Document Listing on Repetitive Collections with Guaranteed Performance[*]

Gonzalo Navarro

Center for Biotechnology and Bioengineering, Dept. of Computer Science, University of Chile, Beauchef 851, Santiago, Chile. gnavarro@dcc.uchile.cl

Abstract

We consider document listing on string collections, that is, finding in which strings a given pattern appears. In particular, we focus on repetitive collections: a collection of size N over alphabet $[1,\sigma]$ is composed of D copies of a string of size n, and s single-character edits are applied on the copies. We introduce the first document listing index with size $\tilde{O}(n+s)$, precisely $O((n\lg\sigma+s\lg^2 N)\lg D)$ bits, and with useful worst-case time guarantees: Given a pattern of length m, the index reports the *ndoc* strings where it appears in time $O(m^2+m\lg N(\lg D+\lg^\epsilon N)\cdot \mathit{ndoc})$, for any constant $\epsilon>0$.

1998 ACM Subject Classification E.1 Data Structures; E.4 Coding and Information Theory; H.3 Information Storage and Retrieval

Keywords and phrases Repetitive string collections; Document listing; Grammar compression; Range minimum queries; Succinct data structures

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Document retrieval on general string collections is an area that has recently attracted attention [24]. On the one hand, it is a natural generalization of the basic Information Retrieval tasks carried out on search engines [1, 4], many of which are also useful on Far East languages, collections of genomes, code repositories, multimedia streams, etc. It also enables phrase queries on natural language texts. On the other hand, it raises a number of algorithmic challenges that are not easily addressed with classical pattern matching approaches.

In this paper we focus on one of the simplest document retrieval problems, *document listing* [22]. Let $\mathcal{D}$ be a collection of D documents of total length N. We want to build an index on $\mathcal{D}$ such that, later, given a search pattern P of length m, we report the identifiers of all the *ndoc* documents where P appears. Given that P may occur $\mathit{occ} \gg \mathit{ndoc}$ times in $\mathcal{D}$, resorting to pattern matching, that is, finding all the *occ* occurrences and then listing the distinct documents where they appear, can be utterly inefficient. Optimal $O(m+\mathit{ndoc})$ time document listing solutions appeared only in 2002 [22], although they use too much space. There are also more recent statistically compressed indices [29, 15] with a small time penalty.

In particular, we are interested in *highly repetitive* string collections [23], which are formed by a few distinct documents and a number of near-copies of those. Such collections arise, for example, when sequencing the genomes of thousands of individuals of a few species, when managing versioned collections of documents like Wikipedia, and in versioned software repositories. Although many of the fastest-growing datasets are indeed repetitive, this is

[*] Supported in part by Fondecyt grant 1-170048 and Basal Funds FB0001, Conicyt, Chile.

an underdeveloped area: most succinct indices for string collections are based on statistical compression, and these fail to exploit repetitiveness [19].

1.1 Our contribution

There are few document listing indices that profit from repetitiveness. A simple model to analyze them is as follows [21, 12, 23]: Assume there is a single document of size n on alphabet $[1, \sigma]$, and $D-1$ copies of it, on which s single-character edits are arbitrarily distributed, forming a collection of size $N \approx nD$. This models, for example, collections of genomes and their single-point mutations. The gold standard to measure space usage on repetitive collections is the size of the *Lempel-Ziv parsing* [20]. If we parse the concatenation of the strings in such a repetitive collection, we obtain at most $z = n/\lg_\sigma n + O(s) \ll N$ phrases. Therefore, while a statistical compressor would require basically $N \lg \sigma$ bits if the base document is incompressible [19], we can aim to reach as little as $O(n \lg \sigma + s \lg N)$ bits by expoiting repetitiveness via Lempel-Ziv compression.

This might be too optimistic for an index, however, as there is no known way to extract substrings efficiently from Lempel-Ziv compressed text. Instead, *grammar compression* allows extracting any text symbol in logarithmic time using $O(r \lg N)$ bits, where r is the size of the grammar [3, 31]. It is possible to obtain a grammar of size $r = O(z \lg(N/z))$ [5, 16], which using standard methods [28] can be tweaked to $r = n/\lg_\sigma N + s \lg N$ under our repetitiveness model. Thus the space we might aim at for indexing is $O(n \lg \sigma + s \lg^2 N)$ bits.

Although they perform reasonably well in practice, none of the preceding structures for document listing on repetitive collections [8, 12] offer good worst-case time guarantees combined with space guarantees that are appropriate for repetitive collections, that is, growing with $n+s$ rather than with N. Those offering search times of the form $O(\mathrm{poly}(m, \lg N) \cdot ndoc)$ require space of the form $O(N/\mathrm{poly}(\lg N))$. In this paper we present the *first index offering good guarantees in space and time*. Namely, our index

1. uses $O((n \lg \sigma + s \lg^2 N) \lg D)$ bits of space, and
2. performs document listing in time $O(m^2 + m \lg N (\lg D + \lg^\epsilon N) \cdot ndoc)$, for any constant $\epsilon > 0$.

That is, our index is an $O(\lg D)$ space factor away from what could be hoped from a grammar-based index. We actually build on a grammar-based document listing index [8] that stores lists of the documents where each nonterminal appears, and strenghten it by rearranging the nonterminals in different orders, following a wavelet tree [14] deployment that guarantees that only $O(m \lg r)$ ranges of lists have to be merged at query time. We do not store the lists themselves in various orders, but just succinct range minimum query (RMQ) data structures [11] that allow implementing document listing on ranges of lists [29]. Those RMQ structures are further compressed because their underlying data has long increasing runs, so the structures are reduced with techniques analogous to those developed for the ILCP data structure [12]. The space reduction brings new issues, however, because we cannot afford storing the underlying RMQ sequences. These problems are circumvented with a new, tailored, technique to extract the distinct documents in a range.

2 Related work

The first optimal-time and linear-space solution to document listing is due to Muthukrishnan [22], who solves the problem in $O(m + ndoc)$ time using an index of $O(N \lg N)$ bits of space. Later solutions [29, 15] improved the space to essentially the statistical entropy of $\mathcal{D}$, at the price of multiplying the times by low-order polylogs of N (e.g., $O(m + \lg N \cdot ndoc)$ time with $O(N)$ bits on top of the entropy). As said, however, statistical entropy does not capture repetitiveness well [19], and thus these solutions are not satisfactory in repetitive collections.

There has been a good deal of work on pattern matching indices for repetitive string collections, building on various principles (see [26, Sec 13.2]). However, there has been little work on document retrieval structures for repetitive string collections.

One precedent is Claude and Munro's index based on grammar compression [8]. It builds on a grammar-based pattern-matching index [10] and adds an *inverted index* that explicitly indicates the documents where each nonterminal appears; this inverted index is also grammar-compressed. To obtain the answer, an unbounded number of those lists of documents must be merged. No relevant worst-case time or space guarantees are offered.

Another precedent is ILCP [12], where it is shown that the longest common prefix array (LCP) of repetitive collections has long increasing runs. Then an index of size bounded by the runs in the suffix array [21] and in the LCP array performs document listing in time $O(\mathsf{search}(m) + \mathsf{lookup}(N) \cdot ndoc)$, where search and lookup are the search and lookup time, respectively, of a run-length compressed suffix array [21]. Yet, there are only average-case bounds for the size of the structure in terms of s: $O(n \lg N + s \lg^2 N)$ bits. A more serious problem is that, to obtain $\mathsf{lookup}(N)$ time per document, a suffix array sampling of $O(N \lg N/\mathsf{lookup}(N))$ bits must be stored.

The last previous work is PDL [12], which stores inverted lists at sampled nodes in the suffix tree of $\mathcal{D}$, and then grammar-compresses the set of inverted lists. For a sampling step b, it requires $O((N/b) \lg N)$ bits plus the (unbounded) space of the inverted lists. Searches that lead to the sampled nodes have their answers precomputed, whereas the others cover a suffix array range of size $O(b)$ and are solved by brute force in time $O(b \cdot \mathsf{lookup}(N))$. Again, the suffix array sampling of $O(N \lg N/\mathsf{lookup}(N))$ bits is necessary.

3 Basic Concepts

3.1 Listing the different elements in a range

Let $A[1, t]$ be an array of integers in $[1, D]$. Muthukrishnan [22] gives a structure that, given a range $[i, j]$, lists all the $ndoc$ distinct elements in $A[i, j]$ in time $O(ndoc)$. He defines an array $C[1, t]$ storing in $C[k]$ the largest position $l < k$ where $A[l] = A[k]$, or $C[k] = 0$ if no such position exists. Note that the leftmost positions of the distinct elements in $A[i, j]$ are exactly those k where $C[k] < i$. He then stores a data structure supporting range-minimum queries (RMQs) on C, $\textsc{rmq}_C(i, j) = \mathrm{argmin}_{i \le k \le j} C[k]$ [11]. Given a range $[i, j]$, he computes $k = \textsc{rmq}_C(i, j)$. If $C[k] < i$, then he reports $A[k]$ and continues recursively on $A[i, k-1]$ and $A[k+1, j]$. Whenever it turns out that $C[k] \ge i$ for an interval $[x, y]$, there are no leftmost occurrences of $A[i, j]$ within $A[x, y]$, so this interval can be abandoned. It is easy to see that the algorithm takes $O(ndoc)$ time and uses $O(t \lg t)$ bits of space; the RMQ structure uses just $2t + o(t)$ bits and answers queries in constant time [11].

Furthermore, the RMQ structure does not even access C, so we can replace C by a bitvector $V[1, D]$ to mark which elements have been reported. We set V initially to all zeros and replace the test $C[k] < i$ by $V[A[k]] = 0$, that is, the value $A[k]$ has not yet been reported (these tests are equivalent only if we recurse left and then right in the interval [24]). If so, we report $A[k]$ and set $V[A[k]] \leftarrow 1$. Overall, we need only $O(t + D)$ bits of space on top of A, and still run in $O(ndoc)$ time [29] (V can be reset to zeros by rerunning the query or through lazy initialization). Hon et al. [15] further reduce the extra space to $o(t)$ bits, yet increasing the time, via sampling the array C.

3.2 Wavelet trees

A wavelet tree [14] is a sequence representation that supports, in particular, two-dimensional orthogonal range queries [6, 25]. Let $(1, y_1), (2, y_2), \ldots, (r, y_r)$ be a sequence of points with $y_i \in [1, r]$, and let $S = y_1 y_2 \ldots y_r$ be the y coordinates in order. The wavelet tree is a perfectly balanced binary tree where each node handles a range of y values. The root handles $[1, r]$. If a node handles $[a, b]$ then its left child handles $[a, \mu]$ and its right child handles $[\mu + 1, b]$, with $\mu = \lfloor (a + b)/2 \rfloor$. The leaves handle individual y values. If a node handles range $[a, b]$, then it represents the subsequence $S_{a,b}$ of y coordinates that belong to $[a, b]$. Thus at each level the strings $S_{a,b}$ form a permutation of S. What is stored for each such node is a bitvector $B_{a,b}$ so that $B_{a,b}[i] = 0$ iff $S_{a,b} \leq \mu$, that is, if that value is handled in the left child of the node. Those bitvectors are provided with support for rank and select queries: $\mathsf{rank}_v(B, i)$ is the number of occurrences of bit v in $B[1, i]$, whereas $\mathsf{select}_v(B, j)$ is the position of the jth occurrence of bit v in B. The wavelet tree has height $\lg r$, and its total space requirement for all the bitvectors $B_{a,b}$ is $r \lg r$ bits. The extra structures for rank and select add $o(r \lg r)$ further bits and support the queries in constant time [7]. With the wavelet tree one can recover any y_i value by tracking it down from the root to a leaf, but let us describe a more general procedure.

Let $[x_1, x_2] \times [y_1, y_2]$ be a query range. The number of points that fall in the range can be counted in $O(\lg r)$ time as follows. We start at the root with the range $S[x_1, x_2] = S_{1,r}[x_1, x_2]$. Then we project the range both left and right, towards $S_{1,\mu}[\mathsf{rank}_0(B_{1,r}, x_1 - 1) + 1, \mathsf{rank}_0(B_{1,r}, x_2)]$ and $S_{\mu+1,r}[\mathsf{rank}_1(B_{1,r}, x_1 - 1) + 1, \mathsf{rank}_1(B_{1,r}, x_2)]$, respectively, with $\mu = \lfloor (r + 1)/2 \rfloor$. If some of the ranges is empty, we stop the recursion on that node. If the interval $[a, b]$ handled by a node is disjoint with $[y_1, y_2]$, we also stop. If the interval $[a, b]$ is included in $[y_1, y_2]$, then all the points in the x range qualify, and we simply sum the length of the range to the count. Otherwise, we keep splitting the ranges recursively. It is well known that the range $[y_1, y_2]$ is covered by $O(\lg r)$ wavelet tree nodes, and that we traverse $O(\lg r)$ nodes to reach them. If we also want to report all the corresponding y values, then instead of counting the points found, we track each one individually towards its leaf, in $O(\lg r)$ time. At the leaves, the y values are sorted, so in particular if they are a permutation of $[1, r]$, we know that the ith left-to-right leaf is the value $y = i$. Thus, extracting the $nocc$ results takes time $O((1 + nocc) \lg r)$.

3.3 Range minimum queries on arrays with runs

Let $A[1, t]$ be an array that can be cut into ρ runs of nondecreasing values. Then it is possible to solve RMQs in $O(\lg \lg t)$ time plus $O(1)$ accesses to A using $O(\rho \lg(t/\rho))$ bits. The idea is that the possible minima (breaking ties in favor of the leftmost) in $A[i, j]$ are either $A[i]$ or the positions where runs start in the range. Then, we can use a sparse bitvector $M[1, t]$ marking with $M[k] = 1$ the run heads. We also define an array $A'[1, \rho]$, so that if $M[k] = 1$ then $A'[\mathsf{rank}_1(M, k)] = A[k]$. We do not store A', but just an RMQ structure on it. Hence, the minimum of the run heads in $A[i, j]$ can be found by computing the range of run heads involved, $i' = \mathsf{rank}_1(M, i - 1) + 1$ and $j' = \mathsf{rank}_1(M, j)$, then finding the smallest value among them in A' with $k' = \textsc{rmq}_{A'}(i', j')$, and mapping it back to A with $k = \mathsf{select}_1(M, k')$. Finally, the RMQ answer is either $A[i]$ or $A[k]$, so we access A twice to compare them.

This idea was used by Gagie et al. [12, Sec 3.2] for runs of equal values, but it works verbatim for runs of nondecreasing values. They show how to store M in $\rho \lg(t/\rho) + O(\rho)$ bits so that it solves rank in $O(\lg \lg t)$ time and select in $O(1)$ time, by enriching a sparse bitvector representation [27]. This dominates the space and time of the whole structure.

The idea was used even before by Barbay et al. [2, Thm. 2], for runs of nondecreasing values. They represented M using $\rho \lg(t/\rho) + O(\rho) + o(t)$ bits so that the $O(\lg \lg t)$ time becomes $O(1)$, but we are not be able to afford the $o(t)$ extra bits in this paper.

3.4 Grammar compression

Let $T[1, N]$ be a sequence of symbols over alphabet $[1, \sigma]$. Grammar compressing T means finding a context-free grammar that generates T and only T. The grammar can then be used as a substitute for T, which provides good compression when T is repetitive. We are interested, for simplicty, in grammars in Chomsky normal form, where the rules are of the form $A \rightarrow BC$ or $A \rightarrow a$, where A, B, and C are nonterminals and $a \in [1, \sigma]$ is a terminal symbol. For every grammar, there is a proportionally sized grammar in this form.

A Lempel-Ziv parse [20] of T cuts T into z phrases, so that each phrase $T[i, j]$ appears earlier in $T[i', j']$, with $i' < i$. It is known that the smallest grammar generating T must have at least z rules [28, 5], and that it is possible to convert a Lempel-Ziv parse into a grammar with $r = O(z \lg(N/z))$ rules [28, 5, 30, 17, 18]. Furthermore, such grammars can be balanced, that is, the parse tree is of height $O(\lg N)$. By storing the length of the string to which every nonterminal expands, it is easy to access any substring $T[i, j]$ from its compressed representation in time $O(j - i + \lg N)$ by tracking down the range in the parse tree. This can be done even on an unbalanced grammar [3]. The total space used by this representation, with a grammar of r rules, is $O(r \lg N)$ bits.

3.5 Grammar-based indexing

The pattern-matching index of Claude and Navarro [9] builds on a grammar in Chomsky normal form that generates a text $T[1, N]$, with $r + 1$ rules. Let $s(A)$ be the string generated by nonterminal A. Then they collect the strings $s(A)$ for all those nonterminals, except the initial symbol S. Let $C_1, \ldots, C_r$ be the nonterminals sorted lexicographically by $s(A)$ and let $B_1, \ldots, B_r$ be the nonterminals sorted lexicographically by the reverse strings, $s(A)^{rev}$. They create a set of points in $[1, r] \times [1, r]$ so that (i, j) is a point (corresponding to nonterminal A) if the rule that defines A is $A \rightarrow B_i C_j$. Those points are stored in a wavelet tree.

To search for a pattern $P[1, m]$, they first find the primary occurrences, that is, those that appear when B is concatenated with C in a rule $A \rightarrow BC$. The secondary occurrences, which appear when A is used elsewhere, are found in a way that does not matter for this paper. To find the primary occurrences, they cut P into two nonempty parts $P = P_1 P_2$, in the $m - 1$ possible ways. For each cut, they binary search for P_1^{rev} in the sorted set $s(B_1)^{rev}, \ldots, s(B_r)^{rev}$ and for P_2 in the sorted set $s(C_1), \ldots, s(C_r)$. Let $[x_1, x_2]$ be the interval obtained for P_1 and $[y_1, y_2]$ the one obtained for P_2. Then all the points in $[x_1, x_2] \times [y_1, y_2]$, for all the $m - 1$ partitions of P, are the primary occurrences.

To search for P_1^{rev} or for P_2, the grammar is used to extract the required substrings of T in time $O(m + \lg N)$, so the overall search time to find the *nocc* primary occurrences is $O(m \lg r (m + \lg N) + \lg r \cdot nocc)$. The space used by the structure is $O(r \lg N)$ bits. Within this space one can store Patricia trees on the strings $s(B_i^{rev})$ and $s(C_i)$, to speed up binary searches and reduce the time to $O(m(m + \lg N) + \lg r \cdot nocc)$. Also, one can use the structure of Gasieniec et al. [13] that, within $O(r \lg N)$ further bits, allows extracting any prefix/suffix of any nonterminal in constant time per symbol (see also [10]). Since in our search we only access prefixes/suffixes of whole nonterminals, this further reduces the time to $O(m^2 + \lg r \cdot nocc)$.

Claude and Munro [8] extend this structure to support document listing on a collection $\mathcal{D}$ of D string documents, which are concatenated into a text $T[1, N]$. To each nonterminal

A they associate the increasing list $\ell(A)$ of the identifiers of the documents (integers in $[1, D]$) where A appears. To perform document listing, they find all the primary occurrences $A \to BC$ of all the partitions of P, and merge their lists. There is no useful worst-case time bound for this operation other than $O(nocc \cdot ndoc)$, where $nocc$ can be much larger than $ndoc$. To reduce space, they also grammar-compress the sequence of all the r lists $\ell(A)$. They also give no worst-case space bound for the compressed lists (other than $O(rD \lg D)$ bits).

4 Our Document Listing Index

We build on the basic structure of Claude and Munro [8]. Our main idea is to take advantage of the fact that the *nocc* primary occurrences to detect in Section 3.5 are found as points in the two-dimensional structure, along $O(\lg r)$ ranges within wavelet tree nodes (recall Section 3.2) for each partition of P. Instead of retrieving the *nocc* individual lists, decompressing and merging them [8], we will use the techniques to extract the distinct elements of a range seen in Section 3.1. This will drastically reduce the amount of merging necessary, and will provide useful upper bounds on the document listing time.

4.1 Structure

We store the grammar of T in a way that it allows direct access for pattern searches, as well as the wavelet tree for the points (B_i, C_j), the Patricia trees, and extraction of prefixes/suffixes of nonterminals, all in $O(r \lg N)$ bits.

Consider any sequence $S_{a,b}[1, q]$ at a wavelet tree node handling the range $[a, b]$ (recall that those sequences are not explicitly stored). Each element $S_{a,b}[k] = A_k$ corresponds to a point (i, j) associated with a nonterminal $A_k \to B_i C_j$. Then let $L_{a,b} = \ell(A_1) \cdot \ell(A_2) \cdots \ell(A_q)$ be the concatenation of the inverted lists associated with the nonterminals in $S_{a,b}$, and let $M_{a,b} = 10^{|\ell(A_1)|-1}10^{|\ell(A_2)|-1} \ldots 10^{|\ell(A_q)|-1}$ mark where each list begins in $L_{a,b}$. Now let $C_{a,b}$ be the C-array corresponding to $L_{a,b}$, as described in Section 3.1. As in that section, we do not store $L_{a,b}$ nor $C_{a,b}$, but just the RMQ structure on $C_{a,b}$, which together with $M_{a,b}$ will be used to retrieve the unique documents in a range $S_{a,b}[i, j]$.

Since $M_{a,b}$ has only r 1s out of (at most) rD bits across all the wavelet tree nodes of the same level, it can be stored with $O(r \lg D)$ bits per level [27], and $O(r \lg r \lg D)$ bits overall. On the other hand, as we will show, $C_{a,b}$ is formed by a few increasing runs, say ρ across the wavelet tree nodes of the same level, and therefore we represent its RMQ structure using the technique of Section 3.3. The total space used by those RMQ structures is then $O(\rho \lg r \lg(rD/\rho))$ bits.

Finally, we store the explicit lists $\ell(B_i)$ aligned to the wavelet tree leaves, so that the list of any element in any sequence $S_{a,b}$ is reached in $O(\lg r)$ time by tracking down the element. Those lists, of maximum total length rD, are grammar-compressed as well, just as in the basic scheme [8]. If the grammar has r' rules, then the total compressed size is $O(r' \lg(rD))$ bits to allow for direct access in $O(\lg(rD))$ time, see Section 3.4.

In total, our structure uses $O(r \lg N + r \lg r \lg D + \rho \lg r \lg(rD/\rho) + r' \lg(rD))$ bits.

4.2 Document listing

A document listing query proceeds as follows. We cut P in the $m - 1$ possible ways, and for each way identify the $O(\lg r)$ wavelet tree nodes (and ranges) where the desired points lie. Overall, we have $O(m \lg r)$ ranges and need to take the union of the inverted lists of all the points inside those ranges. We extract the distinct documents in each range and then compute their union. If a range has only one element, then we can track it to the leaves, where its list $\ell(\cdot)$ is stored, and recover it by decompressing the whole list.

Otherwise, we use in principle the document listing technique of Section 3.1. Let $S_{a,b}[i,j]$ be a range from where to obtain the distinct documents. We compute $i' = \mathsf{select}_1(M_{a,b}, i)$ and $j' = \mathsf{select}_1(M_{a,b}, j+1) - 1$, and obtain the distinct elements in $L_{a,b}[i', j']$, by using RMQs on $C_{a,b}[i', j']$. Recall that, as in Section 3.3, we use a run-length compressed RMQ structure on $C_{a,b}$. With this arrangement, every RMQ operation takes time $O(\lg\lg(rD))$ plus the time to accesses two cells in $C_{a,b}$. Those accesses are made to compare a run head with the leftmost element of the query interval, $C_{a,b}[i']$. The problem is that we have not represented the cells of $C_{a,b}$, and cannot easily compute them on the fly.

Barbay et al. [2, Thm. 3] give a sophisticated representation that determines the position of the minimum in $C_{a,b}[i', j']$ without the need to perform the two accesses on $C_{a,b}$. They need $\rho\lg(rD) + \rho\lg(rD/\rho) + O(\rho) + o(rD)$ bits, which unfortunately is too high for us[1].

Instead, we modify the way the distinct elements are obtained, so that comparing the two cells of $C_{a,b}$ is unnecessary. In the same spirit of Sadakane's solution (see Section 3.1) we use a bitvector $V[1, D]$ where we mark the documents already reported. Given a range $S_{a,b}[i,j] = A_i \ldots A_j$, we first track A_i down the wavelet tree, recover and decompress its list $\ell(A_i)$, and mark all of its documents in V. Note that all the documents in the list $\ell(\cdot)$ are different. Now we do the same with A_{i+1}, decompressing $\ell(A_{i+1})$ left to right and marking the documents in V, and so on, until we decompress a document $\ell(A_{i+d})[k]$ that is already marked in V. Only now we use the RMQ technique of Section 3.3 on the interval $C_{a,b}[i', j']$, where $i' = \mathsf{select}_1(M_{a,b}, i+d) - 1 + k$ and $j' = \mathsf{select}_1(M_{a,b}, j+1) - 1$, to obtain the next document to report. This technique, as explained, yields two candidates: one is $L_{a,b}[i'] = \ell(A_{i+d})[k]$ itself, and the other is some run head $L_{a,b}[k']$ whose identity we can obtain from the wavelet tree leaf. But we know that $L_{a,b}[i']$ was already reported, so we act as if the RMQ was always $L_{a,b}[k']$: If the RMQ answer was $L_{a,b}[i']$ then, since it is already reported, we should stop. But in this case, $L_{a,b}[k']$ is also already reported and we do stop anyway. Hence, if $L_{a,b}[k']$ is already reported we stop, and otherwise we report it and continue recursively on the intervals $C_{a,b}[i', k'-1]$ and $C_{a,b}[k'+1, j']$. On the first, we can continue directly, as we still know that $L_{a,b}[i']$ is already reported. On the second interval, instead, we must restore the invariant that the leftmost element was already reported. So we find out with M the list and position $\ell(A_t)[u]$ corresponding to $C_{a,b}[k'+1]$ (i.e., $t = \mathsf{rank}_1(M_{a,b}, k'+1)$ and $u = k'+1-\mathsf{select}_1(M,t)+1$), track A_t down to its leaf in the wavelet tree, and traverse $\ell(A_t)$ from position u onwards, reporting documents until finding one that has been reported. The correctness of this document listing algorithm is proved in Appendix A.

The $m-1$ searches for partitions of P take time $O(m^2)$. In the worst case, extracting each distinct document in the range requires an RMQ computation without access to $C_{a,b}$ ($O(\lg\lg(rD))$ time), tracking an element down the wavelet tree ($O(\lg r)$ time), and extracting an element from its grammar-compressed list $\ell(\cdot)$ ($O(\lg(rD)$ time). This adds up to $O(\lg(rD))$ time per document extracted in a range. In the worst case, however, the same documents are extracted over and over in all the $O(m\lg r)$ ranges, and therefore the final search time is $O(m^2 + m\lg r\lg(rD)\cdot ndoc)$.

[1] Even if we get rid of the $o(rD)$ component, the $\rho\lg(rD)$ term becomes $O(s\lg^3 N)$ in the final space, which is larger than what we manage to obtain. Also, using it does not make our solution faster.

5 Analysis in a Repetitive Scenario

Our structure uses $O(r \lg N + r \lg r \lg D + \rho \lg r \lg(rD/\rho) + r' \lg(rD))$ bits, and performs document listing in time $O(m^2 + m \lg r \lg(rD) \cdot ndoc)$. We now specialize those formulas under our repetitiveness model. Note that our index works on any string collection; we use the simplified model of the $D-1$ copies of a single document of length n, plus the s edits, to obtain analytical results that are easy to interpret in terms of repetitiveness. We also assume a particular strategy to generate the grammars to show that it is possible to obtain the complexities we give; the actual index may use more sophisticated ones.

5.1 Space

We assume $s \geq D-1$, since otherwise there will be identical documents, and this is easily reduced to a smaller collection with multiple identifiers per document. The documents are concatenated into $T[1, N]$, where $N \leq nD + s$. Let us make our grammar for T contain the $N^{1/3}$ nonterminals that generate all the strings of length $\frac{1}{3} \lg_\sigma N$. Then it replaces the first document with $O(n/\lg_\sigma N)$ such nonterminals, and builds a balanced parse tree of height $h = O(\lg n)$ on top of them, with nonterminal symbol S at the root. On the copies, it first covers them with $D-1$ copies of S. Now, for each edit that occurs on a copy, let $A_1, \ldots, A_h$ be the nonterminals from the leaf (where the edit is applied) to the root $A_h = S$. We create new nonterminals $A'_1, \ldots, A'_{h'}$ so that $h' \leq h+1$ and $A'_{h'} = S'$ generates the modified document. All the other nonterminals can be reused. Therefore, the maximum height h' of the final nonterminal S' rooting a modified document is $O(\lg(n+s))$, which is reached when many of the edits apply to a single copy. The final grammar size is then $r = O(N^{1/3} + n/\lg_\sigma N + s \lg(n+s)) = O(n/\lg_\sigma N + s \lg N)$, where we used that either n or s is $\Omega(\sqrt{N})$ because $N \leq nD + s \leq n(s+1) + s$. Once all the edits are applied, we add a balanced tree on top of those r symbols, which asymptotically does not change r (we may also avoid this final tree and access the documents individually, since our accesses never cross document borders).

Let us now bound ρ. If there are no edits, then every nonterminal appears in all the documents, so all the lists are of the form $\ell(A) = 1, 2, \ldots, D$. Therefore, all the corresponding C values are $C[k] = k - D$, and C has just one nondecreasing run (the first D values are 0, and thus included in the run too). Let us consider the effect of an edit operation at some document d. When we update the upward path $A_1, A_2, \ldots, A_h$ and create nonterminals $A'_1, A'_2, \ldots, A'_{h'}$ to reflect the edit, document d may disappear from all the lists $\ell(A_i)$. Each of those (up to) h' disappeared documents produces a change in C, where the cell that pointed to the disappeared position now points earlier, and this may break one run. There are other h' updates due to the creation of the lists for the nonterminals A'_i. Overall, array C undergoes $O(\lg N)$ run breaks per edit, and therefore it has a total of $\rho = O(s \lg N)$ runs.

The analysis of r' is analogous. When there are no edits and $\ell(A) = 1, 2, \ldots, D$ for all nonterminals A, we can represent the lists with a grammar of $O(D)$ symbols generating one list from nonterminal U, and then $r-1$ copies of U. Now, an edit in a document d that removes d from the lists of nonterminals $A_1, \ldots, A_h$ produces $O(\lg N)$ edits in the lists $\ell(A_1), \ldots, \ell(A_h)$ (and new lists $\ell(A'_i)$ as well). As done for the text, the grammar needs to add $O(\lg D)$ nonterminals to modify the copy of U of each list $\ell(A_i)$, from the point where d disappears to the root U. The new lists $\ell(A'_i)$ also fit within the same space. Therefore, the final grammar is of size $r' = O(D + s \lg N \lg D) = O(s \lg N \lg D)$. Instead of adding a balanced grammar tree over the r resulting nonterminals U', we retain direct pointers to those roots. As a result, the lists, of maximum length D, need $O(r' \lg D)$ bits and can be accessed in time $O(\lg D)$. From the wavelet tree, however, we still have to pay also the $O(\lg r)$ time needed to identify the list to access.

Therefore, the total size of the index can be expressed as follows. The $O(r \lg r \lg D)$ bits coming from the sparse bitvectors M, is $O(r \lg N \lg D)$ (since $\lg r = \Theta(\lg(ns)) = \Theta(\lg N)$), and thus it is $O(n \lg \sigma \lg D + s \lg^2 N \lg D)$. This subsumes the $O(r \lg N)$ bits of the grammar and the wavelet tree. The $O(\rho \lg r \lg(rD/\rho))$ bits of the structures C are monotonically increasing with ρ, so since $\rho = s \lg N \leq r$, we can upper bound it by replacing ρ with r, obtaining $O(r \lg r \lg D)$ as in the space for M. Finally, the $O(r' \lg D)$ bits of the explicit inverted lists are $O(s \lg N \lg^2 D)$. Overall, the structures add up to $O((n \lg \sigma + s \lg^2 N) \lg D)$ bits. Note that we can also analyze the space required by Claude and Munro's structure [8], which is $O(r \lg N)$ bits plus the inverted lists, $O(n \lg \sigma + s \lg N (\lg N + \lg^2 D))$ bits. Although smaller than ours, their search time has no useful bounds.

5.2 Time

Our search time is $O(m^2 + m \lg r \lg(rD) \cdot ndoc) = O(m^2 + m \lg^2 N \cdot ndoc)$. The $O(\lg(rD))$ cost corresponds to accessing a list $\ell(A)$ from the wavelet tree, and includes the $O(\lg r)$ time to reach the leaf and the $O(\lg D)$ time to access a position in the grammar-compressed list. It is possible to reduce the $O(\lg r)$ wavelet tree time by spending more space. The trick is to track the positions upwards to the root, not downwards to the leaves, and associate the lists $\ell(A)$ aligned to the root order. It is possible to reach the root position of a symbol in time $O((1/\epsilon) \lg^\epsilon r)$ by using $O((1/\epsilon) r \lg r)$ bits [6, 25], for any $\epsilon > 0$. By using a constant ϵ we obtain our main result.

▶ **Theorem 1.** *Let collection $\mathcal{D}$, of total size N, be formed by an initial document of length n plus $D-1$ copies of it, with s single-character edit operations applied on the copies. Then $\mathcal{D}$ can be represented within $O((n \lg \sigma + s \lg^2 N) \lg D)$ bits, so that the ndoc documents where a pattern of length m appears can be listed in time $O(m^2 + m \lg N (\lg D + \lg^\epsilon N) \cdot ndoc)$, for any constant $\epsilon > 0$.*

We can also obtain other tradeoffs. For example, with $\epsilon = 1/\lg\lg r$ we obtain $O((n \lg \sigma + s \lg^2 N)(\lg D + \lg\lg N))$ bits of space and $O(m^2 + m \lg N (\lg D + \lg\lg N) \cdot ndoc)$ search time.

6 Conclusions

We have presented the first document listing index with worst-case space and time guarantees that are useful for repetitive collections. On a collection of size N formed by an initial document of length n and $D-1$ copies it, with s single-character edits applied on the copies, our index uses $O((n \lg \sigma + s \lg^2 N) \lg D)$ bits and lists the *ndoc* documents where a pattern of length m appears in time $O(m^2 + m \lg N (\lg D + \lg^\epsilon N) \cdot ndoc)$, for any constant $\epsilon > 0$. We also prove a slightly lower space bound on a previous index that had not been analyzed [8], but which has no useful worst-case time bounds for listing.

The space of our index is an $O(\lg D)$ factor away from what can be expected from a grammar-based index. This is the price paid for storing the inverted lists of the nonterminals. An important question is whether this space factor can be removed, that is, if the inverted lists can be represented within the grammar-compressed size of the text itself.

Another interesting question is whether there exists an index (or a better analysis of this index) whose space and time can be bounded on more general repetitiveness measures of the collection, for example in terms of the number z of Lempel-Ziv phrases into which it can be parsed. In our model it holds $z \leq n/\lg_\sigma n + O(s)$, but other kinds of plausible repetitive collections have $s \gg z$, for example if the edits apply to ranges of documents, or if they involve blocks of text inserted, deleted, or moved around. Typical grammar-based pattern matching indices [9, 10] require $O(r \lg N) = O(z \lg^2 N)$ bits in general; it would be good to obtain the same in the document-listing grammar-based indices.

Finally, there is the question of how much of the theoretical improvements over previous work [8] can be translated into practice. This is also a subject of future work. On one hand, our upper bounds are utterly pessimistic when they assume that the same documents will be reported $O(m \lg r)$ times; the average case should be much better. On the other hand, practical improvements are possible over the basic theoretical ideas presented, which should allow us effectively avoid the cases where the previous index deviates significantly from our worst-case time guarantees, without ruining the cases where it performs well. For example, we can list the documents by brute force when the wavelet tree ranges are short, and use the document listing algorithm only on the long ones, where it is worth applying.

References

1 R. Baeza-Yates and B. Ribeiro. *Modern Information Retrieval.* Addison-Wesley, 1999.

2 J. Barbay, J. Fischer, and G. Navarro. LRM-trees: Compressed indices, adaptive sorting, and compressed permutations. *Theoretical Computer Science*, 459:26–41, 2012.

3 P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. S. Rao, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015.

4 S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines.* MIT Press, 2010.

5 M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

6 B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

7 D. R. Clark. *Compact PAT Trees.* PhD thesis, University of Waterloo, Canada, 1996.

8 F. Claude and J. I. Munro. Document listing on versioned documents. In *Proc. 20th SPIRE*, LNCS 8214, pages 72–83, 2013.

9 F. Claude and G. Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, 2010.

10 F. Claude and G. Navarro. Improved grammar-based compressed indexes. In *Proc. 19th SPIRE*, LNCS 7608, pages 180–192, 2012.

11 J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

12 T. Gagie, K. Karhu, G. Navarro, S.J. Puglisi, and J. Sirén. Document listing on repetitive collections. In *Proc. 24th CPM*, LNCS 7922, pages 107–119, 2013.

13 L. Gasieniec, R. Kolpakov, I. Potapov, and P. Sant. Real-time traversal in grammar-based compressed files. In *Proc. 15th DCC*, page 458, 2005.

14 R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 636–645, 2003.

15 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.

16 D. Hucke, M. Lohrey, and C. P. Reh. The smallest grammar problem revisited. In *Proc 23rd SPIRE*, LNCS 9954, pages 35–49, 2016.

17 A. Jez. Approximation of grammar-based compression via recompression. *Theoretical Computer Science*, 592:115–134, 2015.

18 A. Jez. A really simple approximation of smallest grammar. *Theoretical Computer Science*, 616:141–150, 2016.
19 S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.
20 A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.
21 V. Mäkinen, G. Navarro, J. Sirén, and N. Valimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.
22 S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. 13th SODA*, pages 657–666, 2002.
23 G. Navarro. Indexing highly repetitive collections. In *Proc. 23rd IWOCA*, LNCS 7643, pages 274–279, 2012.
24 G. Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys*, 46(4):article 52, 2014.
25 G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.
26 Gonzalo Navarro. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016.
27 D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th ALENEX*, pages 60–70, 2007.
28 W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.
29 K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.
30 H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *Journal of Discrete Algorithms*, 3(2–4):416–430, 2005.
31 E. Verbin and W. Yu. Data structure lower bounds on random access to grammar-compressed strings. In *Proc. 24th CPM*, LNCS 7922, pages 247–258, 2013.

A Proof of Correctness

We prove that our new document listing algorithm is correct. We remind that the algorithm proceeds as follows, to find the distinct elements in $A[sp, ep]$. It starts recursively with $[i, j] = [sp, ep]$ and remembers the documents that have already been reported, globally. To process interval $[i, j]$, it considers $A[i], A[i + 1], \ldots$ until finding an already reported element at $A[d]$. Then it finds the minimum $C[k]$ in $C[d, j]$. If $A[k]$ has been reported already, it stops; otherwise it reports $A[k]$ and proceeds recursively in $A[d, k - 1]$ and $A[k + 1, j]$, in this order. (The algorithm does this without noticing the cases where $k = d$, but this is correct, as explained in Section 4.2).

▶ **Lemma 2.** *The described algorithm reports the ndoc distinct elements in $A[sp, ep]$ in $O(ndoc)$ steps.*

Proof. We prove that the algorithm reports the leftmost occurrence in $A[sp, ep]$ of each distinct element. In particular, we prove by induction on i (and, upon ties, on $j - i$) that, when run on any subrange $[i, j]$ of $[sp, ep]$, (1) every leftmost occurrence in $A[sp, i - 1]$ is already reported before processing $[i, j]$ and (2) every leftmost occurrence in $A[sp, j]$ is reported after processing $[i, j]$. Invariant (1) holds for $[i, j] = [sp, ep]$, and the recursive procedure always produces intervals with nondecreasing values of i. Then the base case $i = j$ is trivial: the algorithm checks $A[i]$ and reports it if it was not reported before. On a larger interval $[i, j]$, the algorithm first reports $d - i$ occurrences of distinct elements in $A[i, d - 1]$.

Since these were not reported before, by invariant (1) they must be leftmost occurrences in $[sp, ep]$, and thus after doing this the invariant (1) holds for any range starting at d.

Now, we compute the position k with minimum $C[k]$ in $C[d, j]$. Note that $A[k]$ is a leftmost occurrence iff $C[k] < sp$. In this case, it has not been reported before and thus it must be reported by the algorithm. The algorithm then recurses on $A[d, k-1]$, reports $A[k]$, and finally recurses on $A[k+1, j]$.[2] Since those subintervals are inside $[i, j]$, we can apply induction. In the call on $A[d, k-1]$, the invariant (1) holds and thus by induction we have that after the call the invariant (2) holds, so all the leftmost occurrences in $A[sp, k-1] = A[sp, d-1] \cdot A[d, k-1]$ have been reported. After we report $A[k]$ too, the invariant (1) also holds for the call on $A[k+1, j]$, so by induction all the leftmost occurrences in $A[sp, j]$ have been reported when the call returns.

In case $C[k] \geq sp$, $A[k]$ is not a leftmost occurrence in $A[sp, ep]$, and moreover there are no leftmost occurrences in $A[d, j]$, so we can stop since all the leftmost occurrences in $A[sp, j] = A[sp, d-1] \cdot A[d, j]$ are already reported. Indeed, if the leftmost occurrence of $A[k]$ is in $A[sp, d-1]$, then we had already reported it by invariant (1), so the algorithm stops.

Then the algorithm is correct. As for the time, clearly the algorithm never reports the same element twice. The sequential part reports $d - i$ documents in time $O(d - i + 1)$. The extra $O(1)$ can be charged to the caller, as well as the $O(1)$ cost of the subranges that do not produce any result. Each calling procedure reports at least one element $A[k]$, so it can absorb those $O(1)$ costs, for a total cost of $O(ndoc)$. ◀

[2] Since $A[k]$ does not appear in $A[d, k-1]$, the algorithm also works if $A[k]$ is reported before the recursive calls, which makes it real-time.