Theoretical Computer Science 483 (2013) 36–50

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

# Colored range queries and document retrieval ☆

Travis Gagie [a], Juha Kärkkäinen [b], Gonzalo Navarro [c,*], Simon J. Puglisi [d]

[a] *Department of Computer Science and Engineering, Aalto University, Finland*
[b] *Department of Computer Science, University of Helsinki, Finland*
[c] *Department of Computer Science, University of Chile, Chile*
[d] *Department of Informatics, King's College London, United Kingdom*

ARTICLE INFO

*Keywords:*
Data structures
1D range queries
Document retrieval
Wavelet trees
Information retrieval

ABSTRACT

Colored range queries are a well-studied topic in computational geometry and database research that, in the past decade, have found exciting applications in information retrieval. In this paper, we give improved time and space bounds for three important one-dimensional colored range queries — colored range listing, colored range top-$k$ queries and colored range counting — and, as a consequence, new bounds for various document retrieval problems on general collections of sequences. Colored range listing is the problem of preprocessing a sequence $S[1, n]$ of colors so that, later, given an interval $[i, i+\ell-1]$, we list the different colors in $S[i, i+\ell-1]$. Colored range top-$k$ queries ask instead for $k$ most frequent colors in the interval. Colored range counting asks for the number of different colors in the interval.

We first describe a framework including almost all recent results on colored range listing and document listing, which suggests new combinations of data structures for these problems. For example, we give the first compressed data structure (using $nH_k(S) + o(n \log \sigma)$ bits, for any $k = o(\log_\sigma n)$, where $H_k(S)$ is the $k$-th order empirical entropy of $S$ and $\sigma$ the number of different colors in $S$) that answers colored range listing queries in constant time per returned result. We also give an efficient data structure for document listing whose size is bounded in terms of the $k$-th order entropy of the library of documents. We then show how (approximate) colored top-$k$ queries can be reduced to (approximate) range-mode queries on subsequences, yielding the first efficient data structure for this problem. Finally, we show how modified wavelet trees can support colored range counting using $nH_0(S) + \mathcal{O}(n) + o(nH_0(S))$ bits, and answer queries in $\mathcal{O}(\log \ell)$ time. As far as we know, this is the first data structure in which the query time depends only on $\ell$ and not on $n$. We also show how our data structure can be made dynamic.

© 2012 Elsevier B.V. All rights reserved.

## 1. Introduction

A *range query* on a sequence $S[1, n]$ of elements in $[1, \sigma]$ takes as arguments two indices $i$ and $j$ and returns information about $S[i, j]$. This information could be, for example, the minimum or maximum value in $S[i, j]$ [16], the element with a

☆ Early parts of this work appeared in SPIRE 2010 [20] and CPM 2011 [19]. Partially funded by Fondecyt grant 1-110066, Chile; by the Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile; by the Academy of Finland grant 118653 (ALGODAN); and by the Australian Research Council. Simon J. Puglisi is supported by a Newton Fellowship. Part of this paper was written while Travis Gagie was at the Department of Computer Science, University of Chile and Simon J. Puglisi was at the School of Computer Science and Information Technology, Royal Melbourne Institute of Technology.

* Corresponding author.
*E-mail addresses:* travis.gagie@aalto.fi (T. Gagie), juha.karkkainen@cs.helsinki.fi (J. Kärkkäinen), gnavarro@dcc.uchile.cl (G. Navarro), simon.j.puglisi@gmail.com (S.J. Puglisi).

0304-3975/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2012.08.004

specified rank in sorted order [22] (e.g., the median [10]), the mode [26], a complete list of the distinct elements [47], the frequencies of the elements [55], a list of the $k$ most frequent elements for a given $k$ [32], or the number of distinct elements [9]. In this paper, motivated by problems in document retrieval, we consider the latter three kinds of problems, which are often referred to as "colored" range queries: colored range listing (with or without color frequencies), colored range top-$k$ queries, and colored range counting. These have been associated, respectively, with very relevant document retrieval queries on general texts [47,55,57,32,22,16,13,21,6]: listing the documents where a pattern appears (possibly computing term frequencies), finding the most relevant documents to a query (under a $tf \times idf$ scheme, for example), and computing document frequencies. Such techniques have been shown to be competitive [13], even beating classical inverted indexes on natural-language texts.

In Section 2, we describe a framework that includes almost all recent results on colored range listing and the related problem of document listing. This framework suggests new combinations of data structures that yield interesting new bounds, including the first constant-time compressed data structures for colored range listing and an efficient data structure for document listing whose space occupancy is bounded in terms of the higher-order entropies of the library of documents. In Section 3, we describe what seems to be the first data structure to support efficient, general approximate colored range top-$k$ queries. By "approximate" we mean that we are given an $\epsilon > 0$ with $S$ and we guarantee that no element we do not list occurs more than $1 + \epsilon$ times more often in the range than any element we list. Finally, in Section 4, we describe a new solution to the colored range counting problem, reducing the space bound from $\mathcal{O}(n \log n)$ bits to $nH_0(S) + \mathcal{O}(n) + o(nH_0(S))$ bits, where $H_0(S) \leq \log \sigma$ is the zero-order empirical entropy of $S$ and $\sigma \leq n$ is the number of colors, and also improving the time bound to $\mathcal{O}(\log \ell)$, where $\ell$ is the length of the query range.[1] The improvements for general colored range queries we present in Sections 3 and 4 are not competitive with the state of the art when mapped to the more specific problem of document retrieval. However, as we discuss in Section 5, data structures for general colored range queries can be applied to information retrieval scenarios that specialized document-retrieval data structures cannot.

## 2. Color and document listing

### 2.1. Related work on color range listing

The problem of colored range listing (CRL) is to preprocess a given sequence $S[1, n]$ over $[1, \sigma]$ such that later, given a range $S[i..j]$, we can quickly list all the distinct elements ("colors") in that range. Many recent data structures for CRL are based on a key idea by Muthukrishnan [47] (see [34] for older work). He defined $C[1, n]$ to be the array in which $C[j]$ is the largest value $i < j$ such that $S[i] = S[j]$, or 0 if there is no such $i$, so that $S[q]$ is the first occurrence of a color in $S[i..j]$ if and only if $i \leq q \leq j$ and $C[q] < i$. He showed how, if we store $C$ in an $\mathcal{O}(n \log n)$-bit data structure due to Gabow et al. [18] that supports $\mathcal{O}(1)$-time range-minimum queries (RMQs), we can quickly find all the values in $C[i..j]$ less than $i$ and, thus, list all the colors in $S[i..j]$. To do this, we find the minimum value $C[q]$ in $C[i..j]$; if it is less than $i$, then we output $S[q]$ and recurse on $S[i..q-1]$ and $S[q+1..j]$. Muthukrishnan's CRL data structure uses $\mathcal{O}(n \log n)$ bits and $\mathcal{O}(1)$ time per color reported.

Välimäki and Mäkinen [57] gave an alternative slower-but-smaller version of Muthukrishnan's CRL data structure, in which they used a $2n + o(n)$ bit, $\mathcal{O}(1)$ time RMQ succinct index due to Fischer and Heun [17] that requires access to $C$. Välimäki and Mäkinen showed how access to $C$ can be implemented by rank and select queries on $S$; specifically, for $1 \leq q \leq n$, $C[q] = \mathsf{select}_{S[q]}(S, \mathsf{rank}_{S[q]}(S, q) - 1)$, where $\mathsf{select}_a(S, r)$ is the position of the $r$th occurrence of $a$ in $S$. Välimäki and Mäkinen stored $S$ in a multiary wavelet tree [14], which takes $nH_0(S) + o(n) \log \sigma$ bits and $\mathcal{O}(1 + \log \sigma / \log \log n)$ time; when $\sigma$ is polylogarithmic in $n$, it takes $nH_0(S) + o(n)$ bits and $\mathcal{O}(1)$ time. The zero-order empirical entropy $H_0(S) = \sum_a \frac{\mathsf{occ}(a,S)}{n} \log \frac{n}{\mathsf{occ}(a,S)}$, where $\mathsf{occ}(a, S)$ is the number of times element $a$ occurs in $S$, is the Shannon entropy of the distribution of elements in $S$.

Altogether, their CRL data structure takes $nH_0(S) + 2n + o(n) \log \sigma$ bits and $\mathcal{O}(1 + \log \sigma / \log \log n)$ time per reported color. They also showed how to compute color frequencies using two rank queries on $S$, $\mathsf{rank}_c(S, j) - \mathsf{rank}_c(S, i - 1)$. Since multiary wavelet trees support rank queries in the same time as accesses, it follows that reporting the color frequencies in the range does not affect their time and space bounds.

Gagie et al. [22] showed that a binary wavelet tree [27] can be used to compute range quantile queries on $S$ in $\mathcal{O}(\log \sigma)$ time, and that these queries can be used to enumerate the distinct elements in $S[i..j]$, eliminating the need for RMQs. A binary wavelet tree for $S$ takes $nH_0(S) + o(n) \log \sigma$ bits and supports access, rank and select in $\mathcal{O}(\log \sigma)$ time; thus, by itself it is a CRL data structure taking $\mathcal{O}(\log \sigma)$ time per reported color. In a subsequent paper, Gagie et al. [21] reduced this time to $\mathcal{O}(\log(\sigma/\mathsf{ncol}))$, where $\mathsf{ncol}$ is the number of colors reported, by replacing range quantile queries with depth-first-search traversal on the wavelet tree. They also used a more compact wavelet tree [23] to reduce the space to $nH_0(S) + o(n)$ bits.

Very recently, Belazzougui and Navarro [6] gave a new solution for colored range listing that uses $n \log \sigma + \mathcal{O}(n \log \log \sigma)$ bits of space and answers queries in $\mathcal{O}(1)$ time per reported color, and another that uses $n \log \sigma + \mathcal{O}(n \log \log \log \sigma)$ bits and $\mathcal{O}(\log \log \sigma)$ time per reported color; both solutions return the colors' frequencies. They replace the structures that solve rank by weaker structures, based on monotone minimum perfect hash functions (mmphfs), that answer queries of the

[1] Our logarithms are base 2 by default.

**Table 1**
Existing and new solutions for color range listing. We give the time to list each color without and with frequency information.

| Source | Space (in bits) | Time per color | Time including frequencies |
|---|---|---|---|
| [47] | $\mathcal{O}(n \log n)$ | $\mathcal{O}(1)$ | |
| [57] | $nH_0(S) + 2n + o(n) \log \sigma$ | $\mathcal{O}\left(1 + \frac{\log \sigma}{\log \log n}\right)$ | $\mathcal{O}\left(1 + \frac{\log \sigma}{\log \log n}\right)$ |
| [22] | $nH_0 + o(n) \log \sigma$ | $\mathcal{O}(\log \sigma)$ | $\mathcal{O}(\log \sigma)$ |
| [21] | $nH_0 + o(n)$ | $\mathcal{O}(\log(\sigma/\mathsf{ncol}))$ | $\mathcal{O}(\log(\sigma/\mathsf{ncol}))$ |
| [6] | $n \log \sigma + \mathcal{O}(n \log \log \sigma)$ | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| [6] | $n \log \sigma + \mathcal{O}(n \log \log \log \sigma)$ | $\mathcal{O}(1)$ | $\mathcal{O}(\log \log \sigma)$ |
| 2+8 | $nH_0(S) + 2n + o(n)$ | $\mathcal{O}\left(1 + \frac{\log \sigma}{\log w}\right)$ | $\mathcal{O}\left(1 + \frac{\log \sigma}{\log w}\right)$ |
| 3+8 | $nH_k(S) + 2n + o(n) \log \sigma$ | $\mathcal{O}(1)$ | $\mathcal{O}\left(\log \frac{\log \sigma}{\log w}\right)$ |
| 4+8 | $nH_0(S) + 2n + o(n)(H_0(S) + 1)$ | $\mathcal{O}\left(\log \frac{\log \sigma}{\log w}\right)$ | $\mathcal{O}\left(\log \frac{\log \sigma}{\log w}\right)$ |
| 5+8 | $nH_0(S) + 2n + o(n)(H_0(S) + 1)$ | $\mathcal{O}(1)$ | $\mathcal{O}(\log \log \sigma \log \log \log \sigma)$ |
| 5+8+10 | $nH_0(S) + o(n)H_0(S) + \mathcal{O}(n \log \log \log \sigma)$ | $\mathcal{O}(1)$ | $\mathcal{O}(\log \log \sigma)$ |
| 5+8+9 | $nH_0(S) + o(n)H_0(S) + \mathcal{O}(n \log \log \sigma)$ | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| 3+8+9 | $nH_k(S) + o(n) \log \sigma + \mathcal{O}(n \log \log \sigma)$ | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |

form $\mathsf{rank}_{S[i]}(S, i)$. They show that these are sufficient if one locates the first and last occurrence of each color in the array. This is achieved using two symmetric RMQ structures. They use this result to give a solution for document listing that takes $\mathcal{O}(n \log \log D)$ bits on top of the CSA and answers queries in $\mathcal{O}(\mathsf{search}(m) + \mathsf{ndoc} \cdot \mathsf{lookup}(n))$ time, and another that takes $\mathcal{O}(n \log \log \log D)$ extra bits and answers queries in $\mathcal{O}(\mathsf{search}(m) + \mathsf{ndoc}(\mathsf{lookup}(n) + \log \log D))$ time; again, both solutions return frequencies.

The top part of Table 1 summarizes the existing solutions for CRL.

### 2.2. Related work on document listing

Muthukrishnan [47] gave his solution to the CRL problem as part of a solution to the problem of document listing (DL), in which we are given a library of documents and asked to preprocess them such that later, given a pattern $p[1, m]$, we can quickly list all the distinct documents containing that pattern (see [43] for older work). Let $T[1, n]$ be the concatenation of the $D$ documents. Muthukrishnan defined the array $E[1, n]$ such that $E[i]$ is the document containing the starting position of the lexicographically $i$th suffix in $T$. All the positions where $p$ occurs in $T$ correspond to starting positions of suffixes that start with $p$, and all those suffixes are listed contiguously in $E$, say in the range $E[i, j]$. It follows that the documents where $p$ appears are those mentioned in $E[i, j]$, and that the multiplicities of the document identifiers in $E[i, j]$ correspond to the frequencies of $p$ in the corresponding documents. Therefore, once we know $i$ and $j$, we can implement a DL query as a CRL query on $E[i..j]$.

To compute $i$ and $j$, Muthukrishnan used a classical stringology data structure called the suffix tree [58,1] of $T$. It occupies $\mathcal{O}(n \log n)$ bits and gives $i$ and $j$ in time $\mathcal{O}(m)$. Thus the DL solution requires $\mathcal{O}(n \log n)$ bits of space and $\mathcal{O}(m + \mathsf{ndoc})$ time to list the $\mathsf{ndoc}$ documents containing $p[1, m]$.

Just as for CRL, the next developments have focused on reducing the space of this solution. The difference with CRL solutions is the $\mathcal{O}(n \log n)$-bit space suffix tree. A smaller structure, the suffix array $A[1, n]$ [41], simply lists the suffixes of $T$ in lexicographical order. The suffixes starting with $p$ form an interval $A[i, j]$. While this structure still requires $\mathcal{O}(n \log n)$ bits of space, there is a wealth of compressed variants of it [48], using as little space as that of the compressed text and including the text. In this paper, we will call them generically compressed suffix arrays (CSAs), and refer to their space in bits as $|\mathsf{CSA}|$. CSAs find the interval of suffixes starting with $p[1, m]$ in times from $\mathcal{O}(m)$ to $\mathcal{O}(m \log n)$, to which we will refer generically as $\mathsf{search}(m)$. Finally, they compute any cell $A[i]$ in time from constant to polylogarithmic in $n$, to which we will refer generically as $\mathsf{lookup}(n)$.

Once the issue of the CSA is sorted out, any CRL solution can be converted into a DL solution. For example, Välimäki and Mäkinen [57] combined their CRL data structure with a CSA, obtaining a DL data structure that takes $|\mathsf{CSA}| + n \log D + 2n + o(n) \log D$ bits and $\mathcal{O}(\mathsf{search}(m) + \mathsf{ndoc}(1 + \log D/\log \log n))$ time. The pattern's frequency in a document $d$ can also be computed within the same time. Finally, they noted that, using one select query per occurrence, they can list the positions of the pattern's occurrences in a specified document. Similarly, Gagie et al.'s [21] CRL solution, combined with a CSA of $T$,

yields a DL data structure that takes $|\mathsf{CSA}|+n\log D+o(n)\log D$ bits and $\mathcal{O}(\mathsf{search}(m)+\mathsf{ndoc}\log(D/\mathsf{ndoc}))$ time, frequencies included.

Finally, Belazzougui and Navarro's [6] CRL solution can be converted into a DL solution that takes $|\mathsf{CSA}|+\mathcal{O}(n\log\log D)$ bits and answers queries in $\mathcal{O}(\mathsf{search}(m)+\mathsf{ndoc}\cdot\mathsf{lookup}(n))$ time, and another that takes $|\mathsf{CSA}|+\mathcal{O}(n\log\log\log D)$ bits and answers queries in $\mathcal{O}(\mathsf{search}(m)+\mathsf{ndoc}(\mathsf{lookup}(n)+\log\log D))$ time. Both solutions return frequencies within the same time bounds.

Sadakane [55] initiated another line that, although it is based on the idea of Muthukrishnan, does not derive directly from a CRL solution. He replaced Gabow et al.'s [18] RMQ data structure by a $4n+o(n)$ bit index that, given a range $C[i..j]$, in $\mathcal{O}(1)$ time and without consulting $C$, returns the position of the minimum value in that range (but not the value itself). He also showed how the CSA and a bit vector $V[1,n]$ can simulate access to $E$: 1s in $V$ mark the positions in $T$ where the documents start; then, for $1\leq q\leq n$, $E[q]=\mathsf{rank}_1(V,\mathsf{CSA}[q])$, where $\mathsf{rank}_1(V,r)$ is the number of 1s in $V[1..r]$. It takes $D\log(n/D)+\mathcal{O}(D)+o(n)$ bits to store $V$ such that a $\mathsf{rank}_1$ query takes $\mathcal{O}(1)$ time [53]. Sadakane did not store $C$ at all so, when listing the distinct documents containing a pattern, he used a $D$-bit string to mark which documents he had already listed. He used a recursion similar to Muthukrishnan's, stopping whenever it finds a document already reported.

Sadakane's DL data structure uses $|\mathsf{CSA}|+4n+D\log(n/D)+\mathcal{O}(D)+o(n)$ bits and $\mathcal{O}(\mathsf{search}(m)+\mathsf{ndoc}\cdot\mathsf{lookup}(n))$ time. He used $|\mathsf{CSA}|+4n+o(n)$ additional bits for data structures to compute the pattern's frequency in each document, increasing the time bound to $\mathcal{O}(\mathsf{search}(m)+\mathsf{ndoc}(\mathsf{lookup}(n)+\log\log\mathsf{ndoc}))$ (assuming $\mathsf{lookup}(n)$ is also the time to find $\mathsf{CSA}^{-1}[q]$, where $\mathsf{CSA}^{-1}$ denotes the inverse permutation of $A$).

Hon et al. [32] described a solution to DL similar to Sadakane's but removing the $\Theta(n)$-bit space term. They pack $\log^{\epsilon}n$ consecutive cells of $C$ into a block and build the RMQ data structure on the block minima (so it takes $\mathcal{O}(n/\log^{\epsilon}n)$ bits of space), and reports (avoiding repetitions) all the documents in the block that holds the minimum. Their whole data structure takes $|\mathsf{CSA}|+D\log(n/D)+\mathcal{O}(D)+o(n)$ bits and answers queries in time $\mathcal{O}(\mathsf{search}(m)+\mathsf{ndoc}\log^{\epsilon}n\cdot\mathsf{lookup}(n))$, for any constant $\epsilon>0$.

They can also return the number of times the pattern occurs in any document by using, like Sadakane [55], one $\mathsf{CSA}_d$ local to each document $d$. These add up to other $|\mathsf{CSA}|$ extra bits. To find out how many times document $d=E[q]$, $i\leq q\leq j$, appears in $E[i..j]$, it maps $q$ to position $p=\mathsf{CSA}[q]-\mathsf{select}_1(V,d)+1$ within document $d$, and then to $q'=\mathsf{CSA}_d^{-1}[p]$. This is the first lexicographic occurrence of the pattern in $\mathsf{CSA}_d$. The last occurrence is found by an exponential search and then by a binary search on $\mathsf{CSA}_d[q'..]$, for the largest $c$ such that $\mathsf{CSA}^{-1}[\mathsf{CSA}_d[q'+c]+\mathsf{select}_1(V,d-1)]\leq j$. Then the answer, $c+1$, is obtained in time $\mathcal{O}(\mathsf{lookup}(n)\log c)=\mathcal{O}(\mathsf{lookup}(n)\log n)$.

### 2.3. New tradeoffs

All the previous solutions have essentially the same ingredients: for CRL, access to $S$, distinct color enumeration on $S$ (implemented via RMQs on $C$ or range quantile queries on $S$) and, to count the number of times each color occurs, rank on $S$; for DL, a suffix tree or CSA for $T$, access to $E$, distinct document enumeration on $E$ and, to report the pattern's frequency in each document, rank on $E$. Solutions for CRL can be used for DL with the addition of a CSA for $T$, setting $S=E$ and $\sigma=D$. Recall that Sadakane's [55] and Hon et al.'s [32] solutions for DL implement access to $E$ using a CSA and bit vector $V$ on $T$, so they do not apply to general CRL.

Our main contribution in this section is the observation that, using new data structures for access, color enumeration and rank, we obtain new bounds for both CRL and DL. This is formalized next.

**Observation 1.** *Suppose we are given a sequence $S[1,n]$ over $[1,\sigma]$ and we store any data structure supporting access on $S$ in time $t_{\mathsf{acc}}$ and any structure supporting distinct enumeration in a range of $S$ in time $t_{\mathsf{enum}}$ per element (and any structure supporting rank on $S$ in time $t_{\mathsf{rank}}$ if computing frequencies is desired). Then later, given $i$ and $j$, we can list the distinct elements in $S[i..j]$ in time $\mathcal{O}(t_{\mathsf{acc}}+t_{\mathsf{enum}})$ per reported element. The cost to list, in addition, the frequency in $S[i..j]$ of a reported element is $\mathcal{O}(t_{\mathsf{rank}})$.*

**Corollary 2.** *Given a concatenation $T[1,n]$ of $D$ documents, we can store either*

- *the CSA for $T$ and data structures supporting access, enumeration and rank on the corresponding array $E[1,n]$ in times $t_{\mathsf{acc}}$, $t_{\mathsf{enum}}$ and $t_{\mathsf{rank}}$, or*
- *the CSA for $T$, a bit vector occupying $D\log(n/D)+\mathcal{O}(D)+o(n)$ bits, and data structures supporting enumeration and rank on $E$ as above,*

*such that, given a pattern of length $m$, we can list the distinct documents containing that pattern in time $\mathcal{O}(\mathsf{search}(m))$ plus $\mathcal{O}(t_{\mathsf{acc}}+t_{\mathsf{enum}}+t_{\mathsf{rank}})$ per reported document, where $t_{\mathsf{acc}}=\mathsf{lookup}(n)$ in the second case and $t_{\mathsf{rank}}$ is required only in order to list the frequencies of the documents.*

A selection of these data structures is shown in Table 2 (for conciseness, we show only the results that are currently the best for our purposes, leaving aside many previous ones [15,5,24,28] on which most of the new ones build). Note that one solution (row 3) achieves high-order entropy space, $nH_k(S)$. This is a lower bound to the bits per symbol emitted by any semistatic statistical compressor that encodes each symbol as a function of the $k$ previous ones [42]. It holds $H_k(S)\leq H_{k-1}(S)\leq H_0(S)\leq\log\sigma$.

**Table 2**
Space and time bounds for some data structures supporting operations on $S[1, n]$ over $[1, \sigma]$, where $w$ is the length in bits of the computer word. The $\mathcal{O}(\sigma \log n)$ extra bits of wavelet trees can be avoided [40] so we have not included them. The space bound in row 3 holds for $k = o(\log_\sigma n)$. In rows 6 and 7, $g$ is the size (in bits) of a given context-free grammar generating $S$ and only $S$. In rows 4 and 6, $\alpha(\cdot)$ is the inverse Ackermann function. Rows 9 and 10 only solve queries of the form $\mathsf{rank}_{S[i]}(S, i)$.

| Row | Source | Space (in bits) | $t_{\mathsf{acc}}$ | $t_{\mathsf{enum}}$ | $t_{\mathsf{rank}}$ |
|---|---|---|---|---|---|
| 1 | [21] | $nH_0(S) + o(n)$ | $\mathcal{O}(\log(\sigma/\mathsf{ncol}))$ | $\mathcal{O}(\log(\sigma/\mathsf{ncol}))$ | $\mathcal{O}(\log(\sigma/\mathsf{ncol}))$ |
| 2 | [7, Theorem 7] | $nH_0(S) + o(n)$ | $\mathcal{O}\left(1 + \frac{\log \sigma}{\log w}\right)$ | | $\mathcal{O}\left(1 + \frac{\log \sigma}{\log w}\right)$ |
| 3 | [7, Theorem 9] | $nH_k(S) + o(n) \log \sigma$ | $\mathcal{O}(1)$ | | $\mathcal{O}\left(\log \frac{\log \sigma}{\log w}\right)$ |
| 4 | [7, Theorem 8] | $nH_0(S) + o(n)(H_0(S) + 1)$ | $\mathcal{O}(\alpha(\sigma))$ | | $\mathcal{O}\left(\log \frac{\log \sigma}{\log w}\right)$ |
| 5 | [4, Theorem 1] | $nH_0(S) + o(n)(H_0(S) + 1)$ | $\mathcal{O}(1)$ | | $\mathcal{O}(\log \log \sigma \log \log \log \sigma)$ |
| 6 | [8, Theorem 1] | $\mathcal{O}(g\, \alpha(g))$ | $\mathcal{O}(\log n)$ | | |
| 7 | [8, Theorem 1] | $\mathcal{O}(g)$ | $\mathcal{O}(\log n \log \log n)$ | | |
| 8 | [16, Theorem 1] | $2n + o(n)$ | | $\mathcal{O}(1)$ | |
| 9 | [6, Theorem 1] | $\mathcal{O}(n \log \log \sigma)$ | | | $\mathcal{O}(1)$ |
| 10 | [6, Theorem 1] | $\mathcal{O}(n \log \log \log \sigma)$ | | | $\mathcal{O}(\log \log \sigma)$ |

If we choose a set of rows covering support for access and enumeration (and rank) then we can answer CRL queries (and return the frequency of each color). The space bound is the sum of the space bounds and the time bound per reported color is $\mathcal{O}(t_{\mathsf{acc}} + t_{\mathsf{enum}} + t_{\mathsf{rank}})$, the latter term for computing frequencies.

The bottom part of Table 1 shows several combinations that improve upon previous results for CRL. The numbers in italics correspond to the rows of Table 2 used. The first rows, 2+8 to 5+8, are combinations of a compressed sequence representation to provide access and rank (by Belazzougui and Navarro [7] or Barbay et al. [4]), with enumeration provided via Fischer's [16] succinct index for RMQ (which does not access the array). All those improve upon the previous solution of Välimäki and Mäkinen [57] in space and time. The next rows, 3/5+8+9/10, incorporate an mmphf to the previous combination, improving the time (in many cases to constant) in exchange for higher space. The result is a compressed variant of Belazzougui and Navarro's [6] solution, and it turns out to be the first compressed CRL data structure (using $nH_k(S) + o(n \log \sigma)$ bits of space) that answers queries in constant time per returned color.

For conciseness, we do not explicitly enumerate the new DL solutions that derive from our new CRL solutions; those should be immediate from Corollary 2. It is also possible to derive new solutions that are specific for DL using the table. For example, an obvious one is that Sadakane's solution [55] improves by using the newer RMQ solution by Fischer [16], which requires $2n + o(n)$ bits instead of Sadakane's original $4n + o(n)$.

The alternatives listed in Table 1 are not formally comparable. There are some solutions that always use less space than others; one can order 2+8 < 4/5+8 < *5+8+10* < 5+8+9, and 3+8 < 3+8+9, but those using more space are faster. In many cases, however, the space comparison depends on the relation between $nH_0(S) + o(n)H_0$ and $H_k(S) + o(n) \log \sigma$, and this depends on the application in which the CRL problem arises. In the particular case of DL, $H_0(E)$ is related to the lengths of the documents; that is, it will be smaller if documents have very different lengths, and will approach $\log D$ when documents are roughly the same size. More interesting is $H_k(E)$, which depends on how predictable is the next document if we have seen the $k$ previous cells in $E$. The more predictable $E$ is in this sense, the lower is $H_k(E)$. While we do not have formal bounds, we expect that $H_k(E)$ will be lower when $E$ is more repetitive. Next we show that $H_k(T)$, the compressibility of $T$, is related to repetitiveness of $E$, and exploit that relation using a completely different tool.

This result is obtained by combining Bille et al.'s [8] grammar-based data structure for access (lines 6/7), Fischer's [16] succinct index for RMQ (line *8*), and the smaller mmphfs [6] for rank (line 10). González and Navarro [25] showed how to build a grammar generating an array that, together with some other small data structures, gives access to the suffix array (SA) $A$. Building Bille et al.'s data structure for this grammar, we obtain an $\mathcal{O}(\log n)$-time data structure for DL whose size is bounded in terms of the high-order entropies of the library of documents. This is described next.

**Theorem 3.** *Given a concatenation $T[1, n]$ of $D$ documents, we can store $T$ in*

$$|\mathsf{CSA}| + \mathcal{O}(n \log \log \log D) + \mathcal{O}\left(\left(n \min(H_k(T), 1) + D\right) \log\left(\frac{1}{\min(H_k(T), 1) + D/n}\right) \alpha(n) \log n\right)$$

*bits, for any $k \leq \beta \log_\sigma n$, constant $0 < \beta < 1$ and $\sigma$ the size of the alphabet of $T$. Then given a pattern of length $m$, we can list the distinct documents containing that pattern in time $\mathcal{O}(\mathsf{search}(m))$ plus $\mathcal{O}(\log n)$ to list each document with its frequency.*

**Proof.** González and Navarro's algorithm takes advantage of the so-called *runs* of the SA, that is, areas $A[i..i + \ell]$ such that there is some other area $A[j..j + \ell]$ where $A[j + k] = A[i + k] + 1$ for all $0 \leq k \leq \ell$. Let $R$ be the number of runs with which

the SA can be covered; it is known that $R \le \min(n, nH_k(T) + \sigma^k)$ for any $k$ [38]. González and Navarro represent the SA differentially so that these areas become true repetitions, and use a grammar-based compression algorithm that represents $A$ using $\mathcal{O}(R\log(n/R))$ rules. We note that, in $E$, those SA runs become identical areas $E[i..i+\ell] = E[j..j+\ell]$ except for at most $D$ cells where the document number can change when we advance by one text position. It follows that, by applying the same compression algorithm [25] to $E$ we obtain $\mathcal{O}((R+D)\log(n/(R+D)))$ rules and hence the space given in the theorem. □

As a final note applying only to document collections, Sadakane's CSA [54] essentially represents a function $\Psi$ such that $A[\Psi(i)] = A[i]+1$, which is stored in compressed form and any value computed in constant time. Thus one advances virtually in the text by successively applying $\Psi$. Now assume we sample $E$ with a step $r$ such that, for any $i$, $E[\Psi^j(i)]$ is sampled for some $0 \le j < r$. Then one computes any $E[i]$ value in time $\mathcal{O}(r)$ by following $\Psi$ until reaching a sampled entry, whose value will be the same as $E[i]$ if we also sample every document end in the text collection. The space is $\mathcal{O}((n/r)\log r) + (n/r)\log D$ for a bitmap marking the sampled cells and an array with the sampled values, respectively. For example, using $r = \log D$ yields access to $E$ (though not rank nor select on it) in the same time as a binary wavelet tree, within bit space $n + o(n)$. Depending on the relation between $n$ and $D$, this can be an interesting alternative to using lookup and marking the document beginnings [55].

## 3. Top-*k* queries

### 3.1. Improving the current-best solution for documents

Hon et al. [30] described a data structure that stores a library $T$ of $D$ documents of total length $n$ in $\mathcal{O}(n\log^2 n)$ bits such that later, given a pattern of length $m$ and an integer $k \ge 1$, we can find the $k$ documents that contain that pattern most frequently, in $\mathcal{O}(m + \log n \log\log n + k)$ time. We call this the document top-$k$ problem (DTK). Hon et al. [32] gave solutions for DTK that store $T$ in $\mathcal{O}(n\log n)$ bits and answer queries in $\mathcal{O}(m + k\log k)$ time, or in $2|\mathsf{CSA}| + o(n) + D\log(n/D) + \mathcal{O}(D)$ bits and $\mathcal{O}\left(\mathsf{search}(m) + k\log^{3+\epsilon} n \cdot \mathsf{lookup}(n)\right)$ time.

The last solution consists of a tree $\tau_k$ built for each $k$ power of 2. For $\tau_k$ they divide $E$ into blocks of size $z = k\log^{2+\epsilon} n$, and $\tau_k$ consists of the suffix tree nodes that are lowest common ancestors (*lca*) of end points of blocks, and transitively all the *lca*s of pairs of those nodes. At each node, $\tau_k$ stores the $k$ most frequent documents within the whole blocks it contains, and their frequencies. Thus each $\tau_k$ requires $\mathcal{O}((n/z)k\log n) = \mathcal{O}\left(n/\log^{1+\epsilon} n\right)$ bits, and all the trees together add up to $\mathcal{O}(n/\log^{\epsilon} n)$ bits. At query time, to find the top-$k$ documents in $E[i..j]$, they increase $k$ to the next power of 2 and find the highest node of $\tau_k$ whose range $[i'..j']$ is contained in $[i..j]$. They show that $i' - i \le z$ and $j - j' \le z$ by the *lca* properties. Then the query is answered by considering the $k$ candidates given by $\tau_k$ and the $\mathcal{O}(z)$ further candidates found at positions of $E[i..i'-1]$ and $E[j'+1..j]$, for each of which they compute the frequency. The total time, considering priority queue operations, is $\mathcal{O}(\mathsf{search}(m) + z(t_{\mathsf{rank}} + \log k) + k\log k) = \mathcal{O}\left(\mathsf{search}(m) + k\log^{3+\epsilon} n \cdot \mathsf{lookup}(n)\right)$. This time bound can be improved to $\mathcal{O}\left(\mathsf{search}(m) + k\log D\log(D/k)\log^{1+\epsilon} n \cdot \mathsf{lookup}(n)\right)$ by noticing that (a) one needs only $\mathcal{O}(\log D)$ powers of 2 for $k$ since $k \le D$; (b) one can store the top-$k$ elements in the $\tau_k$ trees and not their frequency. The $k$ frequencies can be computed at query time without changing the time complexity since $k = o(z)$. Thus the $k$ documents out of $D$ can be stored in increasing order and as gamma-encoded differences, taking $\mathcal{O}(k\log(D/k))$ bits. Therefore we can use smaller blocks of size $z = k\log D\log(D/k)\log^{\epsilon} n$, which are processed faster, and still have $\mathcal{O}(n/\log^{\epsilon} n) = o(n)$ space for the structure.

Note that, for this solution to work for any $\tau_k$, we also need to represent $\tau_k$ in compact form.[2] A succinct tree representation [56] using just $2 + o(1)$ bits per node supports in $\mathcal{O}(1)$ time many operations, including *lca*, *preorder* (whose consecutive values are used to index an array storing the top-$k$ candidate data on each node), and $preorder^{-1}$. For each pair of consecutive block endpoints $p_i$ and $p_{i+1}$ we store the preorder $x_i$ of the sampled tree node $lca(p_i, p_{i+1})$. As $x_i \ge x_{i-1}$, values $x_i + i$ are increasing, and thus can be stored in a structure of $(n/z)\lg\frac{2n}{n/z} + \mathcal{O}(n/z)$ bits that retrieves any $x_i$ in constant time [51].[3] This space is $\mathcal{O}((n/z)\lg z) = o(n)$ bits. With this structure we can find in constant time the lowest sampled node covering a block interval $[L, R]$ as $lca(preorder^{-1}(x_L), preorder^{-1}(x_{R-1}))$.

In addition, we can replace the $|\mathsf{CSA}|$ bits of that solution for computing frequencies, by Grossi et al.'s [28] succinct index for rank, in the spirit of Section 2.[4] This index requires $n\,o(\log D)$ bits of space and computes any rank on $E$ via $\mathcal{O}(\log\log D)$ accesses to $E$. In this way, we achieve a new space bound of $|\mathsf{CSA}| + o(n) + D\log(n/D) + \mathcal{O}(D) + n\,o(\log D)$ bits, which can be better or worse than before, but the time is reduced to $\mathcal{O}(\mathsf{search}(m) + k\log D\log(D/k)\log^{\epsilon} n \cdot \mathsf{lookup}(n))$, for any $\epsilon$ (log–logarithmic terms disappear by adjusting $\epsilon$).

### 3.2. An approximate solution to the general problem

We now give a solution to the approximate colored range top-$k$ problem (CRTK), which asks us to preprocess a given sequence $S$ such that later, given a range $S[i..j]$ and an integer $k \ge 1$, we can return an approximate list of the $k$ elements

2 This was noted and solved [6] after our conference publication. We reproduce that solution here.

3 Using a constant-time rank/select implementation on their internal bitmap $H$ [46].

4 This index was superseded by another [6] in Section 2, and hence not listed there; however the solution dominating it computes a weaker version of rank that is of no use here.

("colors") that occur most frequently in that range. We do not know of any previous efficient solutions to this problem, although finding the $k$ most frequent or important items in various data sets and models is a well-studied problem and there has been work on interesting special cases (see, e.g., [33,36]).

Greve et al. [26] recently gave a data structure that, for any $\epsilon > 0$, stores $S$ in $\mathcal{O}((n/\epsilon)\log n)$ bits such that we can find an element with the property that no element is more than $1+\epsilon$ times more frequent in $S[i, j]$, in $\mathcal{O}(\log(1/\epsilon))$ time. Thus, their data structure solves the approximate CRTK problem for $k = 1$, which is called the approximate range-mode problem. As motivation for studying the approximate range-mode problem, they also proved a lower bound implying that a data structure using $n\log^{\mathcal{O}(1)} n$ space takes $\Omega(\log n/\log\log n)$ time to answer *exact* range-mode queries, and any data structure answering such queries in $\mathcal{O}(1)$ time takes $n^{\Omega(1)}$ space. (The current upper bounds for exact range-mode queries are much larger, however: e.g., the best known data structure using $\mathcal{O}(n)$ words of $\mathcal{O}(n)$ bits each, takes $\mathcal{O}\left(\sqrt{n/\log n}\right)$ query time [12]; the best known data structure taking $\mathcal{O}(1)$ query time uses $\mathcal{O}\left(n^2\log\log n/\log^2 n\right)$ words [52].) We can assume Greve et al.'s data structure also returns the frequency of the approximate mode in $S[i..j]$, since adding a rank data structure for $S$ allows us to compute this and does not change their space bound. We show how to use their data structure as a building block to store $S$ in $\mathcal{O}((n/\epsilon)(H_0(S)+1)\log n)$ bits such that, given an integer $k$, we can approximately list the $k$ most common elements and their frequencies in $\mathcal{O}(k\log\sigma\log(1/\epsilon))$ time.

We first build a binary wavelet tree for $S$ [27]. This is a balanced tree where each node represents a range of $[1, \sigma]$: the root represents the full range, the leaves the individual symbols, and the children of a node represent the left and right halves of the node's range. For each node $v$, let $S_v$ be the subsequence of $S$ consisting of characters labeling the leaves in $v$'s subtree. The original wavelet tree does not store $S_v$, but just a bitmap $B_v$ of length $|S_v|$ telling whether each $S_v[i]$ went to the left or right child. Rank and select over those bitmaps allow accessing any $S[i]$, as well as computing $\mathsf{rank}_a(S, i)$ and $\mathsf{select}_a(S, i)$, in time $\mathcal{O}(\log\sigma)$, and the overall space is $n\log\sigma(1+o(1))$. The tree can also track any range $S[i..j]$ down to any node [40].

Here we do store each subsequence $S_v$ in an instance of Greve et al.'s approximate range-mode data structure. For now, assume $[i, j] = [1, n]$ and that Greve et al.'s data structure returns the exact mode, rather than an approximation. Notice that, if $a_1, \ldots, a_{k'}$ are the $k'$ most frequent elements and $v$ is an ancestor of the leaf labeled $a_{k'}$ but not of those labeled $a_1, \ldots, a_{k'-1}$, then $a_{k'}$ is the mode in $S_v$. Let $V$ be the set of ancestors of $a_1, \ldots, a_{k'-1}$ and let $V'$ be the set of nodes who are not in $V$ themselves but whose siblings are; $V'$ contains the root of the tree if $V$ is empty. We can find $a_{k'}$ by finding the mode of $S_v$ for each $v \in V'$, finding their frequencies in $S$, and taking the most frequent.

We keep the modes for each $v \in V'$ in a priority queue, ordered by their frequencies and with the corresponding nodes of the wavelet tree as auxiliary data. Notice $a_{k'}$ is the head of the queue, so we can find and output it in $\mathcal{O}(1)$ time; let $v$ be the corresponding node, i.e., the node in $V'$ such that the mode of $S_v$ is $a_{k'}$. To update the queue, we delete $a_{k'}$, perform range-mode queries on the siblings of nodes on the path from $v$ to the leaf labeled $a_{k'}$, and add the modes to the queue. There are always $\mathcal{O}(k\log\sigma)$ nodes in the queue (the tree is of height $\mathcal{O}(\log\sigma)$) so, if we use a priority queue allowing $\mathcal{O}(\log(k\log\sigma)) = \mathcal{O}(\log\sigma)$ time deletion and $\mathcal{O}(1)$ time insertion [11], then we can find the $k$ most frequent elements in $S$ in $\mathcal{O}(k\log\sigma\log(1/\epsilon))$ time. We can deal with general $i$ and $j$ by using the wavelet tree to compute the appropriate range in each subsequence [40].

Suppose that, instead of using a balanced wavelet tree, we use one with the same shape as the code-tree for a code with expected codeword length $\mathcal{O}(H_0(S)+1)$. For each occurrence of a symbol $a$ in $S$, there is an occurrence of $a$ in the subsequence $S_v$ for each node $v$ on the path from the root to the leaf labeled $a$. It follows that the total length of the subsequences in the whole tree is $\mathcal{O}(n(H_0(S)+1))$, so storing all the subsequences in instances of Greve et al.'s data structure takes $\mathcal{O}((n/\epsilon)(H_0(S)+1)\log n)$ bits. Using a Huffman-shaped wavelet tree [38] would minimize the total length of the subsequences, but a Huffman tree can be very deep (height $n-1$ for a very skewed distribution), which would compromise our time bound. Therefore, we use an $\mathcal{O}(\log\sigma)$-restricted Huffman tree [44], which yields both the space and time bounds we want.

**Theorem 4.** *Given a sequence $S[1, n]$ over an alphabet of size $\sigma$ and a constant $\epsilon > 0$, we can store $S$ in $\mathcal{O}((n/\epsilon)(H_0(S)+1)\log n)$ bits such that, given i, j and k, we can list k distinct elements such that no element is more than $1+\epsilon$ times more frequent in $S[i..j]$ than any of the ones we list, in $\mathcal{O}(k\log\sigma\log(1/\epsilon))$ time.*

This $(1+\epsilon)$-approximation makes sense in information retrieval scenarios, where top-$k$ queries are understood to be just approximations of the ideal answer.

### *3.3. The K-mining problem*

Muthukrishnan [47] defined (document) $K$-mining (DKM) as the problem of finding all the documents in the library that contain a given pattern at least $K$ times. He gave an $\mathcal{O}\left(n\log^2 n\right)$-bit data structure that, given $K$ and a pattern of length $m$, answers queries in $\mathcal{O}(m)$ time plus $\mathcal{O}(1)$ time per reported document. Hon et al. [30] noted that we can use a binary search with a DTK data structure to solve DKM, with an $\mathcal{O}(\log n)$ slowdown for the queries. They then showed how we can use an $\mathcal{O}\left(n\log^2 n\right)$-bit data structure to find the largest $k$ such that $k$ documents contain the pattern $K$ times, in $\mathcal{O}(\mathsf{search}(m)+\log n\log\log n)$ time. Hon et al. [32] gave an $\mathcal{O}(n\log n)$-bit data structure that answers $K$-mine queries in time $\mathcal{O}(m)$ plus $\mathcal{O}(1)$ per reported document. They also showed how to improve the space bound to $2|\mathsf{CSA}| + o(n) + D\log(n/D)$ bits at the cost of increasing the time $\mathcal{O}\left(\mathsf{search}(m) + k\log^{3+\epsilon} n \cdot \mathsf{lookup}(n)\right)$, which we can improve in much the same way as in Section 3.1. Neither of these solutions applies, however, to general colored range queries.

Since our CRTK data structure outputs elements in (approximately) non-increasing order by frequency in the range, it also solves (approximately) the natural generalization of DKM: i.e., the colored range $K$-mine (CRKM) problem, which asks us to report all the elements that occur at least $K$ times in $S[i..j]$. If we query our data structure until the next element it would report occurs fewer than $(1+\epsilon)K$ times, then we use $\mathcal{O}(\log\sigma \log(1/\epsilon))$ time per reported element, but we may miss some elements that occur between $K$ and $(1+\epsilon)K$ times. Alternatively, if we query our data structure until the next element it would report occurs fewer than $K/(1+\epsilon)$ times, then we find all the elements that occur at least $K$ times, but we can bound our time only in terms of the number of elements that occur at least $K/(1+\epsilon)$ times.

## 4. Counting

### 4.1. Related work

For general colored range counting, we are asked to store a set of $n$ colored points in $\mathbb{R}^d$ such that later, given an axis-aligned box, we can quickly count the distinct colors it contains. Most papers on this problem have focused on $d \geq 2$ dimensions (see, e.g., [35]). We consider the one-dimensional version of the problem. The best solution known for general static one-dimensional colored range counting is an $\mathcal{O}(n)$-word data structure by Bozanis et al. [9] that answers queries in $\mathcal{O}(\log n)$ time. The best dynamic solutions known [37] take, for queries and updates, either $\mathcal{O}(n\log n)$ words and $\mathcal{O}(\log n)$ time or $\mathcal{O}(n)$ words and $\mathcal{O}(\log^2 n)$ time. In this section, we consider the special case in which the colored points are the integers $1, \ldots, n$. Storing these points is equivalent to storing a string $S[1..n]$ over an alphabet whose size $\sigma$ is the number of distinct colors, such that later, given a substring's endpoints, we can quickly count how many distinct characters that substring contains. We describe a data structure for counting colors in strings, one that takes only $nH_0(S)+\mathcal{O}(n)+o(nH_0(S))$ bits, where $H_0(S)$ is the zero-order empirical entropy of $S$. Furthermore, we simultaneously reduce the query time to $\mathcal{O}(\log \ell)$, where $\ell$ is the size of the query range. As far as we know, no other data structure for colored range counting has a non-trivial upper bound depending only on $\ell$.

Our solution is based on the array $C[1..n]$ of Muthukrishnan [47] described in Section 2.1. Recall that each cell $C[q]$ stores the largest value $p < q$ such that $S[p] = S[q]$ (or 0 if no such $p$ exists), and thus $S[q]$ is the first occurrence of that distinct character in $S[i..j]$ if and only if $i \leq q \leq j$ and $C[q] < i$. Therefore, the number of distinct characters in $S[i..j]$ is the number of values in $C[i..j]$ strictly less than $i$. If we store $C$ in a wavelet tree [27], which takes $n\log n + o(n)$ bits [23], then we can count all such values $q \in [i..j]$ such that $C[q] \in [0..i-1]$ in $\mathcal{O}(\log n)$ time; for details see Mäkinen and Navarro [40]. This is already a slight improvement over the bounds we achieve with Bozanis et al.'s data structure [9]. The wavelet tree could be compressed using standard techniques, but this would reflect the compressibility of $C$. Instead, the space can be reduced to $n\log\sigma + \mathcal{O}(n\log\log n)$ bits, close to the size of $S$, by modifying the wavelet tree [20]. We reduce the space further by modifying the representation of $C$ rather than the wavelet trees.

Apart from counting the unique colors, our data structure can support other interesting queries. For example, we can count

- the "new colors" in an interval $S[i..j]$ (those that do not appear to the left of $i$) by counting the number of 0s in $C[i..j]$;
- the colors in an interval $S[i..j]$ whose last occurrence was in another interval $S[i'..j']$, by counting the number of values in $C[i..j]$ that are between $i'$ and $j'$;
- the colors that occur exactly once in $S[i..j]$.

To count the colors that occur exactly once in $S[i..j]$, we use three instances of our data structure. We build the first instance normally, we build the second instance replacing even occurrences (2nd, 4th, etc.) of each character by a special filler character #, and we build the third instance replacing odd occurrences (1st, 3rd, etc.) of each character by this filler; e.g., if $S =$ `abracadabra`, then the three instances are `abracadabra`, `abr#cad###a` and `###a###abr#`, respectively. (Since the second and third instances are for complementary strings, we could merge them fairly easily; we consider them separately for the sake of simplicity.) Given $S[i..j]$, we use the first instance to find the total number $d_{\mathsf{all}}$ of distinct characters in $S[i..j]$, we use the second instance to find the number $d_{\mathsf{odd}}$ of distinct characters that have an odd occurrence in $S[i..j]$, and we use the third instance to find the number $d_{\mathsf{even}}$ of distinct characters that have an even occurrence in $S[i..j]$. The number of distinct characters that have both an odd and an even occurrence is $d_{\mathsf{odd}} + d_{\mathsf{even}} - d_{\mathsf{all}}$, so the number of characters that have only an odd or only an even occurrence — i.e., exactly one occurrence — is $2d_{\mathsf{all}} - d_{\mathsf{odd}} - d_{\mathsf{even}}$.

In document retrieval, i.e., with $S = E$ (see Section 2.1), colored range counting can be used for computing the document frequency of a given pattern, i.e., how many documents contain it. We note that Sadakane [55] gave a faster and more space-efficient data structure for computing the document frequencies of single patterns, but his solution cannot be used for colored range counting in arbitrary strings. In Section 5, we discuss document retrieval scenarios that are supported by our data structures but not by Sadakane's.

In Section 4.2, we describe a simple data structure that takes $n(\log\sigma + \log\log n + 2 + o(1))$ bits and answers queries in $\mathcal{O}(\log n)$ time. In Section 4.3, we extend the ideas from Section 4.2 to build a data structure that takes $nH_0(S) + \mathcal{O}(n) + o(nH_0(S))$ bits and answers queries in $\mathcal{O}(\alpha(n)\log n\log\log n)$ time, where $\alpha$ is the inverse Ackermann function. We adjust our data structure and analysis slightly in Section 4.4, so that our time bounds are in terms of $\ell$, the length of the substring whose distinct colors we are counting, rather than in terms of $n$. In Section 4.5, we reorganize our data structure and improve the query time to $\mathcal{O}(\log \ell)$. For this result, we need a couple of simple but non-standard tricks in implementing wavelet trees;

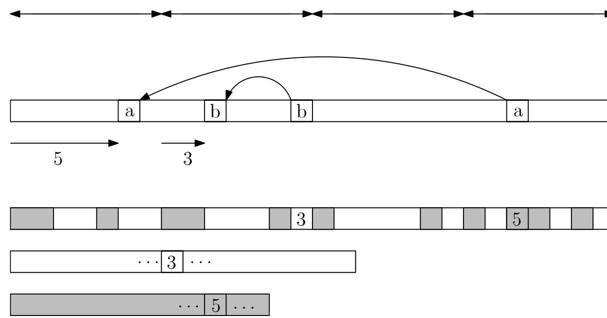


**Fig. 1.** The array $S$, with lines above indicating blocks and arcs indicating characters' previous occurrences; our representation of the $C$ array overlaid on the bitvector, with white indicating intra-block pointers and gray indicating inter-block pointers; and the contents of the two wavelet trees — intra-block pointers in one and inter-block pointers in the other. Notice that, since both copies of $b$ are contained within one block, the distance 3 is measured from the beginning of that block.

previous sections use standard wavelet trees as a black box. In Section 4.6, we show how our data structure can be made dynamic. Specifically, we first show how to achieve the same time bound for querying and a space bound of $\mathcal{O}(n(H_0(S)+1))$ bits while supporting an $\mathcal{O}(\log n)$-time append operation, which is the most natural update when, e.g., maintaining log files. We then show how to support color substitutions and deletions, at the cost of using $\mathcal{O}(\log^2 n)$ time for both queries and updates. Finally, we show how our data structure replaces $S$ (both in the static and dynamic case) by giving access to any $S[i]$ in reasonable time.

### 4.2. Simple blocking

In this section, we give a simple proof that, using two normal wavelet trees and a straightforward encoding of $C$, we need store only $n(\log\sigma + \log\log n + o(1))$ bits to answer queries in $\mathcal{O}(\log n)$ time. Without loss of generality, assume $\sigma = o(n/\log n)$; otherwise, we achieve our desired bound by simply storing $C$ in a single, normal wavelet tree. Our idea is to break $S$ into blocks of length $b = \sigma\log n$ and encode the entry $C[q]$ differently depending on whether the previous occurrence $S[p]$ of the character $S[q]$ is contained in the same block. If $p$ is contained in the same block as $q$, then we write $C[q]$ as the $\lceil\log b\rceil$-bit offset of $p$ within the block; otherwise, we write it as the $\lceil\log n\rceil$-bit binary representation of $p$. Notice that, for each block, there are at most $\sigma$ entries of $C$ encoded as $\lceil\log n\rceil$-bit numbers.

We build a bitvector indicating how each entry of $C$ is encoded, which takes $n + o(n)$ bits. We build one wavelet tree storing all the $\lceil\log b\rceil$-bit encodings, which takes at most $n\log b + o(n) = n(\log\sigma + \log\log n + o(1))$ bits, and another storing all the $\lceil\log n\rceil$-bit encodings, which takes at most $\sigma\lceil n/b\rceil\log n + o(\sigma\lceil n/b\rceil) = n + o(n)$ bits. This is illustrated in Fig. 1. Notice that, if $S[q]$ is the first occurrence of a character in $S[i..j]$ and $C[q]$ is encoded in $\lceil\log b\rceil$ bits, then $q$ must be between $i$ and the end of the block containing $i$. This is because, if $S[q]$ were in a later block, then $C[q] < i$ would be encoded using $\lceil\log n\rceil$ bits. Therefore we can count all such first occurrences in $\mathcal{O}(\log b) = \mathcal{O}(\log\sigma + \log\log n)$ time using the bitvector and the first wavelet tree (looking for positions in $C[i..\lceil i/b\rceil\cdot b]$ with offsets in $[0..(i \bmod b)-1]$). We can count all the other first occurrences in $\mathcal{O}(\log n)$ time using the bitvector and the second wavelet tree (using the normal query after mapping the positions using the bitvector).

**Theorem 5.** *Given a string $S[1..n]$, we can build a data structure that takes $n(\log\sigma + \log\log n + 2 + o(1))$ bits such that later, given a substring's endpoints, in $\mathcal{O}(\log n)$ time we can count how many distinct characters it contains.*

Notice that, if $\sigma \geq 4\log n$, then the data structure we just presented is within a factor of 2 of being succinct. If $\sigma < 4\log n$, then we can store $S$ in a multiary wavelet tree [14], which takes $nH_0(S) + o(n)$ bits, and answer any query by enumerating the characters in the alphabet and, for each one, using two $\mathcal{O}(1)$-time rank queries to see whether it occurs in the given substring.

**Corollary 6.** *Given a string $S[1..n]$, we can build a data structure that takes $2n\log\sigma + o(n)$ bits such that later, given a substring's endpoints, in $\mathcal{O}(\log n)$ time we can count how many distinct characters it contains.*

### 4.3. Multi-size blocking

In this section, we extend our idea from the previous section so that, instead of encoding entries of $C$ differently for only two block sizes — i.e., $\sigma\log n$ and $n$ — we use many block sizes. In particular, we use $\mathcal{O}(\log\log n/\log(1+\delta))$ different block sizes,

$$2^{1+\delta},\ 2^{\max\left((1+\delta)^2,2\right)},\ 2^{\max\left((1+\delta)^3,3\right)},\ 2^{\max\left((1+\delta)^4,4\right)},\ \ldots,\ n,$$

where $\delta \in (0, 1]$ is a value we will specify later. Also, for each block size $b$, we consider $S$ to consist of about $2n/b$ evenly overlapping blocks,

$$S[1..b],\ S[b/2+1..3b/2],\ S[b+1..2b],\ S[3b/2+1..5b/2],\ \ldots,\ S[n-b+1,n].$$

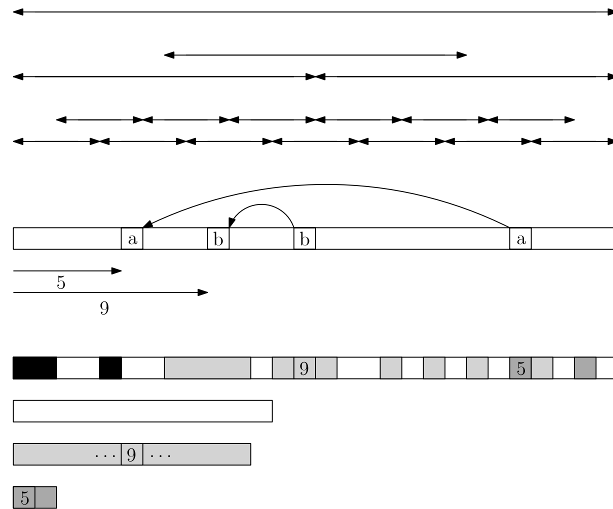


**Fig. 2.** The array $S$, with lines above indicating the overlapping block structure (with blocks of three different sizes, in this case) and arcs indicating characters' previous occurrences; our representation of the $C$ array overlaid on the string $t$, with shades of gray indicating which encoding length is used for each pointer (black for 0s); and the contents of the three wavelet trees — pointers contained in short blocks, pointers contained in medium-length blocks, and pointers contained in long blocks. Notice that, although 9 is larger than 5, the pointer with value 9 has a shorter encoding because both copies of $b$ are contained within the same medium-length block, while the two copies of $a$ are not contained in any single block except the one long block, which contains the whole string.

If $C[q] = p$ and the smallest block containing both $S[p]$ and $S[q]$ has size $b$, then we write $C[q]$ as the $\lceil \log b \rceil$-bit offset of $p$ within the lefthand block of size $b$ containing $S[q]$ (there are at most two such blocks and, if there are two, then they overlap). Since, for some $k$,

$$2^{\max\left((1+\delta)^{k-1}, k-1\right)-1} < q - p + 1 \leq b = 2^{\max\left((1+\delta)^{k}, k\right)},$$

we have $\lceil \log b \rceil < (1+\delta)\log(q-p+1) + 3$. In other words, if $S[p]$ and $S[q]$ are occurrences of a character $a$ that does not occur in $S[p+1..q-1]$, then we use fewer than $(1+\delta)\log(q-p+1)+3$ bits to store $C[q]$. By Jensen's Inequality, since the logarithm is concave, the total number of bits we use to store the offsets for occurrences of $a$ is maximized when those occurrences are evenly spaced and, thus, the space in bits is at most

$$(1+\delta)\sum_{a} \mathsf{occ}(a, S)\log\left(\frac{n}{\mathsf{occ}(a, S)} + 1\right) + 3n = (1+\delta)nH_0(S) + \mathcal{O}(n)\,,$$

where $\mathsf{occ}(a, S)$ is the number of occurrences of $a$ in $S$.

Let $t$ be a string indicating whether each entry of $C[q]$ is 0 and, if not, the block size used for it. We build a multiary wavelet tree [14] storing $t$. Notice we can always encode a block size $b = 2^{\max\left((1+\delta)^{k}, k\right)}$ in $\mathcal{O}(\log k) = \mathcal{O}(\log\log b)$ bits. By the calculations in the paragraph above and another application of Jensen's Inequality, $H_0(t) = \mathcal{O}(\log(H_0(S)+1))$. It follows that, if $H_0(S)$ grows without bound as $n$ goes to infinity, then the size of the wavelet tree for $t$ is $o(nH_0(S))$ bits; otherwise, it is $\mathcal{O}(n)$ bits. As a byproduct, using this wavelet tree, in $\mathcal{O}(1)$ time we can count all the characters whose first appearance in $S$ is in $S[i..j]$.

For each block size $b$, we build a wavelet tree storing all the $\lceil \log b \rceil$-bit encodings. By the same calculation as before, these wavelet trees take a total of $(1+\delta)nH_0(S) + \mathcal{O}(n) + o(nH_0(S))$ bits. This is illustrated in Fig. 2. Notice that, for any block size $b$, if $S[q]$ is the first occurrence of that distinct character in $S[i..j]$ and $C[q]$ is encoded in $\lceil \log b \rceil$ bits, then $q$ must be between $i$ and the end of the righthand block of size $b$ containing $i$. Using the multiary wavelet tree and the wavelet tree for block size $b$, in $\mathcal{O}(\log b)$ time we can count all such characters in the right halves of both the lefthand and the righthand blocks of size $b$ containing $S[i]$. Since these are the only blocks of size $b$ containing $S[i]$ and the right half of the lefthand block is the left half of the righthand block, the sum is the total number of such characters. That is, in $\mathcal{O}(\log b)$ time, we can count all the first occurrences $S[q]$ of distinct characters in $S[i..j]$ such that $C[q]$ is encoded in $\lceil \log b \rceil$ bits. Repeating this for each of the $\mathcal{O}(\log\log n/\log(1+\delta))$ block sizes, in $\mathcal{O}(\log n \log\log n/\log(1+\delta)) = \mathcal{O}((1/\delta)\log n\log\log n)$ time we can count the distinct characters in $S[i..j]$. Choosing $\delta = 1/\alpha(n)$, for example, where $\alpha$ is the inverse Ackermann function, yields a space bound of $(1+1/\alpha(n))nH_0(S) + \mathcal{O}(n) + o(nH_0(S)) = nH_0(S) + \mathcal{O}(n) + o(nH_0(S))$ bits and a time bound of $\mathcal{O}(\alpha(n)\log n\log\log n)$.

**Theorem 7.** *Given a string $S[1..n]$, we can build a data structure that takes $nH_0(S) + \mathcal{O}(n) + o(nH_0(S))$ bits such that later, given a substring's endpoints, in $\mathcal{O}(\alpha(n)\log n\log\log n)$ time we can count how many distinct characters it contains.*

### 4.4. Time independent of $n$

Suppose we are to count the distinct colors in $S[i..j]$, and let $\ell = j - i + 1$. Let $b_{\max}$ be the size of the smallest block in the scheme of Section 4.3 that completely contains $S[i..j]$. Using the technique described in the last paragraph of Section 4.3, we count the entries $C[q] < i$ in $C[i..j]$ that are encoded using a block size at most $b_{\max}$. Since there are $\mathcal{O}(\log\log b_{\max}/\log(1+\delta)) = \mathcal{O}(\alpha(n)\log\log(\ell+1))$ such block sizes and we need $\mathcal{O}(\log b_{\max}) = \mathcal{O}(\log \ell)$ time for each, this takes $\mathcal{O}(\alpha(n)\log\ell\log\log(\ell+1))$ time. Counting the entries encoded with bigger block sizes is made easier by the fact that, if $C[q] = p$ in $C[i..j]$ is encoded using a block size larger than $b_{\max}$, then we must have $p < i$. Therefore, any such big block entry in $C[i..j]$ indicates the first occurrence of some distinct character in $S[i..j]$. Instead of directly counting all big block entries in $C[i..j]$, we count all small block entries in $C[i..j]$ and subtract this count from $\ell$. Using the multiary wavelet tree of $t$, we can count the entries in $C[i..j]$ that are encoded with a given block size in constant time, obtaining a time bound $\mathcal{O}(\alpha(n)\log\log(\ell+1))$ for processing the big block sizes. Thus, without any modification to the data structures, we have improved the query time in Theorem 7 to $\mathcal{O}(\alpha(n)\log\ell\log\log(\ell+1))$. Since $\alpha(n)$ grows very slowly as $n$ increases, our time bound is now almost independent of $n$.

To make our time bound completely independent of $n$, we adjust our block sizes: the first block size $b_1$ is 2; for $i \geq 2$, the $k$th block size is

$$b_k = 2^{\max\left(\prod_{h=1}^{k-1}(1+1/\alpha(b_h)),k\right)}.$$

If the smallest block containing both $S[p]$ and $S[q]$ has size $b_k$ then, since

$$2^{\max\left(\prod_{h=1}^{k-2}(1+1/\alpha(b_h)),k-1\right)-1} < q-p+1 \leq 2^{\max\left(\prod_{h=1}^{k-1}(1+1/\alpha(b_h)),k\right)},$$

we have $\log(q-p+1) < \lceil \log b_k \rceil < (1 + 1/\alpha(b_{k-1}))\log(q-p+1) + 3$. Also notice that, since $b_{k-1}$ can be bounded from below in terms of $b_k$ and $b_k$ can be bounded from below in terms of $q-p$, $\alpha(b_{k-1})$ increases without bound (albeit very slowly) as $q-p$ goes to infinity. Therefore, we use fewer than $\log(q-p+1) + o(\log(q-p+1))$ bits to store $C[q]$. By calculations similar to those in Section 4.3, we still use $nH_0(S) + \mathcal{O}(n) + o(nH_0(S))$ bits in total. Now, however, since $\alpha(b_1) \leq \cdots \leq \alpha(b_k)$, more calculation shows that the number of block sizes up to $b_k$ is $\mathcal{O}(\log\log b_k/\log(1+1/\alpha(b_k)))$, from which it follows that our new time bound is $\mathcal{O}(\alpha(\ell)\log\ell\log\log(\ell+1))$.

**Theorem 8.** *Given a string $S[1..n]$, we can build a data structure that takes $nH_0(S) + \mathcal{O}(n) + o(nH_0(S))$ bits such that later, given a substring's endpoints $i$ and $j$, in $\mathcal{O}(\alpha(\ell)\log\ell\log\log(\ell+1))$ time we can count how many distinct characters it contains, where $\ell = j - i + 1$.*

### 4.5. Reducing time

We now modify the data structure so that instead of having one wavelet tree for each block size, we have a separate wavelet tree for each block. If $C[q] = p$ is encoded using a block size $b$ then one or two blocks of size $b$ contain both $p$ and $q$, and we store the encoding in the wavelet tree of the leftmost block. Notice that $q$ is always in the second half of the block. The total number of bits in the encodings does not change.

A standard wavelet tree implementation technique is to represent each level of a wavelet tree with a single bitvector, which is the concatenation of the bitvectors for individual nodes over that level [14,39]. Here we can similarly use a single bitvector to represent a level over *all* wavelet trees for a given block size. As in the standard case, given the location of the bitvector for a node, we can easily locate the bitvectors for the children. For each block size $b_k$, we provide two additional bitvectors to directly locate nodes, one for the root level and one for the level at height $k$ (where leaves have height 0). The size of such a bitvector is $n_k + v + o(n_k + v)$, where $n_k$ is the length of the level bitvector, which equals the number of entries encoded with block size $b_k$, and $v$ is the number of nodes on the given level. Since $v = \mathcal{O}\left(n/2^k\right)$ for height $k$ and is less or equal for the root level, the size of the locating bitvectors for block size $b_k$ is $\mathcal{O}\left(n_k + n/2^k\right)$, which is $\mathcal{O}(n)$ over all block sizes.

When counting the number of distinct colors in $S[i..j]$, we handle block sizes larger than $b_{\max}$ as before using the multiary wavelet tree for $t$ in $\mathcal{O}(\alpha(\ell)\log\log(\ell+1))$ time. For each block size $b \leq b_{\max}$, we need to query the wavelet trees for the two blocks that contain $i$. For block size $b_{\max}$ we do this as before in $\mathcal{O}(\log b_{\max}) = \mathcal{O}(\log\ell)$ time. Block sizes smaller than $b_{\max}$ are handled differently.

If $B$ is a block of size $b_k < b_{\max}$ that contains $i$, it does not contain $j$. If $C[q] = p$ is stored in the wavelet tree for $B$, then $q < j$. We want to count an entry $C[q] = p$ in $B$ if (i) $q \geq i$ and (ii) $p < i$. Since $p < q$, both conditions cannot be violated simultaneously. Thus we count entries that violate (i) and entries that violate (ii) and subtract the sum from the total number of entries for block $B$. Notice that this does not work for larger block sizes because we need an additional condition $q \leq j$. Using the multiary wavelet tree for $t$ we can count in constant time all entries $C[q] = p$ that are encoded using block size $b_k$ and have $q$ in a given range. This is sufficient to count all entries in $B$ as well as those that violate (i). Counting (ii) can be done by locating the leaf that represents the position $i$ in the wavelet tree for block $B$, so that all the positions to the right at the last level of the wavelet tree for $B$ are those that violate (ii). We locate the leaf by locating its ancestor at height $k$ in constant time and traversing down in $\mathcal{O}(k)$ time. Thus block size $b_k$ can be processed in $\mathcal{O}(k)$ time and all block sizes smaller than $b_{\max}$ in $\mathcal{O}\left((\alpha(\ell)\log\log(\ell+1))^2\right)$ time.

**Theorem 9.** *Given a string $S[1..n]$, we can build a data structure that takes $nH_0(S) + \mathcal{O}(n) + o(nH_0(S))$ bits such that later, given a substring's endpoints $i$ and $j$, in $\mathcal{O}(\log\ell)$ time we can count how many distinct characters it contains, where $\ell = j - i + 1$.*

### 4.6. Dynamism

Suppose we want to append a character $S[n+1]$ to $S$. To maintain $C$, we must append $C[n+1] = p$ to it, where $p$ is the position of the last occurrence of $S[n+1]$ in $S[1..n]$, or 0 if there is no such occurrence. We maintain a separate data structure of $\sigma \log n$ bits to find the last occurrence of any character in $\mathcal{O}(\log \sigma)$ time. We will describe how to append $C[n+1]$ to our representation of $C$ stored in the data structure we gave in Section 4.3 (as appending it to the data structure from Section 4.2 is similar and simpler).

Our first concern is to append to the string $t$ a character indicating whether $p$ is 0 and, if not, the block size used for it. Instead of storing $t$ with a multiary wavelet tree, we now store it with a Huffman-shaped binary wavelet tree [38], with the bitvectors at the internal nodes stored separately from each other (i.e., not concatenated, as would be usual). As long as these bitvectors are each stored with at most linear redundancy, they take a total of at most $\mathcal{O}(n(H_0(t)+1)) \subseteq \mathcal{O}(n \log(H_0(S)+1)+n)$ bits. Also, since $t$ is over an alphabet of size $\mathcal{O}(\log\log n/\log(1+\delta))$, which is $\mathcal{O}\left((\log\log n)^2\right)$ with our choice of $\delta = 1/\alpha(n)$, we can store the shape of the tree using $\mathcal{O}(\log n)$-bit pointers at each internal node without increasing our overall space bound.

To append a character to $t$, we append a bit to each bitvector on the path from the root of the wavelet tree to the leaf labeled with the character we append (we create this leaf if it does not already exist). Each bit indicates whether the next node on the path is the current node's left child or its right child. Many implementations of bitvectors are based on breaking them into blocks (see, e.g., [40] for more discussion) and, thus, make appending relatively easy. Since we allow ourselves linear redundancy, whenever a bitvector outgrows the space allocated to it, we double that space; we use background processing to copy the bitvector into its new location, so that our time bounds are still worst-case. Appending to $t$ takes a total of $\mathcal{O}(\log\log\log n)$ time.

Our other concern is to append $C[n+1]$ to a sequence of values encoded with the same block size $b$, all of which are stored in a wavelet tree. We use essentially the same approach as when appending a character to $t$. One complication is that the sequence of values is no longer guaranteed to be over a small alphabet, so it is not immediately clear how we can use $\mathcal{O}(\log n)$-bit pointers at the internal nodes. If $b$ is small, at most $n/\log n$, then, as with $t$, there is no problem: calculation shows that using pointers in all the wavelet trees for small block sizes increases our space bound by at most $\mathcal{O}(n)$. For the case when $b > n/\log n$, we replace the standard trie shape of wavelet trees with a Patricia trie shape. From the standard wavelet tree, we remove all nodes associated with an empty sequence. If any remaining node has exactly one child, the associated bitvector is all 0s or all 1s and can be encoded with a single bit stored in the closest existing descendant of the node. The resulting wavelet tree shape is a Patricia trie [45], where the number of internal nodes is less than the number of leaves, which is equal to the number of distinct values in the sequence and, thus, at most the length of the sequence. Recall that, if we use $\log b$ bits for each value stored in the wavelet tree for block size $b$, for every $b$, then we use a total of $(1+\delta)nH_0(S)+\mathcal{O}(n)+o(nH_0(S))$ bits. Therefore, if we use $\mathcal{O}(\log n) \subseteq \mathcal{O}(\log b)$ bits for pointers at each internal node, then we use $\mathcal{O}(n(H_0(S)+1))$ bits altogether. Appending to the sequence stored in a wavelet tree for a block size takes $\mathcal{O}(\log n)$ time.

**Theorem 10.** *We can modify the data structure from Theorem 7 such that we achieve the same time bound for querying and a space bound of $\mathcal{O}(n(H_0(S)+1)+\sigma\log n)$ bits while supporting an $\mathcal{O}(\log n)$-time append operation.*

If we modify the data structure from Section 4.3 by replacing all the wavelet trees (including the multiary wavelet tree) with dynamic wavelet trees [29], which support queries, insertions and deletions in $\mathcal{O}\left(\log^2 n/\log\log n\right)$ time,[5] we still use $nH_0(S)+\mathcal{O}(n)+o(nH_0(S))$ bits, but $\mathcal{O}\left(\log^2 n\right)$ time for queries and appends. This data structure can also support color substitutions and deletions in $\mathcal{O}\left(\log^2 n\right)$ time. In order to replace a character $S[q] = a$ by $a'$, we find the last occurrences $S[p]$ and $S[p']$ of $a$ and $a'$ strictly before $S[q]$, and the first occurrences $S[r]$ and $S[r']$ of $a$ and $a'$ strictly after $S[q]$. We update $C$ such that $C[q] = p'$, $C[r] = p$ and $C[r'] = q$, again using $\mathcal{O}\left(\log^2 n\right)$ time.

To quickly find the preceding and succeeding occurrence of a character, we maintain a sampled version $S'$ of $S$ that contains approximately every $(\log\log n)$th occurrence of each distinct character. More precisely, we maintain the invariant that between every sampled occurrence of a character $a$, there is between $\log\log n$ and $2\log\log n$ unsampled occurrences of $a$. Also, the last occurrence of each character is in the sample. We store a dynamic bit vector $F[1, n]$ to mark the sampled positions and a dynamic wavelet tree for $S'$. Since the distribution of symbols is approximately the same in $S$ and $S'$, the space we need is $o(nH_0(S))+\mathcal{O}(n)+\sigma\log n$ bits. We can now find the preceding and succeeding occurrence of a character using a constant number of rank and select queries on $F$ and $S'$ and $\mathcal{O}(\log\log n)$ accesses to $C$. This takes $\mathcal{O}\left(\log^2 n\right)$ time. The same time is also sufficient to modify $S'$ when necessary to maintain the invariants.

To delete a character from $S$, we replace it with a special null character not in the alphabet (which we search for and exclude when performing queries). If $S_d$ is a string $S$ with $d$ extra null characters added, then $(n+d)H_0(S_d)-nH_0(S) \le n+d$. We also maintain a background process that keeps removing null characters. In one $\mathcal{O}\left(\log^2 n\right)$ time step of the process, we move a run of consecutive null characters one step to the right. The background process works in phases. In the beginning of a phase, we find the leftmost null character and start moving it to the right. Any null characters encountered will join

[5] Multiary wavelet trees achieve $\mathcal{O}\left((\log n/\log\log n)^2\right)$ time, but for $C$ we need binary wavelet trees.

the moving group. The phase ends after at most $n$ steps, when the group reaches the end and can be easily removed. For every new deletion, we perform two steps of the background process. This ensures that, if at most half of characters are null characters in the beginning of a phase, the same is true at the end. Thus the number of null characters and the extra bits needed to store them remains $\mathcal{O}(n)$.

**Theorem 11.** *We can modify the data structure from Theorem 7 such that it takes $nH_0(S) + \mathcal{O}(n) + o(nH_0(S)) + \sigma \log n$ bits, and supports queries, appends, color substitutions and deletions in $\mathcal{O}(\log^2 n)$ time.*

We note that our sampled string $S'$, together with $F$ and $C$, indeed replace the original sequence $S$, in the sense that any symbol $S[i]$ can be obtained from $S'$ and $C$ in time $\mathcal{O}(\log^2 n)$, as follows. First check if $F[i] = 1$; if so then $S[i] = S'[rank_1(F, i)]$. Else, do $i \leftarrow C[i]$, which sends us to the previous occurrence in $S$ of the (yet unknown) symbol $c = S[i]$, and iterate. Due to our sampling invariants, after $\mathcal{O}(\log \log n)$ steps we will find a sampled position $i$ such that $F[i] = 1$ (actually we need to make sure that the first occurrence of each symbol is sampled, which adds $\sigma \log n$ bits). Note this technique applies also in the static case, where also with just $\sigma \log n$ extra bits we can obtain any $S[i]$ from our representation, in time $\mathcal{O}(\log n \log \log n)$.

## 5. Concluding remarks

We have presented new and efficient solutions for three natural colored range queries: colored range listing, colored range top-$k$ queries, and colored range counting. Our solutions for colored range listing lead to the fastest compressed data structures for that problem and for document listing; our (approximate) solution for colored range top-$k$ queries is, as far as we know, the first efficient data structure for that problem; and our solution for colored range counting reduces the space bound from $\mathcal{O}(n \log n)$ bits to $nH_0(S)+\mathcal{O}(n)+o(nH_0(S))$ bits while simultaneously improving query time to $\mathcal{O}(\log \ell)$, where $\ell$ is the size of the query range. Although our solutions for general colored range top-$k$ queries and colored range counting do not give improved bounds for the corresponding document retrieval problems, our more general data structures may find applications to other information retrieval scenarios beyond ranges induced by searching for exact patterns in suffix trees or arrays.

A simple example of natural queries not fitting in the restricted model are lexicographic range queries. Imagine we look for patterns lexicographically in the range [“1969”,“2010”] in documents; the result is a suffix array range that does not correspond to any suffix tree node. In this case, existing techniques for document retrieval based on suffix tree properties (such as for computing top-$k$ queries [32] and for computing document frequencies [55]) will not work. The general techniques we have introduced in this article do.

Yet another scenario that is not captured by the suffix tree model is inverted indices for natural language text (as opposed to the general texts addressed in this paper) [3]. Consider that we store the list of documents where each vocabulary word appears, consecutively according to the order of the words in the vocabulary. If queries are simple words, then all the document retrieval problems we have considered are easily solved by storing the documents of each list ordered by decreasing term frequency. Yet, imagine we wish to provide *also* the same functionality on stemmed searching, upon user request at query time. One solution is to group together the vocabulary words sharing the same stem so that, while individual word queries can be handled as usual, stemmed queries are handled by considering the concatenation of the lists of the words sharing the same stem. Then we can regard the concatenation of all inverted lists as the array $E$ and use the general techniques developed in this paper to answer various document queries on stems: Document listing and counting algorithms apply verbatim, while those involving frequencies pose further challenges as each entry in the inverted lists is weighted by the term frequency of the word in the document. Other query operations, from case folding to thesauri expansion, can also be reduced to a proper grouping of lists.

Finally, there are information retrieval scenarios completely different from the text search framework. For example, colored range queries seem a natural tool for query mining [2], where logs of queries posed to search engines are recorded over periods of time, and then analyzed to discover trends in user behavior. By considering that each different query is a color, we can find the most popular queries or the number of distinct queries within any given time period; by considering each visitor as a color, we can find the number of unique visitors within any given time period. There are many other potential queries of interest, which could in turn become new challenging colored range queries.

### 5.1. Postscript

Document retrieval is an active research topic, as demonstrated by some very recent publications improving (and in some cases building upon) our results. Apart from the results described in Section 2, Belazzougui and Navarro [6] built on our work in Section 3.1, and improved our time bounds for the document top-$k$ problem to $\mathcal{O}(\mathsf{search}(m) + k \log k \log(D/k) \log^{\epsilon} n)$ while keeping the same space bounds, and also gave a solution that takes $\mathcal{O}(n \log \log \log D)$ extra bits and answers queries in $\mathcal{O}(\mathsf{search}(m) + k\,\mathsf{lookup}(n) \log k \log^{1+\epsilon} n)$ time. Hon et al. [31] and Navarro and Nekrich [49] also gave solutions for the document top-$k$ problem that use more space than ours but answer queries faster. Hon et al.’s first solution takes $2n \log D + o(n \log D)$ extra bits on top of a CSA and answers queries in $\mathcal{O}(\mathsf{search}(m) + k \log k)$ time; their second solution takes $n \log D + o(n \log D)$ extra bits and answers queries in $\mathcal{O}(\mathsf{search}(m) + k(\log k + (\log \log n)^{2+\epsilon}))$ time. Navarro and Nekrich’s solution takes a total of $\mathcal{O}(n(\log \sigma + \log D + \log \log n))$ bits, where $\sigma$ is the size of the alphabet of the documents,

and answers queries in optimal $\mathcal{O}(m+k)$ time. On the practical side, Navarro et al. [50] implemented the idea in our Theorem 3 and showed it was competitive in practice, achieving significantly less space than the alternative solutions.

## Acknowledgments

Many thanks to Djamal Belazzougui, Veli Mäkinen, Giovanni Manzini and Jorma Tarhio, for helpful discussions, and the referees of the earlier versions of this paper, for helpful comments.

## References

[1] A. Apostolico, The myriad virtues of subword trees, in: Combinatorial Algorithms on Words, Springer-Verlag, 1985, pp. 85–96.
[2] R. Baeza-Yates, Applications of web query mining, in: Proceedings of the 27th European Conference on IR Research, Springer, 2005, pp. 7–22.
[3] R. Baeza-Yates, B. Ribeiro, Modern Information Retrieval, Addison-Wesley, 1999.
[4] J. Barbay, T. Gagie, G. Navarro, Y. Nekrich, Alphabet partitioning for compressed rank/select with applications, in: Proceedings of the 21st International Symposium on Algorithms and Computations, Springer, 2010, pp. 315–326.
[5] J. Barbay, M. He, J.I. Munro, S.S. Rao, Succinct indexes for strings, binary relations and multi-labelled trees, in: Proceedings of the 18th Symposium on Discrete Algorithms, SIAM, 2007, pp. 680–689.
[6] D. Belazzougui, G. Navarro, Improved compressed indexes for full-text document retrieval, in: Proceedings of the 18th Symposium on String Processing and Information Retrieval, Springer, 2011, pp. 386–397.
[7] D. Belazzougui, G. Navarro, New lower and upper bounds for representing sequences, in: Proceedings of the 20th Annual European Symposium on Algorithms, in: LNCS, vol. 7501, Springer, 2012, pp. 181–192.
[8] P. Bille, G.M. Landau, R. Raman, K. Sadakane, S.R. Satti, O. Weimann, Random access to grammar-compressed strings, in: Proceedings of the 22nd Symposium on Discrete Algorithms, SIAM, 2011, pp. 373–389.
[9] P. Bozanis, N. Kitsios, C. Makris, A.K. Tsakalidis, New upper bounds for generalized intersection searching problems, in: Proceedings of the 22nd International Colloquium on Algorithms, Languages and Programming, Springer, 1995, pp. 464–474.
[10] G.S. Brodal, B. Gfeller, A.G. Jørgensen, P. Sanders, Towards optimal range medians, Theoretical Computer Science 412 (2011) 2588–2601.
[11] S. Carlsson, J.I. Munro, P.V. Poblete, An implicit binomial queue with constant insertion time, in: Proceedings of the 1st Scandinavian Workshop on Algorithm Theory, Springer, 1988, pp. 1–13.
[12] T. Chan, S. Durocher, K.G. Larsen, J. Morrison, B.T. Wilkinson, Linear-space data structures for range mode query in arrays, in: Proceedings of the 29th Symposium on Theoretical Aspects of Computer Science, in: Leibnitz Zentrum für Informatik, 2012, pp. 290–301.
[13] J.S. Culpepper, G. Navarro, S.J. Puglisi, A. Turpin, Top-$k$ ranked document search in general text databases, in: Proceedings of the 18th European Symposium on Algorithms, Springer, 2010, pp. 194–205.
[14] P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro, Compressed representations of sequences and full-text indexes, ACM Transactions on Algorithms 3 (2007) article 20.
[15] P. Ferragina, R. Venturini, A simple storage scheme for strings achieving entropy bounds, Theoretical Computer Science 371 (2007) 115–121.
[16] J. Fischer, Optimal succinctness for range minimum queries, in: Proceedings of the 9th Latin American Symposium on Theoretical Informatics, Springer, 2010, pp. 158–169.
[17] J. Fischer, V. Heun, A new succinct representation of RMQ-information and improvements in the enhanced suffix array, in: Proceedings of the 1st Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, Springer, 2007, pp. 459–470.
[18] H.N. Gabow, J.L. Bentley, R.E. Tarjan, Scaling and related techniques for geometry problems, in: Proceedings of the 16th Symposium on Theory of Computing, ACM, 1984, pp. 135–143.
[19] T. Gagie, J. Kärkkäinen, Counting colours in compressed strings, in: Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching, Springer, 2011, pp. 197–207.
[20] T. Gagie, G. Navarro, S.J. Puglisi, Colored range queries and document retrieval, in: Proceedings of the 17th Symposium on String Processing and Information Retrieval, Springer, 2010, pp. 67–81.
[21] T. Gagie, G. Navarro, S.J. Puglisi, New algorithms on wavelet trees and applications to information retrieval, Theoretical Computer Science 426–427 (2012) 25–41.
[22] T. Gagie, S.J. Puglisi, A. Turpin, Range quantile queries: Another virtue of wavelet trees, in: Proceedings of the 16th Symposium on String Processing and Information Retrieval, Springer, 2009, pp. 1–6.
[23] A. Golynski, Optimal lower bounds for rank and select indexes, Theoretical Computer Science 387 (2007) 348–359.
[24] A. Golynski, R. Raman, S. Rao, On the redundancy of succinct data structures, in: Proceedings of the 11th Scandinavian Workshop on Algorithm Theory, Springer, 2008, pp. 148–159.
[25] R. González, G. Navarro, Compressed text indexes with fast locate, in: Proceedings of the 18th Symposium on Combinatorial Pattern Matching, Springer, 2007, pp. 216–227.
[26] M. Greve, A.G. Jørgensen, K.D. Larsen, J. Truelsen, Cell probe lower bounds and approximations for range mode, in: Proceedings of the 37th International Colloquium on Algorithms, Languages and Programming, Springer, 2010, pp. 605–616.
[27] R. Grossi, A. Gupta, J.S. Vitter, High-order entropy-compressed text indexes, in: Proceedings of the 14th Symposium on Discrete Algorithms, SIAM, 2003, pp. 636–645.
[28] R. Grossi, A. Orlandi, R. Raman, Optimal trade-offs for succinct string indexes, in: Proceedings of the 37th International Colloquium on Algorithms, Languages and Programming, Springer, 2010, pp. 678–689.
[29] M. He, I. Munro, Succinct representations of dynamic strings, in: Proceedings of the 17th International Symposium on String Processing and Information Retrieval, Springer, 2010, pp. 334–346.
[30] W. Hon, R. Shah, S. Wu, Efficient index for retrieving top-$k$ most frequent documents, in: Proceedings of the 16th Symposium on String Processing and Information Retrieval, Springer, 2009, pp. 182–193.
[31] W.K. Hon, R. Shah, S.V. Thankachan, Towards an optimal space-and-query-time index for top-$k$ document retrieval, in: Proceedings of the 23rd Symposium on Combinatorial Pattern Matching, Springer, 2012, pp. 173–184.
[32] W.K. Hon, R. Shah, J. Vitter, Space-efficient framework for top-$k$ string retrieval problems, in: Proceedings of the 50th Symposium on Foundations of Computer Science, IEEE, 2009, pp. 713–722.
[33] I.F. Ilyas, G. Beskales, M.A. Soliman, A survey of top-$K$ query processing techniques in relational database systems, ACM Computing Surveys 40 (2008).
[34] R. Janardan, M.A. Lopez, Generalized intersection searching problems, International Journal of Computational Geometry and Applications 3 (1993) 39–69.
[35] H. Kaplan, N. Rubin, M. Sharir, E. Verbin, Efficient colored orthogonal range counting, SIAM Journal on Computing 38 (2008) 982–1011.
[36] M. Karpinski, Y. Nekrich, Top-$K$ color queries for document retrieval, in: Proceedings of the 22nd Symposium on Discrete Algorithms, SIAM, 2011, pp. 401–411.
[37] Y.K. Lai, C.K. Poon, B. Shi, Approximate colored range and point enclosure queries, Journal of Discrete Algorithms 6 (2008) 420–432.
[38] V. Mäkinen, G. Navarro, Succinct suffix arrays based on run-length encoding, Nordic Journal of Computing 12 (2005) 40–66.

[39] V. Mäkinen, G. Navarro, Implicit compression boosting with applications to self-indexing, in: Proceedings of the 14th Symposium on String Processing and Information Retrieval, Springer, 2007, pp. 229–241.
[40] V. Mäkinen, G. Navarro, Rank and select revisited and extended, Theoretical Computer Science 387 (2007) 332–347.
[41] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, SIAM Journal on Computing 22 (1993) 935–948.
[42] G. Manzini, An analysis of the Burrows-Wheeler transform, Journal of the ACM 48 (2001) 407–430.
[43] Y. Matias, S. Muthukrishnan, S.C. Sahinalp, J. Ziv, Augmenting suffix trees, with applications, in: Proceedings of the 6th European Symposium on Algorithms, Springer, 1998, pp. 67–78.
[44] R.L. Milidiú, E.S. Laber, Bounding the inefficiency of length-restricted prefix codes, Algorithmica 31 (2001) 513–529.
[45] D.R. Morrison, PATRICIA — practical algorithm to retrieve information coded in alphanumeric, Journal of the ACM 15 (1968).
[46] I. Munro, Tables, in: Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science, Springer, 1996, pp. 37–42.
[47] S. Muthukrishnan, Efficient algorithms for document retrieval problems, in: Proceedings of the 13th Symposium on Discrete Algorithms, SIAM, 2002, pp. 657–666.
[48] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Computing Surveys 39 (2007).
[49] G. Navarro, Y. Nekrich, Top-$k$ document retrieval in optimal time and linear space, in: Proceedings of the 22nd Symposium on Discrete Algorithms, SIAM, 2012, pp. 1066–1077.
[50] G. Navarro, S.J. Puglisi, D. Valenzuela, Practical compressed document retrieval, in: Proceedings of the 10th International Symposium on Experimental Algorithms, Springer, 2011, pp. 193–205.
[51] D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, in: Proceedings of the Workshop on Algorithm Engineering and Experiments, SIAM, 2007.
[52] H. Petersen, S. Grabowski, Range mode and range median queries in constant time and sub-quadratic space, Information Processing Letters 109 (2009) 225–228.
[53] R. Raman, V. Raman, S. Rao, Succinct indexable dictionaries with applications to encoding $k$-ary trees and multisets, in: Proceedings of the 13th Symposium on Discrete Algorithms, SIAM, 2002, pp. 233–242.
[54] K. Sadakane, New text indexing functionalities of the compressed suffix arrays, Journal of Algorithms 48 (2003) 294–313.
[55] K. Sadakane, Succinct data structures for flexible text retrieval systems, Journal of Discrete Algorithms 5 (2007) 12–22.
[56] K. Sadakane, G. Navarro, Fully-functional succinct trees, in: Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2010, pp. 134–149.
[57] N. Välimäki, V. Mäkinen, Space-efficient algorithms for document retrieval, in: Proceedings of the 18th Symposium on Combinatorial Pattern Matching, Springer, 2007, pp. 205–215.
[58] P. Weiner, Linear pattern matching algorithm, in: Proceedings of the 14th IEEE Symposium on Switching and Automata Theory, IEEE, 1973, pp. 1–11.