

Fast, Small, and Simple Document Listing on Repetitive Text Collections^{*}

Dustin Cobas and Gonzalo Navarro

CeBiB — Center for Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Chile. {dcobas,gnavarro}@dcc.uchile.cl

Abstract. Document listing on string collections is the task of finding all documents where a pattern appears. It is regarded as the most fundamental document retrieval problem, and is useful in various applications. Many of the fastest-growing string collections are composed of very similar documents, such as versioned code and document collections, genome repositories, etc. Plain pattern-matching indexes designed for repetitive text collections achieve orders-of-magnitude reductions in space. Instead, there are not many analogous indexes for document retrieval. In this paper we present a simple document listing index for repetitive string collections of total length n that lists the $ndoc$ distinct documents where a pattern of length m appears in time $\mathcal{O}(m + ndoc \cdot \lg n)$. We exploit the repetitiveness of the *document array* (i.e., the suffix array coarsened to document identifiers) to grammar-compress it while precomputing the answers to nonterminals, and store them in grammar-compressed form as well. Our experimental results show that our index sharply outperforms existing alternatives in the space/time tradeoff map.

1 Introduction

Document retrieval is a family of problems aimed at retrieving *documents* from a set that are relevant to a query *pattern*. In a general setting, both documents and patterns are arbitrary strings. This encompasses the well-known application of natural language and Web searching, but also many others of interest in bioinformatics, software development, multimedia retrieval, etc. [22].

The most fundamental document retrieval problem, on top of which more sophisticated ones are built, is *document listing*. This problem aims at simply returning the list of documents where the pattern appears. An obvious solution to document listing resorts to *pattern matching*: find all the *occ* positions where the pattern appears, and then return the $ndoc$ different documents where those positions lie. This solution requires time $\Omega(occ)$ and the output is of size $\mathcal{O}(ndoc)$, so the approach is very inefficient if $ndoc \ll occ$ (i.e., if the pattern appears many times in the same documents). A better solution, which however applies only in natural language settings, resorts to *inverted indexes* [1]. These restrict the possible patterns to sequences of words and store the list of the documents where

^{*} Funded with basal funds FB0001 and by Fondecyt Grant 1-170048, Conicyt, Chile.

each word appears, thereby solving document listing via intersections of the lists of the pattern words.

Muthukrishnan [20] designed the first linear-space and optimal-time index for general string collections. Given a collection of total length n , he builds an index of $\mathcal{O}(n)$ words that lists the $ndoc$ documents where a pattern of length m appears in time $\mathcal{O}(m + ndoc)$. While linear space is deemed as sufficiently small in classic scenarios, the solution is impractical for very large text collections unless one resorts to disk, which is orders of magnitude slower. Sadakane [26] showed how to reduce the space of Muthukrishnan’s index to that of the statistically-compressed text plus $\mathcal{O}(n)$ bits, while raising the time complexity to only $\mathcal{O}(m + ndoc \cdot \lg n)$ if the appropriate underlying pattern-matching index is used [2].

The sharp growth of text collections is a concern in many recent applications, outperforming Moore’s Law in some cases [27]. Fortunately, many of the fastest-growing text collections are *highly repetitive*: each document can be obtained from a few large blocks of other documents. These collections arise in different areas, such as repositories of genomes of the same species (which differ from each other by a small percentage only) like the 100K-genome project¹, software repositories that store all the versions of the code arranged in a tree or acyclic graph like GitHub², versioned document repositories where each document has a timeline of versions like Wikipedia³, etc. On such text collections, statistical compression is ineffective [14] and even $\mathcal{O}(n)$ bits of extra space can be unaffordable.

Repetitiveness is the key to tackle the fast growth of these collections: their amount of new material grows much slower than their size. For example, version control systems compress those collections by storing the list of edits with respect to some reference document that is stored in plain form, and reconstruct it by applying the edits to the reference version. Much more challenging, however, is to *index* those collections in small space so as to support fast pattern matching or document retrieval tasks. To date, there exist several pattern matching indexes for repetitive text collections (see a couple of studies [21, 10] and references therein). However, there are not many document retrieval indexes for repetitive text collections [5, 8, 23]. Most of these indexes [26, 8] rely on a pattern-matching index needs $\Omega(n)$ bits in order to offer $\mathcal{O}(\lg n)$ time per retrieved document.

In this paper we introduce new simple and efficient document listing indexes aimed at highly repetitive text collections. Like various preceding indexes, we achieve $\mathcal{O}(m + ndoc \cdot \lg n)$ search time, yet our indexes are way faster and/or smaller than previous ones on various repetitive datasets, because they escape from the space/time tradeoff of the pattern-matching index. Our main idea is as follows: we use the *document array* $DA[1..n]$ [20], which projects the entries of the *suffix array* [19] to the document where each position belongs. Document listing boils down to listing the distinct integers in a range $DA[sp..ep]$, where sp and ep are found in time $\mathcal{O}(m)$. Array DA must be grammar-compressible

¹ www.genomicsengland.co.uk/about-genomics-england/the-100000-genomes-project

² github.com/search?q=is:public

³ en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia

since the differential suffix array is known to be so on repetitive texts [11, 10]. We then build a *balanced* binary context-free grammar that generates (only) DA . This allows us retrieve any individual cell of DA in time $\mathcal{O}(\lg n)$ and any range $DA[sp..ep]$ in time $\mathcal{O}(ep - sp + \lg n)$. We can then implement existing indexes [26, 8] within much less space and without affecting their time complexities. Further, we propose a new simple index based on the grammar-compressed array DA . Our compression guarantees that any range $DA[sp..ep]$ is covered by $\mathcal{O}(\lg n)$ nonterminals. For each nonterminal of the grammar, we store the list of the distinct documents appearing in it. The set of all the lists is grammar-compressed as well, as done in previous work [5, 8]. We then merge the lists of the $\mathcal{O}(\lg n)$ nonterminals that cover $DA[sp..ep]$, in time $\mathcal{O}(ndoc \cdot \lg n)$.

2 Preliminaries

A *document* T is a sequence of symbols over an alphabet $\Sigma = [1..\sigma]$, terminated by a special symbol $\$$ that is lexicographically smaller than any symbol of Σ .

A *collection* \mathcal{D} is a set of d documents $\mathcal{D} = \{T_1, \dots, T_d\}$. \mathcal{D} is commonly represented as the concatenation $\mathcal{T} = T_1 T_2 \dots T_d$, of length $|\mathcal{T}| = n$.

A *pattern* P is a string over the same alphabet Σ with length $|P| = m$. It occurs occ times in \mathcal{T} , and appears in $ndoc$ documents.

Text indexes. The *suffix tree* [28] of a string \mathcal{T} is a compressed digital tree storing all the suffixes $\mathcal{T}[i..n]$, for all $1 \leq i \leq n$. The suffix tree node reached by following the symbols of a pattern P is called the *locus* of P and is the ancestor of all the *occ* leaves corresponding to the positions of P in \mathcal{T} . The suffix tree uses $\mathcal{O}(n \lg n)$ bits and lists all the occurrences of P in time $\mathcal{O}(m + occ)$.

The *suffix array* [19] $SA[1..n]$ of a string $\mathcal{T}[1..n]$ is a permutation of the starting positions of all the suffixes of \mathcal{T} in lexicographic order, $\mathcal{T}[SA[i], n] < \mathcal{T}[SA[i+1], n]$ for all $1 \leq i < n$. SA can be binary searched to obtain the range $SA[sp..ep]$ of all the suffixes prefixed by P (note $occ = ep - sp + 1$). Thus the occurrences of P can be listed in time $\mathcal{O}(m \lg n + occ)$. The suffix array takes $n \lg n$ bits.

Compressed suffix arrays (CSAs) [24] are space-efficient representations of the suffix array. They find the interval $[sp..ep]$ corresponding to $P[1..m]$ in time $t_{\text{search}}(m)$, and access any cell $SA[i]$ in time $t_{\text{lookup}}(n)$. Their size in bits, $|CSA|$, is usually bounded by $\mathcal{O}(n \lg \sigma)$.

Grammar compression. Grammar compression of a string $S[1..n]$ replaces it by a context-free grammar (CFG) \mathcal{G} that uniquely generates S . This CFG \mathcal{G} may require less space than the original sequence S , especially when S is repetitive.

Finding the smallest CFG \mathcal{G}^* generating the input S is NP-hard [16], but various $\mathcal{O}(\lg(n/|\mathcal{G}^*|))$ -approximations exist. In particular, we are interested in approximations that are *binary* (i.e., the maximum arity of the parse tree is 2) and *balanced* (i.e., any substring is covered by $\mathcal{O}(\lg n)$ maximal nodes of the parse tree) [25, 3, 13].

3 Related Work

Muthukrishnan [20] proposed the first optimal-time linear-space solution to the document listing problem. He defines the *document array* $DA[1..n]$ of \mathcal{T} , where $DA[i]$ stores the identifier of the document to which $\mathcal{T}[SA[i]]$ belongs. The document listing problem is then translated into computing the $ndoc$ distinct identifiers in the interval $DA[sp..ep]$ corresponding to the pattern P . He uses a suffix tree to find sp and ep in time $\mathcal{O}(m)$, and then an algorithm that finds the $ndoc$ distinct numbers in the range in time $\mathcal{O}(ndoc)$.

Sadakane [26] adapts the method to use much less space. He replaces the suffix tree by a CSA, and mimics the algorithm to find the distinct numbers in $DA[sp..ep]$ using only $\mathcal{O}(n)$ bits of space. Within $|CSA| + \mathcal{O}(n)$ bits, he performs document listing in time $\mathcal{O}(t_{\text{search}}(m) + ndoc \cdot t_{\text{lookup}}(n))$. Using a particular CSA [2] the space is $n \lg \sigma + o(n \lg \sigma) + \mathcal{O}(n)$ bits and the time is $\mathcal{O}(m + ndoc \cdot \lg n)$.

There are many other classical and compact indexes for document listing. We refer the reader to a survey [22] and focus on those aimed at repetitive texts.

Gagie et al. [8] proposed a technique adapting Sadakane’s solution to highly repetitive collections. They show that the technique to find the distinct elements of $DA[sp..ep]$ can be applied almost verbatim on an array they call *interleaved longest-common-prefix array* (*ILCP*). On repetitive collections, this array can be decomposed into a small number ρ of equal values, which allows them represent it in little space. The *ILCP* index requires $|CSA| + \mathcal{O}((\rho + d) \lg n)$ bits of space and solves document listing in time $\mathcal{O}(t_{\text{search}}(m) + ndoc \cdot t_{\text{lookup}}(n))$.

Gagie et al. [8] proposed another radically different approach, called *Pre-computed Document Lists* (*PDL*). The idea is to store the list of the documents where (the corresponding substring of) each suffix tree node appears. Then the search consists of finding the locus of P and returning its list. To reduce space, however, only some sampled nodes store their lists, and so document listing requires *merging* the lists of the maximal sampled nodes descending from the locus node. To further save space, the lists are grammar-compressed.

To bound the query time, the deepest sampled nodes cover at most b leaves, and a factor β restricts the work done per merged document in the unions of the lists. The index then uses $|CSA| + \mathcal{O}((n/b) \lg n)$ bits, and the document listing time is $\mathcal{O}(t_{\text{search}}(m) + ndoc \cdot \beta \cdot h + b \cdot t_{\text{lookup}}(n))$, h being the suffix tree height.

A problem in all revisited CSA-based solutions are the $\Theta((n \lg n)/t_{\text{lookup}}(n))$ extra bits that must be included in $|CSA|$ in order to get $\Theta(t_{\text{lookup}}(n))$ time per document. This space does not decrease with repetitiveness, forcing all these indexes to use $\Omega(n)$ bits to obtain time $\mathcal{O}(t_{\text{search}}(m) + ndoc \cdot \lg n)$, for example.

Claude and Munro [5] propose the first index for document listing based on grammar compression, which escapes from the problem above. They extend a grammar-based pattern-matching index [6] by storing the list of the documents where each nonterminal appears. Those lists are grammar-compressed as well. The index searches for the minimal nonterminals that contain P and merges their lists. While it does not offer relevant space or query time guarantees, the index performs well in practice. Navarro [23] extends this index in order to obtain space guarantees and $\mathcal{O}(m^2 + m \lg^2 n)$ time, but the scheme is difficult to implement.

4 Our Document Listing Index

Like most of the previous work, we solve the document listing problem by computing the $ndoc$ distinct documents in the interval $DA[sp..ep]$ corresponding to the pattern P , found with a CSA in time $\mathcal{O}(t_{\text{search}}(m))$. Instead of also using the CSA to compute the values of DA (and thus facing the problem of using $\Theta((n \lg n)/t_{\text{lookup}}(n))$ bits to compute a cell in time $\Theta(t_{\text{lookup}}(n))$, as it happens in previous work [26, 8]), we store the array DA directly, yet in grammar-compressed form. This is promising because the suffix array of repetitive collections is known to have large areas $SA[i..i + \ell]$ that appear shifted by 1 elsewhere, $SA[j..j + \ell]$, that is, $SA[i + k] = SA[j + k] + 1$ for all $0 \leq k \leq \ell$ [18, 10]. Except for the d entries of SA that point to the ends of the documents, it also holds that $DA[i + k] = DA[j + k]$. Grammar compression is then expected to exploit those large repeated areas in DA .

To answer the queries efficiently, we use an idea similar to the one introduced in PDL [8] and the Grammar-index [5]: precomputing and storing the answers of document listing queries, and grammar-compressing those lists as well. An important difference with them is that PDL stores lists for suffix tree nodes and the Grammar-index stores lists for nonterminals of the grammar of \mathcal{T} . Our index, instead, stores lists for the nonterminals of the grammar of DA . This is much simpler because we do not store a suffix tree topology (like PDL) nor a complex grammar-based pattern-matching index (like the Grammar-index): we simply find the interval $DA[sp..ep]$ using the CSA, fetch the nonterminals covering it, and merge their lists. By using a binary balanced grammar on DA , we ensure that any document is obtained in the merging only $\mathcal{O}(\lg n)$ times, which leads to our worst-case bound of $\mathcal{O}(ndoc \cdot \lg n)$. PDL and the Grammar-index cannot offer such a logarithmic-time guarantee.

4.1 Structure

The first component of our index is a CSA suitable for repetitive collections, of which we are only interested in the functionality of finding the interval $SA[sp..ep]$ corresponding to a pattern $P[1..m]$. For example, we can use the Run-Length CSA (RLCSA) variant of Gagie et al. [10], which offers times $t_{\text{search}}(m) = \mathcal{O}(m \lg \lg_w \sigma)$ within $\mathcal{O}(r \lg n)$ bits, or $t_{\text{search}}(m) = \mathcal{O}(m)$ within $\mathcal{O}(r \lg(n/r) \lg n)$ bits, where r is the number of equal-letter runs in the Burrows-Wheeler Transform of \mathcal{T} . This also upper-bounds the number of areas $SA[i..i + \ell]$ into which SA can be divided such that each area appears elsewhere shifted by 1 [17].

The second component is the grammar \mathcal{G} that generates $DA[1..n]$, which must be binary and balanced. Such grammars can be built so as to ensure that their total size is $\mathcal{O}(r \lg(n/r) \lg n)$ bits [9], of the same order of the first component.

The third component are the lists D_v of the distinct documents that appear in the expansion of each nonterminal v of \mathcal{G} . These lists are stored in ascending order to merge them easily. To reduce their size, the set of sequences D_1, \dots, D_g are grammar-compressed as a whole in a new grammar \mathcal{G}' , ensuring that no

nonterminal of \mathcal{G}' expands beyond a list D_v . Each list D_v can then be obtained in optimal time, $\mathcal{O}(|D_v|)$, from a nonterminal of \mathcal{G}' .

4.2 Document listing

Given a pattern $P[1..m]$, we use the CSA to find the range $[sp..ep]$ where the occurrences of P lie in the suffix array, in time $\mathcal{O}(t_{\text{search}}(m))$. We then find the maximal nodes of the parse tree of DA that cover $DA[sp..ep]$. Finally, we decompress the lists of the nonterminals corresponding to those maximal nodes, and compute their union.

Since \mathcal{G} is binary and balanced, there are $\mathcal{O}(\lg n)$ maximal nonterminals that cover $DA[sp..ep]$ in the parse tree. By storing the length to which each nonterminal of \mathcal{G} expands, we can easily find those $\mathcal{O}(\lg n)$ maximal nonterminals in time $\mathcal{O}(\lg n)$, by (virtually) descending in the parse tree from the initial symbol of \mathcal{G} towards the area $DA[sp..ep]$.

To merge the $\mathcal{O}(\lg n)$ lists of documents in ascending order, we use an atomic heap [7] (see practical considerations in the next section). This data structure performs `insert` and `extractmin` operations in constant amortized time, when storing $\mathcal{O}(\lg^2 n)$ elements. We then insert the heads of the $\mathcal{O}(\lg n)$ lists in the atomic heap, extract the minimum, and insert the next element of its list. If we extract the same document many times, we report only one copy. We then expand and merge the lists D_{v_1}, \dots, D_{v_k} in time $\mathcal{O}(|D_{v_1}| + \dots + |D_{v_k}|)$.

Since each distinct document we report may appear in the $\mathcal{O}(\lg n)$ lists, our document listing solution takes time $\mathcal{O}(t_{\text{search}}(m) + ndoc \cdot \lg n)$. By using the RLCSA that occupies $\mathcal{O}(r \lg(n/r) \lg n)$ bits, the total time is $\mathcal{O}(m + ndoc \cdot \lg n)$.

4.3 Example

Figure 1 shows an example with 3 documents, $T_1 = \text{MINIMUM\$}$, $T_2 = \text{MINIMAL\$}$, and $T_3 = \text{MINIMIZES\$}$. The rightmost column shows \mathcal{T} . The preceding columns show the sorted suffixes, the suffix array SA , and the document array DA , a sequence over $\{1, 2, 3\}$. To the left of DA we show the syntax tree of the grammar we built, with nonterminal symbols 4 to 18. Associated with each nonterminal we write the list of distinct documents to which the nonterminal expands.

A search for the pattern $P = \text{I}$ identifies the suffix array interval $SA[6..12]$, thus we have to report all the distinct documents in $DA[6..12]$. These correspond to two nodes in the grammar, the nonterminals 5 and 6. Thus we merge their lists, $\{1, 2, 3\}$ and $\{1, 2, 3\}$, to obtain the answer $\{1, 2, 3\}$. Note that each of the 3 documents we report was found twice in the lists that cover $DA[6..12]$.

4.4 Plugging-in other indexes

Our grammar-compressed DA , without the lists D_v , can be used to replace the CSA component that requires $\Theta((n \lg n) / t_{\text{lookup}}(n))$ bits to compute a cell in time $\Theta(t_{\text{lookup}}(n))$. These indexes actually access cells $SA[i]$ in order to obtain $DA[i]$. Our compressed DA offers $\mathcal{O}(\lg n)$ access time in $\mathcal{O}(r \lg(n/r) \lg n)$ bits.

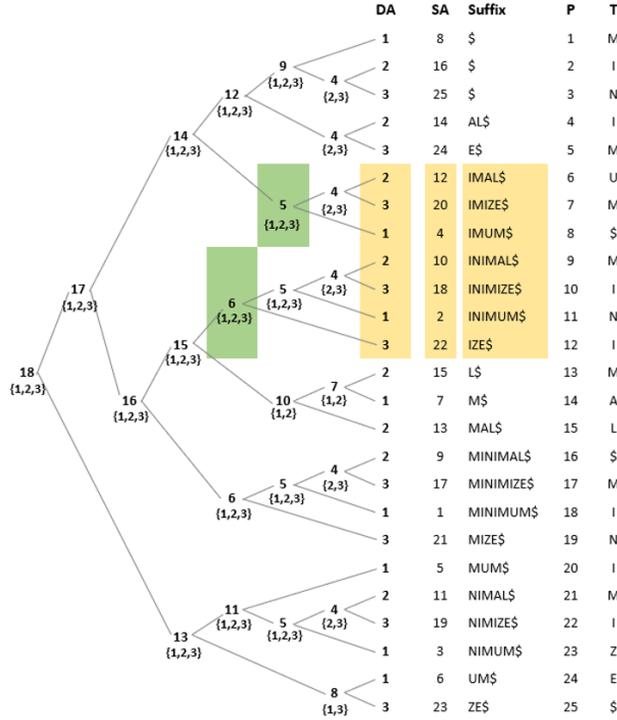


Fig. 1: An example of our document listing structure.

Thus, we can implement Sadakane’s solution [26], as well as ILCP and PDL [8] all answering in time $\mathcal{O}(m + ndoc \cdot \lg n)$, and replacing the $\mathcal{O}((n \lg n) / t_{\text{lookup}}(n))$ part of their $|CSA|$ space by $\mathcal{O}(r \lg(n/r) \lg n)$ bits (which also accounts for the RLCSA variant that finds $[sp..ep]$ in time $\mathcal{O}(m)$). We can also implement the brute-force solution in time $\mathcal{O}(m + occ + \lg n)$ and $\mathcal{O}(r \lg(n/r) \lg n)$ bits by extracting the whole $DA[sp..ep]$.

5 Practical Considerations

5.1 Compressed suffix array

We use a practical RLCSA [18, called RLFM+ in there] that uses $(r \lg \sigma + 2r \lg(n/r))(1 + o(1))$ bits of space and offers search time $t_{\text{search}}(m)$ in $\mathcal{O}(m \lg r) \subseteq \mathcal{O}(m \lg n)$. Since we do not need to compute cells of SA with this structure, we do not need to spend the $\mathcal{O}((n \lg n) / t_{\text{lookup}}(n))$ bits, and as a result the contribution of the RLCSA to the total space is negligible.

5.2 Grammar compressor

We choose Re-Pair [15] to obtain both \mathcal{G} and \mathcal{G}' , since it performs very well in practice. Re-Pair repeatedly replaces the most frequent pair of adjacent symbols with a new nonterminal, until every pair is unique. Upon ties in frequency, we give priority to the pairs whose symbols have been generated earlier, which in practice yielded rather balanced grammars in all the cases we have tried.

Re-Pair yields a binary grammar, but the top-level is a long sequence. We then complete the grammar by artificially adding a parse tree on top of the final sequence left by Re-Pair. To minimize the height of the resulting grammar, we merge first the pairs of nonterminals with shorter parse trees.

We store the g grammar rules as an array G taking $2g \lg(g + d)$ bits, so that if A_i is the i th nonterminal of the grammar, it holds that $A_i \rightarrow A_{G[2i]}A_{G[2i+1]}$.

When building \mathcal{G}' , we concatenate all the lists D_v and separate them with unique numbers larger than d , to ensure that Re-Pair will not produce nonterminals that cross from one list to another. After running Re-Pair, we remove the separators but do not complete the grammars, as all we need is to decompress any D_v in optimal time. We represent all the reduced sets D'_v as a sequence D' , marking the beginning of each set in a bitvector B . The beginning of D'_v is found with operation $select(B, v)$, which finds the v th 1 in B . This operation can be implemented in constant time using $o(|B|)$ further bits [4].

5.3 Sampling

The largest component of our index is the set of compressed lists D'_v . To reduce this space, we store those lists only for sampled nonterminals v of \mathcal{G} . The list of a nonsampled nonterminal v is then obtained by merging those of the highest sampled descendants of v in the parse tree, which yields a space/time tradeoff.

We use a strategy similar to PDL [8], based on parameters b and β . We define a *sampled tree* by sampling some nodes from the parse tree. First, no leaf v of the sampled tree can have an expansion larger than b , so that we spend time $\mathcal{O}(b \lg b)$ to obtain its sorted list directly from \mathcal{G} . To this aim, we sample all the nonterminals v of \mathcal{G} with parent w such that $|D_v| \leq b < |D_w|$. Those are the leaves of the sampled tree, which form a partition of DA .

Second, for any nonsampled node v with $|D_v| > b$, we must be able to build D_v by merging other precomputed lists of total length $\leq \beta |D_v|$. This implies that generating D_v costs $\mathcal{O}(\beta \lg n)$ times more than having D'_v stored and just decompressing it.

We first assume the sampled tree contains all the ancestors of the sampled leaves and then proceed bottom-up in the sampled tree, removing some nodes from it. Any node v with parent w and children u_1, \dots, u_k is removed if $\sum_{i=1}^k |D_{u_i}| \leq \beta \cdot |D_v|$; the nodes u_i then become children of w in the sampled tree.

At query time, if a node v of interest is not sampled, we collect all the lists of its highest sampled descendants. Therefore, on a parse tree of height h we may end up merging many more than the original $\mathcal{O}(h)$ lists D_1, \dots, D_k , but have the guarantee that the merged lists add up to size at most $\beta \cdot (|D_1| + \dots + |D_k|)$.

<i>Collection</i>	<i>Size</i> (<i>n</i>)	<i>RLCSA</i> (bps)	<i>Docs</i> (<i>D</i>)	<i>Doc size</i> (<i>n/D</i>)	<i>Patterns</i>	<i>Occs</i> (<i>occ</i>)	<i>Doc occs</i> (<i>ndoc</i>)	<i>Occs/doc</i> ($\frac{occ}{ndoc}$)
Page	110	0.18	60 1 919 382	7658	781	3	242.75	
	641	0.11	190 3 534 921	14 286	2601	6	444.79	
	1037	0.13	280 3 883 145	20 536	2889	7	429.04	
Revision	110	0.18	8834 13 005	7658	776	371	2.09	
	640	0.11	31 208 21 490	14 284	2592	1065	2.43	
	1035	0.13	65 565 16 552	20 536	2876	1188	2.42	
Influenza	137	0.32	100 000 1436	269	532 739	88 525	6.02	
	321	0.26	227 356 1480	269	1 248 428	202 437	6.17	

Table 1: Statistics for document collections (small, medium, and large variants): *Collection* name; *Size* in megabytes; *RLCSA* bits per symbol (bps); *Docs*, number of documents; *Doc size*, average document length; number of *Patterns*; *Occs*, average number of occurrences; *Doc occs*, average number of document occurrences; *Occs/doc*, average ratio of occurrences to document occurrences.

To merge the lists we use a classical binary heap instead of an atomic heap, so the cost per merged element is $\mathcal{O}(\lg n)$.

We may then spend $k \cdot b \lg b = \mathcal{O}(hb \lg b)$ time in extracting and sorting the lists D_v of size below b . The other lists D_v may lead to merging $\beta |D_v|$ elements. The total cost over the $k = \mathcal{O}(h)$ lists is then $\mathcal{O}(hb \lg b + \beta(|D_1| + \dots + |D_k|) \lg n) \subseteq \mathcal{O}(hb \lg b + ndoc \cdot \beta h \lg n)$. In terms of complexity, if we choose for example $b = \mathcal{O}(\lg n / \lg \lg n)$, $\beta = \mathcal{O}(1)$, and the grammar is balanced, $h = \mathcal{O}(\lg n)$, then the total cost of merging is $\mathcal{O}(ndoc \cdot \lg^2 n)$.

6 Experiments and Results

We evaluate different variants of our indexes and compare them with the state of the art. We use the experimental framework proposed by Gagie et al. [8].

6.1 Document collections

To test various kinds of repetitiveness scenarios, we performed several experiments with real and synthetic datasets. We used the same document collections tested by Gagie et al. [8], available at jtsiren.kapsi.fi/rlcsa. Table 1 summarizes some statistics on the collections and the patterns used in the queries.

Real collections. **Page** and **Revision** are collections formed by all the revisions of some selected pages from the Wikipedia in Finnish language. In **Page**, there is a document for each selected article, that also includes all of its revisions. In the case of **Revision**, each page revision becomes a separate document. **Influenza** is a repetitive collection composed of sequences of the H. influenzae virus genomes.

Synthetic collections. We also used two types of synthetic collections to explore the effect of collection repetitiveness on document listing performance in more detail. **Concat** and **Version** are similar to **Page** and **Revision**, respectively. We use 10 and 100 base documents of length 1000 each, extracted at random from the English file of Pizza&Chili (`pizzachili.dcc.uchile.cl`). Besides, we include variants of each base document, generated using different mutation probabilities (0.001, 0.003, 0.01, and 0.03). A mutation is a replacement by a different random symbol. In collection **Version**, each variant becomes a separate document. In **Concat**, all variants of the same base document form a single document.

Queries. The query patterns for **Page** and **Revision** datasets are Finnish words of length ≥ 5 that occur in the collections. For **Influenza**, the queries are substrings of length 4 extracted from the dataset. In the case of **Concat** and **Version**, the patterns are terms selected from an MSN query log. See Gagie et al. [8] for a more detailed description.

6.2 Compared indexes

Grammar-Compressed Document Array (GCDA). This is our main proposal. We use the balanced Re-Pair compressor implemented by Navarro (`www.dcc.uchile.cl/gnavarro/software/repair.tgz`). To sample the parse tree, we test several parameter configurations for the block size b and factor β .

Brute force (Brute). This family of algorithms is the most basic solution to the document listing problem. They use a CSA to retrieve all the document identifiers in $DA[sp..ep]$, sort them, and report each of them once. **Brute-L** uses the CSA to extract the values $DA[i]$. **Brute-D**, instead, uses an explicit document array DA . Finally, **Brute-C** is our variant using the grammar-compressed DA . From the grammar tree of height h and storing the length of the expansion of each nonterminal, we extract the range $DA[sp..ep]$ in time $\mathcal{O}(h + ep - sp)$.

Sadakane (Sada). **Sada-L** is the original index of Sadakane [26]. **Sada-D** speeds up the query time by explicitly storing DA . **Sada-C** stores DA in grammar-compressed form, where each individual cell $DA[i]$ is extracted in time $\mathcal{O}(h)$.

Interleaved Longest Common Prefix (ILCP). **ILCP-L** implements the ILCP index of Gagie et al. [8] using a run-length encoded ILCP array. **ILCP-D** is a variant that uses the document array instead of the CSA functionality. **ILCP-C** uses, instead, our grammar-compressed DA , which accesses any cell in time $\mathcal{O}(h)$.

Precomputed Document Lists (PDL). **PDL-BC** and **PDL-RP** implement the PDL algorithm proposed by Gagie et al. [8]. **PDL-BC** uses a Web graph compressor [12] on the set of lists, whereas **PDL-RP** uses Re-Pair compression. Both use block size $b = 256$ and factor $\beta = 16$, as recommended by their authors.

Grammar-based (*Grammar*). This is an implementation of the index by Claude and Munro [5]. It uses Re-Pair on the collection \mathcal{T} and on the set of lists. This index is the only tested solution that does not use a CSA.

We implemented GCDA on C++, using several succinct data structures from the SDSL library (github.com/simongog/sdsl-lite). We used existing C++ implementations of the indexes Brute, Sada, ILCP and PDL, which were tested by Gagie et al. [8] (jlttsiren.kapsi.fi/software/doclist.tgz), and modified the versions -C by using DA in grammar-compressed instead of in plain form.

All tested indexes except **Grammar** use a suffix array to compute the interval $[sp..ep]$ corresponding to pattern P . We used a RLCSA implementation (jlttsiren.kapsi.fi/rlcsa) that is optimized for repetitive text collections. To compute entries $SA[i]$, the RLCSA uses a suffix array sampling, which requires significant space as explained. Our index does not use this operation, but it is required for the indexes Brute-L, Sada-L, ILCP-L, and both variants of PDL. We use 32 as the value for this sample rate, as it gave good results in previous tests [8]. The exception is Brute-L, which uses a RLCSA optimized to extract whole ranges $SA[sp..ep]$ [10] (github.com/nicolaprezza/r-index). The column *RLCSA* of Table 1 gives the space used by the RLCSA without suffix array samples.

Our machine has two Intel(R) Xeon(R) CPU E5-2407 processors @ 2.40 GHz and 250 GiB RAM. The operating system was Debian Linux kernel 4.9.0-8-amd64. All indexes were compiled using g++ version 6.3.0 with flags `-O3 -DNDEBUG`.

6.3 Tuning our main index

Figure 2 shows the tradeoff between time and space of GCDA on small real collections. We tested GCDA with 4 different sizes of block b : 128, 256, 512, and 1024. For each block size, we used 3 different factors β (4, 8, and 16), which are represented with increasing color darkness in the plots. The configuration $b = 512$ and $\beta = 4$ shows to be a good general-purpose choice of parameter values, and we stick to it from now on.

The lower-right plot of Figure 2 shows the space required by the main components of our index. As the number of documents in the collection grows and their size decreases, the weight of the grammar-compressed DA , and even more, of the grammar-compressed lists of documents, becomes dominant. Note also that *Influenza* is the least repetitive collection.

6.4 Comparison on real collections

Figures 3 and 4 show the tradeoff between time and space for all tested indexes on the real collections. Our main index, GCDA, and the -C variants of the other indexes we adapted, are clearly dominant in a large portion of the space/time map. Most of the previous indexes are way slower, way larger, or both, than ours. The best previous tradeoffs, PDL-BC and PDL-RP [8], are much closer, but still they are almost always slower and larger than GCDA.

For all versions of *Page*, where there are few large documents and our grammars compress very well, GCDA requires only 0.48–0.56 bits per symbol (bps)

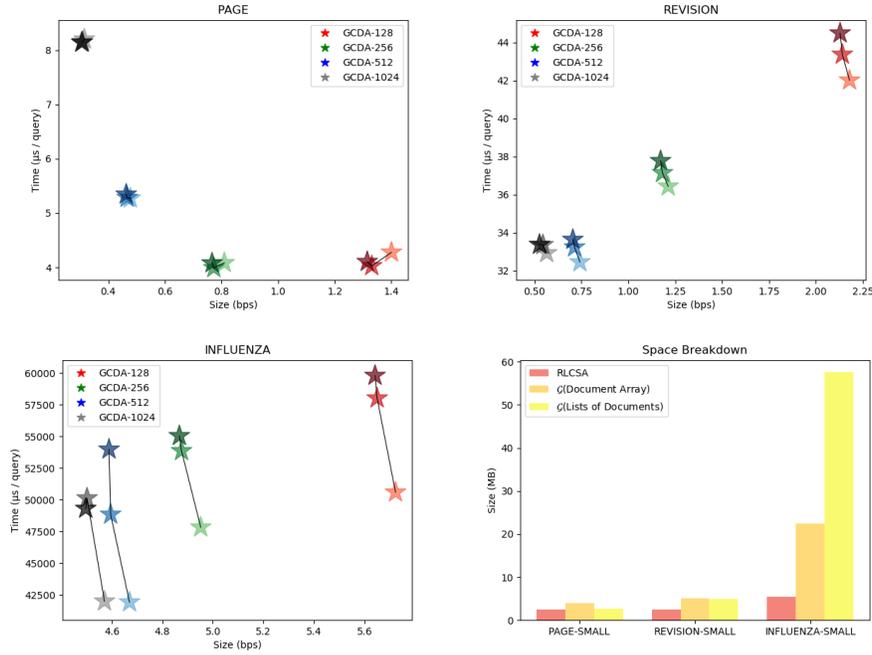


Fig. 2: GCDA on small real collections with different configurations. The x axis shows the total size of the index in bps. The y axis shows the average time per query in μsec . Beware that the plots do not start at zero. The lower-right plot shows the size of the main components of GCDA on the small collections; the y axis shows the size in megabytes.

and answers queries in less than 16 microseconds (μsec). The index using the least space is *Grammar*, which requires 0.21–0.35 bps. *Grammar* is way out of the plot, however, because it requires 1.2–3.4 milliseconds (msec) to solve the queries, that is, 205–235 times slower than GCDA (as in previous work [8], *Grammar* did not build on the largest dataset of *Page*). The next smallest index is our variant *Brute-C*, which uses 0.35–0.55 bps and is generally smaller than GCDA, but slower by a factor of 2.6–6.7. *Brute-L*, occupying 0.38–0.60 bps, is also smaller in some cases, but much slower (180–1080 μsec , out of the plot). GCDA sharply outperforms all the other indexes in space, and also in time (only *Sada-D* is 6% faster in the small collection, yet using 18 times more space). The closest competitors, *PDL-BC* and *PDL-RP*, are 4.4–5.0 times larger and 2.8–5.0 times slower than GCDA.

In the case of *Revision*, where there are more and smaller documents, GCDA uses 0.73–0.88 bps and answers queries in less than 150 μsec . Again *Grammar* uses the least space, 0.26–0.42 bps, but once again at the price of being 8–30 times slower than GCDA. The case of *Brute-L* is analogous: 0.38–0.60 bps but over

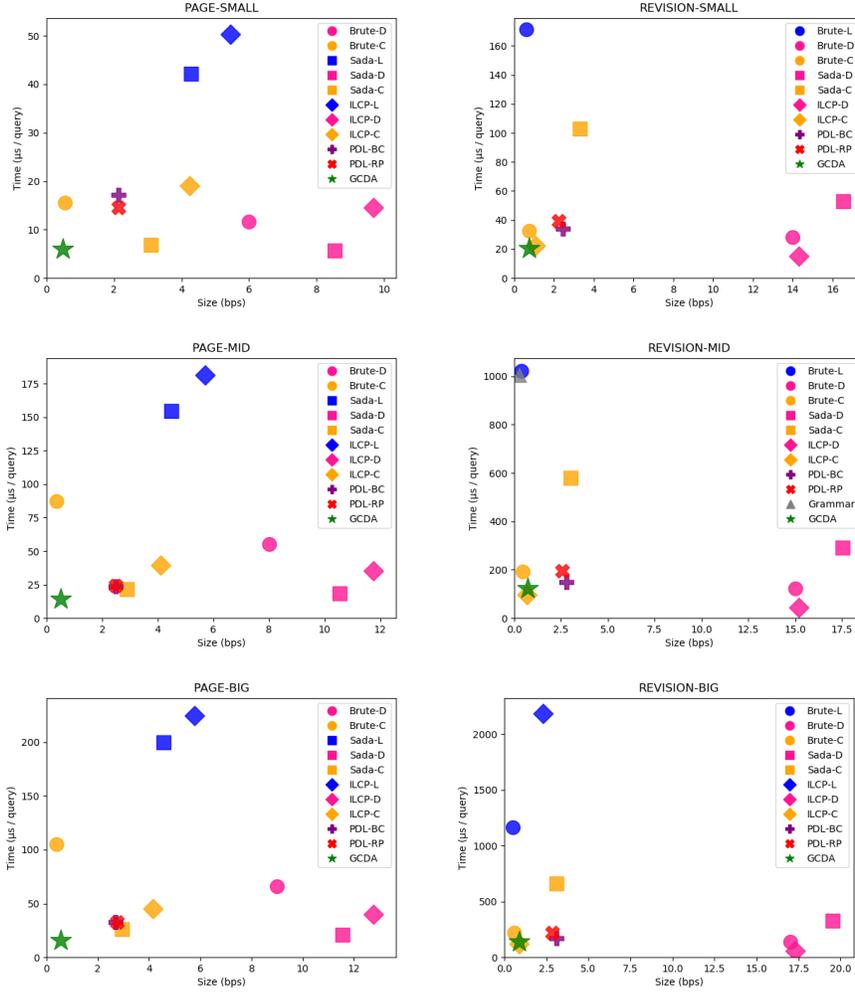


Fig. 3: Document listing indexes on real repetitive collections **Page** and **Revision**. The x axis shows the total size of the index in bps. The y axis shows the average time per query. Combinations with excessively high time are omitted in some plots.

8 times slower than GCDA. Instead, our variant **Brute-C** is a relevant competitor, using 0.45–0.76 bps and being less than 60% slower than GCDA. The other relevant index is our variant **ILCP-C**, using almost the same space and time of GCDA. The group GCDA/Brute-C/ILCP-C forms a clear sweetpoint in this collection. The closest competitors, again PDL-BC and PDL-RP, are 3.1–3.8 times larger and 1.2–1.9 times slower than GCDA.

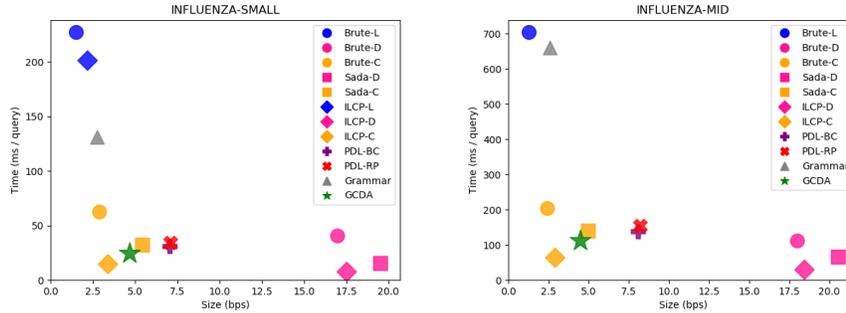


Fig. 4: Document listing indexes on real repetitive collection *Influenza*. The x axis shows the total size of the index in bps. The y axis shows the average time per query. Combinations with excessively high time are omitted in some plots.

Influenza, with many small documents, is the worst case for the indexes. GCDA uses 4.46–4.67 bps and answers queries within 115 msec. Many indexes are smaller than GCDA, but only our variants form a relevant space/time tradeoff: ILCP-C uses 2.88–3.37 bps, Brute-C uses 2.42–2.86 bps, and Sada-C uses 4.96–5.40 bps. All the -C variants obtain competitive times, and ILCP-C even dominates GCDA (it answers queries within 65 msec, taking less than 60% of the time of GCDA). The other indexes outperforming GCDA in time are -D variants, which are at least 3.7 times larger than GCDA and 5.2 times larger than ILCP-C.

6.5 Comparison on synthetic collections

Figure 5 compares the indexes on synthetic collections. These allow us study how the indexes evolve as the repetitiveness decreases, in a scenario of few large documents (*Concat*) and many smaller documents (*Version*). We combine in a single plot the results for different mutation rates of a given collection and number of base documents. The plots show the increasing mutation rates using variations of the same color, from lighter to darker. All the -L variants and Grammar are omitted because they were significantly slower.

On collection *Concat*, GCDA essentially outperforms all the other indexes. In the case of the version composed by 10 base documents, our index obtains the best space/time tradeoff by a wide margin. Only Brute-C is smaller than GCDA, but 8–9 times slower. On the other hand, various indexes are slightly faster than GCDA, but much larger (from Sada-D, which is up to 30% faster but 7 times larger, to Sada-C, which is 15% faster but at least 4 times larger). With the other variant of *Concat* (100 base documents), our index offers the best space and time for all mutation rates. Only PDL-RP is 6% faster in its best case, but 2.2 times larger. Further, GCDA retains its space/time performance as repetitiveness decreases, whereas the competing indexes worsen fast in one or both aspects.

On *Version*, composed by 10 000 documents of length 1000, GCDA is also a dominant solution, retaining its time performance as repetitiveness decreases and

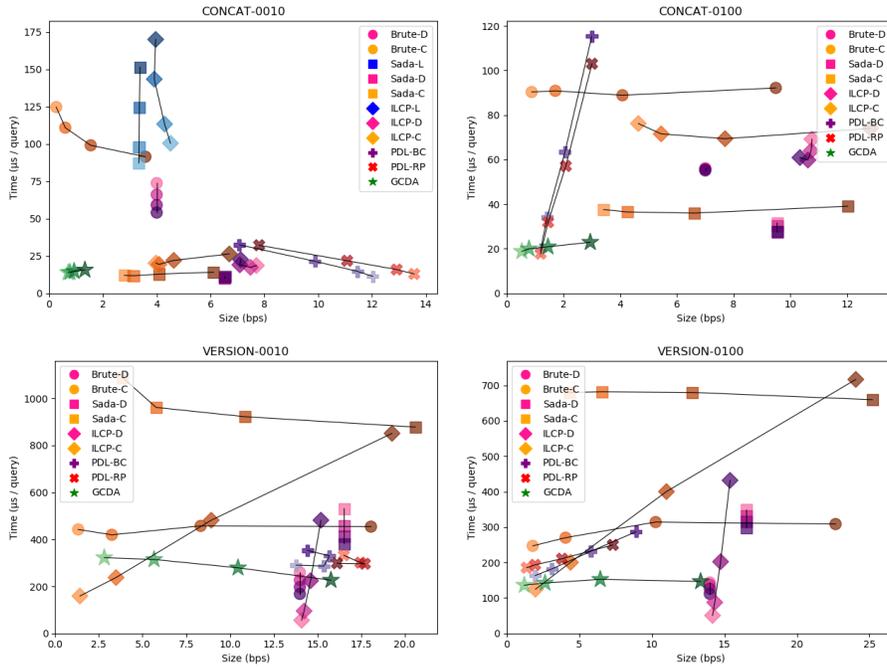


Fig. 5: Document listing on synthetic collections. The x axis shows the total size of the index in bps. The y axis shows the average time per query in μsec . Combinations with excessively high time are omitted in some plots.

outperforming all the -D variants in space up to a mutation rate of 1%. Other competing indexes are our variants Brute-C and ILCP-C (the only one dominating GCDA in some cases), as well as PDL-BC and PDL-RP in the case of 100 base documents. The strange behavior of the PDL indexes in both collections with 10 base documents is briefly discussed in the original article [8].

7 Conclusions

We have presented simple and efficient indexes for document listing on repetitive string collections. They find the $ndoc$ documents where a pattern of length m appears in a collection of size n in time $\mathcal{O}(m + ndoc \cdot \lg n)$. The indexes use grammar-compression of the document array, and perform better as the collection is more repetitive.

Our experimental results show that our main index, GCDA, outperforms the best previous solutions by a fair margin in time and/or space on various repetitive collections. From the previous indexes, only PDL [8] gets close, but it is almost always dominated by GCDA in both space and time. GCDA performs well in space for mutation rates up to 1%, whereas its query time is mostly insensitive to

the repetitiveness. Other previous solutions (especially ILCP [8] and brute force) that we adapted to run on our grammar-compressed document array also display unprecedented performance on repetitive texts, competing with GCDA.

For the final version of this paper, we plan to combine the PDL indexes with a grammar-compressed document array as well, which we omitted for lack of time. A line of future work is to further reduce the space of GCDA and our index variants that use the grammar-compressed document array, by using a more clever encoding of the grammars that may nearly halve their space at a modest increase in time [11]. Another line is to extend the index to support *top-k document retrieval*, that is, find the k documents where P appears most often.

References

1. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley, 2nd edn. (2011)
2. Belazzougui, D., Navarro, G.: Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms* **10**(4), article 23 (2014)
3. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. *IEEE Transactions on Information Theory* **51**(7), 2554–2576 (2005)
4. Clark, D.R.: Compact PAT Trees. Ph.D. thesis, University of Waterloo, Canada (1996)
5. Claude, F., Munro, J.I.: Document listing on versioned documents. In: Proc. 20th Symposium on String Processing and Information Retrieval (SPIRE). pp. 72–83. LNCS 8214 (2013)
6. Claude, F., Navarro, G.: Self-indexed grammar-based compression. *Fundamenta Informaticae* **111**(3), 313–337 (2010)
7. Fredman, M.L., Willard, D.E.: Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences* **48**(3), 533–551 (1994)
8. Gagie, T., Hartikainen, A., Karhu, K., Kärkkäinen, J., Navarro, G., Puglisi, S.J., Sirén, J.: Document retrieval on repetitive collections. *Information Retrieval* **20**, 253–291 (2017)
9. Gagie, T., Navarro, G., Prezza, N.: Fully-functional suffix trees and optimal text searching in BWT-runs bounded space. CoRR **abs/1809.02792** (2018)
10. Gagie, T., Navarro, G., Prezza, N.: Optimal-time text indexing in BWT-runs bounded space. In: Proc. 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 1459–1477 (2018)
11. González, R., Navarro, G., Ferrada, H.: Locally compressed suffix arrays. *ACM Journal of Experimental Algorithmics* **19**(1), article 1 (2014)
12. Hernández, C., Navarro, G.: Compressed representations for web and social graphs. *Knowledge and Information Systems* **40**(2), 279–313 (2014)
13. Jez, A.: A really simple approximation of smallest grammar. *Theoretical Computer Science* **616**, 141–150 (2016)
14. Kreft, S., Navarro, G.: On compressing and indexing repetitive sequences. *Theoretical Computer Science* **483**, 115–133 (2013)
15. Larsson, J., Moffat, A.: Off-line dictionary-based compression. *Proceedings of the IEEE* **88**(11), 1722–1732 (2000)

16. Lehman, E., Shelat, A.: Approximation algorithms for grammar-based compression. In: Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 205–212 (2002)
17. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing* **12**(1), 40–66 (2005)
18. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology* **17**(3), 281–308 (2010)
19. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing* **22**(5), 935–948 (1993)
20. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 657–666 (2002)
21. Navarro, G.: Indexing highly repetitive collections. In: Proc. 23rd International Workshop on Combinatorial Algorithms (IWOCA). pp. 274–279. LNCS 7643 (2012)
22. Navarro, G.: Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys* **46**(4), article 52 (2014)
23. Navarro, G.: Document listing on repetitive collections with guaranteed performance. In: Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM). p. article 4. LIPIcs 78 (2017)
24. Navarro, G., Mäkinen, V.: Compressed full-text indexes. *ACM Computing Surveys* **39**(1), article 2 (2007)
25. Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science* **302**(1-3), 211–222 (2003)
26. Sadakane, K.: Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms* **5**, 12–22 (2007)
27. Stjephen, Z.D., Lee, S.Y., Faghri, F., Campbell, R.H., Chenxiang, Z., Efron, M.J., Iyer, R., Sinha, S., Robinson, G.E.: Big data: Astronomical or genomics? *PLoS Biology* **17**(7), e1002195 (2015)
28. Weiner, P.: Linear pattern matching algorithm. In: Proc. 14th Annual IEEE Symposium on Switching and Automata Theory. pp. 1–11 (1973)