Faster Top-k Document Retrieval in Optimal Space *

Gonzalo Navarro[1] and Sharma V. Thankachan[2]

[1] Department of Computer Science, University of Chile, Chile,
gnavarro@dcc.uchile.cl
[2] Department of Computer Science, Louisiana State University, USA,
thanks@csc.lsu.edu

Abstract. We consider the problem of retrieving the k documents from a collection of strings where a given pattern P appears most often. We show that, by representing the collection using a Compressed Suffix Array CSA, a data structure using the asymptotically optimal $|\mathsf{CSA}| + o(n)$ bits can answer queries in the time needed by CSA to find the suffix array interval of the pattern plus $O(k \lg^2 k \lg^\epsilon n)$ accesses to suffix array cells, for any constant $\epsilon > 0$. This is $\lg n / \lg k$ times faster than the only previous solution using optimal space, $\lg k$ times slower than the fastest structure that uses twice the space, and $\lg^2 k \lg^\epsilon n$ times the lower-bound cost of obtaining k document identifiers from the CSA. To obtain the result we introduce a tool called the *sampled document array*, which can be of independent interest.

1 Introduction

The problem of *top-k document retrieval* is that of preprocessing a text collection so that, given a search pattern $P[1, m]$ 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 [5].

The inverted index has been highly successful to solve those top-k queries in many IR scenarios. However, inverted indexes are bound to text collections that can be easily segmented into "words", so that only whole words can be queried, and the distinct words form a reasonably small set. Inverted indexes store, for each word, the list of the documents where it appears, with the associated relevance. Such a structure is not easily applicable in highly synthetic languages like Finnish or German, where long words are built from particles, and even less in languages where word separators are absent and can only be inferred from the meaning, like Chinese, Korean, etc. Out of resorting to complex segmentation heuristics, a simple solution for those cases is to treat the text as an uninterpreted sequence of symbols and look for any substring in those sequences. The model of a collection of documents (strings) where one can find those where a

* Funded in part by Fondecyt Grant 1-110066.

pattern string is relevant is also appealing in other applications like bioinformatics, chemoinformatics, software repositories, multimedia databases, and so on. Supporting document retrieval queries on those general string collections has proved much more challenging.

Sufix trees [24] and suffix arrays [13] are useful tools to search string collections. However, these structures solve the *pattern matching problem*: they can count or list all the occ individual occurrences of P in the collection. Obtaining the k most relevant documents from that set requires time proportional to occ, usually much much larger than k. Only relatively recently [12, 8, 11, 18, 22] was this problem solved satisfactorily, finally reaching the optimal time $O(m + k)$. Those solutions, like suffix trees, have the drawback of requiring $O(n \lg n)$ bits of space on a collection of length n, whereas the collection itself would require no more than $n \lg \sigma$ bits, if σ is the alphabet size. In practice these indexes require many times the text size, which renders them impractical on moderate and large text collections.

For the pattern matching problem, the space issue began to be solved in year 2000. Recent Compressed Suffix Arrays (CSAs) efficiently answer queries within space asymptotically equal not only to $n \lg \sigma$ bits, but to the size of the compressed text collection [17]. Moreover, those CSAs can retrieve any substring of any document and hence replace the collection: they can be regarded as compressors that support queries. We call their space $|\mathsf{CSA}|$, which can be tought of as the minimum space in which the text collection can be represented.

A similar result for top-k document retrieval has been more elusive. In their seminal paper, Hon et al. [11] showed that, if the relevance is taken as the number of times P appears in the document (a popular choice in IR), the collection can be represented in $2|\mathsf{CSA}| + o(n)$ bits so that queries are solved in time $O(m \lg \lg \sigma + k \lg^{4+\epsilon} n)$, for any constant $\epsilon > 0$ (this complexity assumes that the CSA searches for P in time $O(m \lg \lg \sigma)$ and computes a cell of the suffix array or its inverse in time $O(\lg^{1+\epsilon} n)$; there exists such a CSA achieving high-order entropy compression of the text [1]). After several time improvements that still used $2|\mathsf{CSA}| + o(n)$ bits [6, 3], Hon et al. [10] achieved the best time to date, $O(m \lg \lg \sigma + k \lg k \lg^{1+\epsilon} n)$. Finally, Tsur [23] reduced the space to the asymptotically optimal $|\mathsf{CSA}| + o(n)$ bits, yet with higher time, $O(m \lg \lg \sigma + k \lg k \lg^{2+\epsilon} n)$.

In this paper we (almost) obtain the best from both solutions. We maintain the space in the optimal $|\mathsf{CSA}| + o(n)$ bits, and obtain search time $O(m \lg \lg \sigma + k \lg^2 k \lg^{1+\epsilon} n)$, almost $\lg n$ times faster than the current space-optimal solution and only a $\lg k$ factor away from the fastest one (that uses twice the space). To obtain the result, we introduce a data structure called the *sampled document array*, which may have independent interest.

2 Compressed Top-k Retrieval Indexes

Consider a collection of D strings $\{T_1, T_2, \ldots, T_D\}$ over alphabet $[1, \sigma]$, called *documents*, concatenated into a text $T[1, n] = T_1\$T_2\$ \ldots T_D\$$, where $\$ = 0$ is a

special symbol. Consider the suffix tree [24] of T, the suffix array [13] $A[1,n]$ of T, and a Compressed Suffix Array [17] CSA that is able to (1) given a pattern $P[1,m]$, find the area $A[sp,ep]$ of suffixes starting with P in time $t_{\mathsf{search}}(m)$, and (2) given a position i, compute $A[i]$ in time t_{SA}. For example, there is a CSA with $t_{\mathsf{search}}(m) = O(m \lg\lg \sigma)$ and $t_{\mathsf{SA}} = O(\lg^{1+\epsilon} n)$ for any constant $\epsilon > 0$ and using $|\mathsf{CSA}| = nH_h(T)(1+o(1)) + o(n)$ bits of space [1], and another with $t_{\mathsf{search}}(m) = O(m)$ and $t_{\mathsf{SA}} = O(\lg n)$ using $|\mathsf{CSA}| = nH_h(T)(1+o(1))+O(n)$ bits of space, where $H_h(T) \leq \lg \sigma$ is the per-symbol h-th order empirical entropy of T [14] (this is a lower bound on compressibility using any reasonable statistical model). In this paper we focus on the *top-k (most frequent documents) retrieval problem*: given a pattern $P[1,m]$, return the k documents where P appears most often. As explained, this is a reasonable relevance measure, especially when just one pattern is involved.

Each suffix tree leaf (or suffix array cell) can be associated to the document T_d where the corresponding suffix starts. We call $\mathsf{tf}(v,d)$ the number of leaves associated to document d that descend from suffix tree node v (i.e., the number of times the string label of v appears in document 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 d with highest $\mathsf{tf}(v,d)$ values. Note that the problem could be solved by attaching the answer to any suffix tree node, but the space would be $O(kn \lg n)$ bits, and work only up to the chosen k value. Now we describe the solutions we build on to obtain our result.

Hon, Shah and Vitter's solution. Hon et al.'s [11] structure is built (in principle) for a fixed k value. We choose a grouping factor $b = k \lg^{2+\epsilon} n$ and *mark* every bth leaf in the suffix tree (we use a slightly simplified description of their method [19]). 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 $O(n/b)$ nodes. For every marked suffix tree 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 there are at most $2b$ leaves covered by v but not by u (we will denote $v \setminus u$ that leaf set). Therefore they traverse those leaves using the CSA, and for each one they (1) compute the corresponding document 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).

To carry out (1) on the ith suffix tree leaf, they first compute $A[i]$ in $O(t_{\mathsf{SA}})$ time, and then convert it into a document number by storing a bitmap $B[1,n]$ that marks with a 1 the document beginnings in T [21]. So the document is $d = \mathsf{rank}(B, A[i])$, where $\mathsf{rank}(B,j)$ counts the number of 1s in $B[1,j]$. Since B has D 1s, 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 [20]. To carry out (2) they need additional $|\mathsf{CSA}|$ bits (see Sadakane [21]), and time $O(t_{\mathsf{SA}} \lg n)$. The node $u \in \tau_k$ is found using the CSA plus a constant-time LCA on τ_k for the leftmost and rightmost marked leaves in $[sp,ep]$, whereas the leaves covered by v are simply $[sp,ep]$. Thus the total query time is $O(t_{\mathsf{search}}(m) +$

$b\,t_{\mathsf{SA}} \lg n) = O(t_{\mathsf{search}}(m) + k\,t_{\mathsf{SA}} \lg^{3+\epsilon} n)$. On the two CSAs we have mentioned, this is $O(t_{\mathsf{search}}(m) + k \lg^{4+\epsilon} n)$.

As storing the top-k list needs $O(k \lg n)$ bits, the space for τ_k is $O((n/b)k \lg n) = O(n/\lg^{1+\epsilon} n)$ 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 $O(n/\lg^{\epsilon} n) = o(n)$ bits. Therefore the total space is $2|\mathsf{CSA}| + o(n)$ bits.

Several subsequent improvements [6, 3, 10] reduced the time to $O(t_{\mathsf{search}}(m) + k\,t_{\mathsf{SA}} \lg k \lg^{\epsilon} n)$, yet still using $2|\mathsf{CSA}| + o(n)$ bits of space, that is, twice the space of an optimal (under the hth order empirical entropy model) representation of the collection. Only this year [23] the space was reduced to the optimal $|\mathsf{CSA}| + o(n)$ bits, yet the time raises to $O(t_{\mathsf{search}}(m) + k\,t_{\mathsf{SA}} \lg k \lg^{1+\epsilon} n)$.

Tsur's optimal-space index. Building on ideas of Belazzougui et al. [3], Tsur [23] managed to reduce the space to the asymptotically optimal $|\mathsf{CSA}| + o(n)$ bits. Let $u' \in \tau_k$ be the parent of u in τ_k, that is, its nearest marked ancestor in the suffix tree. Tsur proved that, from the $O(b)$ leaves of $u' \setminus u$, only $O(\sqrt{bk})$ 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 u, associated to u. When one traverses the $O(b)$ leaves in $v \setminus u$, one (1) computes the document d as before, (2) if it is not stored as a candidate for u one can just ignore it, (3) if it is in the list then one just increases its frequency by 1. At the end one has enough information to answer the top-k query, without the need of the second $|\mathsf{CSA}|$ bits to compute frequencies below v.

If $b = k\ell$, the number of candidates is $t = O(\sqrt{bk}) = O(k\sqrt{\ell})$. One can encode them efficiently by storing, for each candidate d, the position of one leaf corresponding to d in the area covered by $u' \setminus u$. Those leaf positions are sorted and stored differentially: Let $0 < p_1 < p_2 < \ldots < p_t < 2b$ 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 [4], which occupy $\sum 2 \lg x_i = O(t \lg(b/t)) = O(k\sqrt{\ell} \lg \ell)$ 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 $O((n/b)(k \lg n + k\sqrt{\ell} \lg \ell)) = O(n((\lg n)/\ell + (\lg \ell)/\sqrt{\ell}))$ bits. By choosing $\ell = \lg k \lg^{1+\epsilon} n$, and since $\lg k \leq \lg n$, this is $O(n/(\lg k \lg^{\epsilon/2} n))$. Added over all the k values that are powers of 2, this is $O(n/\lg^{\epsilon/2} n) \sum_{i=1}^{\lg D} 1/i = O(n \lg \lg D/\lg^{\epsilon/2} n) = o(n)$ bits.

The total time is $O(t_{\mathsf{search}}(m) + b\,t_{\mathsf{SA}}) = O(t_{\mathsf{search}}(m) + k\,t_{\mathsf{SA}} \lg k \lg^{1+\epsilon} n)$. For the two CSAs we have described, this is $O(t_{\mathsf{search}}(m) + k \lg k \lg^{2+\epsilon} n)$.

Hon, Shah, Thankachan and Vitter's fastest index. Hon et al. [10] obtained the fastest solution to date using $2|\mathsf{CSA}| + o(n)$ bits of space. For this sake they consider two independent blocking values, $c < b$. For block value b 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 $O(c)$ suffix tree 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 $w \setminus u$ 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 $O(k \lg(b/k))$ bits, by storing as before one of their positions in $w \setminus u$, 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 $O((n/b)k \lg n) = O(n/\lg^{1+\epsilon} n)$ bits using $b = k \lg^{2+\epsilon} n$, which added over all the powers of 2 for k gives $O(n/\lg^{\epsilon} n) = o(n)$ bits, as before. For the ρ_k trees they require $O((n/c)k \lg(b/k))$ bits, which using $c = k \lg k \lg^{\epsilon} n$ gives $O(n \lg\lg n/(\lg k \lg^{\epsilon} n))$ bits. Added over the powers of 2 for k this gives $O(n \lg\lg n/\lg^{\epsilon} n) \sum_{i=1}^{\lg D} 1/i = O(n \lg\lg n \lg\lg D/\lg^{\epsilon} n) = o(n)$ bits.

The time is dominated by that of traversing $O(c)$ cells. Using some speedups [3] over the basic technique [11], the time is $O(t_{\mathsf{SA}} \lg\lg n)$ per cell, for a total of $O(t_{\mathsf{search}}(m) + k\, t_{\mathsf{SA}} \lg k \lg^{\epsilon} n)$ for any constant $\epsilon > 0$. Over the two CSAs we have described, this is $O(t_{\mathsf{search}}(m) + k \lg k \lg^{1+\epsilon} n)$.

3 A Faster Space-Optimal Representation

We build upon the schemes of Tsur [23] and Hon et al. [10]. 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 second $|\mathsf{CSA}|$ bits. Without this data, the structure 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 documents d found in $v \setminus w$.

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 $O(k\sqrt{\ell} \lg\lg n)$ 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.

3.1 The Sampled Document Array

The *document array* $E[1, n]$ of T contains at $E[i]$ the document to which $A[i]$ belongs [15]. It is a convenient structure but it requires $n \lg D$ bits of space. We store just a sampled version of it.

Definition 1. *The* sampled document array *is an array $E'[1, n']$ that stores every sth occurrence of each document d in E, for a sampling step s. That is, if*

$\mathsf{rank}_d(E,i)$ *is the number of times* d *occurs in* $E[1,i]$*, the cell* $E[i]$ *is stored in* E' *iff* $\mathsf{rank}_{E[i]}(E,i)$ *is a multiple of* s*. Note that* $n' \leq n/s$*.*

To E' we associate a bitmap $S[1,n]$ that marks the positions in E that are sampled in E'. The following lemma follows easily.

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

Proof. The area $E[sp,ep]$ includes y sampled occurrences of d. For the last $y-1$, their $s-1$ preceding non-sampled occurrences are also in $E[sp,ep]$. The $s-1$ occurrences preceding the first sampled one could be before sp, and thus $x \geq (y-1)s+1$. Alternatively, all the ys occurrences corresponding to the y sampled ones could be in the range, which could also include up to $s-1$ non-sampled occurrences to their right, yet their sampled successor could be after ep, thus $x \leq ys + (s-1)$. □

To use this lemma we store E' using a representation [7] that requires $n' \lg D + o(n' \lg D)$ and computes $\mathsf{rank}_d(E',i)$ in time $O(\lg \lg D)$. Further, we represent S in compressed form [20] so that it requires $n' \lg(n/n') + O(n') + o(n)$ bits and supports $\mathsf{rank}(S,i)$ in constant time. We use $s = \lg^2 n$, thus $n' = O(n/\lg^2 n)$ and the space for both E' and S is $o(n)$. Using this representation, we can compute y in Lemma 1 as $\mathsf{rank}_d(E', \mathsf{rank}(S,ep)) - \mathsf{rank}_d(E', \mathsf{rank}(S,sp-1))$ in time $O(\lg \lg D)$.

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 E' 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 second $|\mathsf{CSA}|$ bits of Hon et al. [10].

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

At query time we store the top-k documents of w, 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 $v \setminus w$, accessing the CSA 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 is dominated by computing the $O(c)$ CSA cells, plus $O(\lg\lg D)$ time per cell to compute $\mathsf{rank}_d(E', i)$ and $O(1)$ to operate the dictionary[3]. This adds up to $O(k(t_{\mathsf{SA}} + \lg\lg D)\lg^2 k \lg^\epsilon n)$, and the latter term absorbs the $\lg\lg D$.

Theorem 1. *The top-k most frequent documents problem, on a collection of length n, for a pattern of length m, can be solved using $|\mathsf{CSA}| + D\lg(n/D) + O(D) + o(n)$ bits and in $O(t_{\mathsf{search}}(m) + k\, t_{\mathsf{SA}} \lg^2 k \lg^\epsilon n)$ time, for any constant $\epsilon > 0$. Here CSA is a compressed suffix array over the collection, $t_{\mathsf{search}}(m)$ 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 [1, 2] whose size is related to H_h, the per-symbol empirical entropy of the text collection, for any $h \leq \alpha \lg_\sigma n$ 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 $nH_h(1 + o(1)) + o(n)$ bits and in $O(m \lg\lg\sigma + k \lg^2 k \lg^{1+\epsilon} n)$ time, for any constant $\epsilon > 0$.*

Corollary 2. *The top-k most frequent documents problem can be solved using $nH_h(1 + o(1)) + O(n)$ bits and in $O(m + k \lg n(\lg k \lg\lg n \lg\lg D)^2)$ time.*

4 Final Remarks

Reaching asymptotic space optimality (under the hth order empirical entropy model) for top-k document retrieval indexes is a very recent achievement. In this work we have improved the time of that space-optimal solution [23]. Our time complexity is a $\lg^2 k \lg^\epsilon n$ factor away from the minimum time required to obtain k document identifiers using the CSAs, and a $\lg k$ factor away from the fastest available solution that uses $2|\mathsf{CSA}| + o(n)$ bits [10].

It is natural to ask if those limits can be reached. Especially if the first limit is matched, this problem could be finally considered closed in the scenario of using optimal space based on CSAs. We believe, however, that a $\lg k$ factor in the time is the unavoidable price of allowing k to be specified at query time, whereas reaching the time of the currently fastest solution [10] seems feasible.

The other natural question is how much space is necessary to obtain the optimal $O(m + k)$ time. The best current space used to achieve this time is $O(n(\lg D + \lg\sigma))$ [18]. While it seems that $n \lg D$ bits are unavoidable in this case, there have been some efforts to use only $|\mathsf{CSA}| + n\lg D + o(n \lg D)$ bits [9]. However the time achieved is not yet the optimal.

[3] 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 $O(\lg^\delta D \lg\lg D)$ bits per internal node. They occupy overall $O(\sqrt{ck}\lg n) = O(k \lg k \lg^{1+\epsilon/2} n)$ bits, which is the space we use to answer the query. See [16, App. E] for more details.

References

1. J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet partitioning for compressed rank/select and applications. In *Proc. 21st ISAAC*, pages 315–326 (part II), 2010.
2. D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. In *Proc. 19th ESA*, pages 748–759, 2011.
3. D. Belazzougui, G. Navarro, and D. Valenzuela. Improved compressed indexes for full-text document retrieval. *J. Discr. Alg.*, 18:3–13, 2013.
4. T. Bell, J. Cleary, and I. Witten. *Text compression.* Prentice Hall, 1990.
5. S. Büttcher, C. Clarke, and G. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines.* MIT Press, 2010.
6. T. Gagie, J. Kärkkäinen, G. Navarro, and S.J. Puglisi. Colored range queries and document retrieval. *Theo. Comp. Sci.*, 483:36–50, 2013.
7. A. Golynski, I. Munro, and S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th SODA*, pages 368–373, 2006.
8. W.-K. Hon, M. Patil, R. Shah, and S.-Bin Wu. Efficient index for retrieving top-k most frequent documents. *J. Discr. Alg.*, 8(4):402–417, 2010.
9. W.-K. Hon, R. Shah, and S. Thankachan. Towards an optimal space-and-query-time index for top-k document retrieval. In *Proc. 23rd CPM*, pages 173–184, 2012.
10. W.-K. Hon, R. Shah, S. Thankachan, and J. Vitter. Faster compressed top-k document retrieval. In *Proc. 23rd DCC*, pages 341–350, 2013.
11. W.-K. Hon, R. Shah, and J. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. 50th FOCS*, pages 713–722, 2009.
12. W.-K. Hon, R. Shah, and S.-B. Wu. Efficient index for retrieving top-k most frequent documents. In *Proc. 16th SPIRE*, pages 182–193, 2009.
13. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comp.*, 22(5):935–948, 1993.
14. G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.
15. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc 13th SODA*, pages 657–666, 2002.
16. G. Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *CoRR*, arXiv:1304.6023v5, 2013.
17. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comp. Surv.*, 39(1):art. 2, 2007.
18. G. Navarro and Y. Nekrich. Top-k document retrieval in optimal time and linear space. In *Proc. 23rd SODA*, pages 1066–1078, 2012.
19. G. Navarro and D. Valenzuela. Space-efficient top-k document retrieval. In *Proc. 11th SEA*, pages 307–319, 2012.
20. R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Alg.*, 3(4):art. 43, 2007.
21. K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discr. Alg.*, 5:12–22, 2007.
22. R. Shah, C. Sheng, S. V. Thankachan, and J. Vitter. Top-k document retrieval in external memory. In *Proc. 21st ESA*, 2013. To appear.
23. D. Tsur. Top-k document retrieval in optimal space. *Inf. Proc. Lett.*, 113(12):440–443, 2013.
24. P. Weiner. Linear pattern matching algorithm. In *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.