Theoretical Computer Science 542 (2014) 83–97

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

New space/time tradeoffs for top-k document retrieval on sequences ☆

Gonzalo Navarro [a,*], Sharma V. Thankachan [b]

[a] Department of Computer Science, University of Chile, Chile
[b] Department of Computer Science, University of Waterloo, Canada

ARTICLE INFO

Article history:
Received 29 November 2013
Received in revised form 21 March 2014
Accepted 10 May 2014
Communicated by R. Giancarlo

Keywords:
Document retrieval
Top-k queries
String databases
Compressed data structures

ABSTRACT

We address the problem of indexing a collection $\mathcal{D} = \{T_1, T_2, \ldots, T_D\}$ of D string documents of total length n, so that we can efficiently answer *top-k queries*: retrieve k documents most relevant to a pattern P of length p given at query time. There exist linear-space data structures, that is, using $O(n)$ words, that answer such queries in optimal $O(p + k)$ time for an ample set of notions of relevance. However, using linear space is not sufficiently good for large text collections. In this paper we explore how far the space/time tradeoff for this problem can be pushed. We obtain three results: (1) When relevance is measured as term frequency (number of times P appears in a document T_i), an index occupying $|\mathsf{CSA}| + o(n)$ bits answers the query in time $O(t_{\mathsf{search}}(p) + k \lg^2 k \lg^{\varepsilon} n)$, where CSA is a compressed suffix array indexing $\mathcal{D}$, $t_{\mathsf{search}}(p)$ is its time to find the suffix array interval of P, and $\varepsilon > 0$ is any constant. (2) With the same measure of relevance, an index occupying $|\mathsf{CSA}| + n \lg D + o(n \lg \sigma + n \lg D)$ bits answers the query in time $O(t_{\mathsf{search}}(p) + k \lg^* k)$, where $\lg^* k$ is the iterated logarithm of k. (3) When the relevance depends only on the documents, an index occupying $|\mathsf{CSA}| + O(n \lg \lg n)$ bits answers the query in $O(t_{\mathsf{search}}(p) + k t_{\mathsf{SA}})$ time, where t_{SA} is the time the CSA needs to retrieve a suffix array cell. On our way, we obtain some other results of independent interest.

1. Introduction

Top-k document retrieval is the problem of preprocessing a text collection $\mathcal{D} = \{T_1, T_2, \ldots, T_D\}$ of D string documents of total length n so that, given a search pattern $P[1..p]$ and a threshold k, we retrieve the k documents most "relevant" to P, for some definition of relevance. This is the basic problem of search engines and forms the core of the Information Retrieval (IR) field [1]. A widely used notion of relevance is the tf–idf model, which in the case of a single pattern P boils down to the *term frequency* measure, that is, the number of times P appears in a document. Another popular scenario is to give each document a fixed *importance* independent of P, such as Google's PageRank.

The inverted index successfully solves top-k queries in various IR scenarios. However, it applies to text collections that can be segmented into "words", so that only whole words can be queried. This excludes East Asian languages such as Chinese and Korean, where automatic segmenting is an open problem, and is troublesome even in languages such as German and Finnish. A simple solution for those cases is to treat the text as a plain sequence of symbols and look for any substring in those sequences. This string model is also appealing in applications like bioinformatics, software repositories, multimedia

☆ Early parts of this work appeared in SPIRE 2013 and ISAAC 2013. Work supported by Fondecyt Grant 1-110066, Chile.
* Corresponding author.
E-mail addresses: gnavarro@dcc.uchile.cl (G. Navarro), thanks@uwaterloo.ca (S.V. Thankachan).

http://dx.doi.org/10.1016/j.tcs.2014.05.005
0304-3975/

Table 1
Most relevant space/time tradeoffs achieved for top-k most frequent document retrieval. Note that $D\lg(n/D)+O(D)=o(n)$ if $D=o(n)$. The rows prefixed with a '*' are those not dominated by others after this work.

	Source	Time	Space
	[7]	$O(p+k\lg k)$	$O(n\lg n)$
*	[8]	$O(p+k)$	$O(n\lg D+n\lg\sigma)$
	[14]	$O(t_{\mathsf{search}}(p)+k\lg D\lg(D/k)\lg^{\varepsilon} n)$	$\|\mathsf{CSA}\|+n\lg D+o(n\lg D)$
	[15]	$O(t_{\mathsf{search}}(p)+k\lg k\lg(D/k)\lg^{\varepsilon} n)$	$\|\mathsf{CSA}\|+n\lg D+o(n\lg D)$
	From [16]	$O(t_{\mathsf{search}}(p)+k\lg k\lg^{\varepsilon} n)$	$\|\mathsf{CSA}\|+n\lg D+o(n\lg D)$
	[17]	$O(t_{\mathsf{search}}(p)+(\lg\lg n)^4+k\lg\lg n)$	$\|\mathsf{CSA}\|+2n\lg D+o(n\lg D)$
	[17]	$O(t_{\mathsf{search}}(p)+(\lg\lg n)^6+k(\lg\sigma\lg\lg n)^{1+\varepsilon})$	$\|\mathsf{CSA}\|+n\lg D+o(n\lg D)$
*	This paper	$O(t_{\mathsf{search}}(p)+k\lg^* n)$	$\|\mathsf{CSA}\|+n\lg D+o(n\lg D)$
*	This paper	$O(t_{\mathsf{search}}(p)+k\lg^* k)$	$\|\mathsf{CSA}\|+n\lg D+o(n\lg D+n\lg\sigma)$
	[14]	$O(t_{\mathsf{search}}(p)+kt_{\mathsf{SA}}\lg D\lg(D/k)\lg^{\varepsilon} n)$	$\|\mathsf{CSA}\|+O(n\lg D/\lg\lg D)$
	[15]	$O(t_{\mathsf{search}}(p)+kt_{\mathsf{SA}}\lg k\lg(D/k)\lg^{\varepsilon} n)$	$\|\mathsf{CSA}\|+O(n\lg D/\lg\lg D)$
	[15]	$O(t_{\mathsf{search}}(p)+kt_{\mathsf{SA}}\lg k\lg^{1+\varepsilon} n)$	$\|\mathsf{CSA}\|+O(n\lg\lg\lg D)$
	[7]	$O(t_{\mathsf{search}}(p)+kt_{\mathsf{SA}}\lg^{3+\varepsilon} n)$	$2\|\mathsf{CSA}\|+D\lg(n/D)+O(D)+o(n)$
	[14]	$O(t_{\mathsf{search}}(p)+kt_{\mathsf{SA}}\lg D\lg(D/k)\lg^{1+\varepsilon} n)$	$2\|\mathsf{CSA}\|+D\lg(n/D)+O(D)+o(n)$
	[15]	$O(t_{\mathsf{search}}(p)+kt_{\mathsf{SA}}\lg k\lg(D/k)\lg^{\varepsilon} n)$	$2\|\mathsf{CSA}\|+D\lg(n/D)+O(D)+o(n)$
*	[16]	$O(t_{\mathsf{search}}(p)+kt_{\mathsf{SA}}\lg k\lg^{\varepsilon} n)$	$2\|\mathsf{CSA}\|+D\lg(n/D)+O(D)+o(n)$
	[18]	$O(t_{\mathsf{search}}(p)+kt_{\mathsf{SA}}\lg k\lg^{1+\varepsilon} n)$	$\|\mathsf{CSA}\|+D\lg(n/D)+O(D)+o(n)$
*	This paper	$O(t_{\mathsf{search}}(p)+kt_{\mathsf{SA}}\lg^2 k\lg^{\varepsilon} n)$	$\|\mathsf{CSA}\|+D\lg(n/D)+O(D)+o(n)$

sequences, activity logs, and many others [2]. Supporting document retrieval on those general string collections has proved much more challenging.

Suffix trees [3] and suffix arrays [4] are useful data structures to search general string collections. These structures solve the *pattern matching problem*, that is, count or list all the occ individual occurrences of P in the collection. Suffix trees can count the number of occurrences of P in optimal time $t_{\mathsf{search}}(p)=O(p)$, whereas suffix arrays need time $t_{\mathsf{search}}(p)=O(p\lg n)$, or $t_{\mathsf{search}}(p)=O(p+\lg n)$ with some enhancements. After counting them, the occurrences can be listed in optimal $O(\mathrm{occ})$ additional time. The k most relevant documents could then be obtained from that set, but this would require time $\Omega(\mathrm{occ})$, which can be much larger than k.

Top-k most frequent document retrieval Only recently [5–9] was this top-k problem solved satisfactorily, finally reaching the optimal time $O(p+k)$. Those solutions, much like suffix trees and arrays, have the drawback of requiring $O(n\lg n)$ bits of space, whereas the collection itself would require no more than $n\lg\sigma$ bits, where σ is the alphabet size. This renders these indexes impractical on large text collections.

Compressed Suffix Arrays (CSAs) [10] satisfactorily solve the pattern matching problem within the size of the *compressed* text collection, under some entropy model such as the hth order empirical entropy $nH_h\le n\lg\sigma$ [11]. They can, in addition, retrieve any substring of any document, and hence they replace the collection with a compressed version that in addition supports queries. We call their space $|\mathsf{CSA}|$. Some CSAs [12] reach space $|\mathsf{CSA}|=nH_h+o(nH_h)+o(n)$, which under the hth order entropy model is asymptotically the minimum space in which the text collection itself can be represented, and reach counting time $t_{\mathsf{search}}(p)=O(p\lg\lg\sigma)$. Their time to retrieve any suffix array position is $t_{\mathsf{SA}}=O(\lg^{1+\varepsilon} n)$ for any constant $\varepsilon>0$, so they list the occ occurrences of P in time $O(\mathrm{occ}\,t_{\mathsf{SA}})$. Others [13] achieve the optimal counting time $t_{\mathsf{search}}(p)=O(p)$ using slightly more space, $|\mathsf{CSA}|=nH_h+o(nH_h)+O(n)$ bits, and reach $t_{\mathsf{SA}}=O(\lg n)$ (so listing takes $O(\mathrm{occ}\lg n)$ time).

Similar compressed solutions for top-k queries have been sought, but the results have not been so clean. In the discussion that follows we will assume for simplicity that $D=o(n)$, $t_{\mathsf{search}}(p)=p$, and $t_{\mathsf{SA}}=\lg^{1+\varepsilon} n$; the full results can be seen in Table 1. In their seminal paper [7], Hon et al. gave the first top-k indexes using compressed space. They used $2|\mathsf{CSA}|+o(n)$ bits and $O(p+k\lg^{4+\varepsilon} n)$ time. Gagie et al. [14] and Belazzougui et al. [15] made several technical improvements on top of the basic idea, pushing the time down to $O(p+k\lg k\lg^{2+\varepsilon} n)$. Finally, Hon et al. [16] reduced the time to the current best within this space, $O(p+k\lg k\lg^{1+\varepsilon} n)$. This result, however, is not fully satisfactory in terms of space, as it uses $2|\mathsf{CSA}|$ bits,[1] that is, about twice the minimum necessary space. Tsur [18] managed to reduce the space to the asymptotically optimal $|\mathsf{CSA}|+o(n)$ bits, and solved top-k queries within time $O(p+k\lg k\lg^{2+\varepsilon} n)$. This is only a $\lg n$ factor away from the best result achieved with $2|\mathsf{CSA}|$ bits [16].

The time in all those solutions involves at least accessing CSA cells to return the results, and thus they have a time component of the form $\Omega(kt_{\mathsf{SA}})$. The optimal-time solution [8] reaches $O(p+k)$ time, but it uses $O(n(\lg D+\lg\sigma))$ bits. There has been some research about reducing that space while retaining fast query times. The idea is to use a so-called

[1] In fact, it is $|\mathsf{CSA}|$ bits for a global CSA, plus $|\mathsf{CSA}_d|$ bits for the local CSA of each document T_d, which depending on the CSA used could be more, the same, or less than $2|\mathsf{CSA}|$.

Table 2
Most relevant space/time tradeoffs achieved for top-k most important document retrieval. Note that $D\lg(n/D)+O(D)=o(n)$ if $D=o(n)$. Our solutions marked (*) are valid (and relevant) only for $D=O(\text{polylog}\, n)$.

Source	Time	Space
[26]	$O(p+k)$	$O(n\lg n)$
[26]	$O(t_{\mathsf{search}}(p)+k)$	$\|\mathsf{CSA}\|+O(n\lg D)$
[27]	$O(t_{\mathsf{search}}(p)+k\lg(D/k))$	$\|\mathsf{CSA}\|+n\lg D+o(n\lg D)$
This paper	$O(t_{\mathsf{search}}(p)+kt_{\mathsf{SA}})$	$\|\mathsf{CSA}\|+O(n\lg\lg n)$
This paper(*)	$O(t_{\mathsf{search}}(p)+t_{\mathsf{SA}}(k+\lg n)\lg\lg n)$	$\|\mathsf{CSA}\|+O(n\lg D/\lg\lg D+n\lg\lg\lg n)$
This paper(*)	$O(t_{\mathsf{search}}(p)+t_{\mathsf{SA}}(k+\lg n)(\lg\lg n)^2)$	$\|\mathsf{CSA}\|+O(n\lg\lg\lg n)$
[15]	$O(t_{\mathsf{search}}(p)+kt_{\mathsf{SA}}\lg k\lg^{\varepsilon} n)$	$\|\mathsf{CSA}\|+D\lg(n/D)+O(D)+o(n)$

document array [19,20], which uses $n\lg D+o(n\lg D)$ bits on top of the CSA, and circumvents the use of the CSA to access the k results. These solutions are called *compact*. For example, using $|\mathsf{CSA}|+n\lg D+o(n\lg D)$ bits instead of $2|\mathsf{CSA}|$, one can reduce the time of the fastest solution [16] down to $O(p+k\lg k\lg^{\varepsilon} n)$, which already outperforms some previous solutions of this kind [14,15]. However, the real improvement was obtained by Hon et al. [17], who reduced the time below any $O(k\lg^{\varepsilon} n)$ for small alphabets or using $2n\lg D$ bits. It was also shown that it was possible to obtain fast results using $n\,o(\lg D)$ bits, by simulating the document array via the global CSA and some sublinear extra data [14,15].

Konow and Navarro [21] achieved $O(p+(\lg\lg n)^2+k\lg\lg n)$ time within $|\mathsf{CSA}|+(n\lg D+4n\lg\lg n)(1+o(1))$ bits, but the result holds only almost surely on typical texts, not in the worst case. Their index, on the other hand, turns out to be very competitive in practice. There exist several other indexes of practical interest [22,6,23–25].

Top-k most important document retrieval If we assign a fixed importance to each document, independent of the search pattern, the problem becomes simpler, but still nontrivial. Table 2 lists the space-time tradeoffs achieved.

With a suffix tree, this problem can be reduced to having an array of *colors* and finding k heaviest colors in a given array range, considering that a given color may appear repeated several times in the range. In our case, the array is the document array, documents are colors, and weights are their importance. Karpinski and Nekrich [26] solved the problem in linear space and optimal time $O(k)$, which with the help of a suffix tree yields an $O(p+k)$ time and linear-space solution to the top-k most important documents problem. If, instead, we use a CSA, the space becomes $|\mathsf{CSA}|+O(n\lg D)$ bits and the time becomes $O(t_{\mathsf{search}}(p)+k)$. If we assign identifiers to the colors in decreasing weight order, the problem becomes that of listing the k colors with lowest identifiers in an array range. Gagie et al. [27] showed how to list these using a particular sequence representation of the document array, thus reducing the space to $|\mathsf{CSA}|+n\lg D+o(n\lg D)$ bits, but raising the time to $O(t_{\mathsf{search}}(p)+k\lg(D/k))$. Finally, Belazzougui et al. [15] reduced the space to the asymptotically optimal $|\mathsf{CSA}|+D\lg(n/D)+O(D)+o(n)$, raising the time to $O(t_{\mathsf{search}}(p)+kt_{\mathsf{SA}}\lg k\lg^{\varepsilon} n)$. Their solution is based on a dual sampling that was later successfully applied by Hon et al. [16] to the more complex case of top-k most frequent documents problem.

Our contributions A relevant question is how far are these results from optimal, more specifically, how much space is needed to obtain optimal time and how fast can an index be within asymptotically optimal space. In this paper we obtain new results that push the state of the art much closer to giving a definitive answer to those questions. Our concrete results are the following.

1. We describe a compressed index for top-k most frequent retrieval using $|\mathsf{CSA}|+D\lg(n/D)+O(D)+o(n)$ bits, which is the asymptotically optimal $|\mathsf{CSA}|+o(n)$ when $D=o(n)$. The index answers queries in time $O(t_{\mathsf{search}}(p)+kt_{\mathsf{SA}}\lg^2 k\lg^{\varepsilon} n)$ (Theorem 1). This time improves upon the best current result on optimal space [18] by a factor of $\lg n/\lg k$, supersedes all the results using $n\,o(\lg D)$ further bits, and it is only $\lg k$ times slower than the best solution that uses about twice the space [16]. Actually, the complexity becomes very close to the minimum time $\Omega(t_{\mathsf{search}}(p)+kt_{\mathsf{SA}})$ needed to solve these queries when the accesses to the document identifiers must be done through a CSA. On our way, we introduce a structure we call the *sampled document array*, which might be of independent interest.
2. We describe a compact top-k index for top-k most frequent retrieval using $|\mathsf{CSA}|+n\lg D+o(n\lg D)$ bits that answers queries in time $O(t_{\mathsf{search}}(p)+k\lg^* n)$, where $\lg^* n$ is the iterated logarithm of n (Theorem 2). Note that, within this space, we can accomodate CSAs that achieve $t_{\mathsf{search}}(p)=p$ [13], and therefore the time is $O(p+k\lg^* n)$, outperforming all previous compact indices and getting very close to the optimal $O(p+k)$. Even closer, we show how to reach time $O(p+k\lg^* k)$ within $|\mathsf{CSA}|+n\lg D+o(n\lg D+n\lg\sigma)$ bits of space (Theorem 3). The extra space, $o(n\lg\sigma)$, is usually negligible compared to $o(n\lg D)$ and is also included in many CSAs [10], albeit not in the most recent ones [12,13]. We remark that the top-k results are not returned in sorted order of frequency.
3. We present a new tradeoff for top-k most important retrieval, which is time-optimal in a sense. It uses less space than what is needed to obtain a document identifier without using the CSA. Within this space, it obtains the optimal time $O(t_{\mathsf{search}}(p)+kt_{\mathsf{SA}})$. Its space, $|\mathsf{CSA}|+O(n\lg\lg n)$ bits, on the other hand, is away from optimal, and in particular uninteresting for small $D=O(\text{polylog}\, n)$. For such D values we obtain other tradeoffs, with slightly higher time and space as low as $|\mathsf{CSA}|+O(n\lg\lg\lg n)$. The only index that reaches optimal space [15] is $O(\lg k\lg^{\varepsilon} n)$ times slower. On our way, we obtain a new result on reporting the heaviest points in a range over polylog-height grids, which might be of independent interest. We remark that the top-k results are not returned in sorted order of importance.

We start in Section 2 with a review of the basic concepts and previous ideas that are necessary to understand the rest of the paper. Then Section 3 describes our space-optimal top-k index, culminating in Theorem 1 and its application in some particular cases. Sections 4 to 5 describe the solution that reaches almost optimal time, finishing with Theorem 2, and Section 6 improves the time even further (Theorem 3). In Sections 7 and 8 we describe our solutions for top-k most important retrieval (Theorem 4 and corollaries). Finally, we conclude in Section 9.

2. Basic concepts and related work

We recall the setup of the problem: We can preprocess a collection $\mathcal{D} = \{T_1, T_2, \ldots, T_D\}$ of D string documents of total length n, over an alphabet $[1..\sigma]$. At query time, we are given a pattern $P[1..p]$ and retrieve k document identifiers where P is most relevant. Our notion of relevance will be either the popular *term frequency* $\mathsf{tf}(P, d)$, that is, the number of times P occurs in T_d, or a fixed *importance* assigned to each T_d independently of P.

For technical convenience we will assume that the last character of each document is $\$ = 0$, a special symbol that is smaller than all the others in the alphabet.

2.1. Suffix trees and arrays

Let $T[1..n] = T_1T_2T_3...T_D$ be the text obtained by concatenating all the documents in $\mathcal{D}$. Each substring $T[i..n]$, with $i \in [1..n]$, is called a *suffix* of T. The *suffix tree* [3] for T (or, equivalently, the *generalized suffix tree (GST)* of $\mathcal{D}$) is a lexicographic arrangement of all these n suffixes in a compact trie structure, where the ith leftmost leaf represents the ith lexicographically smallest suffix. Each edge in the suffix tree is labeled by a string, and $path(v)$ of a node v is the concatenation of edge labels along the path from the *root* of GST to node v. We identify GST nodes with their preorder ranks, therefore $path(x)$ also refers to the GST node with preorder rank x. Let ℓ_i, for $i \in [1..n]$, represent the (preorder rank of) the ith leftmost leaf in GST. Then $path(\ell_i)$ represents the ith lexicographically smallest suffix of T. A node x is called the *locus* of a pattern P if it is the node closest to the *root* with $path(x)$ prefixed by P. We call $L(x)$ the set of leaves that descend from node x, and $L(x \backslash y) = L(x) \setminus L(y)$ for a node y that descends from x.

The *suffix array* [4] $\mathsf{SA}[1..n]$ of $T[1..n]$ is an array where $\mathsf{SA}[i]$ is the starting position (in T) of the ith lexicographically smallest suffix of T. An important property of SA is that the starting positions of all the suffixes with the same prefix are always stored in a contiguous region of SA. Based on this property, we define the *suffix range* of P in SA to be the maximal range $[sp..ep]$ such that for all $i \in [sp..ep]$, $\mathsf{SA}[i]$ is the starting point of a suffix of T prefixed by P. Which is the same, the suffix range of P is the set $L(x)$, where x is the locus of P in GST.

A compressed representation of SA is called a Compressed Suffix Array (CSA) [10,12,13]. We call $|\mathsf{CSA}|$ its size in bits (it holds $|\mathsf{CSA}| \leq n \lg \sigma + o(n \lg \sigma)$ for most CSAs, and in most cases they use space close to that of the compressed text T). We also call $t_{\mathsf{search}}(p)$ the time the CSA needs to compute the suffix range $[sp..ep]$ of any given pattern $P[1..p]$, and t_{SA} the time it needs to compute any cell $\mathsf{SA}[i]$.

The tree topology of GST can be represented in (at most) $4n + o(n)$ bits, using representations (e.g., [28]) with constant-time support of the operations $parent(x)$ (the parent of node x), $lca(x, y)$ (the lowest common ancestor of nodes x and y), $left\text{-}leaf(x)/right\text{-}leaf(x)$ (the leftmost/rightmost leaf in the subtree rooted at node x), and $leaf(i)$ (the ith leftmost leaf), and mapping from nodes to their preorder ranks and back. The total space of a CSA and the GST topology is thus $|\mathsf{CSA}| + O(n)$ bits.

Given a node $v \in \mathsf{GST}$, we call $\mathsf{tf}(v, d)$ the number of leaves in $L(v)$ associated to document T_d. This is the same as the number of occurrences of $path(v)$ in T_d. Then the top-k retrieval problem can be solved by first finding the locus v of pattern P, and then retrieving the k documents T_d with highest $\mathsf{tf}(v, d)$ values. Note that the problem could be solved by attaching the answer to all suffix tree nodes, but the space would be $O(kn \lg n)$ bits, and would work only up to a chosen k.

2.2. Rank, select, and document arrays

Given a sequence $S[1..n]$ on an alphabet $[1, \sigma]$, we will use representations that require $n \lg \sigma + o(n \lg \sigma)$ bits and support the following operations

- $access_S(i)$ returns $S[i]$;
- $rank_S(r, i)$ returns the number of occurrences of $r \in [1..\sigma]$ in $S[1..i]$; and
- $select_S(r, j)$ returns i where $S[i] = r$ and $rank_S(r, i) = j$, that is, the position of the jth occurrence of r in S.

When $|\sigma| = 2$, that is, S is a bitvector, there exist representations using $n + o(n)$ bits and supporting the three operations in $O(1)$ time [29,30]. There exist also compressed representations: if there are only m 1s in S (similarly 0s), the constant-time complexities can be maintained while using only $m \lg \frac{n}{m} + O(m) + o(n)$ bits of space [31]. If we only perform $rank_S(r, i)$ where $S[i] = 1$ and $select_S(1, i)$, an "indexed dictionary" [31] achieves constant time using only $m \lg \frac{n}{m} + O(m + \lg \lg n)$ bits. If we only need constant-time $select_S(1, i)$ queries, $O(m \lg \frac{n}{m})$ bits are sufficient [32]. By default, on bitvectors we will assume $rank_S(i) = rank_S(1, i)$ and $select_S(j) = select_S(1, j)$.

For general sequences, we will use the following result.

Lemma 1. *(See [33].) A string $S[1..n]$ over alphabet $[1..\sigma]$ can be represented in $n\lg\sigma + o(n\lg\sigma)$ bits and support queries $access_S$, $rank_S$ and $select_S$ in times $O(1)$, $O(\lg\frac{\lg\sigma}{\lg\lg n})$ and $O(f(n,\sigma))$, respectively, where $f(n,\sigma)=\omega(1)$ is any non-constant function. Alternatively, the times for $access_S$ and $select_S$ can be exchanged.*

The so-called *partial rank* query, where we ask for the number of occurrences of symbol $S[i]$ up to its position i, can be supported more efficiently than general *rank* queries.

Lemma 2. *(See [15].) Operation $rank_S(S[i], i)$ can be supported in $O(1)$ time by storing $O(n\lg\lg\sigma) = o(n\lg\sigma)$ additional bits on top of S.*

Define a bitvector $B[1..n]$, such that $B[i]=1$ iff $T[i]=\$$. Then suffix $T[i..n]$ belongs to document T_d if $d = 1 + rank_B(i)$ [34]. Note that B has D 1s, thus it can be represented using $D\lg(n/D) + O(D) + o(n)$ bits, which is $o(n)$ if $D = o(n)$, and answer *rank* queries in constant time [31].

The *document array* $\mathsf{DA}[1..n]$ [19] is defined as $\mathsf{DA}[j]=d$ if the suffix $\mathsf{SA}[j]$ belongs to document T_d. Moreover, we say that the corresponding leaf node ℓ_j is *marked* with document T_d. The idea of representing DA as a sequence over alphabet $[1..D]$ [20], with support for *access*, *rank*, *select*, and partial *rank*, will be used in this paper. The space of such a representation is $n\lg D + o(n\lg D)$ bits.

2.3. Hon, Shah and Vitter's compressed top-k index

Hon et al.'s [7] structure is built (in principle) for a fixed k value. We choose a grouping factor $g = k\lg^{2+\varepsilon} n$ and *mark* every gth leaf in GST (we use a slightly simplified description of their method [25]). Then we mark the lowest common ancestor (LCA) of every consecutive pair of marked leaves. The tree of marked nodes is called τ_k and has $O(n/g)$ nodes. For every marked GST node v, we store the k pairs $(d, \mathsf{tf}(v,d))$ with highest $\mathsf{tf}(v,d)$. Hon et al. prove that any locus node v contains one maximal marked node u so that $|L(v\backslash u)| < 2g$, or if there is no such u it holds $|L(v)| < 2g$. Therefore they traverse those (at most) $2g$ leaves using the CSA, and for each one they (1) compute the corresponding document T_d, (2) compute the frequency $\mathsf{tf}(v,d)$, (3) add d to the top-k list (or correct its frequency from $\mathsf{tf}(u,d)$ to $\mathsf{tf}(v,d)$ if d was already stored in the precomputed top-k list of u).

Their procedure starts by computing $[sp..ep]$, which is the range covered by the locus node v of P, by looking for P in CSA in time $t_{\mathsf{search}}(p)$. The node $u \in \tau_k$ is then found using a constant-time LCA on τ_k for the leftmost and rightmost marked leaves in $[sp..ep]$ (i.e., the smallest and largest multiple of g in $[sp..ep]$). They obtain the top-k answers to u, as well as the leaf range $[sp'..ep'] \subseteq [sp..ep]$ covered by u, and thus traverse $[sp..sp'-1]$ and $[ep'+1..ep]$ to correct the top-k list of u (running steps (1)–(3) above on each such leaf). To carry out (1) on the ith GST leaf, they first compute $\mathsf{SA}[i]$ in $O(t_{\mathsf{SA}})$ time, and then convert it into a document identifier $d = 1 + rank(B, \mathsf{SA}[i])$, as explained in Section 2.2. To carry out (2) they need an additional CSA_d per document T_d [34], and require time $O(t_{\mathsf{SA}}\lg n)$. Thus the total query time is $O(t_{\mathsf{search}}(p) + g\,t_{\mathsf{SA}}\lg n) = O(t_{\mathsf{search}}(p) + k\,t_{\mathsf{SA}}\lg^{3+\varepsilon} n)$.

As storing the top-k list needs $O(k\lg n)$ bits, the space for τ_k is $O((n/g)k\lg n) = O(n/\lg^{1+\varepsilon} n)$ bits. One τ_k tree is stored for each k power of 2, so that at query time we increase k to the next power of 2 and solve the query within the same time complexity. Summed over all the powers of 2, the space becomes $O(n/\lg^{\varepsilon} n) = o(n)$ bits. Therefore the total space is $2|\mathsf{CSA}| + D\lg(n/D) + O(D) + o(n)$ bits, which is $2|\mathsf{CSA}| + o(n)$ if $D = o(n)$.

Several subsequent improvements [14–16] reduced the time to $O(t_{\mathsf{search}}(p) + k\,t_{\mathsf{SA}}\lg k\lg^{\varepsilon} n)$, yet still using $2|\mathsf{CSA}| + o(n)$ bits of space, that is, about twice the space of an optimal representation of the collection. Only recently [18] the space was reduced to the optimal $|\mathsf{CSA}| + o(n)$ bits, yet the time raises to $O(t_{\mathsf{search}}(p) + k\,t_{\mathsf{SA}}\lg k\lg^{1+\varepsilon} n)$.

2.4. Tsur's optimal-space index

By enhancing previous developments [15], Tsur [18] managed to reduce the space to $|\mathsf{CSA}| + D\lg(n/D) + O(D) + o(n)$ bits, which is the asymptotically optimal $|\mathsf{CSA}| + o(n)$ bits if $D = o(n)$. Let $u' \in \tau_k$ be the parent of u in τ_k, that is, its nearest marked ancestor in GST. Tsur proved that, from the $O(g)$ leaves of $L(u'\backslash u)$, only $O(\sqrt{gk})$ have a chance to become part of the top-k list for a locus node v between u' and u. Thus, they simply store those *candidate* documents, and their frequency in $L(u)$, associated to u. When they traverse the $O(g)$ leaves in $L(v\backslash u)$, they (1) compute the document identifier d as before, (2) if it is not stored as a candidate for u they just ignore it, (3) if it is in the list then they increases its frequency by 1. At the end, they have enough information to answer the top-k query, without the need of the CSA_d structures to compute frequencies below v.

If $g = k\ell$, the number of candidates is $t = O(\sqrt{gk}) = O(k\sqrt{\ell})$. They can be efficiently encoded by storing, for each candidate d, the position of one leaf corresponding to d in the area of $L(u'\backslash u)$. Those leaf positions are sorted and stored differentially: Let $0 < p_1 < p_2 < \ldots < p_t < 2g$ be the ordered positions, then one encodes $x_1, x_2, \ldots, x_t$, where $x_i = p_i - p_{i-1}$ ($p_0 = 0$) using, say, γ-codes [35], which occupy $\sum 2\lg x_i = O(t\lg(g/t)) = O(k\sqrt{\ell}\lg\ell)$ bits by the log-sum inequality. The frequencies are encoded in $O(k\lg n + k\sqrt{\ell}\lg\ell)$ bits (the method is not relevant here).

Therefore, the space for top-k answers plus candidates is $O(k\lg n + k\sqrt{\ell}\lg\ell)$ bits, and the total space for a fixed k equals $O((n/g)(k\lg n + k\sqrt{\ell}\lg\ell)) = O(n((\lg n)/\ell + (\lg\ell)/\sqrt{\ell}))$ bits. By choosing $\ell = \lg k\lg^{1+\varepsilon} n$, and since $\lg k \le \lg n$, this is

$O(n/(\lg k \lg^{\varepsilon/2} n))$. Added over all the k values that are powers of 2, this is $O(n/\lg^{\varepsilon/2} n) \cdot \sum_{i=1}^{\lg D} 1/i = O(n \lg\lg D/\lg^{\varepsilon/2} n) = o(n)$ bits. Considering the $O(\lg\lg k + \lg\lg\lg n)$ time they require to handle the search and update of candidates, the query time is $O(t_{\mathsf{search}}(p) + g(t_{\mathsf{SA}} + \lg\lg k + \lg\lg\lg n)) = O(t_{\mathsf{search}}(p) + k t_{\mathsf{SA}} \lg k \lg^{1+\varepsilon} n)$, where $\lg^{\varepsilon} n$ absorbs the loglogarithmic costs. (Tsur shows how to stretch this result a bit more, so that $\lg^{\varepsilon} n$ can be changed for a polynomial in $\lg\lg n$, but the same can be done on the other indexes and the results become more cumbersome.)

2.5. Hon, Shah, Thankachan and Vitter's faster index

Hon et al. [16] obtained the fastest solution to date using $2|\mathsf{CSA}| + D\lg(n/D) + O(D) + o(n)$ bits of space. For this sake they consider two independent blocking values, $c < g$. For block value g they build the τ_k trees as before. For block value c they build another set of marked trees ρ_k. These trees are finer-grained than the τ_k trees. Now, given the locus node v, there exists a maximal node $w \in \rho_k$ contained in v, and a maximal node $u \in \tau_k$ contained in w. The key idea is to build a list of top-k to top-$2k$ candidates by joining the precomputed results of w and u, and then correct this result by traversing $O(c)$ GST leaves.

Since we have a maximal node $u \in \tau_k$ contained in any node $w \in \rho_k$, we can encode the top-k list of w only for the documents that are not already in the top-k list of u. Note that a document must appear at least once in $L(w\backslash u)$ if it is in the top-k list of w but not in that of u. Thus the additional top-k candidates of w can be encoded using $O(k\lg(g/k))$ bits, by storing as before one of their positions in $L(w\backslash u)$, and encoding the sorted positions differentially. The frequencies do not need to be encoded, since they can be recomputed as for any other candidate.

The space for a τ_k tree is $O((n/g)k\lg n) = O(n/\lg^{1+\varepsilon} n)$ bits using $g = k\lg^{2+\varepsilon} n$, which added over all the powers of 2 for k gives $O(n/\lg^{\varepsilon} n) = o(n)$ bits, as before. For the ρ_k trees they require $O((n/c)k\lg(g/k))$ bits, which using $c = k\lg k\lg^{\varepsilon} n$ gives $O(n\lg\lg n/(\lg k\lg^{\varepsilon} n))$ bits. Added over the powers of 2 for k this gives $O(n\lg\lg n/\lg^{\varepsilon} n) \cdot \sum_{i=1}^{\lg D} 1/i = O(n\lg\lg n\lg\lg D/\lg^{\varepsilon} n) = o(n)$ bits.

The time is dominated by that of traversing $O(c)$ cells. Using some speedups [15] over the basic technique [7], the time is $O(t_{\mathsf{SA}}\lg\lg n)$ per cell, for a total of $O(t_{\mathsf{search}}(p) + k t_{\mathsf{SA}}\lg k\lg^{\varepsilon} n)$ for any constant $\varepsilon > 0$.

3. A faster space-optimal representation

We build upon the schemes of Tsur [18] (Section 2.4) and Hon et al. [16] (Section 2.5). We will use the dual marking mechanism of Hon et al., with trees τ_k and ρ_k, and make it work without using the individual CSA_d structures. Without these, the index gives us the top-k list of the maximal node $w \in \rho_k$ that is below the locus v, but not their frequencies. Similarly, when we traverse the $O(c)$ extra cells to correct the top-k list, we have no way to compute the frequency of the document identifiers d found in $L(v\backslash w)$.

In order to cope with the second problem, we will use the idea of Tsur: there can be only $O(\sqrt{ck})$ candidates that can make it to the top-k list. If $c = k\ell$, this is $O(k\sqrt{\ell})$. Thus we can record their identities by means of their sorted and differentially encoded positions along $O(c)$ leaves, in total space $O(k\sqrt{\ell}\lg\frac{c}{k\sqrt{\ell}}) = O(k\sqrt{\ell}\lg\ell) = O(k\sqrt{\ell}\lg\lg n)$ bits. Now we need a mechanism to store the frequencies, both of the top-k elements and of the $O(k\sqrt{\ell})$ candidates. For this sake we introduce a new data structure, which might have independent interest.

3.1. The sampled document array

The document array $\mathsf{DA}[1..n]$ would allow us to compute the desired frequency for any document d, that is, $\mathrm{tf}(v,d) = rank_{\mathsf{DA}}(d, ep) - rank_{\mathsf{DA}}(d, sp-1)$. The document array uses too much space, however. We will store just a sampled version of it.

Definition 1. The *sampled document array* is an array $\mathsf{DA}'[1..n']$ that stores every sth occurrence of each document identifier d in DA, for a sampling step s. That is, the cell $\mathsf{DA}[i]$ is stored in DA' iff $rank_{\mathsf{DA}}(\mathsf{DA}[i], i)$ is a multiple of s. Note that $n' \le n/s$.

We associate to DA' a bitvector $S[1..n]$ that marks the positions in DA that are sampled in DA'. The next lemma follows easily.

Lemma 3. *Let x be the number of occurrences of a document identifier d in $\mathsf{DA}[sp..ep]$, and let y be the number of occurrences of d in $\mathsf{DA}'[rank_S(sp-1)+1..rank_S(ep)]$. Then $(y-1)s < x < (y+1)s$.*

Proof. The area $\mathsf{DA}[sp..ep]$ includes y sampled occurrences of document d. Each such sampled occurrence is preceded by $s-1$ non-sampled occurrences. For the last $y-1$ sampled occurrences, their $s-1$ preceding non-sampled occurrences are also contained in $\mathsf{DA}[sp..ep]$. Instead, the $s-1$ non-sampled occurrences preceding the first sampled occurrence could be before sp, and thus $x \ge y + (y-1)(s-1) = (y-1)s + 1$.

As for the upper bound, consider that all the $s-1$ non-sampled occurrences preceding each of the y sampled occurrences could lie within the range $\mathsf{DA}[sp..ep]$. Moreover, the range could also include all the $s-1$ non-sampled occurrences that follow the last sampled occurrence, thus $x \le y + y(s-1) + (s-1) = ys + (s-1)$. □

To use this lemma we store DA' using a representation (Lemma 1) that requires $n' \lg D + o(n' \lg D)$ bits and computes $rank_{\mathsf{DA}'}(d, i)$ in time $O(\lg \lg D)$. Further, we represent S in compressed form [31] so that it requires $n' \lg(n/n') + O(n') + o(n)$ bits and supports $rank_S(i)$ in constant time. We use $s = \lg^2 n$, thus $n' = O(n/\lg^2 n)$ and the space for both DA' and S is $o(n)$. Using this representation, we can compute y in Lemma 3 as $rank_{\mathsf{DA}'}(d, rank_S(ep)) - rank_{\mathsf{DA}'}(d, rank_S(sp-1))$ in time $O(\lg \lg D)$. We know that $x = ys \pm O(\lg^2 n)$.

3.2. Completing the index

To retrieve any $\mathsf{tf}(w, d)$ for a top-k document in node $w \in \rho_k$, we use S and DA' to compute the approximation ys in time $O(\lg \lg D)$, and then need to store only $O(\lg s) = O(\lg \lg n)$ bits in w to *correct* this approximate count. Each node $w \in \rho_k$ stores (the correction of) the frequency information of both its top-k documents that appear in the top-k list of its maximal descendant node $u \in \tau_k$, and those that do not (in fact, we do not need frequency information associated to τ_k nodes). Similarly, we need to compute $\mathsf{tf}(w, d)$ for any of the $O(\sqrt{ck})$ candidates to top-k in w, thus we must also store (the correction of) those $O(\sqrt{ck})$ frequencies, which dominate the total space of $O(k\sqrt{\ell} \lg \lg n)$ bits. With this information we can discard the CSA_d structures of Hon et al. [16].

We use $\ell = \lg^2 k \lg^\varepsilon n$. The space for one ρ_k tree is $O((n/c) k\sqrt{\ell} \lg \lg n) = O(n \lg \lg n/\sqrt{\ell}) = O(n \lg \lg n/(\lg k \lg^{\varepsilon/2} n))$ bits. Adding over all the powers of 2 for k yields $O(n \lg \lg n/\lg^{\varepsilon/2} n) \cdot \sum_{i=1}^{\lg D} 1/i = O(n \lg \lg n \lg \lg D/\lg^{\varepsilon/2} n) = o(n)$ bits. Thus the total space is $|\mathsf{CSA}| + o(n)$ bits.

At query time we store the top-k documents of w (part of which are extracted from the top-k list of $u \in \tau_k$ [16]), plus the $O(\sqrt{ck})$ candidates, together with their frequencies in w, in a dictionary using the document identifiers as keys. Then we traverse the $O(c)$ cells of $L(v \backslash w)$, accessing the CSA and bitvector B to determine each document identifier d. If d is not in the dictionary, it can be discarded, otherwise we increment its frequency. At the end, we scan the $O(\sqrt{ck})$ elements of the dictionary and keep the k largest ones. The cost of this procedure includes $O(\sqrt{ck} \lg \lg D) = O(c \lg \lg D/\sqrt{\ell}) = o(c)$ to compute the frequencies of the candidates below w using the frequency correction information and *rank* queries on DA'; $O(c\, t_{\mathsf{SA}})$ to compute the document identifiers of $O(c)$ CSA cells; $O(c)$ time to perform constant-time operations on the dictionary[2]; and $O(\sqrt{ck}) = o(c)$ time to collect all the candidates from the dictionary, find the kth largest frequency θ using linear-time selection [36], and then output elements with frequency θ or higher (more precisely, we report the $k' < k$ candidates with $\mathsf{tf}(v, d) < \theta$ in a first pass, and then report the first $k - k'$ documents we find with $\mathsf{tf}(v, d) = \theta$ in a second pass). This adds up to $O(k\, t_{\mathsf{SA}} \lg^2 k \lg^\varepsilon n)$ time, dominated by the time to compute the CSA cells.

Theorem 1. *The top-k most frequent documents problem, on a collection of length n, for a pattern of length p, can be solved using* $|\mathsf{CSA}| + D \lg(n/D) + O(D) + o(n)$ *bits and in* $O(t_{\mathsf{search}}(p) + k\, t_{\mathsf{SA}} \lg^2 k \lg^\varepsilon n)$ *time, for any constant* $\varepsilon > 0$. *Here* CSA *is a compressed suffix array over the collection,* $t_{\mathsf{search}}(p)$ *is the time* CSA *takes to find the suffix array interval of the pattern, and* t_{SA} *is the time it takes to retrieve any suffix array cell.*

We also give two simplifications using recent CSAs [12,13] whose size is related to H_h, the per-symbol empirical entropy of the text collection, for any $h \le \alpha \lg_\sigma n$ and any constant $0 < \alpha < 1$. For the second, since it uses $O(n)$ extra bits, we set a smaller $c = k(\lg k \lg \lg n \lg \lg D)^2$.

Corollary 1. *The top-k most frequent documents problem, when* $D = o(n)$, *can be solved using* $nH_h + o(nH_h) + o(n)$ *bits and in* $O(p \lg \lg \sigma + k \lg^2 k \lg^{1+\varepsilon} n)$ *time, for any constant* $\varepsilon > 0$.

Corollary 2. *The top-k most frequent documents problem can be solved using* $nH_h + o(nH_h) + O(n)$ *bits and in* $O(p + k \lg n(\lg k \lg \lg n \lg \lg D)^2)$ *time.*

3.3. Construction

The time to build the sampled document array is clearly $O(n)$ plus the time needed to build the data structure of Lemma 1 on a sequence of length n' and alphabet size D. This is dominated by the time to build $O(n'/D)$ perfect hash functions over $O(D)$ elements and a universe of size $O(D)$, which can be done in overall time $O(n' \lg \lg D)$ [37]. Since $n' = O(n/\lg^2 n)$, this extra time is $o(n)$.

The rest of the construction is dominated by the cost to precompute the answers, in particular to find the top-k documents for each sampled node, plus finding the additional candidates.

[2] For example, we can bucket the universe $[1..D]$ in chunks of $\lg^2 D$ elements, and store a B-tree of arity $\sqrt{\lg D}$ and height $O(1)$ for the elements falling in each chunk. The bucket structure adds up to $o(D)$ bits, which can be taken as part of the index. The B-trees are operated in constant time because they store only $O(\sqrt{\lg D} \lg \lg D) = o(\lg n)$ bits per internal node. They occupy overall $O(\sqrt{ck} \lg n) = O(k \lg k \lg^{1+\varepsilon/2} n)$ bits, which is the space we use to answer the query. See [2, App. E] for more details.

For the first part, we can use a data structure [21], which is built in $O(n \lg n)$ time and answers a top-k query in $O((k+\lg n)\lg n)$ time once the locus of the pattern is given.[3] Adding the $O(k \lg n)$ times over all the $O(n/c)$ sampled nodes, for $c = k \lg^2 k \lg^{\varepsilon} n$, gives $O(n \lg^{1-\varepsilon} n / \lg^2 k)$, which added over all the powers of 2 for k gives $O(n \lg^{1-\varepsilon} n)$. Instead, adding the $O(\lg^2 n)$ times gives $O(n \lg^{2-\varepsilon} n/(k \lg^2 k))$, which added over all the powers of 2 for k gives $O(n \lg^{2-\varepsilon} n)$.

Consider now the process of extending the intervals of the sampled nodes to look for further candidates to store. We first build the complete (not the sampled) document array using Lemma 1, in $O(n \lg \lg D)$ time [37]. Now, we consider all the sampled nodes for all the k values that are powers of 2.

For each sampled node v, we put its top-k answers in a min-priority queue Q limited to size k, maintaining also a balanced binary tree K with the elements present in the queue (K will be used to determine whether a document d is in Q, and where), and initially empty balanced trees C and E (C will contain the new candidates found in each area, and E the union of the new candidates). Now we traverse the cells of $L(u \setminus v)$, where u is the parent of v. For each document d found in a cell, we compute $\mathsf{tf}(u,d)$ in time $O(\lg \lg D)$ using the document array, and if this is higher than the lowest term frequency in Q, we insert d in Q, K, and C, displacing the currently lowest value from Q, K and, if present, from C. (It might also be that d was already in the queue and we just update its frequency from $\mathsf{tf}(v,d)$ to $\mathsf{tf}(u,d)$ in Q, without changing K or C, and even that we have found d for the second time and the frequency was already updated to $\mathsf{tf}(u,d)$.) Once we complete the process of $L(u \setminus v)$, we set $v \leftarrow u$, $u \leftarrow parent(u)$, add all the elements of C to E (without duplicates), make C empty, and leave Q and K as is. Then we restart the process, until reaching the nearest marked ancestor of v. At this point, we have in E the set of extra candidates that should be stored in v.

The whole process traverses $O(c)$ cells, and costs $O(c \lg k)$ to operate Q, K and C, and $O(\sqrt{ck} \lg \sqrt{ck})$ to operate E. The encoding of the candidates can be done in $O(\sqrt{ck})$ time. This is carried out over $O(n/c)$ nodes, so the total cost is $O(n(\lg k + \lg(ck)/\sqrt{\ell})) = O(n \lg k)$. Adding over all the powers of 2 for k, we obtain $O(n \lg^2 D)$ time.

The sum of all the pieces of the construction time can thus be upper bounded by $O(n \lg^2 n)$. The extra time to build the CSA is $O(n)$ to $O(n \lg \sigma)$ for all known CSAs.

4. An index with near-optimal time

We now describe an index that answers queries in $O(t_{\mathsf{search}}(p) + k \lg^* n)$ time. Our index will contain a CSA, the topology of its GST using $O(n)$ bits (recall Section 2.1), and the document array DA represented as in Lemmas 1 and 2, for a total of $|\mathsf{CSA}| + n \lg D + o(n \lg D)$ bits. In addition we will store some precomputed answer lists.

We use the grouping scheme of Hon et al. [7] (Section 2.3) to define the τ_k marked trees; however we will use various grouping factors g for the same k. Therefore, this time we will make emphasis on g rather than on k: we will call τ_g the trees obtained with grouping factor g, and the k value (or actually z, its next power of two) will be implicit.

Let $F(x,k)$ represent the list (or set) of top-k documents T_d, along with $\mathsf{tf}(x,d)$, corresponding to a pattern with locus node x in GST. Clearly we cannot afford to maintain $F(x,k)$ for all possible xs and ks. Rather, we will maintain the lists $F(x,z)$ only for marked nodes xs (for various g values) and for zs that are powers of 2. Then $F(x,k)$ for any x and k will be efficiently computed using that sampled data. We now describe how to store and retrieve the sampled lists. The following is a key result in our scheme, and the rest of the section is devoted to prove it.

Lemma 4. *Let $g_h = z(\lg^{(h)} n)^2$ for any $1 \le h < \lg^* n$, where $\lg^{(1)} n = \lg n$, $\lg^{(h)} n = \lg(\lg^{(h-1)} n)$, and $\lg^{(\lg^* n)} n \le 1$. Then $F(x,z)$ for all $x \in \tau_{g_h}$ can be encoded in $s_h = s_{h-1} + O(n/\lg^{(h)} n)$ bits, and $F(x,z)$ for any given $x \in \tau_{g_h}$ can be decoded in time $t_h = t_{h-1} + O(z)$, where $s_1 = O(n/\lg n)$ and $t_1 = O(z)$.*

To prove the lemma, we use induction. Consider the base case $h = 1$. For every $x \in \tau_{g_1}$, we maintain the list $F(x,z)$ explicitly (using $O(\lg n)$ bits per element), along with a pointer to the location where it is stored, in $s_1 = O(|\tau_{g_1}| z \lg n) = O(n/\lg n)$ bits. Thus the list $F(x,z)$, for any $x \in \tau_{g_1}$, can be decoded in time $t_1 = O(z)$.

Now consider that the grouping factor is g_h for $h \ge 2$. As we cannot afford to use $\Theta(\lg n)$ bits per element, we introduce encoding schemes that reduce it to $O(\lg^{(h)} n)$ bits. Thus the overall space for maintaining $F(x,z)$ (in encoded form) for all $x \in \tau_{g_h}$ can be bounded by $O(|\tau_{g_h}| z \lg^{(h)} n) = O(n/\lg^{(h)} n)$ bits. Instead of using pointers as in the base case, we mark in a bitvector $B^h[1..2n]$ the node preorders of GST that belong to τ_{g_h}. Therefore the list $F(x,z)$ of a node $x \in \tau_{g_h}$ is stored in an array at offset $rank_{B^h}(x)$. Since we will only compute *rank* on positions x where $B^h[x] = 1$, an indexed dictionary (Section 2.2) suffices, requiring $O((n/g_h)\lg g_h + \lg \lg n) = o(n/\lg^{(h)} n)$ bits and computes *rank* in time $O(1)$. We now show how to encode the list $F(x,z)$, for $x \in \tau_{g_h}$, in $O(\lg^{(h)} n)$ bits per element, and how to decode it in $t_{h-1} + O(z)$ time.

We maintain a structure STR_h, using s_h bits, for each grouping factor g_h, and decode $F(x,z)$ for $x \in \tau_{g_h}$ recursively, using $O(z)$ time in addition to the time needed to decode $F(y,z)$ for some $y \in \tau_{g_{h-1}}$, as suggested in Lemma 4. As we cannot afford to sort the documents within the targeted query time, it is important to assume a fixed arrangement of documents

[3] They claim construction time $O(n \lg \sigma + n \lg \lg n)$, and query time $O((k + \lg \lg n)\lg \lg n)$, but only on typical texts, where the suffix tree height is $O(\lg n)$. The times we are using here are worst-case. We note there is another structure [9] answering those queries in $O(k)$ time, but its construction time is unclear.

within any particular decoded list $F(\cdot,\cdot)$. That is, each time we decode a specific list, the decoding algorithm must return the elements in the same order.

Let x be a node in τ_{g_h} and y (if it exists) be its highest descendant node in $\tau_{g_{h-1}}$. We show how to encode and decode $F(x,z)$. To decode $F(x,z)$, we first decode the list $F(y,z)$ using STR_{h-1} in time t_{h-1}. From now onwards we have constant-time access to any element the list $F(y,z)$. The list $F(x,z)$ will be partitioned into two disjoint lists:

(i) D_{old}, the documents that are common to $F(x,z)$ and $F(y,z)$.
(ii) D_{new}, the documents that are present in $F(x,z)$, but not in $F(y,z)$.

Encoding and decoding document identifiers in D_{old} We maintain a bit vector $B'[1..z]$, where $B'[i]=1$ iff the ith document in $F(y,z)$ is present in $F(x,z)$. Therefore D_{old} can be decoded by listing those elements in $F(y,z)$ (in the same order as they appear) at positions i where $B'[i]=1$. Thus the space for maintaining the encoded information is z bits and the time for decoding is $O(z)$.

Encoding and decoding document identifiers in D_{new} For each document $d \in D_{new}$, there exists at least one leaf in $L(x\backslash y)$ that is marked with T_d (otherwise $\mathrm{tf}(x,d)=\mathrm{tf}(y,d)$ and d could not be in $F(x,z)$ and not in $F(y,z)$). Therefore, instead of explicitly storing d, it is sufficient to refer to such a leaf. For this we shall store a bit vector $B''[1..|L(x\backslash y)|]$ with all its bits in 0, except for $|D_{new}|$ 1s: for every document $d \in D_{new}$, we set one bit, say $B''[i]=1$, where the ith leaf in $L(x\backslash y)$ is marked with T_d. Since $|B''|=|L(x\backslash y)|<2g_{h-1}$ and the number of 1s is at most z, B'' can be encoded in $O(z\lg(g_{h-1}/z))=O(z\lg^{(h)} n)$ bits with constant time *select* support (Section 2.2). Now, given B'', the documents in D_{new} can be identified in $O(z)$ time as follows: Find all those (at most z) increasing positions i where $B''[i]=1$ using *select* queries. Then, for each such i, find the ith leaf of $L(x\backslash y)$, $\ell_{i'}$, in constant time using the tree operations.[4] Finally, report $\mathsf{DA}[i']$ as a document in D_{new} for each such i' using a constant-time *access* operation on the document array.

As mentioned before, it is important for our (recursive) encoding/decoding algorithm to assume a fixed permutation of elements within any list $F(\cdot,\cdot)$. We use the convention that, in $F(x,z)$, the documents in D_{old} come before the documents in D_{new}. Moreover the documents within D_{old} and D_{new} are in the same order as the decoding algorithm identified them. In conclusion, the list of identifiers of documents in $F(x,z)$ can be encoded in $O(z\lg^{(h)} n)$ bits and decoded in $O(z)$ time, assuming constant-time access to any element in $F(y,z)$. If node y does not exist, we proceed as if $F(y,z)=\emptyset$ and $F(x,z)=D_{new}$. We now consider how to encode the tfs associated with the elements in $F(x,z)$ (i.e., $\mathrm{tf}(x,d)$ for all $d \in F(x,z)$).

Encoding and decoding of frequencies Let d_i, for $i \in [1..z]$, be the ith document in $F(x,z)$, and $f_i=\mathrm{tf}(x,d_i)$. Then, define $\delta_i=f_i-f'_i\geq 0$, where

$$f'_i=\begin{cases}\mathrm{tf}(y,d_i) & \text{if } i\leq |D_{old}| \text{ (i.e., if } d_i\in D_{old}),\\ \mu=\min\{\mathrm{tf}(y,d), d\in F(y,z)\} & \text{if } i>|D_{old}| \text{ (i.e., if } d_i\in D_{new}).\end{cases}$$

The following is an important observation: The number of leaves in $L(x\backslash y)$ marked with document T_{d_i} is $\mathrm{tf}(x,d_i)-\mathrm{tf}(y,d_i)$, which is the same as δ_i for $i\leq|D_{old}|$. For $i>|D_{old}|$, $\mathrm{tf}(x,d_i)-\mathrm{tf}(y,d_i)\geq\delta_i$, otherwise $\mathrm{tf}(y,d_i)>\mu$ and d_i would have qualified as a top-z document in $F(y,z)$ (which is a contradiction as $d_i\in D_{new}$). By combining with the fact that each leaf node is marked with a unique document, we have the inequality $\sum_{i=1}^{z}\delta_i\leq|L(x\backslash y)|<2g_{h-1}$. Therefore, δ_i for all $i\in[1..z]$ can be encoded using a bit vector $B'''=10^{\delta_1}10^{\delta_2}10^{\delta_3}\ldots10^{\delta_z}$ of length at most $2g_{h-1}+z$ with z 1s, in $O(z\lg(g_{h-1}/z))=O(z\lg^{(h)} n)$ bits with constant-time *select* support (Section 2.2).

The decoding algorithm is as follows: compute the f'_is for $i=1\ldots z$ in the ascending order of i. For $i\leq|D_{old}|$, f'_i is given by the frequency associated with the $select_{B'}(i)$th document (which is same as T_{d_i}) in $F(y,z)$. This takes only $O(z)$ time as the number of constant-time *select* operations is $O(z)$, and we have constant-time access to any element and frequency in $F(y,z)$. Next, $\mu=\min\{\mathrm{tf}(y,d), d\in F(y,z)\}$ can be obtained by scanning the list $F(y,z)$ once. Thus all the f'_is are computed in $O(z)$ time. Next we decode each δ_i and add it to f'_i to obtain f_i, for $i=1\ldots z$ in $O(z)$ time, where $\delta_i=select_{B'''}(i)-select_{B'''}(i-1)-1$ is computed in $O(1)$ time. Thus the space for maintaining the frequencies is $O(z\lg^{(h)} n)$ bits and the time for decoding them is $O(z)$.

Adding over the h levels, the total space is $s_h=s_{h-1}+O(n/\lg^{(h)} n)=O(n/\lg^{(h)} n)$ bits and the total decoding time is $t_h=t_{h-1}+O(z)=O(zh)$ (note that $s_1=O(n/\lg n)$ and $t_1=O(z)$). This completes the proof of Lemma 4. Now we have the main ingredient to describe the complete solution, in the next section.

5. Completing the picture

Let $\pi\in[1..\lg^* n)$ be an integer such that $\lg^{(\pi-1)} n\geq\sqrt{\lg^* n}>\lg^{(\pi)} n$, then $\lg^{(\pi)} n=\omega(1)$ (note that $\pi=\lg^* n-\lg^*\sqrt{\lg^* n}=\Theta(\lg^* n)$). Then, by choosing g_π as the grouping factor, the space s_π is $O(n/\lg^{(\pi)} n)=o(n)$ bits. We main-

[4] Compute the leftmost leaves ℓ_{i_x} and ℓ_{i_y}, respectively, of x and y, then $\ell_{i'}$ is ℓ_{i_x+i-1} if $i_x+i-1<i_y$, and $\ell_{j_y+i-(i_y-i_x)}$ otherwise, where ℓ_{j_y} is the rightmost leaf of y.

tain $\lg D$ such structures corresponding to $z = 1, 2, 4, 8, \ldots, 2^{\lfloor \lg D \rfloor}$, in $o(n \lg D)$ total bits. By combining the space bounds of all the components, we obtain the following lemma.

Lemma 5. *The total space requirement of our data structure is* $|\mathsf{CSA}| + n \lg D + o(n \lg D)$ *bits.*

The next lemma gives the total time to extract the sampled results and hints how we will use them.

Lemma 6. *Given any node* $v \in \mathsf{GST}$ *and an integer* k*, our data structure can report the list* $F(u, k)$ *in* $O(k \lg^* n)$ *time, where* u *is a node in the subtree of* v *with* $|L(v \backslash u)| = O(k\sqrt{\lg^* n})$.

Proof. As the first step, round k to $z = 2^{\lceil \lg k \rceil}$, which is the next power of 2. Then identify the highest node u, in the subtree of v, that is marked with respect to the grouping factor g_π: Let ℓ_i and ℓ_j be the leftmost and rightmost leaves of $L(v)$, then $u = lca(\ell_{i'}, \ell_{j'})$ where $i' = g_\pi \cdot \lceil i/g_\pi \rceil$ and $j' = g_\pi \cdot \lfloor j/g_\pi \rfloor$ (there is no u if $i' \geq j'$). This takes constant time on our representation of the GST topology.

Since $g_\pi = z \lg^{(\pi)} n < z\sqrt{\lg^* n}$, it holds $|L(v \backslash u)| = O(g_\pi) = O(z \lg^{(\pi)} n) = O(k\sqrt{\lg^* n})$. As $u \in \tau_{g_\pi}$, the list $F(u, z)$ can be decoded in time $t_\pi = O(z\pi) = O(k \lg^* n)$ from the precomputed lists (from Lemma 4). The final $F(u, k)$ can be obtained by selecting the k highest frequencies in $F(u, z)$, in $O(z) = O(k)$ time just as in Section 3.2. In case u does not exist, we report $F(u, k) = \emptyset$, and even in such a case the inequality $|L(v)| < 2g_\pi$ is guaranteed. □

The construction is similar to the process described in Section 3.3, except that now, for each z (k) value, we build $O(\lg^* n)$ structures for different grouping factors. The construction time then grows slightly, to $O(n \lg^2 n \lg^* n)$.

5.1. Query answering

The query answering algorithm consists of the following steps:

1. Find the locus node v of the input pattern P in GST by first obtaining the suffix range $[sp..ep]$ of P using CSA in $t_{\mathsf{search}}(p)$ time, and then computing the lowest common ancestor v of ℓ_{sp} and ℓ_{ep} in $O(1)$ time.
2. Using Lemma 6, find the node u in the subtree of v, where $|L(v \backslash u)| = O(k\sqrt{\lg^* n})$ and retrieve the list $F(u, k)$ in $O(k \lg^* n)$ time.
3. Every document d in the final output $F(v, k)$ must either belong to $F(u, k)$, or it must be that $d = \mathsf{DA}[i]$ for some leaf $\ell_i \in L(v \backslash u)$. Let us call S_{cand} the union of both sets of candidate documents. Then we compute $\mathsf{tf}(v, d)$ of each document $d \in S_{cand}$.
4. Report k documents in S_{cand} with the highest $\mathsf{tf}(v, r)$ value in time $O(|S_{cand}|) = O(k\sqrt{\lg^* n})$, just as in Section 3.2.

The overall time for Steps 1, 2, and 4 is $O(t_{\mathsf{search}}(p) + k \lg^* n)$. In the remaining part of this section we show how to handle Step 3 efficiently as well, for the documents $d = \mathsf{DA}[i]$ we find in $L(v \backslash u)$. Note that $\mathsf{tf}(v, d)$ can be computed as $rank_{\mathsf{DA}}(d, ep) - rank_{\mathsf{DA}}(d, sp - 1)$ using two *rank* queries on the document array, but those *rank* queries are expensive. Instead, we use a more sophisticated scheme where only the faster *select*, *access*, and partial *rank* queries are used. This is described next.

5.2. Computing scores online

Firstly, we construct a supporting structure, *SUP*, in $O(k \lg^* n)$ time and occupying $o(n \lg D) + O(z \lg n)$ bits, capable of answering the following query in $O(\lg \lg^* n)$ time: for any given d, return $\mathsf{tf}(u, d)$ if $d \in F(u, k)$, otherwise return -1. Let $\Delta = \Theta(\lg^* n)$, then structure *SUP* is a forest of D/Δ balanced binary search trees $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{D/\Delta}$. Initially each $\mathcal{T}_i$ is empty, hence the initial space is $O(\lg n)$ bits per tree (for maintaining a pointer to the location where it is stored), adding up to $O((D/\Delta) \lg n) = o(n \lg D)$ bits, which we consider a part of the index. Next we shall insert each document $d \in F(u, k)$, along with its associated frequency, into tree $\mathcal{T}_{\lceil d/\Delta \rceil}$ of *SUP*. The size of each search tree can grow up to Δ, hence the total insertion time is $O(k \lg \Delta)$. These insertions will increase the space of *SUP* by $O(k \lg n)$ bits, which can be justified as it is the order of the output size. Now we can search for any d in $\mathcal{T}_{d/\Delta}$ and, if $d \in F(u, k)$, we will retrieve $\mathsf{tf}(u, d)$ in $O(\lg \Delta)$ time. Once we finish Step 3, these binary search trees can be set back to their initial empty state by visiting each document $d \in F(u, k)$ and deleting it from the corresponding tree in total $O(k \lg \Delta)$ time. This does not impact the total asymptotic query processing time.

An outline of Step 3 follows: We scan each leaf $\ell_i \in L(v \backslash u)$, and compute $\mathsf{tf}(v, \mathsf{DA}[i])$. Note that there can be many leaves in $L(v \backslash u)$ marked with the same document, but we compute $\mathsf{tf}(v, d)$ of a document d only once (i.e., when we encounter it for the first time). After this, we also scan the documents $d \in F(u, k)$ and compute $\mathsf{tf}(v, d)$ if we have not considered this document in the previous step. However, the scanning of leaves is performed in a carefully chosen order. Let $\ell_{sp'}$ and $\ell_{ep'}$ be the leftmost and rightmost leaves in $L(u)$, and $B[1..D]$ be a bit vector initialized to all 0s (its size is D bits and can be considered a part of the index). A detailed description of Step 3 follows:

3.1 Start scanning the leaves ℓ_i for $i = sp, sp+1, \ldots, sp'-1$, in the *ascending* order of i, then for $i = ep, ep-1, \ldots, ep'+1$, in the *descending* order of i, and do the following: if $B[\mathsf{DA}[i]] = 0$, then set it to 1, compute $\mathsf{tf}(v, \mathsf{DA}[i])$, and store the result $(\mathsf{DA}[i], \mathsf{tf}(v, \mathsf{DA}[i]))$ for Step 4. Note that each time we compute $\mathsf{tf}(v, \mathsf{DA}[i])$, i is either the first or the last occurrence of $\mathsf{DA}[i]$ in $\mathsf{DA}[sp..ep]$. Assume it is the first (the other case is symmetric). We use a constant-time partial *rank* query, $x = rank_{\mathsf{DA}}(\mathsf{DA}[i], i)$. Then, by performing successive $select_{\mathsf{DA}}(\mathsf{DA}[i], j)$ queries for $j = x+1, x+2, \ldots, y$, where $select_{\mathsf{DA}}(\mathsf{DA}[i], y) > ep \geq select_{\mathsf{DA}}(\mathsf{DA}[i], y-1)$, we compute $\mathsf{tf}(v, \mathsf{DA}[i]) = y - x$. The number of *select* queries required is precisely $y - x = \mathsf{tf}(v, \mathsf{DA}[i])$, which can be further reduced as follows:

- If $\mathsf{DA}[i] \in F(u,k)$, retrieve $\mathsf{tf}(u, \mathsf{DA}[i])$ from *SUP* in time $O(\lg\lg^* n)$. As we know that $\mathsf{tf}(u, \mathsf{DA}[i]) \leq \mathsf{tf}(v, \mathsf{DA}[i])$, we start *select* queries from $j = x + \mathsf{tf}(u, \mathsf{DA}[i])$, so the number of *select* queries used to find y is reduced to $\mathsf{tf}(v, \mathsf{DA}[i]) - \mathsf{tf}(u, \mathsf{DA}[i]) = \mathsf{tf}(L(v\backslash u), \mathsf{DA}[i])$, that is, the number of leaves in $L(v\backslash u)$ marked with $T_{\mathsf{DA}[i]}$ (note here we are extending of the tf notation to leaf sets, with the obvious meaning).
- If $\mathsf{DA}[i] \notin F(u,k)$, compute $x' = select_{\mathsf{DA}}(\mathsf{DA}[i], x + \mu - 1)$, where we remind that $\mu = \min\{\mathsf{tf}(u,d), d \in F(u,k)\}$. If $x' > ep$, we conclude that $\mathsf{tf}(v, \mathsf{DA}[i]) < \mu$, and hence $\mathsf{DA}[i]$ can be discarded from being a candidate for the final output. On the other hand, if $x' \leq ep$, the *select* queries can be started from $j = x + \mu$, which reduces the number of *select* queries to $\mathsf{tf}(v, \mathsf{DA}[i]) - \mu \leq \mathsf{tf}(L(v\backslash u), \mathsf{DA}[i])$ (since $\mathsf{DA}[i] \notin F(u,k)$, it holds $\mathsf{tf}(u, \mathsf{DA}[i]) \leq \mu$).

The query time for executing this step can be analyzed as follows: for each i, we perform a query on *SUP*. The computation of $\mathsf{tf}(v, \mathsf{DA}[i])$ requires at most $\mathsf{tf}(L(v\backslash u), \mathsf{DA}[i])$ *select* queries. As we do this computation only once per distinct document, the total number of *select* queries is at most $\sum_r \mathsf{tf}(L(v\backslash u), r) = |L(v\backslash u)|$. By choosing the cost $f(n, D) = \sqrt{\lg^* n}$ for *select* queries, the total time is $O(|L(v\backslash u)|\,(f(n,D) + \lg\lg^* n)) = O(k \lg\lg^* n)$.

3.2 Now scan the documents $d \in F(u,k)$. If $B[d] = 0$, then there exists no leaf in $L(v\backslash u)$ marked with T_d. Thus $\mathsf{tf}(v,d) = \mathsf{tf}(u,d)$ and the pair $(d, \mathsf{tf}(u,d))$ is stored for Step 4. If $B[d] = 1$ then T_d has already been dealt with in the previous pass. The time for accessing $\mathsf{tf}(u,d)$ using *SUP* is $O(\lg\lg^* n)$, hence this step takes $O(k \lg\lg^* n)$ time.

3.3 Reset B to its initial state (all bits set to 0) for supporting queries in future. By revisiting the leaves in $L(v\backslash u)$ and the list $F(u,k)$, we can exactly find out those locations in B where the corresponding bit is 1. The time for this step can be bounded by $O(|L(v\backslash u)| + k) = O(k\sqrt{\lg^* n})$.

Thus the time for Step 3 is $O(k \lg^* n)$, and the result follows.

Theorem 2. *The top-k most frequent documents problem, on a collection of length n, for a pattern of length p, can be solved using* $|\mathsf{CSA}| + n \lg D + o(n \lg D)$ *bits and in near-optimal* $O(t_{\mathsf{search}}(p) + k \lg^* n)$ *time. Here* CSA *is a compressed suffix array over the collection and* $t_{\mathsf{search}}(p)$ *is the time* CSA *takes to find the suffix array interval of the pattern. The documents are delivered in arbitrary order.*

By considering a particular CSA [13] we have the following corollary.

Corollary 3. *The top-k most frequent documents problem can be solved using* $nH_h + n \lg D + o(nH_h + n \lg D)$ *bits and in* $O(p + k \lg^* n)$ *time.*

6. Reducing the time to $O(p + k \lg^* k)$

Note that, when p or k is at least $\lg\lg n$, it already holds $O(t_{\mathsf{search}}(p) + k \lg^* n) = O(t_{\mathsf{search}}(p) + k \lg^* k)$ for any $t_{\mathsf{search}}(p) \geq p$. Therefore, we now concentrate on the case when $\max(p,k) < \lg\lg n$. We use the following result.

Lemma 7. *(See [38].) Given a fixed* κ*, an array* $A[1..n]$ *of n indices can be indexed in* $O(n \lg^2 \kappa)$ *bits for answering the following query in* $O(k)$ *time, without accessing A and for any* $1 \leq k \leq \kappa$*: given i, j, and k, output the positions of the k highest elements in* $A[i..j]$.

Let S_δ be the set of nodes in GST with node depth equal to δ. We start with the description of an $O(n \lg^2 \kappa)$-bit structure for a fixed $\kappa = \lg\lg n$ and a fixed $\delta < \lg\lg n$, for answering top-k queries for any $1 \leq k \leq \kappa$ and those patterns with their locus node belonging to S_δ. First, we construct an array $A[1..n]$ (with all its elements initialized to zero) as follows: For $i = 1 \ldots n$, if the first occurrence of document $\mathsf{DA}[i]$ in $\mathsf{DA}[a..b]$ is at position i, where $[a..b]$ is the suffix range corresponding to a node $u \in S_\delta$, then set $A[i] = \mathsf{tf}(u, \mathsf{DA}[i])$. We do not store this array explicitly, instead we maintain the structure of Lemma 7 over it, requiring $O(n \lg^2 \kappa)$ bits space. Now the list of documents $F(u,k)$ for any locus node $u \in S_\delta$ can be reported in $O(k)$ time as follows: First perform a top-k query on the structure of Lemma 7 with the suffix range $[sp..ep]$. The output will be a set of k locations $j_1, j_2, \ldots, j_k \in [sp..ep]$, and then the identifiers of the top-k documents are $\mathsf{DA}[j_1], \mathsf{DA}[j_2], \ldots, \mathsf{DA}[j_k]$. By maintaining similar structures for all the $\delta \in [1..\lg\lg n)$, any such top-k query with $p < \lg\lg n$ can be answered in $O(t_{\mathsf{search}}(p) + k)$ time. The additional space required is $o(n(\lg\lg n)^3)$ bits, which can be bounded by $o(n \lg\sigma)$ bits if, say, $\lg\sigma \geq \sqrt{\lg n}$. Otherwise, we shall explicitly maintain the top-κ documents corresponding to all patterns of length at most $\lg\lg n$, in decreasing frequency order, using a table of $O(\sigma^{\lg\lg n} \lg\lg n \lg D) = o(n)$ bits. The query time in this case is just $O(k)$.

Thus, by combining the cases, we achieve $O(t_{\mathsf{search}}(p) + k \lg^* k)$ query time.

Theorem 3. *The top-k most frequent documents problem, on a collection of length n, for a pattern of length p, can be solved using* $|\mathsf{CSA}| + n \lg D + o(n \lg \sigma + n \lg D)$ *bits and in near-optimal* $O(t_{\mathsf{search}}(p) + k \lg^* k)$ *time. Here* CSA *is a compressed suffix array over the collection and* $t_{\mathsf{search}}(p)$ *is the time* CSA *takes to find the suffix array interval of the pattern. The documents are delivered in arbitrary order.*

Once again, we obtain a result very close to optimal by considering a particular CSA [13].

Corollary 4. *The top-k most frequent documents problem can be solved using* $nH_h + n \lg D + o(n \lg \sigma + n \lg D)$ *bits and in* $O(p + k \lg^* k)$ *time.*

The construction time of the structure of Lemma 7 is $O(n \lg^2 \kappa)$ plus the time to sort the values, which in this case is $O(n)$ because term frequencies are in $[1..n]$. We build $\lg \lg n$ such structures with $\kappa = \lg \lg n$, so the total extra construction time is $o(n(\lg \lg n)^3)$ time. The structure for the case $\lg \sigma < \sqrt{\lg n}$ can be built in time $O(\sigma^{\lg \lg n} \lg \lg n \lg^2 n) = o(n)$, using the data structure [21] analyzed in Section 3.3. Therefore the construction time $O(n \lg^2 n \lg^* n)$ of the main structure dominates.

7. An index for top-*k* most important documents

We now consider the case where each document has a fixed importance value. For simplicity, assume we reassign document identifiers so that document T_d is the dth most important one. Then, after identifying the interval $\mathsf{DA}[sp..ep]$ for a pattern P, the problem is to report the k distinct smallest document identifiers in $\mathsf{DA}[sp..ep]$.

A property that simplifies the problem is that it is decomposable: We can partition the query range $\mathsf{DA}[sp..ep]$ into two, $\mathsf{DA}[sp..mp]$ and $\mathsf{DA}[mp+1..ep]$, obtain the k smallest identifiers from each half (in arbitrary order), and then merge them to obtain in $O(k)$ further time the k smallest identifiers in $\mathsf{DA}[sp..ep]$. We first explain how to do this merging and then our data structure.

Merging two results sets Assume we have two result sets S_1 and S_2 of size $O(k)$. We maintain a bit vector $V[1..D]$ as a part of our data structure, with $V[d] = 0$ for all d. Then we traverse S_1 and S_2, setting $V[d] \leftarrow 1$ for any $d \in S_1 \cup S_2$. Now we prepare an empty list S, which will contain $S_1 \cup S_2$ without duplicates. We traverse again S_1 and S_2. For each $d \in S_1 \cup S_2$, if $V[d] = 1$, we add d to S and set $V[d] \leftarrow 0$, otherwise we do nothing. At the end S is computed and V is restored to all 0s.

Now we run over S the procedure described near the end of Section 3.2, for selecting the kth element in S in $O(k)$ time and, after knowing the kth smallest identifier θ, collecting all the identifiers smaller than θ from S. The whole process takes $O(k)$ time.

A blocking scheme We virtually partition $\mathsf{DA}[1..n]$ into contiguous blocks of length b, for $b = 1, 2, 4, 8, \ldots, 2^{\lfloor \lg n \rfloor}$. We associate each cell $\mathsf{DA}[i]$ with two values, $C_{prev}[i]$ and $C_{next}[i]$, both in $[1, \lg n]$, so that $C_{prev}[i]$ (respectively, $C_{next}[i]$) is the size of the smallest block containing i and the previous (respectively, next) position j such that $\mathsf{DA}[j] = \mathsf{DA}[i]$.

Now, let 2^h be the size of the smallest block containing $\mathsf{DA}[sp..ep]$, thus there is an mp such that $\mathsf{DA}[sp..mp]$ and $\mathsf{DA}[mp+1..ep]$ are contained in two consecutive blocks of size 2^{h-1}. By the discussion above, it is sufficient to solve top-k queries on those two subranges, that is, to consider only suffixes or prefixes of blocks. We will describe the case of suffixes; prefixes are handled similarly.

Notice that, for any particular document in $\mathsf{DA}[sp..mp]$, there will be exactly one occurrence $i \in [sp..mp]$ (the rightmost) with $C_{next}[i] \geq h$. All the previous ones, $sp \leq j < i$, will have their next occurrences within $[sp..mp]$, which is contained in a block of size 2^{h-1}, and thus $C_{next}[j] \leq h - 1$.

Therefore, the problem can be rephrased as follows: among all those elements $i \in [sp..mp]$ with $C_{next}[i] \geq h$, report k with smallest $\mathsf{DA}[i]$ values. The condition $C_{next}[i] \geq h$ gets rid of possible duplicates.

A geometric problem Consider n two-dimensional points (x_i, y_i) on a short grid of $n \times m$, for $m = O(\mathrm{polylog}\, n)$, and point weights $w_i \in [1..D]$. Queries specify a range $[x_1..x_2] \times [y_1..y_2]$ and an integer k, and the task is to return the k heaviest points in the range. While this problem has been studied [8,39], we show now a different result that is useful for thin grids, which might be of independent interest.

Assume the points are sorted by x-coordinate and that there is exactly one column per point in the grid, so that $x_i = i$ (this simplification is standard and other cases are easily mapped to this one within $O(n)$ extra bits and constant time). Then the grid will be represented via two sequences, $Y[1..n] = y_1 y_2 \ldots y_n$ and $W[1..n] = w_1 w_2 \ldots w_n$.

We will represent W in plain form, whereas Y will be represented with the alternative variant of Lemma 1, so that $select_Y$ is supported in constant time. Note that $rank_Y$ is also supported in constant time because the alphabet size is $\sigma = m = O(\mathrm{polylog}\, n)$, thus $O(\lg \frac{\lg \sigma}{\lg \lg n}) = O(1)$. The total space is $n \lg D + n \lg m + o(n \lg m)$ bits.

The representation of Y is used to define virtual sequences W^y, for $y \in [1..m]$, where $W^y[i] = W[select_Y(y, i)]$ is the ith value in W with $Y[\cdot] = y$. Note that access to any W^y is simulated in constant time.

Then our geometric search can be decomposed into $y_2 - y_1 + 1$ subproblems on $W^y[x_1^y..x_2^y]$, where $x_1^y = rank_Y(y, x_1 - 1) + 1$ and $x_2^y = rank_Y(y, x_2)$, for $y \in [y_1..y_2]$. Consider the (virtual) concatenation $A[1..n] = W^1 \cdot W^2 \cdots W^m$, and the following lemma.

Lemma 8. *(See [7].) We can preprocess an array $A[1..n]$ in linear time and associate to it a data structure of $2n + o(n)$ bits, such that given a set of z non-overlapping ranges $[L_1..R_1], [L_2..R_2], \ldots, [L_z..R_z]$, we can find the smallest k numbers in $A[L_1..R_1] \cup A[L_2..R_2] \cup \cdots \cup A[L_z..R_z]$ (in unsorted order) by performing $O(z + k)$ operations $access_A$.*

Thus, by defining $[L_y..R_y] = [x_1^y..x_2^y]$, we obtain the desired result in $O(k + (y_2 - y_1))$ operations $access_A$, which we transform in constant time into accesses to W. Therefore we obtain the following result.

Lemma 9. *A set of n points on an $n \times m$ grid, with weights in $[1..D]$, where $m = O(\text{polylog}\, n)$, can be represented using $n \lg m(1 + o(1)) + n \lg D$ bits such that, given a query range $[x_1..x_2] \times [y_1..y_2]$ and an integer k, the k heaviest points in the range are obtained in time $O(k + (y_2 - y_1))$.*

The final solution for large k Let $(x_i, y_i) = (i, C_{next}[i])$ be points on a grid $n \times \lg n$, with weights $w_i = D + 1 - \mathsf{DA}[i]$ in $[1..D]$; and the query $[sp..mp] \times [h..\lg n]$. Then we can apply Lemma 9 directly, and obtain the solution to our top-k most important documents problem. The only difference is that we do not have W stored directly, but must obtain $\mathsf{DA}[i]$ via the CSA, in time t_{SA}. Therefore, the time we obtain is $O(t_{\mathsf{search}}(p) + t_{\mathsf{SA}}(k + \lg n))$, and the space is $|\mathsf{CSA}| + n \lg\lg n + o(n \lg\lg n)$ bits. This is already the desired time unless $k < \lg n$; we deal with this case in the sequel.

Structure for $k < \lg n$ We define a set of blocks on top of $\mathsf{DA}[1..n]$ to provide for this case, extending a solution for range minimum queries [40] so that we return up to $\lg n$ range minima. Choose block size $\ell = \lg^3 n$. For all values $i \in [1..n/\ell]$ and $b \in [0..\lg(n/\ell)]$, we store in $F[i, b]$ the smallest $\lg n$ document identifiers in $\mathsf{DA}[i \cdot \ell..(i + 2^b) \cdot \ell - 1]$, in order. This adds up to $O((n/\ell) \cdot \lg n \cdot \lg n \lg D) = O(n)$ bits.

Given a query range $[sp..ep]$, we first compute the maximal range of ℓ-aligned blocks contained in $[sp..ep]$, $sp' = \lceil sp/\ell \rceil$ and $ep' = \lfloor (ep + 1)/\ell \rfloor - 1$. Now we compute the appropriate block length, $b = \lfloor \lg(ep' - sp' + 1) \rfloor$. Then, the area $\mathsf{DA}[sp'\ell..(ep' + 1)\ell - 1]$ is covered by two (possibly overlapping) areas $\mathsf{DA}[sp'\ell..(sp' + 2^b)\ell - 1]$ and $\mathsf{DA}[(ep' - 2^b + 1)\ell..(ep' + 1)\ell - 1]$. Any top-$k$ value in $\mathsf{DA}[sp'\ell..(ep' + 1)\ell - 1]$ must also be top-k in one of those areas (or in both). Thus the set of k smallest identifiers in $\mathsf{DA}[sp'\ell..(ep' + 1)\ell - 1]$ is obtained in $O(k)$ time by merging the first k elements of lists $F[sp', b]$ and $F[ep' - 2^b + 1]$ (there may be repeated identifiers among the two lists).

This leaves us with up to two smaller ranges, of sizes less than ℓ, at the extremes of the query range $[sp..ep]$. Those are handled by storing, for each block of size ℓ, the optimal-time structure of Karpinski and Nekrich [26]. For this sake, we remap the weights (i.e., the document identifiers) to the range $[1..\ell]$ while respecting their relative ordering. We also store a map from the weight assigned to each document d to a local offset i within the block where d appears, $\mathsf{DA}[\ell \cdot r + i] = d$. The mappings add up to $O(\ell \lg \ell)$ bits. Since the universe of weights within a block is of size ℓ, the main data structure [26] takes $O(\ell \lg \ell)$ bits as well. Thus the total space adds up to $O(n \lg \ell) = O(n \lg\lg n)$ bits.

The $O(k)$ candidates returned by this data structure in $O(k)$ time are then converted into document identifiers: first they are converted into local offsets i using the local mapping, and then into document identifiers $\mathsf{DA}[\ell \cdot r + i] = d$ in t_{SA} time. Overall, we obtain top-k values from the tails of the interval $[sp..ep]$ in $O(k t_{\mathsf{SA}})$ time, and finally merge them with those of the central part of the interval (if it is nonempty) in $O(k)$ further time. We have obtained our final result.

Theorem 4. *The top-k most important documents problem, on a collection of length n, for a pattern of length p, can be solved using $|\mathsf{CSA}| + O(n \lg\lg n)$ bits and in $O(t_{\mathsf{search}}(p) + k t_{\mathsf{SA}})$ time. Here CSA is a compressed suffix array over the collection, $t_{\mathsf{search}}(p)$ is the time CSA takes to find the suffix array interval of the pattern, and t_{SA} is the time it takes to retrieve any suffix array cell. The documents are delivered in arbitrary order.*

The construction of this data structure requires $O(n)$ time to build C_{prev} and C_{next}, then $O(n \lg\lg\lg n)$ time to build the representation of Y (as it uses Lemma 1, recall Section 3.3), $O((n/\ell) \lg(n/\ell) \lg n) = o(n)$ to compute table F (as $\ell = \lg^3 n$), and $O((n/\ell)\ell \lg \ell) = O(n \lg\lg n)$ to build the structures of Karpinski and Nekrich [26]. Thus the overall time is $O(n \lg\lg n)$ plus the time to build the CSA of choice.

8. Reducing space

Compared to the solution of Gagie et al. [27], the space used in Theorem 4 is not satisfactory for small enough $\lg D = O(\lg\lg n)$. In this section we show how the space can be further reduced, at some expense in time.

While array DA is represented through the CSA, we spend $O(n \lg\lg n)$ bits in representing C_{prev} and C_{next}. With some effort, these arrays can also be expressed in terms of DA: $C_{next}[i]$ is a function of the first occurrence of $\mathsf{DA}[i]$ in $\mathsf{DA}[i+1..n]$, and C_{prev} of the last occurrence in $\mathsf{DA}[1..i-1]$. These are easily computed if we have support for $rank_{\mathsf{DA}}$ and $select_{\mathsf{DA}}$; note however that we do not have DA in explicit form, so we cannot directly use Lemma 1. Instead, we use the following result.

Lemma 10. *(See [41].) Let $S[1..n]$ be a string over an alphabet of size $\sigma = O(\text{polylog}\, n)$. With $O(n \lg\lg \sigma + (n \lg \sigma)/t)$ bits on top of S, for any $1 \le t \le \sigma$, one can support $rank_S$ and $select_S$ in time dominated by $O(t)$ accesses to S.*

We apply the lemma to string DA, whose alphabet size is $D = O(\text{polylog}\, n)$ and whose access cost is t_{SA}. For example, if we choose $t = \lg\lg D$, we need $O(n \lg D / \lg\lg D)$ bits on top of DA and can support $rank_{\mathsf{DA}}$ and $select_{\mathsf{DA}}$ in time $O(t_{\mathsf{SA}} \lg\lg D)$. Therefore, in this time we provide access to C_{next} and C_{prev} without storing them explicitly.

These arrays correspond to array Y in Lemma 9. We need, however, to support $rank_Y$ and $select_Y$. We can use Lemma 10 once again, this time over Y (whose alphabet size is $\lg n$). By choosing, for example, $t = \lg\lg n / \lg\lg D$, we have $O(n \lg\lg\lg n + n \lg\lg D)$ bits of space and carry out operations $rank_Y$ and $select_Y$ in $O(t)$ accesses to Y, that is, in time $O(t_{\mathsf{SA}} \lg\lg D \cdot (\lg\lg n / \lg\lg D)) = O(t_{\mathsf{SA}} \lg\lg n)$. The total space is $|\mathsf{CSA}| + O(n \lg D / \lg\lg D + n \lg\lg\lg n)$ bits.

Now, using Lemma 9 on this representation we obtain time $O(t_{\mathsf{search}}(p) + t_{\mathsf{SA}}(k + \lg n) \lg\lg n)$. This time we cannot use the special solution for $k < \lg n$, because it needs much space.

Corollary 5. *For $D = O(\text{polylog}\, n)$, the top-$k$ most important documents problem can be solved using $|\mathsf{CSA}| + O(n \lg D / \lg\lg D + n \lg\lg\lg n)$ bits and in $O(t_{\mathsf{search}}(p) + t_{\mathsf{SA}}(k + \lg n) \lg\lg n)$ time.*

A result with the minimum space we can obtain is achieved by choosing $t = \lg D / \lg\lg\lg n$ for DA. The space becomes $O(n \lg\lg\lg n)$, but the time to access Y is $O(t_{\mathsf{SA}} \lg D / \lg\lg\lg n)$. Then we can choose $t = \lg\lg n / \lg\lg\lg n$ to represent Y, reaching $O(n \lg\lg\lg n)$ bits of space and $O(t_{\mathsf{SA}} \lg D \lg\lg n / (\lg\lg\lg n)^2) = O(t_{\mathsf{SA}} (\lg\lg n)^2)$ time.

Corollary 6. *For $D = O(\text{polylog}\, n)$, the top-$k$ most important documents problem can be solved using $|\mathsf{CSA}| + O(n \lg\lg\lg n)$ bits and in $O(t_{\mathsf{search}}(p) + t_{\mathsf{SA}}(k + \lg n)(\lg\lg n)^2)$ time.*

The additional construction time required by Lemma 10 is dominated by the construction of $O(n/\sigma)$ monotone minimum perfect hash functions (mmphf) on $O(\sigma)$ elements over universe $[1..\sigma]$, as well as other structures that require perfect hash functions on $O(\sigma / \lg\sigma)$ elements over universe $[1..\sigma]$. The mmphfs are not allowed to use $\sigma \lg\sigma$ bits, and as a consequence their construction time dominates. The only deterministic construction time known for this case is $O(\sigma \lg^5 \sigma)$ [42]. This adds up to $O(n \lg^5 \sigma) = O(n(\lg\lg n)^5)$ extra time. This may dominate the construction time of the general structure, but it is still low, for example it is $o(n \lg n)$.

9. Conclusions

We have obtained important results about the relation between time and space for the most popular document retrieval problems on general string collections. For top-k most frequent document retrieval, which had been solved in $O(p + k)$ and $O(n(\lg\sigma + \lg D))$ bits (with a large constant) [8], we have shown that it is possible to obtain the almost optimal time $O(p + k \lg^* k)$ time within $n(\lg\sigma + \lg D)(1 + o(1))$, and even less on compressible text collections. On the other hand, we have shown that within the asymptotically optimal space of the compressed data plus $o(n)$ bits, it is possible to solve the problem in time $O(p + k \lg^2 k \lg^{1+\varepsilon} n)$. Both results outperform the current state of the art [17,16,18] by a significant margin (e.g., $O(\lg n / \lg k)$ times faster than the previous structure that uses optimal space [18]) and get very close to answering the question of which are the optimal space/time tradeoffs.

Similarly, for top-k most important document retrieval, we have obtained a data structure with the optimal time $O(t_{\mathsf{search}}(p) + k t_{\mathsf{SA}})$ given that the access to the documents must be carried out through a CSA. Its extra space on top of the CSA is $O(n \lg\lg n)$ bits, away from the asymptotically optimal $o(n)$. For a polylogarithmic number of documents, we reduce the extra space up to $O(n \lg\lg\lg n)$ bits, with a moderate increase in time. The only structure reaching optimal space [15] is slower than ours by an $O(\lg k \lg^{\varepsilon} n)$ factor.

Natural questions on the top-k most frequent problem are whether optimal time can be obtained within $n(\lg\sigma + \lg D)(1 + o(1))$ bits of space, or even less (although the latter seems unlikely), or prove a fundamental limit. Similarly, it is natural to ask which is the best time that can be obtained within optimal space. We believe it might be possible to reduce our $O(p + k \lg^2 k \lg^{1+\varepsilon} n)$ by a $\lg k$ factor, but the other seems to be an unavoidable price to allow for k to be chosen at query time. Once again, it would be interesting to know which is the actual lower bound. Similar questions can be posed on the top-k most important problem, which has been less studied, and on even more open problems, such as top-k proximity search (i.e., find the k documents where two occurrences of P appear closest to each other).

Finally, practical developments should follow these theoretical results. As usual, a good deal of algorithm engineering will probably be needed to obtain competitive practical performance, retaining some aspects of the theoretical ideas and replacing others.

References

[1] S. Büttcher, C.L.A. Clarke, G. Cormack, Information Retrieval: Implementing and Evaluating Search Engines, MIT Press, 2010.
[2] G. Navarro, Spaces, trees and colors: the algorithmic landscape of document retrieval on sequences, ACM Comput. Surv. 46 (4) (2014), article 52.
[3] P. Weiner, Linear pattern matching algorithm, in: Proc. 14th Annual IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.

[4] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, SIAM J. Comput. 22 (5) (1993) 935–948.
[5] W.-K. Hon, R. Shah, S.-B. Wu, Efficient index for retrieving top-*k* most frequent documents, in: Proc. 16th Symposium on String Processing and Information Retrieval (SPIRE), 2009, pp. 182–193.
[6] W.-K. Hon, M. Patil, R. Shah, S.-B. Wu, Efficient index for retrieving top-*k* most frequent documents, J. Discrete Algorithms 8 (4) (2010) 402–417.
[7] W.-K. Hon, R. Shah, J.S. Vitter, Space-efficient framework for top-*k* string retrieval problems, in: Proc. 50th IEEE Annual Symposium on Foundations of Computer Science (FOCS), 2009, pp. 713–722.
[8] G. Navarro, Y. Nekrich, Top-*k* document retrieval in optimal time and linear space, in: Proc. 23rd Annual ACM–SIAM Symposium on Discrete Algorithms (SODA), 2012, pp. 1066–1078.
[9] R. Shah, C. Sheng, S.V. Thankachan, J.S. Vitter, Top-*k* document retrieval in external memory, in: Proc. 21st Annual European Symposium on Algorithms (ESA), in: LNCS, vol. 8125, 2013, pp. 803–814.
[10] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Comput. Surv. 39 (1) (2007), article 2.
[11] G. Manzini, An analysis of the Burrows–Wheeler transform, J. ACM 48 (3) (2001) 407–430.
[12] J. Barbay, T. Gagie, G. Navarro, Y. Nekrich, Alphabet partitioning for compressed rank/select and applications, in: Proc. 21st Annual International Symposium on Algorithms and Computation (ISAAC), in: LNCS, vol. 6507, 2010, pp. 315–326 (part II).
[13] D. Belazzougui, G. Navarro, Alphabet-independent compressed text indexing, in: Proc. 19th Annual European Symposium on Algorithms (ESA), 2011, pp. 748–759.
[14] T. Gagie, J. Kärkkäinen, G. Navarro, S.J. Puglisi, Colored range queries and document retrieval, Theoret. Comput. Sci. 483 (2013) 36–50.
[15] D. Belazzougui, G. Navarro, D. Valenzuela, Improved compressed indexes for full-text document retrieval, J. Discrete Algorithms 18 (2013) 3–13.
[16] W.-K. Hon, R. Shah, S.V. Thankachan, J.S. Vitter, Faster compressed top-*k* document retrieval, in: Proc. 23rd Data Compression Conference (DCC), 2013, pp. 341–350.
[17] W.-K. Hon, R. Shah, S.V. Thankachan, Towards an optimal space-and-query-time index for top-*k* document retrieval, in: Proc. 23rd Annual Symposium on Combinatorial Pattern Matching (CPM), in: LNCS, vol. 7354, 2012, pp. 173–184.
[18] D. Tsur, Top-*k* document retrieval in optimal space, Inform. Process. Lett. 113 (12) (2013) 440–443.
[19] S. Muthukrishnan, Efficient algorithms for document retrieval problems, in: Proc. 13th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA), 2002, pp. 657–666.
[20] N. Välimäki, V. Mäkinen, Space-efficient algorithms for document retrieval, in: Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM), in: LNCS, vol. 4580, 2007, pp. 205–215.
[21] R. Konow, G. Navarro, Faster compact top-*k* document retrieval, in: Proc. 23rd Data Compression Conference (DCC), 2013, pp. 351–360.
[22] J.S. Culpepper, G. Navarro, S.J. Puglisi, A. Turpin, Top-*k* ranked document search in general text databases, in: Proc. 18th Annual European Symposium on Algorithms (ESA), in: LNCS, vol. 6347, 2010, pp. 194–205 (part II).
[23] M. Patil, S.V. Thankachan, R. Shah, W.-K. Hon, J.S. Vitter, S. Chandrasekaran, Inverted indexes for phrases and strings, in: Proc. 34th International ACM Conference on Research and Development in Information Retrieval (SIGIR), 2011, pp. 555–564.
[24] J.S. Culpepper, M. Petri, F. Scholer, Efficient in-memory top-*k* document retrieval, in: Proc. 35th International ACM Conference on Research and Development in Information Retrieval (SIGIR), 2012, pp. 225–234.
[25] G. Navarro, D. Valenzuela, Space-efficient top-*k* document retrieval, in: Proc. 11th International Symposium on Experimental Algorithms (SEA), in: LNCS, vol. 7276, 2012, pp. 307–319.
[26] M. Karpinski, Y. Nekrich, Top-*k* color queries for document retrieval, in: Proc. 22nd Annual ACM–SIAM Symposium on Discrete Algorithms (SODA), 2011, pp. 401–411.
[27] T. Gagie, G. Navarro, S.J. Puglisi, New algorithms on wavelet trees and applications to information retrieval, Theoret. Comput. Sci. 426–427 (2012) 25–41.
[28] K. Sadakane, G. Navarro, Fully-functional succinct trees, in: Proc. 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA), 2010, pp. 134–149.
[29] J.I. Munro, Tables, in: Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), in: LNCS, vol. 1180, 1996, pp. 37–42.
[30] D. Clark, Compact PAT trees, Ph.D. Thesis, University of Waterloo, Canada, 1996.
[31] R. Raman, V. Raman, S.S. Rao, Succinct indexable dictionaries with applications to encoding *k*-ary trees, prefix sums and multisets, ACM Trans. Algorithms 3 (4) (2007), article 43.
[32] D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, in: Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX), 2007, pp. 60–70.
[33] D. Belazzougui, G. Navarro, New lower and upper bounds for representing sequences, in: Proc. 20th Annual European Symposium on Algorithms (ESA), in: LNCS, vol. 7501, 2012, pp. 181–192.
[34] K. Sadakane, Succinct data structures for flexible text retrieval systems, J. Discrete Algorithms 5 (2007) 12–22.
[35] T.C. Bell, J.G. Cleary, I.H. Witten, Text Compression, Prentice Hall, 1990.
[36] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest, R.E. Tarjan, Time bounds for selection, J. Comput. Syst. Sci. 7 (4) (1973) 448–461.
[37] M. Ružić, Constructing efficient dictionaries in close to sorting time, in: Proc. 35th International Colloquium on Automata, Languages and Programming (ICALP), in: LNCS, vol. 5125, 2008, pp. 84–95 (part I).
[38] R. Grossi, J. Iacono, G. Navarro, R. Raman, S.S. Rao, Encodings for range selection and top-*k* queries, in: Proc. 21st Annual European Symposium on Algorithms (ESA), in: LNCS, vol. 8125, 2013, pp. 553–564.
[39] G. Navarro, Y. Nekrich, L.M.S. Russo, Space-efficient data-analysis queries on grids, Theoret. Comput. Sci. 482 (2013) 60–72.
[40] M. Bender, M. Farach-Colton, The LCA problem revisited, in: Proc. 4th Latin American Theoretical Informatics Symposium (LATIN), in: LNCS, vol. 1776, 2000, pp. 88–94.
[41] R. Grossi, A. Orlandi, R. Raman, Optimal trade-offs for succinct string indexes, in: Proc. 37th International Colloquim on Automata, Languages and Programming (ICALP), 2010, pp. 678–689.
[42] N. Alon, M. Naor, Derandomization, witnesses for boolean matrix multiplication and construction of perfect hash functions, Algorithmica 16 (4–5) (1996) 434–449.