# Tailoring r-index for Document Listing Towards Metagenomics Applications

Cobas, Dustin

Springer Nature Switzerland AG
2020

Cobas , D , Mäkinen , V & Rossi , M 2020 , Tailoring r-index for Document Listing Towards Metagenomics Applications . in C Boucher & S V Thankachan (eds) , String Processing and Information Retrieval : 27th International Symposium, SPIRE 2020 Orlando, FL, USA, þÿOctober 13 15, 2020, Proceedings . Lecture Notes in Computer Scienc
subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) , vol. 12303 LNCS , Springer Nature Switzerland AG , Cham , pp. 291-306 , International Symposium on String Processing and Information Retrieval , Orlando , Florida , United States , 13/10/2020 . https://doi.org/10.1007/978-3-030-59212-7_21

http://hdl.handle.net/10138/354341
https://doi.org/10.1007/978-3-030-59212-7_21

# Tailoring r-index for document listing towards metagenomics applications

Dustin Cobas[1][0000−0001−6081−694X], Veli Mäkinen[2][0000−0003−4454−1493], and Massimiliano Rossi[3][0000−0002−3012−1394]

[1] CeBiB — Center for Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Santiago, Chile
dcobas@dcc.uchile.cl

[2] Department of Computer Science, University of Helsinki, Helsinki, Finland
veli.makinen@helsinki.fi

[3] Department of Computer and Information Science and Engineering, University of Florida, Gainesville, USA
rossi.m@ufl.edu

**Abstract.** A basic problem in *metagenomics* is to assign a sequenced read to the correct species in the reference collection. In typical applications in genomic epidemiology and viral metagenomics the reference collection consists of a set of species with each species represented by its highly similar strains. It has been recently shown that accurate read assignment can be achieved with $k$-mer hashing-based *pseudoalignment*: a read is assigned to species A if each of its $k$-mer hits to a reference collection is located only on strains of A. We study the underlying primitives required in pseudoalignment and related tasks. We propose three space-efficient solutions building upon the *document listing with frequencies* problem. All the solutions use an *r-index* (Gagie *et al.*, SODA 2018) as an underlying index structure for the text obtained as concatenation of the set of species, as well as for each species. Given $t$ species whose concatenation length is $n$, and whose Burrows-Wheeler transform contains $r$ runs, our first solution, based on a grammar-compressed document array with precomputed queries at non terminal symbols, reports the frequencies for the `ndoc` distinct documents in which the pattern of length $m$ occurs in $\mathcal{O}(m + \log(n)\texttt{ndoc})$ time. Our second solution is also based on a grammar-compressed document array, but enhanced with bitvectors and reports the frequencies in $\mathcal{O}(m + ((t/w)\log n + \log(n/r))\texttt{ndoc})$ time, over a machine with wordsize $w$. Our third solution, based on the interleaved LCP array, answers the same query in $\mathcal{O}(m + \log(n/r)\texttt{ndoc})$ time. We implemented our solutions and tested them on real-world and synthetic datasets. The results show that all the solutions are fast on highly-repetitive data, and the size overhead introduced by the indexes are comparable with the size of the $r$-index.

**Keywords:** Metagenomics · r-index · document listing.

## 1 Introduction

Metagenomics is the study of genomic material recovered directly from environmental samples. Thus, conversely to genomic samples, metagenomic samples consist of genome sequences of a community of organisms sharing the same environment, highlighting the microbial diversity in the environmental samples. The samples of genome sequences are collected using shotgun sequencing. This creates a mixture of genome fragments from all organisms in the environment. One important step in metagenomics is to assign each fragment to its owner, allowing to identify and quantify species. This step is called read assignment [19], and it is the basic step in most metagenomic analysis workflows such as in genomic epidemiology [25], and viral epidemiology [6].

Read assigners were first implemented using computationally expensive read aligners [19,38,23]. In [37] the authors showed that similar results are achieved replacing the read aligners with the computationally less expensive $k$-mer hashing methods. Read assigners based on $k$-mer set indexing are referred to as *pseudoaligners*. Efficient indexing of $k$-mer sets, including *colored de Bruijn graphs* [20], has been deeply investigated and we refer the reader to the survey [27] for further reading. Pseudoaligners such as Kallisto [4], MetaKallisto [34], and Themisto [25] use *colored de Bruijn graphs* and are based on the following pseudoalignment criterion. Given a set of references $T_1, \ldots, T_t$ (representing $t$ distinct species), and read $P$, the read $P$ is pseudoaligned with $T_i$ if there exists a $k$-mer of $P$ that occurs in $T_i$ and for all other $k$-mers $u$ of $P$, either $u$ occurs in $T_i$ or $u$ does not occur in $T_1, \ldots, T_t$. This approach is motivated by the fact that the species are usually quite dissimilar, but the strains inside the species are highly similar.

In this paper, we study some basic primitives that are required in different variations of the pseudoalignment criteria. We argue that the specific criterion given above is just one example of a family of criteria, and it is important to study the general framework rather than tailoring the methods to a very narrow setting. Towards this goal of obtaining general results, instead of studying directly $k$-mers of a pattern, we focus here on searching the complete pattern. We continue the discussion in Sect. 6 on how to integrate the results with $k$-mer based criteria.

We modelled this read assignment problem as a *document listing with frequencies* problem, where the set of species is a collection and each species is a document formed by the concatenation of its strains. Given a pattern $P$ we want to report all documents where $P$ occurs, and their frequencies. This problem was first introduced in [35] and further refined in [3] and [15] (details in Sect. 3). We propose three solutions. All solutions use an $r$-index [14] as text index for the concatenation of all documents. The first solution is an extension to frequencies of the solution proposed in [9] in which a grammar-compressed document array is used, and for each non terminal node, precomputed answers are stored. The second and the third solution are based on the *term frequency* approach presented in [33] which uses an additional index for all documents. The key idea is to find the leftmost and rightmost occurrence of the pattern $P$ in the index of each document, by searching the pattern in the index of the concatenation of all documents. To do this, the second solution uses the grammar-compressed document array of [9] enhanced with bitvectors at non terminal nodes marking which descendant contains the leftmost and rightmost occurrence of the pattern in each document. The third solution relies on a modified version of the interleaved longest common prefix array [13]. We implemented our solutions and we tested them using real-world and synthetic datasets.

## 2 Basics

A string $S[1..n]$ is a sequence of $n$ characters over an alphabet $\Sigma$ of size $\sigma = |\Sigma|$. A document $T$ is a string terminated by a special symbol $\$ \notin \Sigma$ that is lexicographically smaller than all characters in $\Sigma$. A collection $D = \{T_1, T_2, \ldots, T_t\}$ is a set of $t$ documents, which is usually represented as the concatenation of its documents, i.e. $\mathcal{D}[1..n] = T_1 T_2 \cdots T_t$. When it is clear from the context, we will refer to $T_i$ as document $i$. Given a string $S[1..n]$, let $\mathtt{rank}_c(S, i)$ be the number of occurrences of symbol $c$ in $S[1..i]$, and let $\mathtt{select}_c(S, j)$ be the position of the $j$-th symbol $c$ in $S[1..n]$. When string $S$ is from alphabet $\{0, 1\}$, we call it a bitvector. For bitvector $S$ it holds $\mathtt{rank}_0(S, i) = i - \mathtt{rank}_1(S, i)$.

Given a string $S$ over an alphabet $\sigma$, the *suffix array* [26] $\mathsf{SA}[1..n]$ of $S$ is an array of integers providing the starting position of the suffixes of $S$ sorted in lexicographic order. The *inverse suffix array* $\mathsf{ISA}[1..n]$ of $S$ is an array of integers that, for each suffix of $S$, provides the position of the suffix in the suffix array. In particular we have that for all $1 \le i \le n$, $\mathsf{SA}[\mathsf{ISA}[i]] = i$.

A *compressed suffix array* [31] $\mathsf{CSA}[1..n]$ is a space-efficient representation of the suffix array whose size $|\mathsf{CSA}|$ in bits is usually bounded by $\mathcal{O}(n \log \sigma)$. We denote by $t_{search}(m)$ the time to find the interval of the suffix array corresponding to all occurrences of $P[1..m]$, while by $t_{lookup}(n)$ the time necessary to access any value $\mathsf{SA}[i]$.

The $r$-index [14] is a compressed text index whose main components are a run-length encoded *Burrows-Wheeler* transform ($\mathsf{BWT}$) [5] and the sample of the suffix array at the beginning and at the end of each run of the $\mathsf{BWT}$. We denote by $r$ the number of equal character runs of the $\mathsf{BWT}$. The $r$-index of the document $T[1..n]$ can be computed in $\mathcal{O}(n)$ time and occupies $\mathcal{O}(r \log(n/r))$ space. We can find all occurrences of a given pattern $P[1..m]$ in the document $T[1..n]$ in time $\mathcal{O}(m + occ)$ time. The $r$-index supports $\mathsf{SA}$ and $\mathsf{ISA}$ queries in $\mathcal{O}(\log(n/r))$ time and $\mathcal{O}(r \log(n/r))$ space[4].

Given a collection $D = \{T_1, \ldots, T_t\}$ of $t$ documents and its concatenation $\mathcal{D} = T_1 T_2 \cdots T_t$ of length $n$, the *document array* [28] $\mathsf{DA}[1..n]$ stores in each position $i$ the index of the document which the suffix $\mathsf{SA}[i]$ belongs to.

Given a document $T[1..n]$, the *longest common prefix* array $\mathsf{LCP}_T[1..n]$ stores in each position $2 \le i \le n$ the length of the longest common prefix between the two strings $T[\mathsf{SA}[i-1]..n]$ and $T[\mathsf{SA}[i]..n]$.

[4] Throughout the paper, we report the space in words, where not otherwise specified.

Given a collection $D = \{T_1, \ldots, T_t\}$ whose concatenation is $\mathcal{D}[1..n]$, the interleaved longest-common-prefix array $\mathsf{ILCP}[1..n]$ is defined in [13] as the interleaving of the $\mathsf{LCP}$ arrays of the documents $T_1, \ldots, T_t$ in the order they appear in the suffix array of $\mathcal{D}$, i.e., if $\mathsf{SA}[i]$ is the lexicographically $j$-th suffix of the $k$-th document, $\mathsf{ILCP}[i] = \mathsf{LCP}_k[j]$. Let the $\mathsf{ILCP}$ array be run-length encoded in $\rho$ runs. Then, it can be represented using two arrays: $\mathsf{LILCP}[1..\rho]$ contains the prefix sums of the lengths of the $\rho$ runs; $\mathsf{VILCP}[1..\rho]$ contains the values of these runs. Furthermore, the $\mathsf{LILCP}$ array can be replaced by a sparse bitvector $\mathsf{L}[1..n]$ such that $\mathsf{LILCP}[i] = \mathtt{select}_1(\mathsf{L}, i)$.

Given a string $S[1..n]$, a *straight line grammar* for $S$ is a context-free grammar $\mathcal{G}$ that uniquely generates the string $S$. We denote by $\mathcal{T}$ the parse tree of $S$. Given a node $t \in \mathcal{T}$, $t$ is a *terminal* node if $t$ has no children, $t$ is a *non terminal* node otherwise. Each node $t \in \mathcal{T}$ uniquely identifies an interval of $S$ denoted by $S[\ell_t..r_t]$. For the ease of explanation we say that a character $c$ occurs in $t$ by meaning that the character $c$ occurs in $S[\ell_t..r_t]$. The parse tree $\mathcal{T}$ is *binary* if its maximum arity is 2, and $\mathcal{T}$ is *balanced* if every substring is covered by $\mathcal{O}(\log n)$ *maximal nodes*, which are the highest nodes of the tree whose expansions form a partition of the substring. Computing the smallest grammar is an NP-hard problem [22], but various $\mathcal{O}(\log(n/\mathcal{G}^*))$-approximation exists. We consider those that are binary and balanced [32,7,21].

## 3 Related Work

In this section we define three problems and report solutions and techniques from the literature that are used in our approach. For a complete overview we refer the reader to the survey [29].

*Problem 1 (Document listing).* Given a collection $D = \{T_1, T_2, \ldots, T_t\}$, and a pattern $P$, return the set of documents $L \subseteq D$ where $P$ occurs.

Muthukrishnan [28] proposed the first solution to Problem 1 in optimal time and linear space. He defined the *document array* $\mathsf{DA}$ and used a suffix tree [36] to find all occurrences of the pattern $P$ represented as an interval $[s_p..e_p]$. Then, he proposed a recursive algorithm to find all distinct documents $\mathtt{ndoc}$ in $\mathsf{DA}[s_p..e_p]$ in optimal time $\mathcal{O}(\mathtt{ndoc})$.

Sadakane [33] replaced the suffix tree with a compressed suffix array $\mathsf{CSA}$ and the document array with a bitvector marking the starting position of each document in text order. He also replaced the data structures to find all distinct documents $\mathtt{ndoc}$ in $\mathsf{DA}[s_p..e_p]$ with a succinct version using only $\mathcal{O}(n)$ bits. With this solution, Problem 1 can be solved in $\mathcal{O}(t_{search}(m) + \mathtt{ndoc}t_{lookup}(n))$ using a data structures of $|\mathsf{CSA}| + \mathcal{O}(n)$ bits.

Gagie *et al.* [13] introduced the $\mathsf{ILCP}$ array whose property stated in Lemma 1 allows to apply almost verbatim the technique used by Sadakane to find distinct elements in $\mathsf{DA}[s_p..e_p]$. The solution uses a run-length compressed suffix array $\mathsf{RLCSA}$ [24] which allows to answer the queries of Problem 1 in $\mathcal{O}(t_{search}(m) + \mathtt{ndoc}t_{lookup}(n))$ time.

Claude and Munro [8] proposed the first grammar-based document listing later improved by Navarro in [30]. Cobas and Navarro [9], later proposed a practical variant in which they store the *document array* as a binary balanced straight line grammar. Then, they precompute and store the answers for all non terminal nodes of the grammar. The queries are answered by using a $\mathsf{CSA}$ to find the interval $\mathsf{DA}[s_p..e_p]$ and merging the precomputed answers for the $\mathcal{O}(\log n)$ non terminal symbols covering $\mathsf{DA}[s_p..e_p]$. This leads to a solution that solves Problem 1 in $\mathcal{O}(t_{search}(m) + \mathtt{ndoc} \log n)$ time.

*Problem 2 (Term frequency).* Given $D = \{T_1, T_2, \ldots, T_t\}$, and a pattern $P$, for each document $T \in D$ return the number of occurrences of $P$ in $T$.

Sadakane [33], addressed also the *term frequency* problem. The solution to Problem 1 is enhanced building a compressed suffix array $\mathsf{CSA}$ for each document. Given the interval $[s_p..e_p]$ of all occurrences of the pattern $P$, he uses the data structure to find the distinct documents in $\mathsf{DA}[s_p..e_p]$ and their leftmost and rightmost occurrences. Those positions are then mapped into an interval in the $\mathsf{CSA}$ of the document. The sizes of these intervals represent the frequencies of the documents. This approach solves Problem 2 in $\mathcal{O}(t_{search}(m) + \mathtt{ndoc} t_{lookup}(n))$ time.

*Problem 3 (Document listing with frequencies).* Given $D = \{T_1, T_2, \ldots, T_t\}$, and a pattern $P$, return the set of documents where $P$ occurs and their frequencies.

Välimäki and Mäkinen [35] first proposed Problem 3 and showed that the document listing problem can be solved using a rank and select data structure on the document array, to simulate Muthukrishnan's [28] solution. In addition, after locating the interval $\mathsf{SA}[s_p..e_p]$ of all occurrences of $P$ in $\mathcal{D}$, the frequencies for each distinct document in $\mathsf{DA}[s_p..e_p]$ are computed using a rank array on the document array, i.e., the number of occurrences of $P$ in document $T_i$ are $\mathtt{rank}_i(\mathsf{DA}, s_e) - \mathtt{rank}_i(\mathsf{DA}, s_p - 1)$. Using a *wavelet tree* [18] to represent the document array, given a pattern $P[1..m]$, Problem 3 can be solved in $\mathcal{O}(t_{search}(m) + \mathtt{ndoc} \log t)$ time.

Belazzougui *et al.* [3] built a *monotone minimum perfect hash function* [1] on the document array. Combining Muthukrishnan's [28] and Sadakane's [33] approaches, it is possible to find the leftmost and rightmost occurrence of the pattern $P$ in the $i$-th document. Using the constant time rank on the document array, Problem 3 can be solved in $\mathcal{O}(t_{search}(m) + \mathtt{ndoc})$ time.

Gagie *et al.* [15] proposed a solution based on *wavelet trees* [18], that does not rely on Muthukrishnan's [28] solution. The idea is to use the *range quantile* [16] problem to find the $i$-th smallest value in the range $\mathsf{DA}[s_p..e_p]$. Then, retrieve its frequency as the length of the interval corresponding to $[s_p..e_p]$ in its leaf in the wavelet tree. With this approach Problem 3 can be solved in $\mathcal{O}(t_{search}(m) + \mathtt{ndoc} \log t)$ time.

# 4 The document listing with frequencies

We are now ready to describe our *document listing with frequencies* approaches. We propose three different solutions, which rearrange and adapt different concepts of previous work. The first solution is based on the solution for the *document listing* proposed in [9]. We grammar compress $\mathsf{DA}$, and for all non terminal nodes, we precompute and store the results of *document listing with frequencies* queries. The second solution combines Sadakane's approach [33] for the *term frequency* problem, with the grammar compressed document array. We enhance the grammar compressed document array with bitvectors in each non terminal, to locate the leftmost and rightmost occurrences of each document in the corresponding interval in the document array. The third solution combines Sadakane's approach [33] for the *term frequency* problem with the $\mathsf{ILCP}$ array. In this case we use two copies of the $\mathsf{ILCP}$ array to locate the leftmost and rightmost occurrences of each document in the corresponding interval in the document array.

As a common step in all three approaches, given a collection $D = \{T_1[1..n_1], \ldots, T_t[1..n_t]\}$, we build one $r$-index for the concatenation of the documents $\mathcal{D}$. Given the pattern $P[1..m]$, in order to find the frequencies of the occurrences of the pattern in each document, we first find all occurrences of the pattern $P$ in the concatenation of all documents $\mathcal{D}$ using the $r$-index in $\mathcal{O}(m)$ time and $\mathcal{O}(r \log(n/r))$ space. All occurrences of the pattern $P$ are identified as an interval in the suffix array of $\mathcal{D}$, i.e. $\mathsf{SA}[s_p..e_p]$.

For the second and the third approach we also build an $r$-index for each document $T_i$, for $1 \leq i \leq t$. The $r$-index for $T_1, \ldots, T_t$ can be built in $\mathcal{O}(\sum_{i=1}^{t} n_i) = \mathcal{O}(n)$ time and occupying $\mathcal{O}(\sum_{i=1}^{t} r_i \log(n_i/r_i)) = \mathcal{O}(Rt \log(n/r_k))$ space, where $r_i$ is the number of runs in the $\mathsf{BWT}$ of $T_i$, $R = \sum_{i=1}^{t} r_i$, and $k$ =argmin$(r_1, \ldots, r_t)$.

## 4.1 Precomputed document list with frequencies

Following the ideas for the *document listing* problem proposed in [9], we grammar compress $\mathsf{DA}$ producing a binary and balanced grammar of $\nu$ non-terminals, that can be stored in $\mathcal{O}(r \log(n/r))$ space [14]. Let $\mathcal{T}$ be the parse tree of the document array $\mathsf{DA}[1..n]$, given a non terminal node $nt \in \mathcal{T}$ let $\mathsf{DA}[s_{nt}..e_{nt}]$ be its expansion. For all non terminal nodes $nt \in \mathcal{T}$, we precompute and store the list $D_{nt}$ of the distinct documents in $\mathsf{DA}[s_{nt}..e_{nt}]$ with their frequencies. The lists are stored in ascending order.

**Query.** Given the range $[s_p..e_p]$ of all occurrences of $P$, we find maximal nodes of the parse tree $\mathcal{T}$ that cover $\mathsf{DA}[s_p..e_p]$. Since the grammar is binary and balanced, the number of maximal non terminal nodes covering $\mathsf{DA}[s_p..e_p]$ is $\mathcal{O}(\log n)$. Those nodes can be found in $\mathcal{O}(\log n)$ time traversing the parse tree $\mathcal{T}$ from the root towards the interval $\mathsf{DA}[s_p..e_p]$. We use an atomic heap [12] to merge the $\mathcal{O}(\log n)$ lists and compute the frequencies of the documents, by inserting the head of each list in the heap; extracting the minimum and inserting the next element from the same list. While extracting the document, we compute the frequencies for each document. The atomic heap allows to insert end extract the minimum in constant amortized time, thus the total time to compute the output is $\mathcal{O}(\texttt{ndoc} \log n)$ since each document can appear in each list.

Summarizing, we can answer to Problem 3 in $\mathcal{O}(m + \texttt{ndoc} \log n)$ time, using $\mathcal{O}(r \log(n/r) + t \times \nu)$ space.

### 4.2 Grammar-compressed document array with bitvectors

Let $\mathcal{T}$ be the parse tree of the document array $\mathsf{DA}[1..n]$ with $\nu$ non-terminals. For each non terminal node $nt \in \mathcal{T}$ we store if the $i$-th document occurs in the expansion of $nt$ and, if so, whether the leftmost (resp. rightmost) occurrence is in the left child or in the right child of $nt$. Let $\ell$ and $r$ be the left child and right child of $nt$, respectively. The above information can be stored into two bitvectors $\mathsf{L}_{nt}$ and $\mathsf{R}_{nt}$ of length $t$, such that for all documents $i = 1, \ldots, t$, $\mathsf{L}_{nt}[i] = 0$ if the leftmost occurrence of the $i$-th document is in $\ell$, and 1 otherwise, and $\mathsf{R}_{nt}[i] = 1$ if the rightmost occurrence of the $i$-th document is in $r$, and 0 otherwise. Note that if $\mathsf{L}_{nt}[i] > \mathsf{R}_{nt}[i]$, then the $i$-th document does not occur in $nt$.

For the $i$-th document it holds that $\mathsf{L}_{nt}[i] = \mathsf{L}_{\ell}[i] \wedge \overline{\mathsf{R}_{\ell}[i]}$ and $\mathsf{R}_{nt}[i] = \overline{\mathsf{L}_{r}[i]} \vee \mathsf{R}_{r}[i]$ where $\overline{x}$ is $1 - x$. We compute $\mathsf{L}_{nt}$ and $\mathsf{R}_{nt}$ for each non terminal node in a bottom up fashion and we store them. Considering that non terminal nodes associated to the same non terminal symbol have the same subtree, we can compute the $\mathsf{L}_{nt}$ and $\mathsf{R}_{nt}$ bitvectors only once for each non terminal symbol. Thus, the whole running time of the algorithm is $\mathcal{O}((t/w) \times \nu)$ using bit parallelism on words of $w$ bits.

**Query.** Let $t_1, \ldots, t_k$ be the $k = \mathcal{O}(\log n)$ maximal non terminals that cover the interval corresponding to $\mathsf{DA}[s_p..e_p]$. We build a binary tree $\mathcal{T}'$ having as leaves the nodes corresponding to $t_1, \ldots, t_k$. Each internal node stores a pair of bitvectors $L$ and $R$, computed using the rules described above. The height of $\mathcal{T}'$ is $\mathcal{O}(\log \log n)$. To retrieve the leftmost and rightmost occurrences of each document, we start from the root of $\mathcal{T}'$, for each document present in the root, we descend the tree, using the information stored in the bitvectors, to find first the leftmost, and then the rightmost occurrence of the document.

We perform exactly two traversals of the tree for each document that occurs at least once in the interval, since the $\mathsf{L}$ and $\mathsf{R}$ bitvectors store the information that a document does not appear in the interval of the node. Using bit parallelism on words of size $w$, we can find the leftmost and rightmost occurrence of each document in $\mathcal{O}(\texttt{ndoc}(t/w)(\log n + \log \log n))$ time.

Once we have computed the leftmost and rightmost occurrences $\ell_i$ and $r_i$ for each document $i$, we use random access to $\mathsf{SA}$ of the $r$-index to find their corresponding suffix values $\mathsf{SA}[\ell_i]$ and $\mathsf{SA}[r_i]$ in the concatenation of the documents. We, then, find the corresponding suffix values in the document $T_i$, and, using random access to $\mathsf{ISA}$ we find the leftmost and rightmost occurrence $\ell'_i$ and $r'_i$ in the suffix array of the document $T_i$. The size of this interval is the number of occurrences of the pattern $P$ in $T_i$, i.e. $r'_i - \ell'_i + 1$.

Keeping all together, we can answer queries to Problem 3 in $\mathcal{O}(m+((t/w)\log n+\log(n/r))\texttt{ndoc})$ time, using $\mathcal{O}(r\log(n/r)+Rt\log(n/r_k)+(t/w)\times\nu)$ space.

### 4.3 Double run-length encoded ILCP

We first introduce a variation of the interleaved LCP array (ILCP) introduced in [13] called *double run-length encoded* ILCP, denoted by $\mathsf{ILCP}^{\bigstar}$. The $\mathsf{ILCP}^{\bigstar}$ is composed by the array $\mathsf{VILCP}^{\bigstar}$ storing the values of the runs, and the array $\mathsf{LILCP}^{\bigstar}$ storing their lengths. Given the run-length encoded ILCP array for the collection $D=\{T_1,T_2.\dots,T_t\}$ we merge together consecutive runs whose elements are from the same document, keeping the smallest value as the value of the run. Formally, let $\rho$ be the number of runs of ILCP, let $\ell_1=1$ and $r_1=\mathsf{LILCP}[1]$, and for all $i=2,\dots,\rho$ let $\ell_i=\sum_{j=1}^{i-1}\mathsf{LILCP}[j]$ and $r_i=\ell_i+\mathsf{LILCP}[i]-1$. Moreover, for all $1\le i\le j\le n$, let $|\mathsf{DA}[i..j]|=|\{\mathsf{DA}[k]\mid i\le k\le j\}|$ .

**Definition 1.** *Let us assume that we have computed the run-length encoding up to position $i$ of* VILCP*, the next run of* $\mathsf{ILCP}^{\bigstar}$ *is defined as follows. Let $\ell=max\{k\mid|\mathsf{DA}[\ell_i..r_k]|=1\}$ if $|\mathsf{DA}[\ell_i..r_i]|=1$ and $0$ otherwise. Then $\mathsf{VILCP}^{\bigstar}[j]=\min\{\mathsf{VILCP}[i..i+\ell]\}$, and $\mathsf{LILCP}^{\bigstar}[j]=\sum_{k=i}^{i+\ell}\mathsf{LILCP}[k]$.*

The ILCP has a nice property described in [13] that we are going to recall.

**Lemma 1 ([13, Lemma 1]).** *Given a collection $D=\{T_1,\dots,T_t\}$ whose concatenation is $\mathcal{D}[1..n]$, let SA be its suffix array, and let DA be its document array. Let $\mathsf{SA}[s_p..e_p]$ be the interval corresponding to the occurrences of the pattern $P[1..m]$ in $\mathcal{D}$. Then, the leftmost occurrences of the distinct document identifiers in $\mathsf{DA}[s_p..e_p]$ are in the same positions as the values strictly less than $m$ in $\mathsf{ILCP}[s_p..e_p]$.*

Extending Lemma 1 to $\mathsf{ILCP}^{\bigstar}$ we have that:

**Lemma 2.** *Given a collection $D=\{T_1,\dots,T_t\}$ whose concatenation is $\mathcal{D}[1..n]$, let SA be its suffix array, and let DA be its document array. Let $\mathsf{SA}[s_p..e_p]$ be the interval corresponding to the occurrences of the pattern $P[1..m]$ in $\mathcal{D}$. Then, the leftmost occurrences of the distinct document identifiers in $\mathsf{DA}[s_p..e_p]$ are in the same positions as the values strictly less than $m$ in $\mathsf{ILCP}^{\bigstar}[s_p..e_p]$. If there are two values smaller than $m$ for one document, we consider the leftmost one.*

*Proof.* For the runs of $\mathsf{ILCP}^{\bigstar}$ that are also runs of ILCP, the property of Lemma 1 holds. We have to show that the same property holds also for runs of values from the same document.

Let $[s_p..e_p]$ be the interval of all occurrences of $P$ in the text. If a *same-document* run has value greater than or equals to $m$, then all occurrences in the run have ILCP value larger than or equals to $m$, hence by Lemma 1 the property is satisfied. If the considered run has value strictly smaller than $m$ we have to consider three cases. The first case to consider is if the run is entirely included in $\mathsf{ILCP}[s_p..e_p]$, than the head of the run is the value strictly less than $m$, otherwise the head of the run would not be in the interval $\mathsf{ILCP}[s_p..e_p]$. The second case to consider is if the run is not entirely included in $\mathsf{ILCP}[s_p..e_p]$, and the run is broken by the left boundary of the interval, then, the leftmost occurrence of the document is in $s_p$. The last case is if the run is broken by the right boundary of the interval, then, if there is another run containing a value smaller than $m$ for document $i$, by Lemma 1 the leftmost occurrence is the head of the other run, otherwise the leftmost occurrence is the head of the run crossing the right boundary.

Thus, considering the last run in the interval as a special case, we can apply the same approach as in [13]. Then we consider the last run, checking if it is a *same-document* run or not, and if it is, we check if the same document has already been found by the algorithm.

We build the double run-length encoded $\mathsf{LCP}$ array on $\mathcal{D}$. We, then, build a *range minimum query* data structure [11] on $\mathsf{VILCP}^\bigstar$ and a bitvector $\mathsf{L}[1..n]$ such that $\mathsf{LILCP}^\bigstar[i] = \mathtt{select}_1(\mathsf{L}, i)$. This allows, together with Lemma 2, to use Sadakane's approach to find distinct documents to $\mathsf{VILCP}^\bigstar$. This allows us to retrieve the leftmost occurrences of the distinct documents. To retrieve the rightmost occurrence, we build the $\mathsf{ILCP}$ array using the *right* $\mathsf{LCP}$, i.e. the $\mathsf{LCP}$ array defined as follows. We store in each position $1 \leq i \leq n-1$ the length of the longest common prefix between the two strings $T[\mathsf{SA}[i]..n]$ and $T[\mathsf{SA}[i+1]..n]$. In this case, we have that the rightmost occurrences of the distinct documents in $\mathsf{DA}[s_p..e_p]$ correspond to values of the $\mathsf{ILCP}$ strictly smaller than $m$. In particular, all properties that apply to the $\mathsf{ILCP}$ array also apply to the $\mathsf{ILCP}$ array defined array using the *right* $\mathsf{LCP}$. We, then, also double run-length encode it.

**Query.** Given the interval $[s_p..e_p]$, as in [13], we apply Sadakane's technique to find distinct elements in $\mathsf{DA}$, to find distinct values in both the *double run-length encoded* $\mathsf{ILCP}$ arrays. Provided the positions of the leftmost and rightmost occurrences of each document, we then use the $r$-index to find the corresponding value of the suffix array. We map those positions back in the original document, and, using random access to $\mathsf{ISA}$ of the document, we obtain the interval $[s'_p..e'_p]$ in the suffix array of the document, whose size corresponds to the frequency of the document.

Keeping all together, we can answer queries to Problem 3 in $\mathcal{O}(m+\log(n/r)\mathtt{ndoc})$ time, using $\mathcal{O}(r\log(n/r) + Rt\log(n/r_k) + |\mathsf{ILCP}^\bigstar s|)$ space, where $|\mathsf{ILCP}^\bigstar s|$ is the size of both the $\mathsf{ILCP}^\bigstar$ arrays.

## 5 Experimental result

We implemented the data structures and measured their performance on real-world datasets. Experiments were performed on a server with Intel(R) Xeon(R) CPU E5-2407 processors @ 2.40 GHz and 250 GiB RAM running Debian Linux kernel `4.9.0-11-amd64`. The compiler was `g++` version 6.3.0 with `-O3 -DNDEBUG`

Table 1: Statistics for document collections (small, medium, and large variants): *Collection* name; *Size* in megabytes; *R-Index* bits per symbol (bps); *Docs*, number of documents; *Seqs*, average number of sequences (or versions) per each document; number of *Patterns*; For the synthetic collections (second group), we sum-up variants that use 10 or 100 base documents with the different mutation probabilities.

| *Collection* | *Size* | *R-Index* | *Docs* | *Seqs* | *Patterns* |
|---|---|---|---|---|---|
| `Species` | 105 | 11.79 | 3 | 10 | 7658 |
| | 631 | 3.15 | 3 | 60 | 20 536 |
| `Page` | 110 | 0.60 | 60 | 147 | 7658 |
| | 641 | 0.38 | 190 | 164 | 14 286 |
| `Concat` | 95 | | 10 | 1000 | 7538–10 832 |
| | 95 | | 100 | 100 | 10 614–13 165 |

options. Runtimes were recorded with Google Benchmark framework[5]. The source code is available online at: `github.com/duscob/dret`

**Datasets.** To evaluate our proposals, we experimented on different real and synthetic datasets. We used a variation of the dataset described by Mäklin *et al.* [25], and some of the datasets tested by Cobas and Navarro [9]. These are available at `zenodo.org` and `jltsiren.kapsi.fi/RLCSA`, respectively. Table 1 summarizes some statistics on the collections and patterns used in the queries.

*Real datasets.* We used two repetitive datasets from real-life scenarios: `Species` and `Page`. `Species` collection is composed of sequences of *Enterococcus faecalis*[6], *Escherichia coli*[7] and *Staphylococcus aureus*[8] species. We created three documents, one per species, containing sequences of different strains of the corresponding species. We created two variants of `Species` dataset with 10 and 60 strains per document. `Page` is a collection composed of pages extracted from Finnish-language Wikipedia. Each document groups an article and all its previous revisions. We tested on two variants of `Page` collection of different sizes: the smaller composed of 60 pages and 8834 revisions, and the bigger with 190 pages and 31208 revision.

*Synthetic datasets.* Synthetic collections allow us to explore the performance of our solutions on different repetitive scenarios. We experimented on the `Concat` datasets, very similar to `Page`. Each `Concat` collection contains $d = \{10, 100\}$ documents. Each document groups a base document and $10000/d$ versions of this. We generate the different versions of a base document with a mutation probability $R$. Notice that we have a `Concat` dataset for each combination of $d = \{10, 100\}$ and $R = \{0.001, 0.003, 0.01, 0.03\}$. A mutation is a substitution by a different random symbol. The base documents sequences of 1000 symbols randomly extracted from English file of Pizza&Chili [10].

[5] github.com/google/benchmark

[6] DOI: 10.5281/zenodo.3724100

[7] DOI: 10.5281/zenodo.3724112

[8] DOI: 10.5281/zenodo.3724135

*Queries.* The query patterns for `Species` collections are substrings of lengths $m = \{8, 12, 16\}$ extracted from the dataset. In the case of `Page` datasets, the patterns are Finnish words of length $m \geq 5$ that appears in the collections. For `Concat` collections, the queries are terms selected from an MSN query log. See Gagie *et al.* [13] for more details.

**Implementation details.** All our implementations use the $r$-index as text index. We use the implementation of [14] available at `github.com/nicolaprezza/r-index`. Since the implementation does not support random access to the *suffix array* SA and to the *inverse suffix array* ISA, we used a grammar-compressed *differential suffix array* and *differential inverse suffix array* — the differential versions store the difference between two consecutive values of the array —. Mäkinen *et al.* [24] show that SA of repetitive collections contains large *self-repetitions* which are suitable to be compressed using a grammar compressor like balanced Re-Pair.

Since we use the random access to SA and ISA to retrieve the frequencies of the distinct documents, we implemented also a variant using a *wavelet tree* on the document array, as in [35], to support the `rank` functionalities over the *document array* DA. For our experiments, we use the `sdsl-lite` [17] implementation of the wavelet tree.

**Algorithms.** We plugged-in our proposal with two different approaches to calculate the frequencies from the occurrences. All implementations marked with -ISA uses the random access to SA and ISA to retrieve the frequencies, while the one marked with -WT uses the *wavelet tree*.

- GCDA-PDL: *Grammar-Compressed Document Array with Precomputed Document Lists*. Solution described in Section 4.1, using balanced Re-Pair[9] for DA and sampling the sparse tree as in [9].
- GCDA: *Grammar-Compressed Document Array*. Solution described in Section 4.2, using balanced Re-Pair for DA and bit-vectors stored in the nonterminals. We implemented the variants: GCDA-ISAs and GCDA-WT.
- ILCP: *Interleaved Longest Common Prefix*. Solution described in Section 4.3, using ILCP array (not double run-length encoded). We implemented the variants: ILCP-ISAs and ILCP-WT.
- ILCP$^\bigstar$: *double run-length encoded Interleaved Longest Common Prefix*. Solution described in Section 4.3, using ILCP$^\bigstar$ array. We implemented the variants: ILCP$^\bigstar$-ISAs and ILCP$^\bigstar$-WT.
- Sada: *Sadakane*. The algorithm proposed in [33]. We provide the variants: Sada-ISAs and Sada-WT.
- R-Index: r-*index*. Bruteforce algorithm that scans all occurrences of the pattern, counting the frequencies.

[9] www.dcc.uchile.cl/gnavarro/software/repair.tgz

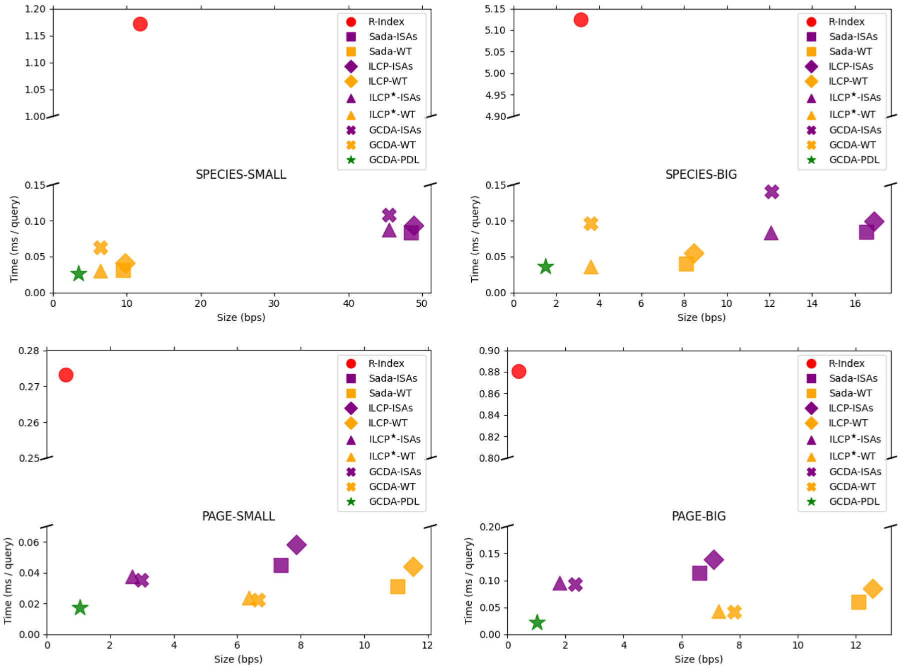

Fig. 1: Document listing with frequencies on `Species` and `Page` datasets. The $x$ axis shows the total size of the index in bits per symbol (bps). The broken $y$ axis shows the average time per query.

Note that in all our algorithms we do not use the random access to SA and ISA of the $r$-index, thus we do not need to store the samples. The only exception is R-Index which needs the samples to compute the frequencies.

**Results.** Figure 1 contains our experimental results for document listing with frequencies on real datasets. We show the trade-off between time and space for all tested indexes on different variants of the collections `Species` and `Page`.

The two variants of `Species` collections are composed of few large documents (only three, one per species). In this scenario, GCDA-PDL proves to be the best solution, finding the document frequencies in 27–36 microseconds ($\mu$sec) per each pattern in average, and requiring only 1.5–3.5 bits per symbol (bps). GCDA-PDL is the fastest and smallest index, requiring even less space than R-Index, since GCDA-PDL does not store the samples. The large size of the sampling scheme for collections with low repetitiveness has also been observed in [14]. The best competitor is ILCP$^\star$-WT, being almost as fast (30–36 $\mu$sec per query) as GCDA-PDL, but requiring 1.85–2.4 times more space. In these collections, -WT indexes perform better than -ISAs solutions. They can answer the queries at least 1.45 times faster, while they are 2–7 times smaller. In terms of space, GCDA-WT

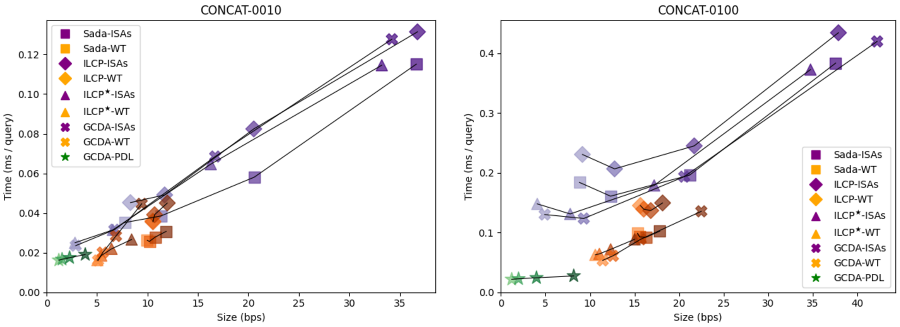

Fig. 2: Document listing with frequencies on synthetic collection `Concat`. The $x$ axis shows the total size of the index in bits per symbols (bps). The $y$ axis shows the average time per query. `R-index` is omitted from the plots due to its excessively high time.

represents a good option, improving even the space required by R-Index in some cases, but much slower than GCDA-PDL and ILCP$^{\star}$-WT.

`Page` collections that contain more documents than `Species` collections: 60 documents in its small version and 190 in the bigger one. Again GCDA-PDL turns up as the best index. It uses less than 1.05 bps and answers the queries in 17–22 $\mu$sec. R-Index requires the least space among the solutions, 0.38–0.60 bps, but is 15.86–40.35 times slower. The second overall-best index is ILCP$^{\star}$-ISAs, with 1.80–2.69 bps and query times of 37–95 $\mu$sec, closely followed by GCDA-ISAs. On the `Page` variants, -WT indexes are faster than its counterparts -ISAs, but 1.47–4.05 times bigger.

On real datasets GCDA-PDL outperforms the rest of the competitors, but the ILCP$^{\star}$-variants are also relevant solutions obtaining a good space/time tradeoff.

The comparison of the indexes on synthetic collections `Concat` are shown in Figure 2. These kinds of collections allow us to observe the indexes' behavior as the repetitiveness varies. Each plot combines the results for the different mutation probabilities of a given collection and number of base documents. The plots show the increasing mutation rates using variations of the same color, from lighter to darker.

GCDA-PDL outperforms all the other indexes. For the collections composed of 10 base documents, our index obtains the best space/time tradeoff, requiring 1.22–3.84 bps with a query time of 16–19 $\mu$sec. Only GCDA-WT and ILCP$^{\star}$-WT obtain competitive query times, but they are 2.20–4.20 times bigger. R-Index requires the least space for lower mutation rates, but it is 79–83 times slower than GCDA-PDL (note that the R-Index data for this collection is not shown in Figure 2 due to its high query times). In the case of the collections composed of 100 base documents, GCDA-PDL dominates the space/time map.

## 6 Discussion

Future work includes the integration of the results with real pseudoaligners. A trivial approach for such integration is to query each $k$-mer of a pattern with our methods, and check if a single document (species) receives positive term frequency. This approach multiplies the $O(m)$ part of the running time with $O(k)$, in addition to affecting the output-sensitive part of the running time. To avoid the $O(k)$ multiplier, we need to maintain the frequencies in a sliding window of length $k$ through the pattern. Such solution requires the techniques of the fully-functional bidirectional BWT index [2] extended to work on the $r$-index. However, one could also modify the pseudoalignment criterion into looking at maximal runs of $k$-mer hits, in the order of the (reverse) pattern. For this, our methods are readily applicable: just do backward search with the pattern $P$ until obtaining an empty interval with suffix $P[i..m]$. Report term frequency of $P[i+1..m]$ if $m-i \geq k$. Continue analogous process backward searching $P[1..i]$. If all the maximal runs of $k$-mer hits report a single document (species) $T_i$, assign $P$ to $T_i$. The $O(m)$ part of the running time remains unaffected, and the output-sensitive part remains smaller than with the sliding window approach.

## Acknowledgments

We wish to thank Antti Honkela and Tommi Mäklin for introducing us the need for better solutions to the pseudo-alignment problem. Some initial solutions were discussed during summer 2019 with Jarno Alanko, Travis Gagie, and Gonzalo Navarro at the Dagstuhl seminar: 25 Years of the Burrows-Wheeler Transform. This led to the plan of tackling this problem during the visits (supported by the EU's Horizon 2020 research and innovation programme under Marie Skłodowska-Curie grant agreement No 690941 (BIRDS) and the Academy of Finland (grant 309048)) of DC and MR to Helsinki. MR is supported by the National Science Foundation (NSF) IIS (Grant No. 1618814). DC is supported by the National Agency for Research and Development (ANID)/Scholarship Program/DOCTORADO BECAS CHILE/2020-21200906 and by Google's Latin America Research Awards 2019. We also wish to thank the anonymous reviewers for their insightful comments and suggestions.

## References

1. Belazzougui, D., Boldi, P., Pagh, R., Vigna, S.: Monotone minimal perfect hashing: searching a sorted table with $O(1)$ accesses. In: Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009. pp. 785–794. SIAM (2009)
2. Belazzougui, D., Cunial, F.: Fully-functional bidirectional Burrows-Wheeler indexes and infinite-order de Bruijn graphs. In: 30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019. LIPIcs, vol. 128, pp. 10:1–10:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)

3. Belazzougui, D., Navarro, G., Valenzuela, D.: Improved compressed indexes for full-text document retrieval. J. Discrete Algorithms **18**, 3–13 (2013)
4. Bray, N.L., Pimentel, H., Melsted, P., Pachter, L.: Near-optimal probabilistic rna-seq quantification. Nature biotechnology **34**(5), 525–527 (2016)
5. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation (1994)
6. Carroll, D., Daszak, P., Wolfe, N.D., Gao, G.F., Morel, C.M., Morzaria, S., Pablos-Méndez, A., Tomori, O., Mazet, J.A.: The global virome project. Science **359**(6378), 872–874 (2018)
7. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Trans. Inf. Theory **51**(7), 2554–2576 (2005)
8. Claude, F., Munro, J.I.: Document listing on versioned documents. In: Proceedings of the 20th International Symposium on String Processing and Information Retrieval, SPIRE 2013. Lecture Notes in Computer Science, vol. 8214, pp. 72–83. Springer (2013)
9. Cobas, D., Navarro, G.: Fast, small, and simple document listing on repetitive text collections. In: Proceedings of the 26th International Symposium on String Processing and Information Retrieval (SPIRE 2019). LNCS, vol. 11811, pp. 482–498. Springer (2019)
10. Pizza & Chili repetitive corpus: Available at `http://pizzachili.dcc.uchile.cl/repcorpus.html`, accessed 16 April 2020
11. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comput. **40**(2), 465–492 (2011)
12. Fredman, M.L., Willard, D.E.: Trans-dichotomous algorithms for minimum spanning trees and shortest paths. Journal of Computer and System Sciences **48**(3), 533–551 (1994)
13. Gagie, T., Hartikainen, A., Karhu, K., Kärkkäinen, J., Navarro, G., Puglisi, S.J., Sirén, J.: Document retrieval on repetitive string collections. Information Retrieval Journal **20**(3), 253–291 (2017)
14. Gagie, T., Navarro, G., Prezza, N.: Fully functional suffix trees and optimal text searching in BWT-runs bounded space. J. ACM **67**(1), 2:1–2:54 (2020)
15. Gagie, T., Navarro, G., Puglisi, S.J.: New algorithms on wavelet trees and applications to information retrieval. Theor. Comput. Sci. **426**, 25–41 (2012)
16. Gagie, T., Puglisi, S.J., Turpin, A.: Range quantile queries: Another virtue of wavelet trees. In: Proceedings of the 16th International Symposium on String Processing and Information Retrieval SPIRE 2009. Lecture Notes in Computer Science, vol. 5721, pp. 1–6. Springer (2009)
17. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. In: Proceedngs of the 13th International Symposium on Experimental Algorithms, (SEA 2014). pp. 326–337 (2014)
18. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA. pp. 841–850. ACM/SIAM (2003)
19. Huson, D.H., Auch, A.F., Qi, J., Schuster, S.C.: Megan analysis of metagenomic data. Genome research **17**(3), 377–386 (2007)
20. Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., McVean, G.: De novo assembly and genotyping of variants using colored de bruijn graphs. Nature genetics **44**(2), 226–232 (2012)

21. Jez, A.: A really simple approximation of smallest grammar. Theor. Comput. Sci. **616**, 141–150 (2016)
22. Lehman, E., Shelat, A.: Approximation algorithms for grammar-based compression. In: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms. pp. 205–212. Society for Industrial and Applied Mathematics (2002)
23. Lindner, M.S., Renard, B.Y.: Metagenomic abundance estimation and diagnostic testing on species level. Nucleic acids research **41**(1), e10–e10 (2013)
24. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and Retrieval of Highly Repetitive Sequence Collections. Journal of Computational Biology **17**(3), 281–308 (2010)
25. Mäklin, T., Kallonen, T., Alanko, J., Mäkinen, V., Corander, J., Honkela, A.: Genomic epidemiology with mixed samples. BioRxiv (2020), supplement: Pseudoalignment in the mGEMS pipeline.
26. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993)
27. Marchet, C., Boucher, C., Puglisi, S.J., Medvedev, P., Salson, M., Chikhi, R.: Data structures based on k-mers for querying large collections of sequencing datasets. bioRxiv p. 866756 (2019)
28. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA). pp. 657–666. Society for Industrial and Applied Mathematics (2002)
29. Navarro, G.: Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. ACM Computing Surveys (CSUR) **46**(4), 52 (2014)
30. Navarro, G.: Document listing on repetitive collections with guaranteed performance. Theoretical Computer Science **772**, 58–72 (2019)
31. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comput. Surv. **39**(1), 2 (2007)
32. Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theor. Comput. Sci. **302**(1-3), 211–222 (2003)
33. Sadakane, K.: Succinct data structures for flexible text retrieval systems. Journal of discrete Algorithms **5**(1), 12–22 (2007)
34. Schaeffer, L., Pimentel, H., Bray, N., Melsted, P., Pachter, L.: Pseudoalignment for metagenomic read assignment. Bioinform. **33**(14), 2082–2088 (2017)
35. Välimäki, N., Mäkinen, V.: Space-efficient algorithms for document retrieval. In: Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching CPM 2007. Lecture Notes in Computer Science, vol. 4580, pp. 205–215. Springer (2007)
36. Weiner, P.: Linear pattern matching algorithms. In: 14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973. pp. 1–11. IEEE Computer Society (1973)
37. Wood, D.E., Salzberg, S.L.: Kraken: ultrafast metagenomic sequence classification using exact alignments. Genome biology **15**(3), R46 (2014)
38. Xia, L.C., Cram, J.A., Chen, T., Fuhrman, J.A., Sun, F.: Accurate genome relative abundance estimation based on shotgun metagenomic reads. PloS one **6**(12) (2011)