is $(4/\tau')w^\beta/4 \le \sigma' w^\beta = w^{2\beta}$, and thus $2\beta \lg w$ bits suffice to represent each counter. The σ' counters then require $2\beta w^\beta \lg w$ bits and can be maintained in a computer word (although we will store them somewhat spaced for technical reasons). We can read the elements in S_v by chunks of w^β symbols, and compute in constant time the corresponding counters for those symbols. Then we sum the current counters and the counters for the chunk, all in constant time because they are fields in a single computer word. The range is then processed in time $\mathcal{O}(1/\tau')$.

To compute the counters corresponding to w^β symbols, we extend the popcounting algorithm of Belazzougui and Navarro [6, Sec. 4.1]; assume we extract the w^β symbols from S_v and have them packed in the lowest $k\ell$ bits of a computer word X, where $k = w^\beta$ is the number of symbols and $\ell = \lg \sigma'$ is the number of bits used per symbol. We first create σ' copies of the sequence at distance $2k\ell$ of each other: $X \leftarrow X \cdot (0^{2k\ell-1}1)^{\sigma'}$. In each copy we will count the occurrences of a different symbol. To have the $(i+1)$th copy count the occurrences of symbol i, for $0 \le i < \sigma'$, we perform

$$X \;\leftarrow\; X \;\textsc{xor}\; 0^{k\ell}((\sigma'-1)_\ell)^k \ldots 0^{k\ell}(2_\ell)^k\; 0^{k\ell}(1_\ell)^k\; 0^{k\ell}(0_\ell)^k,$$

where i_ℓ is number i written in ℓ bits. Thus in the $(i+1)$th copy the symbols equal to i become zero and the others nonzero. We then set a 1 at the highest bit of the symbols equal to i in the $(i+1)$th copy, with

$$X \;\leftarrow\; (Y - (X \;\textsc{and not}\; Y)) \;\textsc{and}\; Y \;\textsc{and not}\; X,$$

where $Y = (0^{k\ell}(10^{\ell-1})^k)^{\sigma'}$.[3] Now we add all the 1s in each copy with $X \leftarrow X \cdot 0^{k\ell(2\sigma'-1)}(0^{\ell-1}1)^k$. This spreads several sums across the $2k\ell$ bits of each copy, and in particular the kth sum adds

[3] This could have been simply $X \leftarrow (Y-X)$ AND Y if there was an unused highest bit set to zero in the $(\lg \sigma')$-length fields of X. Instead, we have to use this more complex formula that first zeroes the highest bit of the fields and later considers them separately.

up all the 1s of the copy. Each sum requires $\lg k$ bits, which is precisely the ℓ bits we have allocated per field. Finally, we isolate the desired counters using $X \leftarrow X$ AND $(0^{k\ell}1^{\ell}0^{(k-1)\ell})^{\sigma'}$. The σ' counters are not contiguous in the computer word, but we still can afford to store them spaced: we use $2k\ell\sigma' = 2\beta w^{2\beta} \lg w$ bits, which since $\beta \le 1/4$, is always less than w.

Since the range is of length at most $w^{2\beta}$, the cumulative counters need $\lg(w^{2\beta}) = 2\ell$ bits. We will store them in a computer word A separated by $2k\ell$ bits so that we can directly add the resulting word X after processing a chunk of w^{β} symbols of the range in S_v: $A \leftarrow A+X$. If the last chunk is of length $l < w^{\beta}$, we complete it with zeros and then subtract those spurious $w^{\beta} - l$ occurrences from the first counter, $A \leftarrow A - (w^{\beta} - l) \cdot 2^{(k-1)\ell}$.

The last challenge is to output the counters that are at least $y = \lfloor \tau'(j-i+1) \rfloor + 1$ after processing the range. We use

$$A \leftarrow A + (2^{2\ell} - y) \cdot (0^{k\ell+\ell-1}10^{(k-1)\ell})^{\sigma'}$$

so that the counters reaching y will overflow to the next bit. We isolate those overflow bits with $A \leftarrow A$ AND $(0^{(k-1)\ell-1}10^{(k+1)\ell})^{\sigma'}$, so that we have to report the symbol i if and only if A AND $0^{(k(2\sigma'-2i-1)-1)\ell-1}10^{(k(2i+1)+1)\ell} \neq 0$. We then repeatedly isolate the lowest bit of A with

$$D \leftarrow (A \text{ XOR } (A-1)) \text{ AND } (0^{(k-1)\ell-1}10^{(k+1)\ell})^{\sigma'},$$

and then remove it with $A \leftarrow A$ AND $(A-1)$, until $A = 0$. Once we have a position isolated in D, we find the position in constant time by using a monotone minimum perfect hash function over the set $\{2^{(k(2i+1)+1)\ell},\ 0 \le i < \sigma'\}$, which uses $\mathcal{O}(\sigma' \lg w) = o(w)$ bits [2]. Only one such data structure is needed for all the sequences, and it takes less space than a single systemwide pointer.