Document Listing on Versioned Documents*

Francisco Claude[1,2,3] and J. Ian Munro[3]

[1] Akori S.A.
Santiago, Chile
[2] Escuela de Informática y Telecomunicaciones
Universidad Diego Portales, Chile
[3] David R. Cheriton School of Computer Science
University of Waterloo, Canada

Abstract. Representing versioned documents, such as Wikipedia history, web archives, genome databases, backups, is challenging when we want to support searching for an exact substring and retrieve the documents that contain the substring. This problem is called *document listing*.

We present an index for the document listing problem on versioned documents. Our index is the first one based on grammar-compression. This allows for good results on repetitive collections, whereas standard techniques cannot achieve competitive space for solving the same problem.

Our index can also be addapted to work in a more standard way, allowing users to search for word-based phrase queries and conjunctive queries at the same time.

Finally, we discuss extensions that may be possible in the future, for example, supporting ranking capabilities within the index itself.

1 Introduction

Highly repetitive collections are becoming more and more common. We have a lot of versioned information on the Web; good examples of this are software repositories and Wikipedia. It is also expected that in the future we will have to provide storage for genome sequences of many individuals of the same species, perhaps millions of people. This last scenario is interesting because within the same species, the sequences share close to 99.99%, making the collection highly repetitive [15].

Being capable of storing archive data with historic information on how documents evolve is a challenging task by itself, but we also need to provide searching capabilities to make this information easily available for people when needed. In this work we focus on the *document listing* problem for such collections.

Formally speaking, the document listing problem is defined as follows: Given a collection of documents $\mathcal{D} = \{T_1, T_2, \ldots, T_d\}$, and a query string P, we want to retrieve the documents that contain P as a substring. We could add a ranking

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-3-319-02432-5_33

* First author funded in part by Google U.S./Canada PhD Fellowship. Second author funded in part by NSERC and the Canada Research Chairs Programme.

function f, such that we retrieve the documents ordered by $f(T_i, P)$, and even limit the size of the resulting set by a given parameter k, these are called top-k queries. Some examples of ranking functions include TF-IDF and `closeness` [1].

One important point to clarify is the difference between standard text-indexing and the document listing problem. Usually text indexes allow you to search for a pattern in a text and report the position where the pattern occurs. If we concatenate all the documents and use a classical index, we can retrieve the documents that contain the pattern. The drawback is that we are forced to iterate over all occurrences of the pattern, which means we can pay a huge overhead for just one document if it contains the pattern multiple times. This is the main difference that renders classical text indexes unsuitable for certain instances of the problem. For natural language, the problem has been usually simplified by using an inverted index. For every word w in the language, we have a list $L[w] = \{T_{i_1}, T_{i_2}, \dots, T_{i_\ell}\}$ listing all documents that contain w, plus extra information to compute the ranking function. Answering a query $\mathcal{Q} = \{P_1, P_2, \dots, P_q\}$ corresponds to obtaining a subset of the elements in $\cap_{i=1}^{q} L[P_i]$.

This solution has been shown to be effective in space, retrieval time, and quality, but it lacks the freedom we would expect in other domains. For example, if we consider a collection of DNA sequences, the concept of a word is not well defined. For this reason, solutions in one domain may be completely useless in others. We also see this phenomenon in languages where the separation among words is not clearly defined, or hard to determine automatically.

The main idea behind our proposal is as follows: Given a collection of documents, we compress the whole set of texts using a grammar-compressor [13,21,4]. The resulting file is indexed using the result of [7]. Then we augment the structure with a set of inverted lists for non-terminal symbols. This inverted lists store the documents that contain each non-terminal. The queries are answered by first asking the text index to produce the minimum set of non-terminals that match the pattern for which we have to look into their inverted lists.

Once we have all the inverted lists, we compute the union of those, generating the final result, an inverted list for the pattern that was given as a query.

The main contributions of this paper are:

- We show how to extend a grammar-compressed index to support document listing in a simple and clean way. The index also supports access to any document of the collection, verbatim, so it completely replaces the original input. Building our index on top of any grammar-compressor allows us to achieve good space for repetitive sequences, which is the case of versioned documents. In addition to achieving good space [8,5], a straight-forward grammar representation allows for fast decompression, and therefore, access to the content being indexed [6,8,5].
- The resulting structure supports retrieving the inverted list for an arbitrary pattern. This is particularly interesting, since all the algorithms developed for plain posting lists can be applied to the output of our searches. This allows to easily extend our result to support conjunctive queries.

- We can apply the same result for words in natural language, allowing a new index. This index does not support full-text document listing, but solves the problem of searching for phrases, a problem that is also hard to handle with traditional inverted indexes. Due to lack of space, we ommit the experimental results for this particular application of our result.
- Our final index does not only allow document listing. We discuss how to extend it to compute other pieces of information commonly used by ranking functions: Term frequencies for each document and positional information on where patterns occur inside each document.

2 Related Work

Most of the items in our index are built using grammar compression and indexes. A grammar-compressed representation of a sequence corresponds to a context-free grammar that generates one single text, the one being compressed. For purposes of this work, the following definition suffices.

Definition 1 (Grammar-compressed seq.). *Given a grammar $\mathcal{G} = (\mathcal{X} = \{X_1, X_2, \ldots, X_n\}, \sigma, \Gamma : \mathcal{X} \to \mathcal{X}^+ \cup \sigma, s)$, where:*

- *$\mathcal{X}$ represents the set of non-terminal symbols.*
- *σ corresponds to the set of terminal symbols.*
- *Γ is the set of rules that transform a non-terminal into a sequence of non-terminals or just one terminal symbol. We do not allow cycles in the rules, and that is enough to make sure the grammar generates only one sequence.*
- *s corresponds to the identifier of the start symbol X_s.*

We define $\mathcal{F}(X_i)$ as the result of recursively replacing all non-terminals until obtaining a sequence of terminal symbols. We also refer to $\mathcal{F}(X_i)^R$ as $\mathcal{F}(X_i)$ read from right to left (i.e., reversed).

We say that $\mathcal{G}$ compresses $T = t_1t_2 \ldots t_u$, iff $\mathcal{F}(X_s) = T$.

We call N the sum of the sizes of all the right sides in the grammar, that is

$$N = \sum_{i=1}^{n} |\Gamma(X_i)|$$

We also refer to the height of the grammar as the longest path from the starting symbol to a terminal symbol in the parse tree.

We rely on the grammar-based index proposed by Claude and Navarro [7] to support one of the steps in our searching procedure. We explain in more detail the pieces needed in Section 2.1.

2.1 Grammar Indexes

We first explain the basics of the index proposed by Claude and Navarro [7]. The index takes as input a free-context grammar that generates a single sequence. We call $\mathcal{G}$ the grammar, composed of a set of non-terminals $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$, an initial symbol X_s and a set of rules Γ, that map non-terminals to a sequence of non-terminals or just one single terminal symbol.

The grammar is first preprocessed to remove duplicate rules, and embed rules that are mentioned only once inside the rule that mentions them. This does not increase the size of the grammar, but allows to bound some of the running times further.

The main result of [7] is summarized in Theorem 1. We next explain the structures we need in this paper, omitting some of the details for the sake of readability.

Theorem 1. *[7] Let a sequence $T[1..u]$ be represented by a context free grammar with n symbols, size N and height h. Then, for any $0 < \epsilon \leq 1$, there exists a data structure using at most $2N \lg n + N \lg u + \epsilon\, n \lg n + o(N \lg n)$ bits that finds the occ occurrences of any pattern $P[1..m]$ in T in time $O((m^2/\epsilon) \lg \left(\frac{\lg u}{\lg n}\right) + (m + occ) \lg n)$. It can extract any substring of length ℓ from T in time $O(\ell + h \lg(N/h))$. The structure can be built in $O(u + N \lg N)$ time and $O(u \lg u)$ bits of working space.*

For the construction of the index, we first preprocess the grammar and reassign the identifiers of each non-terminal so that they are sorted lexicographically by the reverse of the string they generate, i.e., $\mathcal{F}(X_i)^R$. We number the non-terminals in sorted order, that is, $\mathcal{F}(X_i)^R \leq \mathcal{F}(X_j)^R$ iff $i \leq j$. We then create a bitmap Y where we assign a 1 to position i iff X_i generates just a single terminal symbol. We augment this bitmap to support the following operations:

- $access_Y(i)$: retrieves the bit at position i in Y.
- $rank_Y(b, i)$: counts the number of times bit b appears up to position i in Y.
- $select_Y(b, j)$: retrieves the position of the j-th occurrence of bit b in Y.

We can represent the bitmap Y and support all three operations in constant time using the method of Raman, Raman, and Rao [17]. This representation requires $nH_0(Y) + o(n)$, where $H_0(Y)$ represents the zero order entropy of the bitmap[1].

By using Y we can know whether a rule generates more non-terminals or just one single terminal symbol. Given X_i, if $access_Y(i) = 1$, then we know it generates a terminal symbol. Furthermore, if we assume terminal symbols are contiguous, we know that X_i generates $rank_Y(1, i)$. It is also possible to obtain the non-terminal X_j that generates symbol a by computing $j = select_Y(1, a)$.

In addition, for each proper suffix of each rule, we assign an id, and then reassign them according to the lexicographical order of the strings generated by those proper suffixes. We will call this SuffPerm. In other words, SuffPerm stores at position i the i-th proper suffix of a rule in lexicographical order.

[1] The zeroth order entropy of a bitmap of length n with m ones is defined as $\frac{m}{n} \lg \frac{n}{m} + \frac{n-m}{n} \lg \frac{n}{n-m}$. This is bounded above by 1.

Finally, we create a labeled binary relation $\mathcal{R}$ that maps `SuffPerm`$[i]$ with j through a label k if rule j appears before the suffix represented by `SuffPerm`$[i]$ in rule k.

We want to support range searching in $\mathcal{R}$. Wavelet trees [12] are a good alternative, access takes $O(\lg n)$ time and range searching takes $O(\lg n)$ per element reported. Wavelet trees, in this context, require $n \lg n(1+o(1))$ bits of space. The time can be further improved to $O(\lg n/\lg\lg n)$ (access and element reported by the range search) within the same space bounds as the standard wavelet trees [2].

In the original paper [7], the grammar is represented as a tree, where we have $N-n$ leaves. In order to have efficient navigation and access to the rules, the tree is represented using the method of Benoit et al. [3], adding a simple trick to allow fast access to the definition of any non-terminal symbol [7]. In our case a simple plain representation of the grammar is enough, we do not need to navigate the parse tree upwards, and the theoretic solution for fast access works slower than traversing a plain representation in practice.

Given a pattern $P = p_1 p_2 \ldots p_m$, we can find two different types of occurrences inside the grammar. The first kind, called *primary occurrences*, are those non-terminals that contain the pattern because two or more rules generated by it, after being concatenated, generate the pattern. The second kind, called *secondary occurrences* are those non-terminals that contain P because they generate a single rule that contain P. Note that actually one non-terminal may be both at the same time, primary and secondary, but for that, the non-terminal must have at least two different occurrences of P.

To find the primary occurrences of a pattern $P = p_1 p_2 \ldots p_m$, we try the m possible partitions: $p_1 \cdot p_2 \ldots p_m$, $p_1 p_2 \cdot p_3 \ldots p_m$, up to $p_1 \ldots p_{m-1} \cdot p_m$. For each partition $P = P_1 \cdot P_2$, we perform a binary search on the rules to determine which ones finish with P_1. Then we perform a binary search over the suffixes of rules, `SuffPerm`, to find suffixes of rules that begin with P_2. Finally, using the binary relation $\mathcal{R}$, we can perform a range search to retrieve the non-terminals that contain elements that start with P_2 preceded by elements that end with P_1.

Secondary occurrences are obtained by following up the primary occurrences in the parse tree. As we will explain later, we only care about primary occurrences in this work, that is why we do not deal with an efficient representation for the parse tree to track secondary occurrences.

Claude and Navarro show how to represent `SuffPerm` in little space on top of the binary relation, and also how to extract prefixes of suffixes of rules in linear time. We do not need the technical details of these results, it suffices to know the running time of each step. The binary search for P_1 requires $O(m \lg n)$ time. The binary search for P_2 requires $O(m \lg N)$ time. Finally, retrieving the primary occurrences requires $O(\lg n/\lg\lg n)$ time per element retrieved.

Retrieving all $\mathrm{occ_p}$ primary occurrences requires $O(m^2 \lg N + \mathrm{occ_p} \lg n/\lg\lg n)$ time.

2.2 Re-pair

Due to its simplicity, we chose Re-Pair as the grammar compression [13] for evaluating our index. It is important to point out that other grammar compressors may achieve better results, yet their implementation for large scale is still an issue. It is also possible to trade compression speed and space for compression ratio using an approximate version [6].

We post-process the result of Re-Pair to make the final grammar smaller. For each rule X_i that generates a set of non-terminals, if it is mentioned only once in the grammar by rule X_j, we expand X_i where X_j mentions it, and remove X_i. We repeat this process until each rule is mentioned at least twice in the grammar.

This is required by the index, but it also has the nice property that matches the dictionary compression algorithm proposed by González and Navarro [11], that has shown to improve the final result considerably (see [11,6]).

3 The Index

In this section we describe how we build the index, augment it to support document listing, and finally how queries are answered.

3.1 Construction for Primary Occurrences

We take the whole collection $\mathcal{D} = \{T_1, T_2, \ldots, T_d\}$, and generate a single sequence

$$T = \$_0 T_1 \$_1 T_2 \$_2 \ldots \$_{d-2} T_{d-1} \$_{d-1} T_d,$$

where $\$_i$ are symbols that do not appear anywhere else in the collection.

When we compress this sequence with Re-Pair, we are sure that no rule spans from one document to the other, since the $\$_i$ symbols cannot form pairs that appear twice. We then remove the $\$_i$ elements, and generate one rule per document, containing all the elements left between the \$s in X_s. After that, we replace X_s by a new rule that generates the new rules we just created, in order. This allows us to have direct access to a rule that generates the whole content for any document. Our grammar, after this preprocessing, has the following form:

- X_s generates d non-terminals, $X_{t_1}, X_{t_2}, \ldots, X_{t_d}$, where $\mathcal{F}(X_{t_i}) = T_i$.
- X_{t_i} generates the symbols between $\$_{i-1}$ and $\$_i$ in the original X_s generated by Re-Pair.

When building the index, we leave X_s outside the permutation `SuffPerm`. This does not only save space, but makes sure that whenever we find a primary occurrence, it is contained inside a single document, and not formed by the concatenation of two.

To access the i-th document in the collection, we just expand the i-th non-terminal generated by s. This allows us to retrieve documents in time proportional to their length (amortized if we don't use the result from [7]).

Note that we can adapt other grammar-based compressors to this scheme. An interesting option is to just simply compress each document separately with a compressor that generates an SLP (rules restricted to generate two non-terminals or just one terminal), and then apply the merge algorithm of Wan [20]. This will generate a grammar that satisfies the conditions above, and by applying the same preprocessing before constructing the index, we can optimize the output even further.

3.2 Adding Inverted Lists

For each non-terminal, we store an inverted list of the documents containing that non-terminal. Note that this requires at most $n \times d$ bits, and we expect n to be small. Yet this is still not satisfactory. If two versions share much of their content, they will appear in a very similar set of lists, since they will be formed by the same non-terminals.

To exploit this, we again use grammar-compression on the sequence of lists. We could use any space-efficient representation of lists, but for repetitive ones, this particular solution has proven to work well in practice [8,5].

We refer to $L[X_i]$ to the list of documents containing non-terminal X_i and will call L the set of inverted lists. We represent the inverted lists in the same way as we represent the documents, this allows to access an entire list in time proportional to its length.

It is interesting to relate the size of this inverted lists to the size of the original sequence. It turns out that under reasonable assumptions, these lists can be represented space efficiently. We see the inverted lists as a grid, where coordinate (i, j) is a 1 iff non-terminal i is contained in document j. Let t be the number of points in this grid. We need $t \lg \frac{nd}{t} + O(t)$ bits to represent the grid[2].

We know that $n \leq t$, therefore, the space is bounded by $t \lg d$, which is the same as the solution by Välimaki and Mäkinen requires for the document array [19]. We can further bound the space by considering the worst possible space for the grid. The space is maximized when $t = \frac{nd}{e}$. In this case, the total space required by the grid is $O(t)$ bits.

On the other hand, we can also bound the length of the text in terms of t. We know that each point on the grid represents at least one occurrence of a rule in the collection, therefore, $u \geq t$. This means that the total extra space for the grid is bounded by the length of the collection in bits, in other words $\frac{\mathcal{D}}{\lg \sigma}$ bits.

3.3 Full-Text Document Listing

Having built the grammar-index, and the inverted lists, the searching becomes quite straight-forward. We search for the nonterminals that contain primary occurrences of the pattern, and compute the union of the inverted lists associated to those nonterminals.

At this stage we need to compute the union of sets, in contrast with the usual operation we encounter between inverted lists, which is the intersection. Furthermore, our case is a bit more complicated. We have a grammar-compressed

[2] This is a simple information theoretic lower bound, there exist representations that achieve this [9], and some that do better on repetitive cases [5], as in our case.

version of the lists, and thus we want to make use of this fact, both to keep the space low, and to improve the query time.

Given a set of non-terminals representing the primary occurrences of the pattern, we will create a dynamic dictionary containing those elements, called *seen*, and a queue containing the same elements, we call this queue *remaining*. The merge procedure generates a dictionary containing all the elements, and is shown in Algorithm 1.

```
   Data: Set $V = \{v_1, v_2, \ldots, v_n\}$, Lists $\mathcal{G} = (\mathcal{X}, \Gamma, \sigma, s)$
   Result: $R = (d_{i_1}, d_{i_2}, \ldots, d_{i_k})$
 1 remaining $\leftarrow \emptyset$
 2 seen $\leftarrow \emptyset$
 3 $R \leftarrow \emptyset$
 4 for $v \in V$ do
 5     remaining $\leftarrow$ remaining $\cup \{\mathcal{X}_v\}$
 6     seen $\leftarrow$ seen $\cup \{\mathcal{X}_v\}$
 7 while remaining $\neq \emptyset$ do
 8     $x \leftarrow$ GetMax(remaining)
 9     remaining $\leftarrow$ remaining $- \{x\}$
10     if $x$ is terminal then
11         $R \leftarrow R \cup \{x\}$
12     for $x^j$ in $\Gamma(x)$ do
13         if $x^j \notin$ seen then
14             seen $\leftarrow$ seen $\cup \{x^j\}$
15             remaining $\leftarrow$ remaining $\cup \{x^j\}$
16 return $L$
```

Algorithm 1. Computing the union of the lists for a set of non-terminals

The worst case running time of this algorithm is $O(\text{occ}_\text{p} \times \textit{output})$. Section 4 shows that occ_p is in general small, and also that our heuristic of keeping track of previously seen non-terminals allows us to save processing time; it exploits the regularities seen between the lists. If two lists contain basically the same elements, we will only explore one of them, since we will encounter a non-terminal we have already seen.

It is quite straight-forward to see why we only find primary occurrences. Secondary occurrences contain documents we already reported as primary occurrences, so processing only primary occurrences maintains the correctness of the result while cutting down the time.

3.4 Adding Ranking Information

The index can be augmented with extra information in a similar way as inverted lists, with a couple of restrictions. We can augment the inverted lists that associate each non-terminal symbol with the documents that contain it with score values. In particular, frequencies offer a property that is easy to exploit here.

When we augment the lists L with frequencies, we can just add up all the values associated with primary occurrences of a certain document and we will obtain precisely the number of occurrences of the pattern in the whole document. We include the details of this algorithm in the Appendix.

We may not need to store the frequencies for each possible occurrence of a document in the inverted lists. We could store an approximation of the frequency to approximate the term frequency and save space, by storing values from a smaller universe.

We can also use the result of Claude and Navarro [7] to support locating the occurrences of the pattern in the collection. This allows to obtain positional information for the query when required. Another option here is to approximate the locations of multiple patterns depending on the primary occurrences. This line of work is out of the scope of this article.

4 Experimental Results

4.1 Practical Considerations

For the practical implementation, we did not implement the real-time access to prefixes/suffixes of rules as described in [7]. We just store the grammar as a set of arrays describing each rule. We also do not need the tree in practice, since we are not tracking occurrences upwards.

The binary relation is represented using a wavelet tree, as implemented in LIBCDS[3]. We also make use of the arrays implemented in the library. We use Navarro's implementation of Re-Pair [4], which runs in linear time. As containers we use the standard `C++ STL` containers. For sets we use `set`, and for unsorted sequences, we use `vector`.

4.2 Experimental Setup

To test our index we downloaded the first part of Wikipedia in English[5], and sampled documents from it uniformly at random. For each document selected, we extracted all its versions. This was done using `anonymous'` library.

We also generated synthetic collections composed of symbols A, C, G and T. This is to mimic the compression of genome databases. The process of generation is the following: Generate a random sequence T_1 of length n, and then generate $d-1$ copies of T_1 and mutate $x\%$ of it.

Table 1 shows the main characteristics of our datasets. The compression ratio may not be very descriptive given that the sequences are highly repetitive. For this reason, we include the compression ratio achieved by `anonymous'` Re-Pair implementation. This does not include any post-processing, and just represents the original sequences, therefore, it is only a guideline on how much the text could be compressed.

[3] Available at `http://libcds.recoded.cl`
[4] Available at `http://www.dcc.uchile.cl/gnavarro/software/`
[5] `enwiki-20110722-pages-meta-history1.xml`

Table 1. Datasets

Dataset	size	# docs	versions/doc (avg)	mutation rate	Re-Pair
Wiki1	69MB	8	582	-	0.36MB
Wiki2	600MB	20	772.85	-	3.45MB
Wiki3	1.5GB	36	831.08	-	5.50MB
DNA1	1000MB	1	1000	0.01%	4.5MB
DNA2	1000MB	1	1000	0.005%	2.09MB
DNA3	1000MB	1	1000	0.0026%	1.17MB

Table 2. Space required for our index for each dataset, separated by components

Collection	T	Lists	SuffPerm	$\mathcal{R}$	Total	Compr.
Wiki1	0.39MB	0.49MB	0.39MB	0.39MB	1.66MB	2.43%
Wiki2	1.75MB	2.14MB	1.69MB	1.71MB	7.29MB	1.22%
Wiki3	3.19MB	4.37MB	3.12MB	3.06MB	13.73MB	0.90%
DNA1	3.21MB	4.76MB	2.94MB	3.03MB	13.95MB	1.40%
DNA2	1.99MB	2.80MB	1.78MB	1.91MB	8.47MB	0.85%
DNA3	1.26MB	1.59MB	1.15MB	1.23MB	5.23MB	0.52%

We generated queries by taking a version uniformly at random, and then choosing a substring uniformly at random from that particular version.

The machine used for generating the indexes and measuring time has 2 Intel(R) Xeon(R) CPU X5660 processors running at 2.80GHz, 11TB of hard drive and 24GB of RAM. The machine is running Ubuntu Linux 11.04 with kernel `2.6.38-13-generic` for `x86_64`. All our code is implemented and `C++` and was compiled using `gcc` version 4.5.2 with flags `-O3 -DNDEBUG`. Our code is available for download from `http://fclaude.recoded.cl/projects`.

4.3 Full-Text Document Listing

Table 2 shows the sizes of our index for the different collections. We can see that our indexes, for the Wikipedia samples and the DNA synthetic data, are around 4 to 4.5 times the size of the collection when we compress it using Re-Pair. This

Table 3. Time per element retrieved in microseconds for patterns of length $m = 4, 8, 16, 32$, averaged over $10,000$ queries

Collection	$m = 4$	$m = 8$	$m = 16$	$m = 32$
Wiki1	0.60	1.36	3.37	7.38
Wiki2	0.51	0.72	1.72	4.03
Wiki3	0.54	0.83	2.40	6.23
DNA1	20.03	1.86	3.05	6.05
DNA2	12.42	1.35	2.17	4.06
DNA3	8.05	1.06	1.59	2.90

means, within this space, we are replacing the collection and supporting search operations on top of it. Table 3 shows the time in microseconds per element retrieved. This was averaged over $10,000$ queries.

4.4 Comparison to Related Work

The document listing problem was first solved in linear space by Muthukrishnan [14]. Sadakane [18] proposed a different time/space tradeoff, and later Mäkinen and Välimäki [19] and Navarro et al. [16] proposed practical solutions to the problem. All these solutions are not designed for repetitive collections. Only recently, Gagie et al. [10] proposed a solution in this scenario. We measured their results with default parameters for the Wiki collections. They offer a different tradeoff than our solution. We provide superior space, our index is 3.56, 8.22, and 10.86 times smaller for Wiki1, Wiki2, and Wiki3 respectively. On the other hand, their query time is much lower, 11–17 times faster for Wiki1, 18–22 times faster for Wiki2, and 20–31 for Wiki3. We measured patterns of length $4, 8$ and 16, since the patterns of length 32 produced inconsistent results in their index, showing less occurrences than documents reported. We also excluded patterns of length 16 from Wiki3 for the same reason. When compared to the solution by Navarro et al. [16], we are 16 to 62 times smaller, considering only Wiki1 and a preffix of Wiki2.

5 Conclusions

We have presented a new index for representing highly repetitive collections. This index can be used in two different scenarios: (1) Indexing a collection to support document listing of exact substrings; (2) Indexing a collection and support phrase searches for words existing in the collection.

The results show that while providing competitive time complexities, we achieve space considerably smaller than previous results. This opens a new line for storing historic information on documents while supporting efficient search operations.

It is easy to relate to our index in terms on the inverted lists. In the symbol-based version, we can build the inverted index for any possible substring using our index. Furthermore, when we tokenize the text, and index the word identifiers, our index is just a grammar-compressed representation of the inverted lists, augmented with extra information to support phrase search operations on top of it, allowing to produce the inverted list of an arbitrary phrase.

Our work also leaves some challenging open problems. First, the union of all non-terminals that represent primary occurrences has no good theoretical bound, yet is reasonable in practice. Is it possible to modify the structure or the grammar in order to provide a reasonable bound, say we do not visit more than k symbols per element in the resulting set? Another interesting problem not considered in this work, is whether we could support approximate searches, allowing to retrieve the phrases or substrings that are most similar to the query. This is important, since typos may have a huge effect in the result.

References

1. Baeza-Yates, R.A., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
2. Barbay, J., Claude, F., Navarro, G.: Compact binary relation representations with rich functionality. CoRR abs/1201.3602 (2012)
3. Benoit, D., Demaine, E., Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Representing trees of higher degree. Algorithmica 43(4), 275–292 (2005)
4. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Trans. Inf. Theo. 51(7), 2554–2576 (2005)
5. Claude, F., Fariña, A., Martínez-Prieto, M., Navarro, G.: Indexes for highly repetitive document collections. In: CIKM, pp. 463–468 (2011)
6. Claude, F., Navarro, G.: A fast and compact Web graph representation. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 118–129. Springer, Heidelberg (2007)
7. Claude, F., Navarro, G.: Improved grammar-based compressed indexes. In: Calderón-Benavides, L., González-Caro, C., Chávez, E., Ziviani, N. (eds.) SPIRE 2012. LNCS, vol. 7608, pp. 180–192. Springer, Heidelberg (2012)
8. Claude, F., Fariña, A., Martínez-Prieto, M.A., Navarro, G.: Compressed q-gram indexing for highly repetitive biological sequences. In: BIBE, pp. 86–91 (2010)
9. Farzan, A., Gagie, T., Navarro, G.: Entropy-bounded representation of point grids. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010, Part II. LNCS, vol. 6507, pp. 327–338. Springer, Heidelberg (2010)
10. Gagie, T., Karhu, K., Navarro, G., Puglisi, S.J., Sirén, J.: Document listing on repetitive collections. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 107–119. Springer, Heidelberg (2013)
11. González, R., Navarro, G.: Compressed text indexes with fast locate. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 216–227. Springer, Heidelberg (2007)
12. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: SODA, pp. 841–850. Society for Industrial and Applied Mathematics, Philadelphia (2003)
13. Larsson, J., Moffat, A.: Off-line dictionary-based compression. Proc. of the IEEE 88(11), 1722–1732 (2000)
14. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: FOCS, pp. 657–666 (2002)
15. Navarro, G.: Indexing highly repetitive collections. In: Smyth, B. (ed.) IWOCA 2012. LNCS, vol. 7643, pp. 274–279. Springer, Heidelberg (2012)
16. Navarro, G., Puglisi, S.J., Valenzuela, D.: Practical compressed document retrieval. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 193–205. Springer, Heidelberg (2011)
17. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: SODA, pp. 233–242 (2002)
18. Sadakane, K.: Succinct data structures for flexible text retrieval systems. Journal of Discrete Algorithms 5(1), 12–22 (2007)
19. Välimäki, N., Mäkinen, V.: Space-efficient algorithms for document retrieval. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 205–215. Springer, Heidelberg (2007)
20. Wan, R.: Browsing and searching compressed documents. Ph.D. thesis, The University of Melbourne (2003)
21. Ziv, J., Lempel, A.: Compression of individual sequences via variable length coding. IEEE Trans. Inf. Theo. 24(5), 530–536 (1978)