A Lempel-Ziv Compressed Structure for Document Listing ⋆

Héctor Ferrada and Gonzalo Navarro

Department of Computer Science, University of Chile.
{hferrada,gnavarro}@dcc.uchile.cl

Abstract. Document listing is the problem of preprocessing a set of sequences, called documents, so that later, given a short string called the pattern, we retrieve the documents where the pattern appears. While optimal-time and linear-space solutions exist, the current emphasis is in reducing the space requirements. Current document listing solutions build on compressed suffix arrays. This paper is the first attempt to solve the problem using a Lempel-Ziv compressed index of the text collections. We show that the resulting solution is very fast to output most of the resulting documents, taking more time for the final ones. This makes this index particularly useful for interactive scenarios or when listing some documents is sufficient. Yet, it also offers a competitive space/time tradeoff when returning the full answers.

1 Introduction

The classical Information Retrieval (IR) problems aimed at natural language text collections can be naturally generalized to general sequence collections. Such *general document retrieval* problems are of interest in various areas like bioinformatics, multimedia databases, software repositories, and so on [17]. Moreover, IR on Oriental languages like Chinese and Korean also regards the texts as general sequences, since inverted indexes do not handle well those languages.

In this paper we focus on the simplest document retrieval problem, called *document listing*. Given D *documents*, which are strings $d_1 \dots d_D$ over an alphabet of size σ, each terminated with a special symbol \$, we preprocess them to build an *index*. Later, given a pattern $p[1,m]$ over the same alphabet, we must list the ndoc documents where p appears.

Muthukrishnan [15] solved this problem in optimal time $O(m + \mathsf{ndoc})$, using an index of $O(n)$ words of space, where $n = \sum |d_i|$ is the total length of the documents. This space usage, albeit linear, is very large in practice. Much subsequent research focused on reducing the space requirements. One research line [23, 8, 7, 19] achieved about $O(m + \mathsf{ndoc} \lg D)$ time and $|\mathsf{CSA}| + n \lg D + O(n)$ bits of space, where CSA is a compressed suffix array [18] of T. The CSA has a space close to that of the compressed text and can replace it. They achieve in practice fast document listing, but the extra space $n \lg D$ is still considerable. A

⋆ Partially funded by Fondecyt grant 1-110066, Chile.

second research line [22, 11] reduced the space to $|\mathsf{CSA}| + o(n)$ bits, but with the higher listing time $O(m + \mathsf{ndoc}\ \lg^{1+\varepsilon} n)$.

In this paper we propose a novel alternative, which obtains low time and low extra space. We build on the idea of the LZ-index [16, 1] so as to produce a document listing index that is small thanks to LZ78 compression, whereas it can list the documents fast. While the theoretical upper bounds we can prove, $5|\mathsf{LZ78}| + O(n \lg^2 \sigma / \lg n)$ bits (where $|\mathsf{LZ78}| \approx |\mathsf{CSA}|$ is the size of the LZ78-compressed text) and $O(m^2 \lg n + \mathsf{ndoc}\, m \lg^2 n)$ time, are not too good, they are overly pessimistic. Indeed, a good part of the occurrences are listed in $O(1)$ time each. We show that the index is very fast to list those first occurrences (which usually form most of the output), becoming slower to output the final ones. This makes it ideal for interactive scenarios, where one wishes to show some results to the user as fast as possible, and there is much more time to produce further results while the user browses the first ones. Another scenario is when only a partial or approximate answer is sufficient, that is, when one simply wants to find several documents where the pattern appears. However, the index also offers a very competitive space/time combination when returning the full set of answers.

2 Related Work

Muthukrishnan [15] solved the document listing problem in optimal time $O(m + \mathsf{ndoc})$, using an index of $O(n)$ words of space. Let $T[1, n]$ be the concatenation of the D documents. Let $A[1, n]$ be the suffix array [13] of T. Muthukrishnan defined the so-called *document array* $E[1, n]$, where $E[i]$ is the identifier of the document containing the suffix $A[i]$. A new array $C[1, n]$ is defined over E as $C[i] = \max\{1 \leq k < i, E[k] = E[i]\} \cup \{0\}$, that is, the position of the previous occurrence of $E[i]$ in E, or 0 if there is no previous occurrence. Array C is then preprocessed for range minimum queries (RMQs), which are of the form $\textsc{rmq}_C(i, j) = \mathrm{argmin}_{i \leq k \leq j} C[k]$, that is, it gives the position of the minimum value in $C[i, j]$. RMQs can be solved in constant time after a linear-time preprocessing (see, e.g., [6]). Once the interval $A[sp, ep]$ of the suffixes starting with p is determined, the problem becomes that of listing the distinct values in the interval $E[sp, ep]$. The interval is found in time $O(m)$ using a suffix tree [24]. The ndoc distinct values are listed in time $O(\mathsf{ndoc})$ using the observation that the first occurrence $E[k]$ of each distinct value in $E[sp, ep]$ satisfies $C[k] < sp$. Then the process recursively finds the smallest values of $C[sp, ep]$: It first computes $k = \textsc{rmq}_C(sp, ep)$ and reports $E[k]$, then it continues recursively with $C[sp, k-1]$ and $C[k+1, ep]$. The recursion stops at any branch where $C[k] \geq sp$.

While this solution is time-optimal, it requires much space, $O(n \lg n)$ bits. Subsequent work has focused on reducing the space, giving away the optimality.

Välimäki and Mäkinen [23] proposed a low-space implementation of Muthukrishnan's structure. They used a $2n + o(n)$ bit, constant time RMQ succinct index [6] that still required access to C. They showed that access to C can be implemented by *rank* and *select* queries on E, where $\mathsf{rank}_c(E, i)$ is the number of occurrences of symbol c in $E[1, i]$ and $\mathsf{select}_c(E, j)$ is the position in E of the

jth occurrence of c. Then it holds $C[i] = \mathsf{select}_{E[i]}(E, \mathsf{rank}_{E[i]}(E, i-1))$ if we assume that $\mathsf{select}_c(E, 0) = 0$. By representing E with a multiary wavelet tree [5, 9], the space is $n \lg D + o(n)$ bits and the operations are carried out in time $O(1 + \lg D / \lg \lg n)$. Finally, the suffix tree is replaced by a compressed suffix array (CSA), of which there are many choices [18]. A recent one [3] requires $|\mathsf{CSA}| = nH_k(T) + o(nH_k(T)) + O(n)$ bits of space and finds the interval $[sp, ep]$ in time $t_{\mathsf{tsearch}}(m) = O(m)$. A slightly smaller one [2] reaches $|\mathsf{CSA}| = nH_k(T) + o(nH_k(T)) + o(n)$ bits and $t_{\mathsf{tsearch}} = O(m \lg \lg \sigma)$. Here $H_k(T)$ is the empirical kth order entropy of T [12]. Overall, their solution requires $|\mathsf{CSA}| + n \lg D + O(n)$ bits and solves the problem in time $O(t_{\mathsf{tsearch}}(m) + \mathsf{ndoc}(1 + \lg D / \lg \lg n))$.

Gagie et al. [8, 7] showed that a wavelet tree [10] can be used for document listing without any need of RMQs, but just a DFS traversal. Their index can use $|\mathsf{CSA}| + n \lg D + o(n)$ bits and their document listing time is $O(t_{\mathsf{tsearch}}(m) + \mathsf{ndoc}\ \lg(D/\mathsf{ndoc}))$. Navarro et al. [19] achieved nearly 50% compression of the wavelet tree in practice, at the price of nearly doubling the time required (these wavelet-tree based indices also solve more complex queries).

Sadakane [22] initiated another line based on the idea of Muthukrishnan, but avoiding the large $n \lg D$-bit term in the space. He replaced the RMQ solution by a constant-time one that does not need access to C. His structure needed $4n + o(n)$ bits, but more recent ones [6] require $2n + o(n)$ bits. The other use for C is to determine where to stop the recursion. Sadakane used instead a bitmap $V[1, D]$ where the already reported documents are marked. Once a branch of the recursion attempts to report a marked document, it is pruned. Finally, array E is only needed to list the document identifiers. This is done with a bit vector $B[1, n]$ that marks the positions in T where the documents start; then it holds $E[k] = \mathsf{rank}_1(B, A[k])$. Value $A[k]$ is computed by the CSA, for example in time $O(\lg^{1+\varepsilon} n)$ for any constant $\varepsilon > 0$ [2, 3]. Bitmap B can be represented in $D \lg(n/D) + O(D) + o(n)$ bits with rank queries supported in constant time [20]. Overall, the data structure requires only $|\mathsf{CSA}| + 2n + D \lg(n/D) + O(D) + o(n) = |\mathsf{CSA}| + O(n)$ bits and $O(t_{\mathsf{tsearch}}(m) + \mathsf{ndoc}\ \lg^{1+\varepsilon} n)$ time. Hon et al. [11] achieved a further reduction to $|\mathsf{CSA}| + D \lg(n/D) + O(D) + o(n)$ bits, within the same asymptotic time, by running the RMQs over blocks of $\lg^{\varepsilon} n$ cells (see Navarro [17] for comments on the correctness of this solution).

As it can be seen, all the approaches build on the suffix array. Our new approach uses instead the LZ-index, a compressed text index not based on suffix arrays but on the LZ78 compression [25] of the text.

3 The LZ-Index

The algorithm LZ78 builds a dictionary of phrases (text substrings), with the aim of replacing strings by pointers to their previous occurrences in the text. The dictionary grows as the text is processed, and the result is a sequence of n' distinct phrases ($n' \leq n / \lg_{\sigma} n$). The phrases are formed by scanning the text left to right. In each step, the method finds the longest prefix of the remaining text that is a phrase of the dictionary. It then creates a new phrase formed by

the phrase found plus the symbol following it in the remaining text. This is represented by a pointer to the dictionary and the extra character. The number of bits output by the compressor is $|\mathsf{LZ78}| = n'(\lg n + \lg \sigma) \le n\,H_k(T) + o(n \lg \sigma)$ for $k = o(\lg_\sigma n)$ [12]. The LZ-index [16] is a compressed text index built on the LZ78 parsing of the text, and it supports locating the occurrences of a pattern $p[1,m]$ in T. The index is formed by the following components.

1. ***LZTrie***: a trie composed of all the phrases produced by the LZ78 parsing. Note that the set of phrases is prefix-closed (the prefix of a phrase is also a phrase), so LZTrie has n' nodes. It stores the phrase identifiers of each node.
2. ***RevTrie***: a trie storing the reversed phrases. It is not prefix-closed, so there are *empty* nodes not associated to phrases. We collapse unary paths of empty nodes. The trie has $n_{rev} = n' + n_e \le 2n'$ nodes, where n_e empty nodes remain after collapsing. The phrase numbers of the n' nonempty nodes are stored.
3. ***Node***: an array mapping from phrase numbers to their preorder in LZTrie.
4. ***Range***: an $n' \times n'$ grid where the rows represent the phrases and the columns the reverse phrases, both in lexicographic order. If the $(k+1)$th text phrase is at row i and the kth at column j, then there is a point at (i,j) in the grid.

Thus the LZ-index uses $4|\mathsf{LZ78}|(1+o(1))$ bits of space. To search for the occurrences of pattern $p[1,m]$ we divide them into three classes: (1) those completely inside a phrase, (2) those spanning two phrases, (3) those spanning 3 phrases or more. Those are found separately.

- ***Type 1***. Search for p^r (the reversed pattern) in RevTrie, arriving at node v^r. Each node u^r descending from v^r (including v^r) corresponds to an occurrence of type 1 where p appears at the end of the phrase. The other occurrences of type 1 are the nodes u' that descend from u in LZTrie, where u corresponds to u^r. Thus, for each node u^r that is nonempty, we read the phrase id f_u of u^r, compute $u = Node(f_u)$, and report all the phrase ids in the subtree of u. This takes $O(m + occ_1)$ time, reporting the occ_1 occurrences of type 1. See Fig. 1, ignoring for now Doc_{lz}, Doc_{rev}, and $LDoc_{rev}$.
- ***Type 2***. Partition $p = p_{start} \cdot p_{end}$ in the $m-1$ possible ways, searching for p^r_{start} in RevTrie and for p_{end} in LZTrie. The subtrees found define column and row ranges in the grid *Range*, and each point in the range is a type 2 occurrence. The phrase identifiers are obtained from those stored in LZTrie using the rows of the reported points. Using a linear-space geometric data structure, the total time is $O(m^2)$ for the m searches in LZTrie and RevTrie, $O(m \lg n)$ for the m range searches, and $O(occ_2 \lg n)$ for reporting the occ_2 points found.
- ***Type 3***. Since phrases are unique, each $p[i,j]$ equal to a phrase leads to at most one occurrence of type 3. We search LZTrie incrementally for the $O(m^2)$ pattern substrings $p[i,j]$ and find their phrase ids, if any. Then we find concatenations of consecutive phrases that together form a maximal substring $p[i,j] = b_k \ldots b_l$. Finally, we check if the phrases b_k-1 and b_l+1 are equal to the strings $p[1,i-1]$ and $p[j+1,m]$, respectively. For the second we check that the subtree of phrase $p[j+1,m]$ in LZTrie contains $Node(l+1)$. For

the first we check if the column range of the node for $p[1, i-1]^r$ in RevTrie has a point at row $Node(k)$, corresponding to LZTrie (the m searches in RevTrie are computed once). Thus these occurrences require $O(m^2 \lg n)$ time [1].

The total search time for the *occ* occurrences is $O(m^2 \lg n + occ \lg n)$.

Wavelet trees. The geometric data structure we use in practice is a wavelet tree [10]. It is a pefect binary tree where the points are sorted in row order at the root and in column order in the bottom. The coordinates are not explicitly stored. At the root, a bitmap marks with a 0 or a 1 whether each point belongs to the left or right half of the grid, respectively. Those on the left/right side of the grid are then recursively subdivided at the left/right child of the root node. The wavelet tree uses in total $n' \lg n'$ bits.

To support range searches, the bitmaps are enhanced with rank/select data structures. Both can be computed in constant time and $o(n')$ extra bits [14]. To find the points in a range $[i, i'] \times [j, j']$ (rows $\times$ columns), we start with $B[i, i']$ in the root bitmap, and project the interval to the left/right children, towards the new interval $[rank_{0/1}(B, i-1)+1, rank_{0/1}(B, i')]$. We continue splitting the interval, stopping when it becomes empty, or the wavelet tree node has no intersection with the columns $[j, j']$, or it is fully included in $[j, j']$. In the last case, all the values in the current bitmap interval are points in the range. They can be counted directly, or reported one by one by tracking them to the leaves, to know their column values, for example. As any range is decomposed into $O(\lg n')$ wavelet tree nodes that have in total $O(\lg n')$ ancestors, counting the points in the range takes $O(\lg n')$ time and reporting each of them requires $O(\lg n')$ time.

4 A Novel LZ-Index Based Document Listing Structure

We now adapt the LZ-index to carry out document listing instead of reporting all the occurrences of a pattern p. The general search strategy will be as follows. For occurrences of type 1, we store the RMQ of the expansion of RevTrie with the subtree of LZTrie that corresponds to each node. This requires $O(n)$ bits and allows us to apply Muthukrishnan's algorithm [22] directly. For type 2, we enhance the bitmaps of the wavelet tree of *Range* with RMQ data structures for their documents. We can then apply Muthukrishnan's algorithm on any of the $O(\lg n')$ nodes into which the range is decomposed. For occurrences of type 3 we find their documents one by one. The total time will be $O(m^2 \lg n + \mathsf{ndoc}\, m \lg^2 n)$.

Structure. We modify the LZ78 parsing so that no phrase crosses a document boundary. Now consider the LZTrie and RevTrie structures of the original LZ-index resulting from this parsing. We store the following structures.

- ***RevTrie.*** We represent only the topology and the letters of RevTrie and LZTrie, just in order to be able to navigate RevTrie and to search it for patterns in constant time per symbol [1]. The structure requires $3n' \lg \sigma + O(n')$ bits. (In the implementation we do not represent LZTrie, but all the nodes of RevTrie, which in the worst case can be n but in practice are not.)

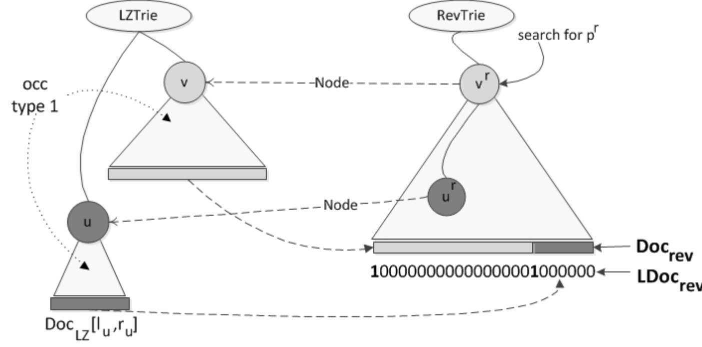

Fig. 1. Our structures for occurrences of type 1.

- ***Doc.*** Let us define Doc_{lz}, the sequence of n' document identifiers of the LZTrie phrases in preorder. We save Doc_{lz} explicitly with $n' \lceil \lg D \rceil$ bits. This is equivalent to the document array [15], but restricted to phrases. Now we define Doc_{rev}, a sequence of n document identifiers built as follows. We traverse RevTrie in preorder, and for each phrase node v^r, let v be the corresponding LZTrie node. Let $Doc_{lz}[l_v, r_v]$ be the range in Doc_{lz} of all the descendants of v (included). We append $Doc_{lz}[l_v, r_v]$ to Doc_{rev}. Doc_{rev} will not be stored, but just its RMQ structure, so as to run Muthukrishnan's algorithm [15] over Doc_{rev}. This RMQ structure answers queries in $O(1)$ time without accessing Doc_{rev} and uses $2n + o(n)$ bits [6].[1] Finally we store a bitmap $LDoc_{rev}[1, n]$, which marks the Doc_{rev} positions where the intervals $Doc_{lz}[l_v, r_v]$ start. In total we store $n' \lceil \lg D \rceil + 3n + o(n)$ bits. Fig. 1 illustrates.
- ***Node.*** Now this is a mapping from RevTrie to LZTrie. If the node v^r in RevTrie with nonempty preorder i corresponds to the node v in LZTrie with preorder j, then $Node[i] = j$. Array $Node$ uses $n' \lceil \lg n' \rceil$ bits.
- ***Range.*** An enhanced binary wavelet tree. Each wavelet tree node implicitly represents a sequence of points. Now consider the array of their corresponding documents. We store, in addition to the bitmap, the RMQ structure corresponding to Muthukrishnan's algorithm [15] on its (virtual) array of documents. The total space of $Range$ is then $3n' \lg n' + o(n' \lg n')$ bits.

Overall, our structure requires $4n' \lg n' + n' \lg D + 3n' \lg \sigma + o(n' \lg n') + 3n + o(n) \leq 5nH_k(T) + 3n + o(n \lg \sigma)$ bits (and $\leq 4nH_k(T) + 3n + o(n \lg \sigma)$ if $\lg D = o(\lg n)$). This is close to the original LZ-index size [16]. We describe the document listing procedure now.

Type 1 occurrences. We search for $p[1, m]^r$ in RevTrie, arriving at node v with preorder j_v. We find the interval $I = Doc_{rev}[s_v, e_v]$ containing all the occurrences of type 1, where $s_v = select_1(LDoc_{rev}, j_v)$ and $e_v = select_1(LDoc_{rev}, j_v + subtree\text{-}size(j_v)) - 1$. Next, we report all the distinct documents in I with

[1] The length is n because n is the internal path length (sum of all node depths) in LZTrie. Each LZTrie node is appended to Doc_{rev} once per ancestor it has in LZTrie.

Muthukrishnan's algorithm using RMQs. For each new position $Doc_{rev}[pos]$ reported by an RMQ, we determine the nonempty preorder $j = rank_1(LDoc_{rev}, pos)$ of the RevTrie node holding that position, and the preorder of this node in LZTrie, $i = Node[j]$. The difference $d = pos - select_1(LDoc_{rev}, j)$ provides the offset of this position within the leaf interval of the LZTrie node with preorder i. Thus, the document is $Doc_{lz}[i+d]$. The time is $O(m + \mathsf{ndoc}_1)$, where ndoc_1 is the number of distinct documents containing at least one occurrence of type 1.

Type 2 occurrences. We consider all the $m-1$ partitions $p = p_{start} \cdot p_{end}$. For each one, we search RevTrie for p^r_{start}, arriving at node v^r with preorder interval $[j, j']$. To find the LZTrie interval we do as follows. We search RevTrie for p^r_{end}. If it does not exist, or it leads to an empty node, then p_{end} is not a phrase and there are no phrases starting with p_{end} (as phrases are built incrementally letter by letter). If instead we reach node u^r, with nonempty preorder t, then $i = Node[t]$ is the LZTrie preorder of the corresponding node u, which represents p_{end}. It is also the left end of the preorder interval of the descendants of u. We compute the size of the interval using $LDoc_{rev}$: $\ell = select_1(LDoc_{rev}, t+1) - select_1(LDoc_{rev}, t)$, then $i' = i + \ell - 1$ and the row interval for the search in *Range* is $[i, i']$.

Now we identify in *Range* the $O(\lg n')$ wavelet tree nodes that cover the interval $[j, j']$, and the ranges where interval $[i, i']$ is projected on their bitmaps. Each of these $O(\lg n')$ intervals represent documents with occurrences of type 2, and we list the documents in each by running Muthukrishnan's algorithm over the RMQ structures that enhance the bitmaps. For each document, which is found in $O(1)$ time, we need $O(\lg n')$ time to reach the corresponding leaf and find its identifier in Doc_{lz}. Although unlikely, in the worst case we can output the same document in each of the $O(\lg n')$ intervals for each of the $m-1$ partitions, which gives $O(m^2)$ time for the RevTrie searches plus a (very pessimistic) worst-case bound of $O(\mathsf{ndoc}_2\, m \lg^2 n')$ time for the ndoc_2 occurrences of type 2.

Type 3 occurrences. We wish to apply the same algorithm of the original LZindex and then output the documents, yet we have fewer data structures now. First, all the searches for all the substrings $p[i, j]$ are carried out in RevTrie, in time $O(m^2)$, and we record the RevTrie and LZTrie preorder values of each (the latter using *Node* from the RevTrie node). For each i, we store in array A_i the information for the substrings of the form $p[i, j]$, sorted by LZTrie preorder value. Now note that we have not stored phrase numbers, yet we can still use *Range* to determine the LZTrie preorder t of the phrase following that of $p[i, j]$, which has RevTrie preorder t^r. If we traverse the wavelet tree of *Range* starting at position t^r in the root bitmap and track it to the leaves, the final position is precisely t. This operation takes $O(\lg n')$ time. Now we can binary search A_{j+1} for LZTrie preorder t, and if we find it corresponding to a phrase $p[j+1, j']$, we can concatenate $p[i, j]$ to get $p[i, j']$. Therefore we can carry out the same process for finding maximal concatenations [16], in total time $O(m^2 \lg n)$.

Finally, we have to check if $p[1..i-1]$ precedes the maximal concatenation and if $p[j+1, m]$ follows it. The first question is equivalent to computing whether the preorder interval for $p[1..i-1]^r$ in RevTrie is connected with the LZTrie preorder value t of the first phrase in the maximal concatenation. The second question

corresponds to computing the LZTrie preorder interval of $p[j+1, m]$ (which can be done using RevTrie, as before) and then asking if the RevTrie preorder value t^r of the last phrase in the maximal concatenation is connected with some point in the LZTrie interval. These tests add up $O(m \lg n)$ time.

This adds up to the promised total time of $O(m^2 \lg n + \mathsf{ndoc}\, m \lg^2 n)$. Note, however, that the occurrences of type 1 are reported very early, in time $O(m + \mathsf{ndoc}_1)$. If the text is generated by an ergodic source, the occurrences of any pattern p appear regularly, every d positions on average (e.g., $d = \sigma^m$ if the symbols are generated uniformly and independently). On the other hand, since $n' \le n/\lg_\sigma n$, only $O((n/d)m/\lg_\sigma n)$ of those occurrences hit a phrase boundary on average. This means that that a fraction of $1-O(m/\lg_\sigma n)$ of the occurrences are of type 1, and also $\mathsf{ndoc}_2 = O(\mathsf{ndoc}\, m/\lg_\sigma n) = o(\mathsf{ndoc})$ if $m = o(\lg_\sigma n)$. Thus we report almost all of the occurrences in $O(1)$ time each. If we just lose those $o(\mathsf{ndoc})$ occurrences not of type 1, our time is the optimal $O(m+\mathsf{ndoc})$! We show in the next section that, indeed, our index is particularly competitive to show the first occurrences (those of type 1), which are the most for short patterns.

5 Experimental Results

We consider the following document collections, following previous work [19].

- `ClueChin`: A 2.3 MB sample of ClueWeb09 (`boston.lti.cs.cmu.edu/Data/clueweb09`), formed by 23 Web pages in Chinese.
- `ClueWiki`: A 141 MB sample of ClueWeb09, formed by 3,334 Web pages from the English Wikipedia (same source as the previous).
- `KGS`: A 75 MB collection of 18,838 sgf-formatted Go game records from year 2009 (`www.u-go.net/gamerecords`).
- `Proteins`: A 60 MB collection formed by 143,244 sequences of Human and Mouse proteins (`www.ebi.ac.uk/swissprot`).

Our machine is an Intel Xeon with 8 processors of 2.4GHz and 12MB cache, with 96GB RAM. It runs Linux 2.6.32-46-server, and we use gcc with full optimization. We choose 40,000 patterns of lengths $m = 3$ and $m = 8$ extracted randomly from the collection.

Table 1 gives the space obtained by our LZ-Index structure on those collections. `ClueWiki` and `KGS` are the most compressible ones, reaching 11–12 bpc, whereas `ClueChin` and `Proteins` are the least compressible ones. All are, as roughly expected from the space analysis, 4.3–5.3×|LZ78|. We show how |LZ78| relates to n/n', and how it roughly coincides with the output size of *Compress*, a classical LZW Unix compressor.

In the more compressible collections, *RevTrie* uses less than 20% of the space, *Doc* uses slightly more than 30%, *Node* slightly more than 10%, and *Range* uses almost 40%. The distribution varies a bit on the less compressible collections, where the fraction of *Node* and *Range* increases, reaching 50%. Note that component *Range* can be omitted if we only want to list the occurrences of type 1, in which case the index size is reduced by 40%–50%.

Component	`ClueChin`		`ClueWiki`		`KGS`		`Proteins`	
RevTrie	2.429	(15%)	1.725	(16%)	2.091	(18%)	2.154	(9%)
topology	*0.396*		*0.182*		*0.247*		*0.461*	
labels	*1.793*		*1.396*		*1.613*		*1.530*	
empty nodes	*0.240*		*0.147*		*0.231*		*0.163*	
Doc	3.594	(22%)	3.529	(33%)	3.864	(33%)	5.777	(25%)
Doc_{lz}	*0.638*		*0.696*		*1.002*		*2.788*	
Doc_{rev} RMQ	*2.331*		*2.336*		*2.360*		*2.348*	
$LDoc_{rev}$	*0.625*		*0.497*		*0.502*		*0.641*	
Node	2.424	(15%)	1.335	(12%)	1.403	(12%)	3.717	(16%)
Range	7.938	(48%)	4.279	(39%)	4.423	(37%)	11.748	(50%)
Total *LZ-Index* (/|LZ78|)	16.386	(4.27×)	10.870	(5.21×)	11.831	(5.35×)	23.550	(4.90×)
|LZ78| (avg. phrase length)	3.840	(7.81)	2.088	(17.24)	2.211	(14.93)	4.805	(6.45)
Compress	2.927		2.733		1.851		4.610	

Table 1. Space breakdown of the main components of our LZ-Index based structure. The numbers are in bpc. Main components are in bold and their space is the sum of the second-level components (bpc in italics). The percentages are w.r.t. the total LZ-Index size, whose line indicates its ratio over |LZ78|. The |LZ78| line, in turn, gives also (n/n'). The last line gives the bpc of a real LZ78-like compression program.

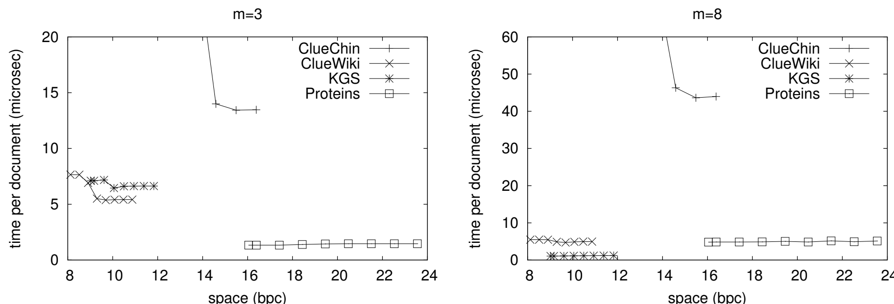

Fig. 2. Space versus listing time per document output. The tradeoff is obtained by not representing RMQ information on the last levels of the wavelet tree of *Range*.

A way to reduce the space without sacrificing functionality is to remove the RMQ structures at the last levels of the wavelet tree of *Range*. In those levels we simply obtain all the documents one by one. The query covers at most two ranges per level, those at the last levels are the smallest, and they are closest to the bottom, so obtaining the document identifiers is faster. Thus, removing those structures should not impact much the time. Fig. 2 confirms that the time is practically unaffected when the first levels are removed, while the space improves noticeably. From now on we will remove the RMQ structures on the last 6 levels of `ClueChin`, 12 levels of `ClueWiki` and `KGS`, and all the levels in `Proteins`.

Table 2 shows the number of documents listed by the queries. In these relatively small collections we list a good percentage of the documents, with the exception of `Proteins`, which has many more documents and then a document listing query is selective enough. From the listed documents, many are obtained as type 1 occurrences (75%–100% for $m = 3$ and 50%–95% for $m = 8$). This shows that we could obtain a significant part of the result using just the fastest listing and without representing *Range*.

Occurrences	ClueChin		ClueWiki		KGS		Proteins	
$m = 3$	14.20	(62%)	2,732.41	(82%)	15,799.10	(84%)	12,106.90	(8%)
Type 1	*13.60*	*(96%)*	*2,727.93*	*(99%)*	*15,132.60*	*(96%)*	*9,185.01*	*(76%)*
Type 2	*0.598*	*(4%)*	*25.06*	*(1%)*	*667.40*	*(4%)*	*2,921.90*	*(24%)*
Type 3	*0.002*	*(0%)*	*0.001*	*(0%)*	*0.022*	*(0%)*	*0.015*	*(0%)*
$m = 8$	6.52	(28%)	1,742.52	(52%)	4,285.02	(23%)	89.45	(0%)
Type 1	*5.02*	*(77%)*	*1,646.97*	*(95%)*	*2,943.00*	*(69%)*	*46.27*	*(52%)*
Type 2	*1.28*	*(20%)*	*94.79*	*(5%)*	*1,338.74*	*(31%)*	*42.49*	*(48%)*
Type 3	*0.208*	*(3%)*	*0.724*	*(0%)*	*3.29*	*(0%)*	*0.981*	*(0%)*

Table 2. Number of occurrences of each type, for pattern lengths $m = 3$ and $m = 8$. Global percentages are w.r.t. the total number of documents, whereas local percentages (in italics) are w.r.t. the total number of occurrences found.

Fig. 3 compares our LZ-Index structures in three modes: the full mode where it returns all the occurrences, a mode where it can return all the occurrences but we take the time needed to return only the occurrences of type 1, and use the minimum space for *Range* (called "up to type 1"), and a mode where it can only return the occurrences of type 1 as it does not store *Range* at all (called "only type 1"). We also compare Sadakane's document listing [22] we implemented on top of Sadakane's CSA [21] obtained from *PizzaChili*[2], and showing three points using suffix array sampling steps 32, 64, and 128. Finally, we include the variant using document arrays as plain wavelet trees [23], as RePair-compressed wavelet trees, and an intermediate between both called "alpha", as implemented by their authors [19] and using Sadakane's CSA with no sampling to minimize space (the sampling is not needed here).

It can be seen that Sadakane's technique uses less space than our smallest LZ-Index variant, but it is orders of magnitude slower (except on `ClueChin`), even on this CSA that is the fastest [4] to compute $A[i]$. The wavelet trees dominate our LZ-Index variants on `ClueChin`, because it has very few documents and thus the wavelet trees are small and fast. On the other collections, instead, wavelet trees use much more space than our LZ-Index variants. Indeed, in all but the toy collection `ClueChin`, even the LZ-Index in full mode is a relevant alternative, whereas the approximate ones offer even better space/time performance.

6 Final Remarks

We have introduced the first document listing data structure based on Lempel-Ziv compression. Apart from offering a competitive space/time tradeoff in general, an interesting feature of the index is its ability to retrieve a large number of documents very fast. This makes it an ideal choice in interactive scenarios, where one must show some answers immediately and others can be calculated in the background, and in cases where only some answers are sufficient.

We plan to extend our ideas to top-k document retrieval. Since the bulk of the occurrences are type 1, considering only those for computing top-k would yield very fast an answer that will usually be very accurate.

[2] From site `pizzachili.di.unipi.it` or `pizzachili.dcc.uchile.cl`

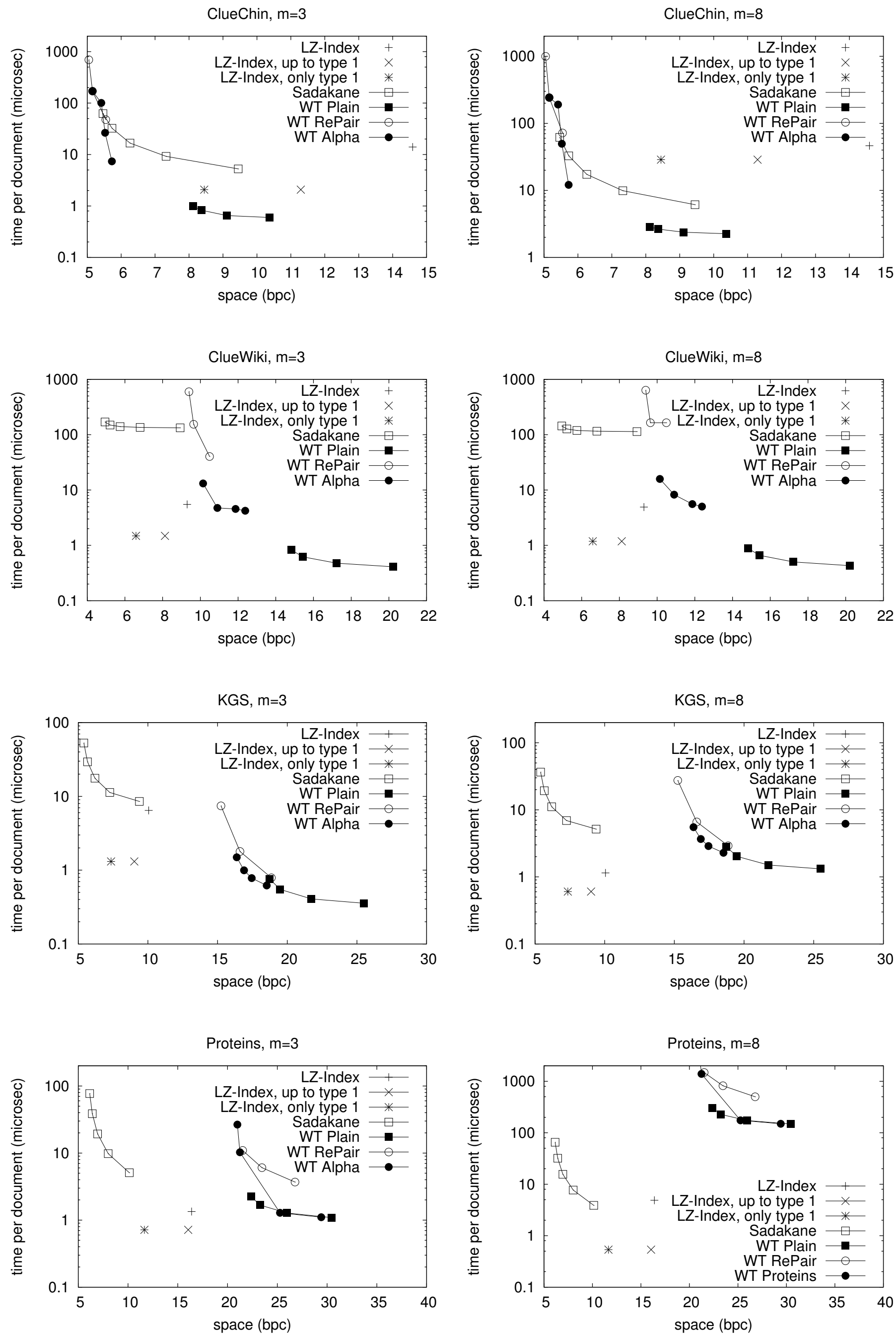

Fig. 3. Space versus listing time (logscale) per document output, for various indexes.

References

1. D. Arroyuelo, G. Navarro, and K. Sadakane. Stronger Lempel-Ziv based compressed text indexing. *Algorithmica*, 62(1):54–101, 2012.
2. J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet partitioning for compressed rank/select with applications. In *Proc. 21st ISAAC*, pages 315–326, 2010.
3. D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. In *Proc. 19th ESA*, pages 748–759, 2011.
4. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM J. Exp. Alg.*, 13:art. 12, 2009.
5. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Alg.*, 3(2):article 20, 2007.
6. J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal of Computing*, 40(2):465–492, 2011.
7. T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theor. Comp. Sci.*, 426-427:25–41, 2012.
8. T. Gagie, S. J. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proc. 16th SPIRE*, pages 1–6, 2009.
9. A. Golynski, R. Grossi, A. Gupta, R. Raman, and S. S. Rao. On the size of succinct indices. In *Proc. 15th ESA*, pages 371–382, 2007.
10. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 636–645, 2003.
11. W.-K. Hon, R. Shah, and J. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. 50th FOCS*, pages 713–722, 2009.
12. S. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM J. Comp.*, 29(3):893–911, 2000.
13. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comp.*, 22(5):935–948, 1993.
14. I. Munro. Tables. In *Proc. 16th FSTTCS*, pages 37–42, 1996.
15. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. 13th SODA*, pages 657–666, 2002.
16. G. Navarro. Indexing text using the ziv-lempel trie. *J.Disc.Alg.*, 2(1):87–114, 2004.
17. G. Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *CoRR*, arXiv:1304.6023v1, 2013.
18. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comp. Surv.*, 39(1):art. 2, 2007.
19. G. Navarro, S. J. Puglisi, and D. Valenzuela. Practical compressed document retrieval. In *Proc. 10th SEA*, pages 193–205, 2011.
20. R. Raman, V. Raman, and S.S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Alg.*, 3(4):art. 43, 2007.
21. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Alg.*, 48(2):294–313, 2003.
22. K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Disc. Alg.*, 5(1):12–22, 2007.
23. N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *Proc. 18th CPM*, pages 205–215, 2007.
24. P. Weiner. Linear pattern matching algorithm. In *Proc. 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.
25. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theor.*, 24(5):530–536, 1978.