Space-efficient Algorithms for Document Retrieval

Niko Välimäki and Veli Mäkinen *

Department of Computer Science, University of Helsinki, Finland.
{nvalimak,vmakinen}@cs.helsinki.fi

Abstract. We study the *Document Listing* problem, where a collection D of documents $d_1, \ldots, d_k$ of total length $\sum_i d_i = n$ is to be preprocessed, so that one can later efficiently list all the ndoc documents containing a given query pattern P of length m as a substring. Muthukrishnan (SODA 2002) gave an optimal solution to the problem; with $O(n)$ time preprocessing, one can answer the queries in $O(m + \text{ndoc})$ time. In this paper, we improve the space-requirement of the Muthukrishnan's solution from $O(n \log n)$ bits to $|CSA| + 2n + n \log k(1 + o(1))$ bits, where $|CSA| \leq n \log |\Sigma|(1 + o(1))$ is the size of any suitable *compressed suffix array* (CSA), and Σ is the underlying alphabet of documents. The time requirement depends on the CSA used, but we can obtain e.g. the optimal $O(m+\text{ndoc})$ time when $|\Sigma|, k = O(\text{polylog}(n))$. For general $|\Sigma|, k$ the time requirement becomes $O(m \log |\Sigma| + \text{ndoc} \log k)$. Sadakane (ISAAC 2002) has developed a similar space-efficient variant of the Muthukrishnan's solution; we obtain a better time requirement in most cases, but a slightly worse space requirement.

1 Introduction and Related Work

The *inverted file* is by far the most often utilized data structure in the Information Retrieval domain, being a fundamental part of Internet search engines such as Google, Yahoo, and MSN Search. In its simplest form, an inverted file consists of a set of *words*, where each word is associated with the list of documents containing it in a given document collection D. The *Document Listing* problem is then solved by printing out the document list associated with the given query word.

This approach works as such only for documents consisting of distinct words, such as natural language documents. If one considers the more general case, where a document consists of a sequence of symbols without any word boundaries, then the inverted file approach may turn out to be infeasible. There are basically two alternative approaches to proceed; (i) create an inverted file over all substrings of D; or (ii) create an inverted file over all *q-grams* of D, i.e., over all substrings of D of length q.

* Funded by the Academy of Finland under grant 108219.

Approach (i) suffers from (at least) quadratic space requirement, and approach (ii) suffers from the slow time requirement affected by the size of intermediate results; In approach (ii) the search for a pattern P of length $m > q$ proceeds by covering the pattern with q-grams, and searching each q-gram from the inverted file. In this case, each q-gram needs to be associated with a list of exact occurrence positions inside the documents. One can then merge the occurrence lists to spot the co-occurrences of the q-grams covering P. The total time requirement is determined by the length of the longest intermediate occurrence list associated with a q-gram of P. This can be significantly more than the number of occurrences of whole P.

Despite the above mentioned weakness, a q-gram based inverted file is very effective in practice, and it has found its place in important applications such as in a popular search tool BLAST for genome data [1]. One of the attractive properties of an inverted file is that it is easily compressible while still supporting fast queries [13]. In practice, an inverted file occupies space close to that of a compressed document collection.

Nevertheless, it is both practically and theoretically interesting to study index structures having the same attractive space requirement properties as inverted files and at the same time having provably good performance on queries.

If one can afford to use more space, then so-called *full-text indexes* such as *suffix arrays* and *suffix trees* [3] provide good alternatives to inverted files. These structures can be used, among many other things, to efficiently solve the *Occurrence Listing* problem, where all occurrence positions of a given pattern P are sought for from the document collection.

Recently, several *compressed full-text indexes* have been proposed that achieve the same good space requirement properties as inverted files [10]. These indexes also provide provably good performance in queries. Many of these indexes have been implemented and experimented, and shown effective in practice as well.[1]

Puglisi et al. [11] made an important sanity-check study on the practical efficiency of compressed full-text indexes when compared to inverted files. They concluded that, indeed, pattern searches are typically faster using compressed full-text indexes. However, when the number of occurrence positions becomes high (e.g. more than $5,000$ on English text [11]), inverted files become faster. This experimental observation is consistent with the theoretical properties of the indexes; In compressed full-text indexes each occurrence position must be decoded, and this typically takes $O(\log n)$ time per occurrence. On the other hand, the shorter the pattern, the more occurrences there are, meaning that in a q-gram based inverted file, the intermediate occurrence lists are not significantly larger than the final result.

Let us now get back to the original motivation — the Document Listing problem. Unlike for the Occurrence Listing problem, there is yet no simultaneously space- and time-efficient solution known for this problem. Interestingly, even achieving a good time requirement is nontrivial. One can, of course, solve the problem by pruning the answer given by any solution to the Occurrence

[1] See http://pizzachili.dcc.uchile.cl/

Listing problem. This is clearly not optimal, since the *number of occurrences*, nocc, may be arbitrarily greater than the *number of documents*, ndoc, containing these occurrences!

Matias et al. [8] gave the first efficient solution to the Document Listing problem; with $O(n)$ time preprocessing of a collection D of documents $d_1, \ldots, d_k$ of total length $\sum_i d_i = n$, they could answer the document listing query on a pattern P of length m in $O(m \log k + \text{ndoc})$ time. The algorithm uses a *generalized suffix tree* augmented with extra edges making it a directed acyclic graph.

Muthukrishnan [9] simplified the solution in [8] by replacing the augmented edges with a divide and conquer strategy. This simplification resulted into optimal query time $O(m + \text{ndoc})$.

The space requirements of both the above solutions are $O(n \log n)$ bits. This is significantly more than the collection size, $O(n \log |\Sigma|)$ bits, where Σ is the underlying alphabet. More importantly, the space usage is much more than that of an inverted file.

Sadakane [12] has developed a space-efficient version of the Muthukrishnan's algorithm. He obtains a structure taking $|CSA| + 4n + O(k \log \frac{n}{k}) + o(n)$ bits. However, his structure is not anymore time-optimal, as document listing queries take $O(m + \text{ndoc} \log^{\epsilon} n)$ time on a constant alphabet, where $|CSA|$ is the size of any *compressed suffix array* that supports *backward search* and $O(\log^{\epsilon} n)$ time retrieval of one suffix array value. Here $\epsilon > 0$ is a constant affecting the size of such compressed suffix array.

Very recently, Fischer and Heun [6] have shown how the space of the Sadakane's structure can be reduced to $|CSA| + 2n + O(k \log \frac{n}{k}) + o(n)$ bits; they also improve the extra space needed to build the Sadakane's structure from $O(n \log n)$ bits to $O(n \log |\Sigma|)$ bits.

In this paper, we give an alternative space-efficient variant of Muthukrishnan's structure that is capable of supporting document listing queries in optimal time under realistic parameter settings. Our structure takes $|CSA| + 2n + n \log k(1 + o(1))$ bits, where $|CSA| \leq n \log |\Sigma|(1 + o(1))$ is the size of any compressed suffix array supporting backward search. (We do not require the CSA to support the retrieval of suffix array values as the Sadakane's structure.) The time requirement depends on the underlying CSA used, but we can obtain e.g. optimal $O(m + \text{ndoc})$ time when $\Sigma, k = O(\text{polylog}(n))$. For general Σ, k the time requirement becomes $O(m \log |\Sigma| + \text{ndoc} \log k)$.

We also show that a recent data structure by Bast et al. [2] proposed for *output-sensitive autocompletion search* can be used for the Document Listing problem. The space requirement is asymptotically the same as above but the time requirement is slightly inferior.

We provide some preliminary experimental evidence to show that our structure is significantly faster than an inverted file, especially when ndoc $<<$ nocc.

2 Preliminaries

A *string* $T = t_1 t_2 \cdots t_n$ is a sequence of *symbols* from an ordered *alphabet* Σ. A *substring* of T is any string $T_{i...j} = t_i t_{i+1} \cdots t_j$, where $1 \leq i \leq j \leq n$. A *suffix* of T is any substring $T_{i...n}$, where $1 \leq i \leq n$. A *prefix* of T is any substring $T_{1...j}$, where $1 \leq j \leq n$. A *pattern* is a short string over the alphabet Σ. We say that the pattern $P = p_1 p_2 \cdots p_m$ *occurs* at the position j of the *text* alias *document* string T iff $p_1 = t_j, p_2 = t_{j+1}, \ldots, p_m = t_{j+m-1}$. Length of a document T is denoted $|T|$.

Definition 1 (Document Listing problem). *Given a collection D of documents $d_1, d_2, \ldots, d_k$ of total length $\sum_{i=1}^{k} |d_i| = n$, the Document Listing problem is to build an index for D such that one can later efficiently support the* document listing query *of listing for any given pattern P of length m the documents that contain P as a substring.*

The output of the document listing query in Def. 1 is a subset of document identifiers $\{1, 2, \ldots, k\}$.

We say that a *space-optimal solution* to the Document Listing problem is an index that occupies $n \log \Sigma(1 + o(1))$ bits. This is the asymptotic lower-bound given by the Kolmogorov complexity for any representation of D. Here we count the representation of D as part of the index. For compressible documents, it is possible to achieve better bounds.

Likewise, we say that a *time-optimal solution* to the Document Listing problem is an index that can be constructed in $O(n)$ time, and that supports listing of documents in $O(m + \mathrm{ndoc})$ time, where ndoc is the size of the output.

3 Time-Optimal Document Listing

Muthukrishnan [9] obtained a time-optimal solution to the document listing problem, but the solution was yet not space-optimal. In the sequel, we show that one can adjust the solution to obtain the space-optimality as well in certain parameter settings.

Let us describe the Muthukrishnan's solution in a level suitable for our purposes. We use a generalized suffix array instead of the generalized suffix tree used in the original proposal.

Let the document collection $d_1, d_2, \ldots, d_k$ be represented as a concatenation $D = d_1 d_2 \cdots d_k$. A *generalized suffix array* for the document collection D is then an array $A[1 \ldots n]$ containing the permutation of $1, 2, \ldots, n$ such that $D_{A[i],n} <_b D_{A[i+1],n}$ for $1 \leq i < n$. Here $<_b$ is the normal lexicographic order $<$ of strings, except that the document boundaries are handled separately; the order of suffixes is computed assuming (virtually) a special symbol $\$_i$ inserted after each document d_i, such that $\$_1 < \$_2 < \cdots < \$_k < c$, where $c \in \Sigma$.[2]

[2] Concrete insertion of symbols $\$_i$ is the standard definition. However, here it would make the alphabet size grow to $|\Sigma| + k$, affecting the later results. Therefore we opt for the virtual handling of boundaries.

Using two binary searches on A, one can easily locate the maximal range $[sp, ep]$ of A such that the pattern P is a prefix of all the suffixes $D_{A[sp],n}$, $D_{A[sp+1],n}$, ..., $D_{A[ep],n}$. Now, the remaining task is to report the ndoc documents containing those suffixes without having to spend time on each occurrence.

Muthukrishnan introduces a divide and conquer strategy on two arrays $C[1 \dots n]$ and $E[1 \dots n]$ for this task. The array E simply lists the document numbers of each suffix in the order they appear in the suffix array A, that is, $E[i] = j$ if $A[i]$ points inside the document d_j in the concatenation D. The array C is defined as

$$C[i] = \max\{j \mid j < i, E[i] = E[j]\} \tag{1}$$

(if there is no such $j < i$, then $C[i] = -1$). The algorithm is based on the following observation.

Lemma 1 ([9]). *Let $[sp, ep]$ be the maximal range of A corresponding to suffixes that have pattern P as a prefix. The document k' contains P if and only if there exists precisely one $j \in [sp, ep]$ such that $E[j] = k'$ and $C[j] < sp$.*

To proceed, the table C is assumed to be preprocessed for constant time *Range Minimum Queries (RMQ)*. That is, on a given interval I, one can compute $\min_{i \in I} C[i]$ in constant time (as well as the argument i giving the minimum). This preprocessing can be done in $O(n)$ time [4].

The divide and conquer algorithm starts by finding the $i \in [sp, ep]$ such that $C[i]$ is the smallest (using the constant time RMQ). If $C[i] > sp$ then there is no document to report and the process stops. Otherwise, we output $E[i]$ and repeat the same process recursively on $[sp, i-1]$ and on $[i+1, ep]$. One can see that all the documents containing P are reported and each of them only once [9].

By replacing the generalized suffix array with a generalized suffix tree, one can find the range $[sp, ep]$ in $O(m)$ time on a constant size alphabet (on general alphabets, this takes $O(m \log |\Sigma|)$ time). Afterward, the reporting of the documents takes $O(\text{ndoc})$ time.

Theorem 1 ([9]). *Document Listing problem can be solved using an index structure occupying $O(n \log n)$ bits and having an $O(n)$ time construction algorithm. The index supports document listing queries in $O(m + ndoc)$ time on constant size alphabets. On general alphabets, the query time becomes $O(m \log |\Sigma| + ndoc)$.*

4 Space-Optimal Document Listing

We will derive a space-efficient version of the index structure derived in the previous section. This is accomplished by representing the arrays A, C, E, as well as the structure for RMQ, compressed. The algorithm for answering the queries stays the same.

Representing A. Instead of suffix array A we can use any compressed suffix array supporting the backward search [10]. Different time/space tradeoffs are possible, e.g. one can find the range $[sp, ep]$ of A containing the occurrences of P in $O(m)$ time, when $|\Sigma| = O(\text{polylog}(n))$, using an index of size $nH_h + o(n \log |\Sigma|)$ bits [5]. Here $H_h = H_h(D) \leq \log |\Sigma|$ is the *h-th order empirical entropy* of the text collection D (lower bound for the average number of bits needed to code a symbol using a fixed code table for each h-context in D). The given space bound holds for small h (see [5]), but for our purposes it is enough to use an estimate $H_h \leq \log |\Sigma|$ that is independent of h. That is, the index size can be expressed as $n \log |\Sigma|(1+o(1))$ bits. The space bound is valid for general alphabets as well, but the time requirement becomes $O(m \log |\Sigma|)$.

Representing C and E. The crucial observation is that the array C is not needed at all, but instead it can be represented implicitly via the array E. Recall the definition of C in Eq. (1). We can re-express the definition as

$$C[i] = select_{E[i]}(E, rank_{E[i]}(E, i) - 1), \tag{2}$$

where $rank_{k'}(E, i)$ gives the number of times the value k' appears in $E[1, i]$ and $select_{k'}(E, j)$ gives the index of E containing the j-th occurrence of the value k' (and we define $select_{k'}(E, 0) = -1$ to handle the boundary case). It is easy to see that Eqs. (1) and (2) are identical; both express the link from the value $E[i]$ to its predecessor in E.

The array E can be seen as a sequence of symbols from the alphabet $\Sigma' = \{1, 2, \ldots, k\}$. The functions *rank* and *select* on such sequences are an essential part of the index structure we are already using as a representation of the suffix array A [5]. That is, we can represent E using $n \log |\Sigma'|(1 + o(1)) = n \log k(1 + o(1))$ bits of space for a so-called *generalized wavelet tree* [5]. Each value of E as well as the queries $rank_{k'}(E, i)$ and $select_{k'}(E, j)$ can then be computed in constant time when $k = O(\text{polylog}(n))$. On general $k \leq n$, the space stays the same, but the time requirement becomes $O(\log k)$.

Representing RMQ structure. The algorithm requires range minimum queries on C. As C is now implicitly represented via E, some modifications to the existing RMQ structures are needed. Sadakane [12] gives a succinct representation of the RMQ structure in [4] requiring $4n+o(n)$ bits on top of the array indexed. Fischer and Heun [6] have recently found another constant time RMQ representation occupying only $2n + o(n)$ bits on top of the array indexed. We can use either of these representations on top of our implicit representation of C. Explicit values of C are mainly needed only during construction; queries access a constant number of values in C, which can then be computed from the generalized wavelet tree representation of E.

We have obtained the following result.

Theorem 2. *The Document Listing problem can be solved using an index structure occupying* $n \log |\Sigma|(1 + o(1)) + 2n + n \log k(1 + o(1))$ *bits and having an* $O(n \log |\Sigma| + n \log k)$ *time construction algorithm. The index supports document*

listing queries in $O(m + ndoc)$ time when $|\Sigma|, k = O(polylog(n))$. On general alphabets and on general document collection sizes $k \leq n$, the query time component $O(m)$ becomes $O(m \log |\Sigma|)$ and $O(ndoc)$ becomes $O(ndoc \log k)$, respectively.

Proof. The space and query time bounds should be clear from the above discussion. The construction time is achieved by building first the suffix array of D using e.g a linear time construction algorithm, and then building the generalized wavelet tree on the Burrows-Wheeler transform and other structures to form the compressed representation of the suffix array [5]. The array E and its generalized wavelet tree are constructed similarly. The bottleneck is the generalized wavelet tree construction. Although not explicitly stated in [5], it can be constructed in time linear in the final result size in bits. □

Notice that the obtained structure is space-optimal when $k = o(|\Sigma|)$ and $|\Sigma| = O(\text{polylog}(n))$.

The space requirement of the $O(n \log |\Sigma|)$ time construction algorithm is $O(n \log n)$ bits. A slower construction algorithm, taking $O(n \log n \log |\Sigma|)$ time, that uses the same asymptotic space as the final structure, is easy to derive using the dynamic wavelet tree proposed in [7]. Also the RMQ-solution by Fischer and Heun can be constructed within these space and time limits.

4.1 Extended functionality.

So far our structure can list the documents containing the pattern. A useful extension would be to *list the occurrences inside the selected documents*.

For motivation, imagine that the collection represents a file system; files are concatenated into D in the order of a depth-first traversal. A search on the file system would return the documents containing the query word. A user could select documents of interest from the list, and ask to see the occurrence positions. Most likely the user would like to see a *context* around each occurrence to judge the relevance.

Also, to guide the selection of relevant documents, it would be good to have the matching documents listed in the order of expected relevance; one way would be to *list the documents in the order of number of occurrences of the pattern*.

We can support the above described functionalities with our index. First, to have the matching documents listed in the order of relevance, one may use the fact that the number of occurrences $\text{nocc}_{k'}$ in document k' can be computed by

$$\text{nocc}_{k'} = rank_{k'}(E, ep) - rank_{k'}(E, sp - 1). \tag{3}$$

After sorting the numbers $\text{nocc}_{k'}$, one has achieved the goal.

Second, to list the occurrences lying in a selected document (or in the range of documents lying in a subdirectory), one may use the existing functionalities of compressed suffix arrays to list *all* the occurrences. To make this faster than just pruning the list, one can proceed as follows. Let k' be a selected document known to contain occurrences of P. To find the last occurrence inside the suffix

array range $[sp, ep]$, one can query $i = select_{k'}(E, rank_{k'}(E, ep))$ and compute the occurrence position $A[i]$ by decoding the entry from the compressed suffix array. The process is continued step-by-step with $i = select_{k'}(E, rank_{k'}(E, i-1))$ until $i < sp$. At each step one can also output a context around each occurrence.

5 Autocompletion Search and Document Listing

Recently, Bast et al. [2] studied a related problem of providing a space-efficient index for the so-called *output-sensitive autocompletion search* problem.

Consider a user typing a query in a document retrieval tool. The tool can provide repeatedly a list of matching documents while the user is still completing the query. This interactive feature is called *autocompletion*.

To avoid the trivial solution of starting the query on each new symbol from scratch, Bast et al. proposed an online output-sensitive method that focuses the query W on the so-far matching documents, say $D' \subset D$. They developed an index structure supporting this query assuming a text collection consisting of distinct words. As they mention [2, p. 153], the structure can be extended to the case of full-text sequences by using e.g. a *suffix array* on top of their index.

Let us now consider how the structure of Bast et al., when applied to the full-text setting, can be used to solve the Document Listing problem. Their `AUTOTREE` structure is basically a succinct version of the following; a balanced binary tree built on the lexicographically ordered suffixes of D such that each node lists the documents containing suffixes in its subtree. In fact, the succinct coding they propose (`TREE+BITVEC`) is almost identical to a balanced wavelet tree built on our array E! However, the difference comes in the queries supported; they engineer the representation suitable for fast autocompletion searches and do not exploit the divide and conquer strategy to speed up the search. Instead they avoid reporting the same occurrence repeatedly by pruning the tree suitably.

Nevertheless, one cannot obtain exactly as fast reporting time for the Document Listing problem using `AUTOTREE` as what we obtain in Theorem 2; the reason is that `AUTOTREE` outputs word-in-document pairs where the prefix of the word matches the query pattern. In our case, this means outputting all suffixes whose prefix matches the pattern (that is, all occurrences). However, one can adjust the search algorithm in [2, page 154, step 2] to work only on the $O(\log n)$ nodes covering the search range; in the worst case, each document can appear in each of those nodes and be reported $O(\log n)$ times. This gives $O(\mathrm{ndoc} \log n)$ reporting time which is still inferior to our structure. The space usage of both structures are closely the same, since in our terminology the size of `AUTOTREE` is $n\lceil \log k \rceil$ bits (their N equals our number of suffixes n, and their n is our k).

6 Comparison to Sadakane's Solution

Our solution is very similar to the Sadakane's solution [12]. The difference is in the use of generalized wavelet trees to represent the document numbers associated with the suffixes. Sadakane is able to represent this information in

$O(k \log \frac{n}{k})$ bits, as we use $n \log k(1 + o(1))$ bits. However, to retrieve the document numbers, he needs to use the expensive $O(\log^{\epsilon} n)$ time operation to retrieve a suffix array value. Choosing a small value of ϵ affects the multiplicative constant factor in the size of the underlying compressed suffix array inversely. We can do this in constant time using generalized wavelet tree, when the number of documents is $k = \text{polylog}(n)$.

7 Preliminary Experimental Results

Extrapolating from the experimental results in [11] for the Occurrence Listing problem, one could expect that our structure is superior to inverted files in typical document listing settings, that is, where the inverted file needs to examine all occurrences and our index can work directly on the document level. The space should be quite close as well; Inverted files use more space in representing the document collection as such, while in our index the document collection is compressed inside the index. Our $n \log k(1 + o(1))$ may exceed the space needed for the inverted file, when k is large.

We have a preliminary implementation ready of our structure, which is yet not fully optimized for time or space usage. However, it is enough for validating the claim of being faster when the number of occurrence positions nocc is large but the number of matching documents ndoc is small. To see this, we compared our structure to the same inverted file implementation as used in [11]. We used different size prefixes of a catenated English text collection as the input and partitioned each prefix to k equal size "documents", with varying k. We selected randomly from the text collection two sets of patterns, each containing 1000 patterns of length $m = 3$ and $m = 4$, respectively. The small pattern length was used to guarantee that ndoc << nocc. Table 1 shows the results for the inverted file, and Table 2 shows the results for our structure. The running times are the total time needed for the 1000 queries, and values nocc/pattern and ndoc/pattern are the average output sizes.

Table 1. Running times and index sizes for inverted file. The results correspond to Occurrence Listing queries; the time needed for pruning the Document Listing result would be negligible. We used parameter values Block size = 64, q-gram size = 3, and list intersection limit = 10000 inside the inverted file implementation, see [11].

		$m = 3$		$m = 4$	
|text| (MB)	|index| (MB)	time (s)	nocc/pattern	time (s)	nocc/pattern
1	2.00	0.35	1334.5	0.13	267.4
25	49.12	8.61	29085.4	2.23	3077.1
50	98.11	17.46	57136.0	4.29	6428.9

The results are as expected: The running times of both structures depend almost linearly on their output sizes. When the input size grows, but the number

Table 2. Running times and index sizes for our structure, with varying k.

			$m = 3$		$m = 4$	
|text| (MB)	k	|index| (MB)	time (s)	ndoc/pattern	time (s)	ndoc/pattern
1	1	2.12	0.0029	1	0.0036	1
1	50	3.13	0.35	40.7	0.21	27.9
1	100	3.31	0.68	72.9	0.35	44.0
1	150	3.41	0.95	100.0	0.45	55.7
1	200	3.47	1.21	123.4	0.54	65.0
1	250	3.52	1.41	144.3	0.61	72.8
25	200	84.76	3.1	180.1	2.1	132.2
50	200	169.29	3.7	185.1	2.7	148.6

of documents (i.e. the maximum output size for our structure) stays the same, our structure becomes faster than the inverted file. On short collections the running times are almost the same.

This result with short patterns, combined with the observation in [11] that compressed suffix arrays are faster than inverted files for Occurrence Listing queries on long patterns, gives reason to argue that an index based on compressed suffix arrays may be an attractive alternative to inverted files as a generic building block for flexible Information Retrieval tasks.

8 Future Work

Muthukrishnan [9] studied many other important Information Retrieval tasks such as *document mining* (finding the documents where a given pattern appears more often than a given threshold) and *proximity queries* (where the occurrences of patterns appearing near to each others are searched for). The solutions to these problems use the same kind of ideas as for the Document Listing problem, but are somewhat more complicated, and hence more difficult to make space-efficient. It is an interesting future challenge to derive space-efficient solutions to these problems such that they would become competitive with inverted file -based solutions in practice.

Acknowledgement

We wish to thank Gonzalo Navarro for bringing [9] to our attention and anonymous reviewers for bringing [8] and[12] to our attention.

References

1. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

2. H. Bast, C. W. Mortensen, and I. Weber. Output-sensitive autocompletion search. In *Proceedings of the 13th International Conference on String Processing and Information Retrieval (SPIRE 2006)*, pages 150–162, 2006.
3. M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2002.
4. M. Farach-Colton and M. A. Bender. The lca problem revisited. In *Proc. Latin American Theoretical Informatics (LATIN)*, pages 88–94, 2000.
5. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representation of sequences and full-text indexes. *ACM Transactions on Algorithms*, 2007. To appear.
6. J. Fischer and V. Heun. A new succinct representation of rmq-information and improvements in the enhanced suffix array. In *Proceedings of the International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE'07)*, LNCS. Springer, 2007. To appear.
7. V. Mäkinen and G. Navarro. Dynamic entropy compressed sequences and full-text indexes. In *Proc. Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009, pages 306–317, 2006.
8. Y. Matias, S. Muthukrishnan, S. C. Sahinalpk, and J. Ziv. Augmenting suffix trees with applications. In *Proceedings of the 6th Annual European Symposium on Algorithms (ESA 1998)*, LNCS 1461, pages 67–78. Springer-Verlag, 1998.
9. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA'02)*, pages 657–666, 2002.
10. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article 2, 2007.
11. S. J. Puglisi, W. F. Smyth, and A. Turpin. Inverted files versus suffix arrays for locating patterns in primary memory. In *Proceedings of the 13th International Conference on String Processing and Information Retrieval (SPIRE 2006)*, pages 122–133, 2006.
12. K. Sadakane. Space-efficient data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007. Earlier in ISAAC 2002.
13. Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.