A

General Document Retrieval in Compact Space

GONZALO NAVARRO, University of Chile, Chile
SIMON J. PUGLISI, University of Helsinki, Finland
DANIEL VALENZUELA, University of Chile, Chile

Given a collection of documents and a query pattern, *document retrieval* is the problem of obtaining documents that are relevant to the query. The collection is available beforehand so that a data structure, called an index, can be built on it to speed up queries. While initially restricted to natural language text collections, document retrieval problems arise nowadays in applications like bioinformatics, multimedia databases and Web mining. This requires a more general setup where text and pattern can be general sequences of symbols, and the classical inverted indexes developed for words cannot be applied. While linear-space time-optimal solutions have been developed for most interesting queries in this general case, space usage is a serious problem in practice.

In this article we develop compact data structures that solve various important document retrieval problems on general text collections. More specifically, we provide practical solutions for listing the documents where a query pattern appears, together with its frequency in each document, and for listing k documents where a query pattern appears most frequently. Some of our techniques build on existing theoretical proposals, while others are new. In particular, we introduce a novel grammar-based compressed bitmap representation that may be of independent interest when dealing with repetitive sequences.

Ours are the first practical indexes that use less space when the text collection is compressible. Our experimental results show that, on various real-life text collections, our data structures are significantly smaller than the most space-efficient previous solutions, using up to half the space without noticeably increasing the query time. Overall, document listing can be carried out in 10 to 40 milliseconds for patterns that appear 100 to 10,000 times in the collection, whereas top-k retrieval is carried out in k to $10\,k$ milliseconds.

Categories and Subject Descriptors: E.1 [**Data structures**]; E.2 [**Data storage representations**]; E.4 [**Coding and information theory**]: Data compaction and compression; H.3 [**Information storage and retrieval**]

General Terms: Algorithms

Additional Key Words and Phrases: Document retrieval, wavelet trees, top-k retrieval, compressed text databases

1. INTRODUCTION

Enormous repositories of data are arising in almost every area of human activity. Some examples are Web pages, genomic sequences, multimedia sequences, sensor data, financial data, click-through data and query logs, to name a few. Managing and extracting meaningful information from this data requires sophisticated procedures. One of

The work of G.N. and D.V. was funded in part by the Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile, and by Fondecyt Grant 1-110066, Chile. The work of S.J.P. was funded by a Newton Fellowship.
Authors' address: G. Navarro, D. Valenzuela, Department of Computer Science, University of Chile, Santiago, Chile. {gnavarro|dvalenzu}@dcc.uchile.cl. S.J. Puglisi, Department of Computer Science, University of Helsinki, Finland. simon.j.puglisi@gmail.com.
Partial preliminary versions of this paper appeared in *Proc. SEA 2011* and *Proc. SEA 2012*.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

the most basic operations required by virtually all the more complex analysis tasks is that of finding the areas in the data that are *relevant* to certain *patterns*.

When the data to be searched is available beforehand, it is useful to preprocess the data to build an *index* data structure, to improve the speed and effectiveness with which later queries can be answered. The fastest index will pre-store the answer to every possible query. This, however, will be extremely space-inefficient in most cases. On the other hand, using no index at all will reduce the extra space to zero but the time to answer the queries will be at best linear in the size of the data (e.g., a sequential search over the visible Web would take days, while any decent search engine takes less than a second). The challenge of indexing can be thought of as how to achieve relevant space-time trade-offs in between those extremes.

In this article we focus on a simple but common scenario. The data is assumed to be *sequences* over a certain alphabet and to be cut into *documents* (e.g., Web pages, genes, proteins, musical pieces, sensed regions, time periods). Patterns are just short sequences over the same alphabet (e.g., a DNA motif, a protein of interest, a short MIDI sequence that implies influence or plagiarism, a telling numeric pattern in stock behavior, a query that helps analyze user behavior in Web query logs), and relevance is measured just by the frequency of the pattern in the documents. Then the problem becomes finding documents where a pattern string appears frequently. These are generically called *document retrieval problems*. The rest of this section describes these problems in more detail, and summarizes our contributions.

1.1. Document Retrieval Challenges

Document listing is probably the most basic document retrieval problem. Given a collection of documents and a query pattern, document listing consists in obtaining all the documents in which the pattern occurs. This is different from the much better known *full-text search* problem, where all the occurrences of a pattern must be listed. Since there may be thousands of occurrences per document, using full-text searching to solve document listing can be very inefficient.

When the outcome of the search goes directly to the final user (e.g., in Web search engines), document listing might still deliver too many results. In those cases, it is more useful to *rank* the documents by some *relevance* criterion. That is, the documents where the pattern appears are sorted by relevance and only the k most relevant ones are returned. This is called *top-k document retrieval*. One of the simplest, yet common, relevance criteria is the *term frequency*, which is the number of occurrences of the pattern in the document.

Document retrieval problems are best known on *natural language* text collections, where documents and patterns are sequences of words from a (relatively) small vocabulary. In this restricted case, a good solution (used in all Web search engines) is the inverted index [Baeza-Yates and Ribeiro-Neto 2011]. This stores, for each vocabulary word, the list of documents where it appears together with the relevance of the word in each such document. Hence, the answers to all one-word queries are precomputed, and more complex queries are handled through list unions and intersections. However, the inverted index relies on the assumption that the text is tokenizable into querieable units (words), which is not true in many newer scenarios of interest.

One such scenario is documents in languages with ambiguous word boundaries, such as Chinese and Korean. Search engines usually treat these texts as sequences of symbols, so that queries can retrieve any substring; inverted indexes cannot precompute the answer for every text substring. Similarly, retrieving particles of words in agglutinating languages such as Finnish or German is problematic for inverted indexes.

In other scenarios the data to be indexed do not even have a concept of word, yet document retrieval problems arise naturally. In biotechnology, huge repositories of DNA

sequences are arising. Looking for short patterns (motifs) that appear frequently in those sequences is a critical task in many contexts, like understanding diseases. Similarly, protein sequences are searched for short amino acid sequences that may imply some desired function. In large source code repositories, code analysis requires quickly finding the functions or modules where a given function, variable, or expression is used, among more sophisticated problems. Detecting the use of short parts of a musical piece is useful for tracking author influence, as well as spotting plagiarism. Finding areas in a time series where a pattern appears frequently is useful for various data analysis problems, such as the evolution of trade indicators, stock markets, and other financial data. The same problems arise when analyzing past data coming from sensors of various kinds (meteorological, seismic, weather, and so on). Finally, the analysis of user behavior in terms of queries or click-through data collected on logs over time is useful for social mining behavior over regions or time periods.

When the texts to be indexed are not easily tokenized into words, the most successful solutions for document retrieval are based on suffix arrays [Manber and Myers 1993] and suffix trees [Weiner 1973]. Those structures support pattern searches over general texts, meaning that they allow one to search for an arbitrary concatenation of symbols. A suffix array is an array of all the suffixes of the text stored in lexicographic order. Two binary searches find the interval containing all the suffixes that begin with a query pattern. This interval contains all the occurrences of the pattern in the text. Similarly, suffix trees store the suffixes in a digital tree and find the subtree with the answers in a single traversal from the root. This functionality comes at the cost of high space consumption: suffix arrays and trees require 4 to 20 times the size of the original text. Moreover, these data structures by themselves are not powerful enough to efficiently solve document retrieval problems; they just support efficient full-text searches.

In a foundational work, Muthukrishnan [2002] gave the first time-optimal solution to the document listing problem, as well as efficient solutions to various related document retrieval problems. Muthukrishnan's approach is built on the top of the suffix tree, therefore the space consumption, although linear, is even higher. Similarly, Hon et al. [2009] and Navarro and Nekrich [2012] achieved linear-space and optimal-time solutions for top-k document retrieval, by adding further structures to the suffix tree.

Needless to say, the space requirement of all those solutions sharply limits their applicability. The new millennium witnessed a rich development of the field of *compact data structures*. These aim to provide the same functionality of their classical counterparts, or more, using much less space and hopefully not much more time. In particular, *compressed suffix arrays* have evolved rapidly, enabling the same functionality of classical suffix arrays, while requiring as little space as that of the compressed text. They also allow one to recover any desired text substring, and so, in a sense, replace the text. For this reason, they are also called *self-indexes* [Navarro and Mäkinen 2007].

A self-index is a satisfactory data structure in the sense that it uses nearly the minimum space in which the text collection could be represented (under various popular entropy models), and efficiently solves full-text searches within that space. One could hope to solve document retrieval problems efficiently within the same space. However, this has been achieved only for the simplest document listing problem, where no document relevance information is retrieved. For listing documents with the frequencies of the pattern in each, or for retrieving the top-k documents, there are solutions using twice the minimum space, but we will show that their implementation is not practical. On the other hand, the solutions that are efficient in practice require much more than the minimum space. In this article we contribute with practical solutions to overcome this space/time problem.

1.2. Our Contributions

We propose novel compact data structures and algorithms for two document retrieval problems on general sequence collections: document listing with frequencies and top-k retrieval. A roadmap of the paper, highlighting the main contributions, follows.

— Section 2 gives basic definitions and introduces the fundamental concepts. Then Section 3 introduces the document retrieval concepts and reviews existing approaches. It also describes the document collections we will use in all the experiments.
— In Section 4 we propose a grammar-compressed representation of binary sequences that answers *rank* and *select* queries, which are fundamental for other higher-level tasks. This is the first bitmap representation that takes advantage of the repetitiveness of the bitmap instead of its density or clustering of 1s and 0s, and might be of independent interest. Even on non-repetitive sequences, the representation uses space close to alternative structures optimized for bitmap density or clustering. It is significantly slower to operate, however.
— In Section 5 we use the previous result to develop a grammar-compressed sequence representation with *rank* and *select* functionality. This is useful to represent the *document array*, a basic component of most useful document retrieval solutions, whose repetitiveness is related to the compressibility of the collection. This makes up the first compressible representation of the document array, and reduces its space by up to one half in various real-life collections. While this representation is slower than classical ones, we design various tradeoffs that retain most of the space gains at a negligible penalty in time performance for document listing with frequencies.
— In Section 6 we build on the previous representation and add a small data structure that strengthens it for top-k document retrieval queries. We build on a theoretical compact structure by Hon et al. [2009] and engineer it in various ways, reaching an implementation with negligible extra space that speeds up top-k retrieval queries by a factor of up to 10 compared to using just the plain document array.
— Finally, the Conclusions discuss our contributions and possible future directions.

Our new data structures dominate almost the whole space/time tradeoff map. As a quick figure of the actual time performance achieved, document listing with frequencies can be carried out in 10 to 40 milliseconds for patterns that appear 100 to 10,000 times in the collection, whereas top-k retrieval is carried out in k to $10\,k$ milliseconds.

2. PRELIMINARIES

2.1. Empirical Entropy

In the last 10 years the *empirical entropy* has emerged as a powerful means of measuring the compressibility of sequences, and the space efficiency of data structures that operate on them. Let $S[1,n]$ be a sequence over alphabet $\Sigma = \{1,\ldots,\sigma\}$. For each $c \in \Sigma$, let n_c denote the number of times c appears in S. The zero-order empirical entropy [Manzini 2001] of S is

$$H_0(S) = \sum_{c\in\Sigma} \frac{n_c}{n} \log \frac{n}{n_c} = \log n - \frac{1}{n}\sum_{c\in\Sigma} n_c \log n_c. \quad (1)$$

Now let n_w be the number of occurrences of a (sub)string w in S, and let $S|w$ be the subsequence of S consisting of those characters that immediately precede w in S (treating S as a cyclic string). Then the k^{th} order empirical entropy is

$$H_k(S) = \sum_{w\in\Sigma^k} \frac{n_w}{n} H_0(S|w)\ .$$

The value $nH_k(S)$ above is a lower bound on the number of bits needed to encode S by any compressor that considers a context of size at most k when encoding a symbol.

2.2. Rank and Select

Two basic operations used in almost every succinct data structure are *rank* and *select*. Given a sequence $S[1,n]$ over an alphabet $\Sigma = \{1,\ldots,\sigma\}$, a character $c \in \Sigma$, and integers i,j, $rank_c(S,i)$ is the number of times that c appears in $S[1,i]$, and $select_c(S,j)$ is the position of the j-th occurrence of c in S. On binary sequences we assume $rank(S,i) = rank_1(S,i)$ and $select(S,j) = select_1(S,j)$.

There is a great variety of techniques to answer these queries, depending on the nature of the sequence, for example whether or not it will be compressed, the size of the alphabet, etc.

Given a binary sequence $B[1,n]$, the classic solution [Munro 1996; Clark 1998] is built upon the plain sequence, requiring $o(n)$ additional bits. More advanced solutions acheive zero-order compression of B. For example Raman et al. [2007] describe a data structure (now well-known as ***RRR***) that requires $nH_0(B) + o(n)$ bits, and supports *rank* and *select* operations in constant time. Okanohara and Sadakane [2007] describe another structure, called ***SDArray***, that requires $nH_0(B) + 2m + o(m)$ bits for bitmaps with m 1s, and can support *select* in constant time and *rank* in time $O(\log(n/m))$.

2.3. General Sequences: Wavelet Tree

There are many solutions for the *rank* and *select* problem on general sequences [Grossi et al. 2003; Golynski et al. 2006; Ferragina et al. 2007; Barbay et al. 2011; Barbay et al. 2010]. We will focus on one of the most versatile and useful: the *wavelet tree* [Grossi et al. 2003]. Let us consider a sequence $T = a_1 a_2 \ldots a_n$ over an alphabet Σ as before.

The wavelet tree of T is a binary balanced tree, where each leaf represents a symbol of Σ. The root is associated with the complete sequence T. Its left child is associated with a subsequence obtained by concatenating the symbols a_i of T satisfying $a_i < \sigma/2$. The right child corresponds to the concatenation of every symbol a_i satisfying $a_i \geq \sigma/2$. This relation is maintained recursively up to the leaves, which will be associated with the repetitions of a unique symbol. At each node we store only a binary sequence of the same length of the corresponding sequence, using at each position a 0 to indicate that the corresponding symbol is mapped to the left child, and a 1 to indicate the symbol is mapped to the right child.

If the bitmaps of the nodes support constant-time *rank* and *select* queries, then the wavelet tree supports fast *access*, *rank* and *select* on T.

Access: In order to obtain the value of a_i the algorithm begins at the root, and depending on the value of the root bitmap B at position i, it moves down to the left or to the right child. If the bitmap value is 0 it goes to the left, and replaces $i \leftarrow rank_0(B,i)$. If the bitmap value is 1 it goes to the right child and replaces $i \leftarrow rank_1(B,i)$. When a leaf is reached, the symbol associated with that leaf is the value of a_i.

Rank: To obtain the value of $rank_c(S,i)$ the algorithm is similar: it begins at the root, and goes down updating i as in the previous query, but the path is chosen according to the bits of c instead of looking at $B[i]$. When a leaf is reached, the i value is the answer.

Select: The value of $select_c(S,j)$ is computed as follows: The algorithm begins in the leaf corresponding to the character c, and then moves upwards until reaching the root. When it moves from a node to its parent, j is updated as $j \leftarrow select_0(B,j)$ if the node is a left child, and $j \leftarrow select_1(B,j)$ otherwise. When the root is reached, the final j value is the answer.

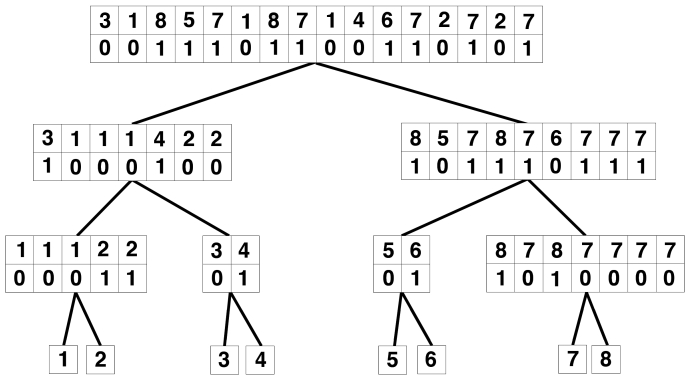


Fig. 1. Wavelet tree of the sequence $T = 3185718714672727$, We show the mapped sequences in the nodes only for clarity, the wavelet tree only stores the bitmaps.

2.4. Grammar Compression of Sequences: RePair

Grammar compression replaces the original sequence by a context-free grammar that uniquely produces the original sequence. Compression is achieved by finding a grammar that requires less space to be represented than the original sequence.

The problem of finding the smallest grammar to represent a sequence is known to be NP-Complete. However, there are very efficient algorithms that asymptotically achieve the entropy of the sequence, LZ78 [Ziv and Lempel 1978], and RePair [Larsson and Moffat 2000] among others.

We focus on RePair because it has proven to fit very well in the field of succinct data structures [González and Navarro 2007]. RePair is an off-line dictionary-based compressor that achieves high-order compression taking advantage of repetitiveness and allowing fast random access [Larsson and Moffat 2000; Navarro and Russo 2008].

RePair looks for the most common pair in the original sequence and replaces that pair with a new symbol, adding the corresponding replacement rule to a dictionary. The process is repeated until no pair appears twice. More formally, given a sequence T over an alphabet of size σ, RePair begins with an empty dictionary $\mathcal{R}$ and

(1) Identifies the symbols a and b in T, such as ab is the most frequent pair in T. If no pair appears twice, the algorithm stops.
(2) Creates a new symbol A, adds a new rule to the dictionary, $\mathcal{R}(A) \rightarrow ab$, and replaces every occurrence of ab in T with A.
(3) Repeats from (1).

As a result, the original sequence T is transformed into a new, compressed, sequence $\mathcal{C}$ (including original symbols as well as newly created ones), and a dictionary $\mathcal{R}$. Note that the new symbols have values larger than σ, thus the compressed sequence alphabet is $\sigma' = \sigma + |\mathcal{R}|$.

To decompress $\mathcal{C}[j]$ we evaluate: if $\mathcal{C}[j] \leq \sigma$, then it is an original symbol, so we return $\mathcal{C}[j]$. Otherwise, we expand it using the rule $\mathcal{R}(\mathcal{C}[j]) \rightarrow ab$ and repeat the process recursively with a and b. In this manner we can expand $\mathcal{C}[j]$ in time $O(|\mathcal{C}[j]|)$.

2.5. Succinct Representation of Trees

How to succinctly encode a tree has been subject to a lot of study [Jacobson 1989; Benoit et al. 2005; Barbay et al. 2011; Sadakane and Navarro 2010; Arroyuelo et al. 2010]. Although there are many proposals that achieve optimal space, that is $2n+o(n)$ bits to encode a tree of n nodes, they offer different query times for different queries. Most solutions are based on Balanced Parentheses [Jacobson 1989; Munro and Raman 2002; Raman et al. 2007; Geary et al. 2006; Sadakane and Navarro 2010], Depth-First Unary Degree Sequences (DFUDS) [Benoit et al. 2005], or Level-Order Unary Degree Sequences (LOUDS) [Jacobson 1989].

Arroyuelo et al. [2010] implemented and compared the major current techniques and showed that, for the functionality it provides, LOUDS is the most promising succinct representation of trees. The $2n+o(n)$ bits of space required can, in practice, be as little as $2.1n$ and the representation solves many operations in constant time (and fast in practice). In particular, it allows fast navigation through labeled children.

In LOUDS, the shape of the tree is stored using a single binary sequence, as follows. Starting with an empty bitstring, every node is visited in level order starting from the root. A node with c children is encoded by writing its arity in unary, that is, appending 1^c0 to the bitstring. Each node is identified with the position in the bitstring where the encoding of the node begins. If the tree is labeled, then all the labels are put together in another sequence, and the labels are indexed by the *rank* of the node in the bitstring.

2.6. Range Minimum Queries

Range minimum queries (RMQ) are useful in the field of succinct data structures [Fischer and Heun 2011; Sadakane 2007]. Given a sequence $A[1,n]$, a range minimum query from i to j asks for the position of the minimum element in the subsequence $A[i,j]$. The RMQ problem consists in building an additional data structure over A that allows one to answer RMQ queries on-line in an efficient manner. There are two possible settings: in the so-called systematic setting, the sequence A is available at query time, whereas in the non-systematic setting A is no longer available during query time.

Sadakane [2007] gave the first known solution for the non-systematic setting, requiring $4n+o(n)$ bits and answering the queries in $O(1)$ time. Later, Fischer and Heun [2011] offered a solution requiring the optimal $2n+o(n)$ bits, and answering the RMQ queries in $O(1)$ time.

2.7. Direct Access Codes

There exist several techniques to represent sequences $S[1,n]$ of numbers using variable-length codes, usually using shorter codes for smaller numbers. Those are useful when most numbers in a sequence are small, but there can also be some large ones.

A recurrent problem in this case is how to directly access the ith number in a sequence. Brisaboa et al. [2013] propose a simple and practical technique called *Direct Access Codes (DACs)*. Given a parameter b, this technique cuts the binary representation of each number into blocks of b bits, possibly wasting some bits at the end of the last block. It then stores a sequence S_1 with all the first blocks of all the numbers, a sequence S_2 with all the second blocks of all the numbers that have more than one block, and so on. It also stores bitmaps $B_\ell[1,|S_\ell|]$ such that $B_\ell[i]=1$ iff the block stored at $S_\ell[i]$ is not the last of its number. Then, the first block of number $S[i]$ is $S_1[i]$. If $B_1[i]=1$, then there is a second block, which is found at $S_2[rank(B_1,i)]$, and so on.

With this technique, given a sequence of n numbers whose binary representations require u bits, we waste at most $u/b+nb$ bits, and recover any number of t bits in time $O(1+t/b)$. Parameter b can be changed at each level (b_ℓ), and it is not hard to tune those b_ℓ values to minimize the total space, using dynamic programming.

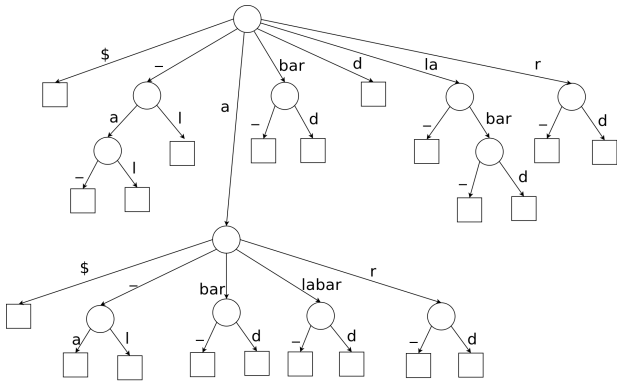


Fig. 2. Suffix Tree of the text “alabar_a_la_alabarda$”.

2.8. Suffix Trees

The suffix tree is a classic full-text index introduced by Weiner [1973], providing a powerful data structure that requires linear space and supports the counting of the occurrences of an arbitrary pattern P in optimal $O(|P|)$ time, as well as locating these *occ* occurrences in $O(|P| + occ)$ time, which is also optimal.

Let us consider a text $T[1, n]$, with a special end-marker $T[n] = \$$ that is lexicographically smaller than any other character in T. We define a *suffix* of T starting at position i as $T[i, n]$. The suffix tree is a digital tree containing every suffix of T. The root corresponds to the empty string, every internal node corresponds to a proper prefix of (at least) two suffixes, and every leaf corresponds to a suffix of T. Each unary path is compressed to ensure that the space requirement is $O(n \log n)$ bits, and every leaf contains a pointer to the corresponding position in the text. Figure 2 shows an example.

To find the occurrences of an arbitrary pattern P in T, the algorithm begins at the root and follows the path corresponding to the characters of P. The search can end in three possible ways:

(1) At some point there is no edge leaving from the current node that matches the characters that follows in P, which means that P does not occur in T;
(2) we read all the characters of P and end up at a tree node, called the *locus* of P (we can also end in the middle of an edge, in which case the locus of P is the node following that edge), in which case all the answers are in the subtree of the locus of P; or
(3) we reach a leaf of the suffix tree without having read the whole P, in which case there is at most one occurrence of P in T, which must be checked by going to the suffix pointed to by the leaf and comparing the rest of P with the rest of the suffix.

2.9. Suffix Arrays

The suffix array [Manber and Myers 1993] is also a classic full-text index that allows us to efficiently count and find the occurrences of an arbitrary pattern in a given text.

The suffix array of $T[1, n] = t_1 t_2 \dots t_n$ is an array $\mathsf{SA}[1, n]$ of pointers to every suffix of T, lexicographically sorted. More specifically, $\mathsf{SA}[i]$ points to the suffix $T[\mathsf{SA}[i], n] = t_{\mathsf{SA}[i]} t_{\mathsf{SA}[i]+1} \dots t_n$, and it holds that $T[\mathsf{SA}[i], n] < T[\mathsf{SA}[i+1], n]$. The suffix array requires $n\lceil \log n \rceil$ bits in addition to the text itself.

To find the occurrences of P using the suffix array, it is important to notice that every substring is the prefix of a suffix. In the suffix array the suffixes are lexicographically sorted, so the occurrences of P will be in a contiguous interval of SA, $\mathsf{SA}[sp, ep]$, such

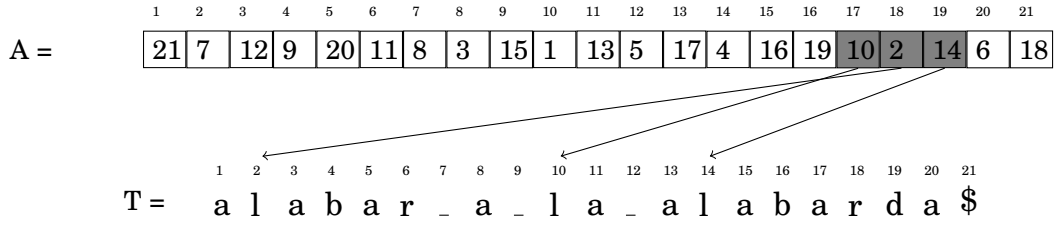


Fig. 3. Suffix Array of the text "alabar_a_la_alabarda$". The shaded interval corresponds to the suffixes that begin with"la".

Table I. Summary of some of the main self-indexes, and their (simplified) time and space complexities. The size is expressed in bits, t_{SA} is the time required to compute either $\mathsf{SA}[i]$ or $\mathsf{SA}^{-1}[i]$, and $search(P)$ is the time required to compute the $[sp, ep]$ interval. Parameter ϵ is any constant greater than 0.

Index	Size	t_{SA}	$search(P)$
CSA [Sadakane 2003]	$\frac{1}{\epsilon}n(H_0+1)+o(n\log\sigma)$	$O(\log^{\epsilon} n)$	$O(\|P\|\log n)$
CSA [Grossi et al. 2003]	$(1+\frac{1}{\epsilon})nH_k+o(n\log\sigma)$	$O(\log^{\epsilon} n)$	$O(\|P\|\log\sigma+\mathrm{polylog}(n))$
FM-Index [Ferragina et al. 2007]	$nH_k+o(n\log\sigma)$	$O(\log^{1+\epsilon} n)$	$O(\|P\|\frac{\log\sigma}{\log\log n})$

that every suffix $t_{\mathsf{SA}[i]}t_{\mathsf{SA}[i]+1}\dots t_n$, for every $sp \le i \le ep$, contains P as a prefix. This interval is easily computed using two binary searches, one to find sp and another one to find ep. It takes $O(|P|\log n)$ time to find the interval. Figure 3 shows an example.

2.10. Self-Indexes

A self-index is a data structure built in a preprocessing phase of a text T, that is able to answer the following queries for an arbitrary pattern P:

— Count(P): Number of occurrences of pattern P in T.
— Locate(P): Position of every occurrence of pattern P in T.
— Access(i): Character t_i.

The last query means that the self-index provides random access to the text, thus it is a replacement for the text. If the space requirement of a self-index is proportional to that of a compressed representation of the text, then the self-index can be thought of as a compressed representation of T and as a full-text index of T *at the same time*.

Most self-indexes emulate a suffix array, and these are divided into two main families: the FM-Index [Ferragina and Manzini 2005; Ferragina et al. 2007; Mäkinen and Navarro 2007], and the Compressed Suffix Array [Grossi and Vitter 2006; Grossi et al. 2003; Sadakane 2003].

For some document retrieval applications of self-indexes we will need not only to answer the queries defined above, but also to efficiently compute $\mathsf{SA}[i]$ and its inverse, $\mathsf{SA}^{-1}[i]$. Table I briefly sketches the space requirement, the time to compute $\mathsf{SA}[i]$ or $\mathsf{SA}^{-1}[i]$, called t_{SA}, and the time to compute the interval $\mathsf{SA}[sp, ep]$, called $search(P)$, for the most prominent self-indexes that emulate suffix arrays. We will refer to any of those self-indexes as *compressed suffix arrays (CSAs)*.

3. RELATED WORK

Consider a collection of D documents $\mathcal{D} = \{d_1, d_2, \dots, d_D\}$, of total length $\sum_{i=1}^{D} |d_i| = n$, and call T their concatenation, which is a sequence over an alphabet $\Sigma = [1..\sigma]$. We assume that every document ends with a special end-marker \$, satisfying $\$ < c, \forall c \in \Sigma$. For an arbitrary pattern P over alphabet Σ, the following queries are defined:

— *Document Listing(P):* Return the n_{doc} documents that contain P.
— *Document Listing(P) with frequencies:* Return the n_{doc} documents that contain P, with the frequency (number of occurrences) of P in each document d, $TF_{P,d}$.
— *Top(P,k):* Return k documents where P occurs most times.

In this section we cover the existing solutions to handle these queries in the scenario of general sequences, focusing on reduced-space approaches.

3.1. Non-Compressed Approaches

Muthukrishnan [2002] proposed the first solution to the document listing problem that is optimal-time and linear-space. He builds on suffix trees (Section 2.8) and introduces a new data structure for document retrieval problems: the so-called *document array*.

Muthukrishnan's solution builds the generalized suffix tree of $\mathcal{D}$, which is a suffix tree containing the suffixes of every document in $\mathcal{D}$. This is equivalent to the suffix tree of the concatenated collection, T. This structure requires $O(n \log n)$ bits.

To obtain the document array of $\mathcal{D}$, $\mathsf{DA}[1, n]$, we consider the suffix array of T, $\mathsf{SA}[1, n]$ (Section 2.9) and, for every position pointed from $\mathsf{SA}[i]$, we store in $\mathsf{DA}[i]$ the document corresponding to that position. The document array requires $n \log D$ bits.

Up to this point, the query can be answered by looking for the pattern P in the suffix tree in $O(|P|)$ time, determining the range $\mathsf{DA}[sp, ep]$ of the occ occurrences of P in DA, and then reporting every different document in $\mathsf{DA}[sp, ep]$. That would require at least $O(|P|+occ)$ time. The problem is that $occ = ep-sp+1$ can be much bigger than n_{doc}, the number of different documents where P occurs. An optimal algorithm would answer the query in $O(|P| + n_{\mathsf{doc}})$ time.

For this purpose, Muthukrishnan defines an array C, which is built from DA, and that for each document stores the position of the previous occurrence of the same document in DA. Thus, C can be seen as a collection of linked lists for every document. More formally, C is defined as $C[i] = max\{j < i, \mathsf{DA}[i] = \mathsf{DA}[j]\}$ or $C[i] = -1$ if such j does not exist. C must be enriched to answer range minimum queries (RMQs, Section 2.6). The space required by the C array and the RMQ structure is $O(n \log n)$ bits.

To understand Muthukrishnan's algorithm it is important to note that, for every different document present in $\mathsf{DA}[sp, ep]$, there exists exactly one position in C pointing to a number smaller than sp. Those positions are going to be used to report the corresponding document. The algorithm works recursively as follows: find the position j with the smallest value in $C[sp, ep]$ using the RMQ structure. If $C[j] \geq sp$, output nothing and return. If $C[j] < sp$, return $\mathsf{DA}[j]$ and continue the procedure with the intervals $\mathsf{DA}[sp, j-1]$ and $\mathsf{DA}[j+1, ep]$ (looking in both subintervals for positions where C points to positions smaller than the original sp boundary). This algorithm clearly reports each distinct document $\mathsf{DA}[j]$ in $[sp, ep]$ where $C[j] < sp$, and no document is reported more than once. The total space required is $O(n \log n)$ bits, which is linear. The total time is $O(|P| + n_{\mathsf{doc}})$, which is optimal.

An optimal solution to top-k document retrieval has required more work. Hon et al. [2009] achieved linear space and $O(|P| + k \log k)$ time. They enrich the suffix tree of T with bidirectional pointers that describe the subgraph corresponding to the suffix tree of each individual document. It is shown that, if P corresponds to the suffix tree node v, then each distinct document where P appears has a pointer from the subtree of v to an ancestor of v. Thus they collect the pointers arriving at nodes in the path from the root to v and extract the k with most occurrences from those candidates. Navarro and Nekrich [2012] achieve linear-space and optimal time $O(|P| + k)$ by reducing this search over candidates to a geometric problem where we want the k heaviest points on a three-sided range over a grid. In both cases, the space is several times that of the suffix tree, which is already high.

While $O(n \log n)$ bits is called "linear-space", it is in practice many times the size of the collection itself, $n \log \sigma$ bits without compression. There have been various approaches to reduce the space of Muthukrishnan's and Hon et al.'s solutions. An easy step is to replace the suffix tree by a self-index. As stated in Section 2.10, we will refer to any of those indexes as a compressed suffix array (CSA). All the algorithms presented in this article use a CSA to obtain the interval $\mathsf{SA}[sp, ep]$ corresponding to the suffixes beginning with P, in a time $search(P)$ that depends on the particular CSA. The challenging part is to reduce the space of the other data structures, in particular the document array DA. The rest of this section surveys the main techniques for document listing and top-k retrieval queries using reduced space.

3.2. Wavelet Tree Based Approaches

Document listing with frequencies. Välimäki and Mäkinen [2007] were the first in using a wavelet tree (Section 2.3) to represent DA. While the wavelet tree takes essentially the same space of the plain representation, $n \log D + o(n \log D)$ bits, they showed that it can be used to emulate the array C using *rank* and *select* on DA (Section 2.2), with $C[i] = select_{\mathsf{DA}[i]}(\mathsf{DA}, rank_{\mathsf{DA}[i]}(\mathsf{DA}, i) - 1)$. Therefore, C is not needed anymore. The time for document listing becomes $O(search(P) + n_{\mathsf{doc}} \log D)$. The wavelet tree also allows them to compute the frequency of P of any document d as $TF_{P,d} = rank_d(\mathsf{DA}, ep) - rank_d(\mathsf{DA}, sp - 1)$, in $O(\log D)$ time as well. The RMQ data structure is still necessary to emulate Muthukrishnan's algorithm.

Later, Gagie et al. [2009] showed that the wavelet tree was powerful enough to get rid of the whole Muthukrishnan's algorithm. They defined a new operation over wavelet trees, the so-called *range quantile query (RQQ)*. Given the wavelet tree of a sequence DA, a range quantile query takes a rank k and an interval $\mathsf{DA}[sp, ep]$ and returns the kth smallest number in $\mathsf{DA}[sp, ep]$. To answer such query $RQQ(\mathsf{DA}[sp, ep], k)$, they begin in the root of the wavelet tree and compute the number of 0s in the interval corresponding to $\mathsf{DA}[sp, ep]$, $n_z = rank_0(B_{root}, ep) - rank_0(B_{root}, sp - 1)$, where B_{root} is the bitmap associated to the root node. If $n_z \geq k$, then the target number will be found in the left subtree of the root. Therefore, the algorithm updates $sp = rank_0(B_{root}, sp - 1) + 1$, $ep = rank_0(B_{root}, ep)$, and continue on the left subtree. If $n_z < k$, the target is in the right subtree, so the algorithm updates $k = k - n_z$, $sp = rank_1(B_{root}, sp-1)+1$ and $ep = rank_1(B_{root}, ep)$, and continues on the right. When a leaf is reached, its corresponding value is the kth smallest in the subinterval.

To solve the document listing problem, Gagie et al. used the wavelet tree of the document array, and applied successive range quantile queries. The first document is obtained as $d_1 = RQQ(\mathsf{DA}[sp, ep], 1)$, the second is $d_2 = RQQ(\mathsf{DA}[sp, ep], 1 + TF_{P,d_1})$, and in general $d_j = RQQ(\mathsf{DA}[sp, ep], 1 + \sum_{i<j} TF_{P,d_i})$. The frequencies are computed also in $O(\log D)$ time, $TF_{P,d_j} = rank_{d_j}(D, ep) - rank_{d_j}(D, sp - 1)$. Each range quantile query takes $O(\log D)$ time, therefore the total time to enumerate all the documents in $\mathsf{DA}[sp, ep]$ is $O(n_{\mathsf{doc}} \log D)$.

Culpepper et al. [2010] showed that Gagie et al.'s algorithm could be simplified to a depth-first search, which improves the performance. They also made a practical improvement that obtains the frequencies for free.

The algorithm begins in the root of the wavelet tree and calculates the number of 0s (resp. 1s) in $B_{root}[sp, ep]$, n_0 (resp. n_1), using two *rank* operations. If n_0 (resp. n_1) is not zero, the algorithm continues to the left (resp. right) child of the node. When the algorithm reaches a leaf, the document value encoded in that leaf is reported, and if the leaf was a left child (resp. right), then the n_0 (resp. n_1) value calculated in its parent is reported as the document frequency.

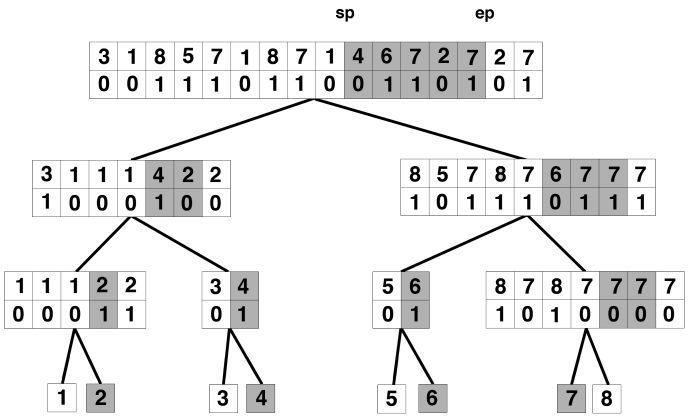


Fig. 4. Wavelet tree representation of a document array and depth-first search traversal to perform document listing with frequencies. Shaded regions show the intervals $[sp, ep]$ mapped in each node.

The DFS traversal visits the same nodes that are visited by the successive range quantile queries. However, the range quantiles visit some nodes many times (e.g., the root is visited n_{doc} times). Calculation shows that the worst-case complexity is $O(n_{\mathsf{doc}} \log(D/n_{\mathsf{doc}}))$ [Gagie et al. 2012]. Figure 4 shows an example.

Top-k document retrieval. Culpepper et al. [2010] also explored different heuristics to solve top-k document retrieval queries using a wavelet tree. ***Quantile probing*** is based on the observation that in a sorted array X, if there exists a d with frequency larger than f in X, then there exists at least one j such that $X[j \cdot f] = d$. Instead of sorting the document array, Culpepper et al. use range quantile queries to access the document with a given rank in $\mathsf{DA}[sp, ep]$.

The algorithm makes successively more refined passes using the above observation, and stores candidates in a min-heap of size k. The first pass finds the document with rank $m/2$, where $m = ep - sp + 1$, computes its frequency, and inserts it into the heap. In the second pass, the elements probed are those with ranks $m/4$ and $3m/4$, their frequencies are computed, and each is inserted in the heap, if not already present. If the heap exceeds the size k, then the smallest element of the heap is removed. Each pass considers the new quantiles of the form $jm/2^i$. If after the ith pass the heap has k elements, and its minimum element has a frequency larger than $m/2^{i+1}$, then the candidates in the heap are the actual top-k documents and the algorithm terminates.

Greedy traversal was the other top-k heuristic presented by Culpepper et al. [2010], and it performed best in practice. The key idea is that, if a document d has a high $TF_{P,d}$ value, then the $[sp, ep]$ intervals on the path to that document leaf in the wavelet tree should also be long.

By choosing the nodes to inspect in a greedy fashion (i.e., longer $[sp, ep]$ intervals first) the algorithm will first reach the leaf (and report its corresponding document) with highest TF value. The next document reported will be the second highest score, and so on until reporting k documents with highest TF value. The algorithm uses a max-heap where wavelet tree nodes to traverse are maintained. Initially it contains only the root node with the associated interval $[sp, ep]$ and key $ep - sp + 1$. In each step, the longest interval is extracted from the heap. If it is a leaf, the document is re-

ported, as explained. Otherwise, the two children of the node, with their corresponding intervals, are inserted in the heap. In this way, the algorithm can avoid inspecting a large part of the tree that would be explored in a depth-first search traversal. Although very good in practice, there are no time guarantees for this algorithm beyond the DFS worst-case complexity, $O(n_{\mathsf{doc}} \log(D/n_{\mathsf{doc}}))$.

3.3. CSA Based Approaches

Document listing. A parallel development started with Sadakane [2007]. He proposed to store a bitmap $B[1,n]$ marking with a 1 the positions in T where the documents start. B should be enriched to answer *rank* and *select* queries (Section 2.2). Sadakane used the fact that DA can be easily simulated using a CSA of T and the bitmap B, computing $\mathsf{DA}[i] = rank(B, \mathsf{SA}[i])$, using very little extra space on top of the CSA: A compressed representation of B requires just $D\log(n/D) + O(D) + o(n)$ bits, supporting *rank* in constant time [Raman et al. 2007].

To emulate Muthukrishnan's algorithm, Sadakane showed that C can be discarded as well, because just RMQ queries on C are needed. He proposed an RMQ structure using $4n + o(n)$ bits that does not need to access C (a more recent proposal [Fischer and Heun 2011] adresses the same problem using just $2n + o(n)$ bits, see Section 2.6). Therefore, the overall space required for document listing was that of the CSA plus $O(n)$ bits. The time is $O(search(P) + n_{\mathsf{doc}} t_{\mathsf{SA}})$, where t_{SA} is the time required by the CSA to compute $\mathsf{SA}[i]$ (Section 2.9). Both in theory and in practice, this solution is competitive in time and uses much less space than those based on wavelet trees, yet it only solves basic document listing queries.

Hon et al. [2009] showed how to reduce the extra space to just $o(n)$ by sparsifying the RMQ structure, impacting the time with a factor $\log^{\epsilon} n$ for listing each document, for some constant $0 < \epsilon < 1$. To accomplish this, instead of building the RMQ structure over C, Hon et al. built the RMQ structure over a sampled version of C, C^*. They consider chunks of size $b = \log^{\epsilon} n$ in C, and store in C^* the smallest value of each chunk, that is, the leftmost pointer. The queries are solved in a similar way: using the RMQ data structure they find all the values in $C^*[\lceil sp/b \rceil, \lfloor ep/b \rfloor]$ that are smaller than sp. Each such a value still corresponds to an actual document to be reported, but now they need to check its complete chunk of size b (as well as the two extremes of $[sp, ep]$, not convering whole chunks) looking for other values smaller than sp. Therefore, Hon et al. answer document listing queries using $|\mathsf{CSA}| + D\log(n/D) + O(D) + o(n)$ bits, and the total time becomes $O(search(P) + n_{\mathsf{doc}} t_{\mathsf{SA}} \log^{\epsilon} n)$.

Document listing with frequencies. Sadakane [2007] also proposed a method to calculate the frequency of each document: He stores an individual CSA, CSA_d, for each document d of the collection. By computing SA and SA^{-1} a constant number of times over the global CSA and that of document d, it is possible to compute frequencies on document d. The idea is to determine the positions $[sp_d, ep_d]$ in SA_d corresponding to the suffixes beginning with P in document d, and then report $TF_{P,d} = ep_d - sp_d + 1$. The naive approach is, after solving the document listing problem, for each document d in the output compute its frequency by searching for the pattern in CSA_d. This would take $O(n_{\mathsf{doc}}(search(P) + t_{\mathsf{SA}}))$ time.

Sadakane improves that time by finding the leftmost and rightmost occurrences of each document d in $\mathsf{DA}[sp, ep]$, and then mapping the extremes to $\mathsf{SA}_d[sp_d, ep_d]$. He defines an array C' that is analogous to C. The array C is regarded as a set of linked lists for each document. C' represents the linked lists in the opposite direction, that is $C'[i] = min\{j > i, \mathsf{DA}[j] = \mathsf{DA}[i]\}$ or $D + 1$ if no such j exists. Then, a procedure analogous to that on C, but using range maximum queries on C', enumerates the

rightmost indices j of all the distinct values in $\mathsf{DA}[sp, ep]$. As for C, C' is not stored but only the RMQ structure over it.

After matching the documents in both lists, Sadakane has the leftmost l_d and the rightmost r_d indices in $[sp, ep]$ where $\mathsf{DA}[l_d] = \mathsf{DA}[r_d] = d$, for each document d. It is now possible to map these values to the corresponding positions $[sp_d, ep_d]$ in SA_d with the following calculation: let $x = \mathsf{SA}[l_d]$ and $y = \mathsf{SA}[r_d]$ be the corresponding positions in the text T. Those are calculated in time $O(t_{\mathsf{SA}})$ using the global CSA. We then obtain the position z in T that corresponds to the beginning of the document d in constant time using $z = \mathit{select}(B, d)$. Then $x' = x - z$ and $y' = y - z$ are the positions in the document d that correspond to x and y. Finally, $sp_d = \mathsf{SA}_d^{-1}[x']$ and $ep_d = \mathsf{SA}_d^{-1}[y']$ are computed using CSA_d in time $O(t_{\mathsf{SA}})$.

Thus, Sadakane solved the document listing with frequencies problem in $O(\mathit{search}(P) + n_{\mathsf{doc}}(t_{\mathsf{SA}} + \log\log n_{\mathsf{doc}}))$ time (using a sorting algorithm by Andersson et al. [1995]). The space needed is $2|\mathsf{CSA}| + O(n)$ bits.

Top-k document retrieval. Hon et al. [2009] proposed a new data structure with worst-case guarantees to answer top-k queries. Their proposal is essentially a sampling of the suffix tree of T storing some extra data and being sparse enough to require just $o(n)$ extra bits.

Hon et al. consider a fixed value of k and a parameter g as the "group size". They traverse the leaves of the suffix tree from left to right, forming groups of size g, and marking the lowest common ancestor (lca) of the leftmost and rightmost leaves of each group: $lca(l_1, l_g)$, $lca(l_{g+1}, l_{2g})$, and so on. Then, they do further marking to ensure that the set of marked nodes is closed under the lca operation, so that the total number of marked nodes becomes at most $2n/g$. At each marked node v, they store a list called ***F-list*** with the k documents where the string represented by v appears most often, together with the actual frequencies in those k documents. They also store the intervals of leaves associated with the node, $[sp_v, ep_v]$. With all the marked nodes they form the so-called τ_k tree, which requires $O((n/g)k \log n)$ bits. Fixing $g = \Theta(k \log^{2+\epsilon} n)$, for an arbitrary $\epsilon > 0$, the space of the τ_k tree is $O(n/\log^{1+\epsilon} n)$. Adding the space required by the τ_k trees for $k = 1, 2, 4, 8, 16, \ldots$, the total amount of space is $O(n/\log^{\epsilon} n) = o(n)$ bits.

To answer a top-k query they first look for the pattern using the CSA, finding the interval $[sp, ep]$. Now, they take k' as the power of two next to k, $k' = 2^{\lceil \log k \rceil}$, and traverse $\tau_{k'}$ looking for the node with the largest interval contained in $[sp, ep]$, namely $[L, R]$. The k' most frequent documents in $[L, R]$ are stored in the ***F-list*** of the node, so attention is placed on the documents not covered by the ***F-list***. They examine each suffix array position $sp, sp + 1, \ldots, L - 1, R + 1, R + 2, \ldots, ep$, and find out their corresponding document d in $O(t_{\mathsf{SA}})$ time per position (using $d = \mathit{rank}_1(B, \mathsf{SA}[i])$). These documents may potentially be the top-k most frequent, so the next step is to calculate the frequency of each of those documents d, in the range $[sp, ep]$.

To obtain the frequencies efficiently, they consider each first occurrence of a document $d = \mathsf{DA}[l_d]$, for $l_d \in [sp, L - 1]$, and map $\mathsf{SA}[l_d]$ to $\mathsf{SA}_d[sp_d]$ as before. Since the suffixes are in the same relative order in SA and SA_d, they carry out an exponential search for the last value ep_d such that $\mathsf{SA}_d[ep_d]$ still maps within $\mathsf{SA}[sp, ep]$, using $\mathsf{SA}^{-1}[z + \mathsf{SA}_d[ep_d]]$ at each step, with $z = \mathit{select}(B, d)$ as before. The total time to compute sp_d and ep_d is $O(t_{\mathsf{SA}} \log n)$. Finally, they return $ep_d - sp_d + 1$ as the frequency of document d in range $[sp, ep]$. The procedure is analogous for the last occurrence of each distinct document in $r_d \in [R + 1, ep]$, now exponentially searching for sp_d to the left.

This procedure is repeated at most $2g$ times because, by construction of τ, both $[sp, L-1]$ and $[R+1, ep]$ are at most of length g. They find the frequencies of these documents, each costing $O(t_{\mathsf{SA}} \log n)$ time, so the total time taken by this is $O(t_{\mathsf{SA}}\, k \log^{3+\epsilon} n)$.

Once they have this set of frequencies (those calculated and those they got from the ***F-list*** of v), they report the top-k among them using the standard linear-time selection algorithm (in $O(2g+k)$ time), followed by a sorting (in $O(k \log k)$ time).

Summarizing, Hon et al. [2009] data structure requires $2|\mathsf{CSA}| + D \log(n/D) + O(D) + o(n)$ bits and retrieves the k most frequent documents in $O(search(P) + k\ t_{\mathsf{SA}}\ \log^{3+\epsilon} n)$ time, for any $\epsilon > 0$.

3.4. Hybrid Approach for top-k Document Retrieval

In a more theoretical work, Gagie et al. [2013] pointed out that Hon et al.'s sparsification technique can run on any other data structure able to (1) tell which document corresponds to a given value of the suffix array, $\mathsf{SA}[i]$, and (2) count how many times the same document appears in any interval $\mathsf{SA}[sp, ep]$.

A structure that is suitable for this task is the document array $\mathsf{DA}[1, n]$, however there is no efficient method for task (2). A natural alternative is the wavelet tree representation of DA. It uses $n \log D + o(n \log D)$ bits and not only computes any $\mathsf{DA}[i]$ in $O(\log D)$ time, but it can also compute operation $rank_d(\mathsf{DA}, j)$ in the same time. This solves operation (2) as $rank_{\mathsf{DA}[i]}(\mathsf{DA}, ep) - rank_{\mathsf{DA}[i]}(\mathsf{DA}, sp-1)$. With the obvious disadvantage of the considerable extra space to represent DA, this solution improves the time of Hon et al. [2009], replacing $t_{\mathsf{SA}} \log n$ by $\log D$ in the query time. Gagie et al. show many other combinations that solve (1) and (2). One of the fastest uses Golynski et al. [2006] representation on DA, which within the same $n \log D + o(n \log D)$ bits reduces $t_{\mathsf{SA}} \log n$ to $\log \log D$ in the time of Hon et al.'s solution. Very recently, Hon et al. [2012] presented new combinations in the line of Gagie et al., using also faster CSAs. Their least space-consuming variant requires $n \log D + o(n \log D)$ bits of extra space on top of the CSA of T, and improves the time to $O(search(P) + k(\log k + (\log \log n)^{2+\epsilon}))$. Although promising, this structure has not yet been implemented.

3.5. Monotone Minimal Perfect Hash Functions

Belazzougui et al. [2012] recently presented a new approach to document retrieval based on monotone minimal perfect hash functions (mmphf) [Belazzougui et al. 2009a; 2009b]. Instead of using individual CSAs or the document array DA to compute frequencies, they use a weaker data structure that takes $O(n \log \log D)$ bits of space.

A mmphf over a bitmap $B[1, n]$ is a data structure able to answer $rank_1$ queries over B, but only for those positions i where $B[i] = 1$. Whenever $B[j] = 0$ the result of $rank_1(B, j)$ is an arbitrary value. This means that the mmphf is unable to tell whether $B[i] = 1$ or 0. As it cannot reconstruct B, the mmphf can be represented within less space than the lower bounds stated in Section 2.2. One variant of mmphfs is able to answer the limited *rank* query in constant time and requires $O(m \log \log(n/m))$ bits, where m is the number of bits set in B. Another variant uses $O(m \log \log \log(n/m))$ bits, and the query time increases to $O(\log \log(n/m))$ [Belazzougui et al. 2009a].

Belazzougui et al.'s approach for document listing with frequencies is similar to the method of Sadakane [2007] (Section 3.3). They use the CSA of the whole collection and the RMQ structures over arrays C and C', but instead of the individual CSAs, they make use of mmphfs. For each document d they mark in a mmphf f_d the positions i such that $\mathsf{DA}[i] = d$ (i.e., $f_d(i) = rank_d(\mathsf{DA}, i)$ if $\mathsf{DA}[i] = d$). Let n_d be the frequency of document d in DA (which is actually the length of document d), then this structure occupies $\sum_d O(n_d \log \log(n/n_d)) = O(n \log \log D)$ bits.

In order to answer the queries, they proceed in the same way of Sadakane [2007], first computing the interval $\mathsf{SA}[sp, ep]$ using the global CSA, and then using the RMQ structures to find the leftmost and rightmost occurrences of each different document in $\mathsf{DA}[sp, ep]$. Then, the frequency of each document d with leftmost and rightmost

Table II. Collections

Collection	Documents (D)	Characters (n)	Avg. doc. length (n/D)	$\|\mathsf{CSA}\|/\log\sigma$
ClueChin	23	2,294,691	99,769	5.34/7.99 = 0.68
ClueWiki	3,334	137,622,191	41,298	4.74/6.98 = 0.68
KGS	18,838	26,351,161	1,399	4.48/6.93 = 0.65
Proteins	143,244	59,103,058	413	6.02/6.57 = 0.92

occurrences l_d and r_d is computed as $TF_{P,d} = f_d(r_d) - f_d(l_d) + 1$ in constant time. This solution yields $|\mathsf{CSA}| + O(n \log\log D)$ bits of space and $O(search(P) + n_{\mathsf{doc}} t_{\mathsf{SA}})$ time for document listing with frequencies. Their other variant uses $O(n \log\log\log D)$ bits and carries out the last computation in $O(\log\log D)$ time, yielding $|\mathsf{CSA}| + O(n \log\log\log D)$ bits of total space and $O(search(P) + n_{\mathsf{doc}}(t_{\mathsf{SA}} + \log\log D))$ overall time. This solution is practical and has been implemented.

Belazzougui et al. also offer a solution for top-k document retrieval. Gagie et al. [2013] (Section 3.4) showed that Hon et al.'s solution may run on top of other data structures, but mmphfs are not able to compute arbitrary frequencies. Their solution enriches the τ_k trees of Hon et al. so that they store enough information to answer the top-k queries using mmphfs instead of the individuals CSAs. Their main result is a data structure that requires $|\mathsf{CSA}| + O(n \log\log\log D)$ bits and answers top-k queries in time $O(search(P) + t_{\mathsf{SA}} k \log k \log^{1+\epsilon} n)$ for any $\epsilon > 0$. They also make several other improvements over previous work, yet these are mostly theoretical.

3.6. Experimental Setup

We conclude this section by presenting the experimental setup that is going to be used for the rest of the article. We will test the different techniques over four collections of different nature, such as English, Chinese, biological, and symbolic sequences.

`ClueChin`. A sample of ClueWeb09 containing a collection of Chinese Web pages (`http://boston.lti.cs.cmu.edu/Data/clueweb09`).
`ClueWiki`. A sample of ClueWeb09 formed by Web pages extracted from the English Wikipedia. It is seen as a sequence of characters, ignoring word structure.
`KGS`. A collection of sgf-formatted Go game records from year 2009 (`http://www.u-go.net/gamerecords`).
`Proteins`. A collection formed by sequences of Human and Mouse proteins (`http://www.ebi.ac.uk/swissprot`).

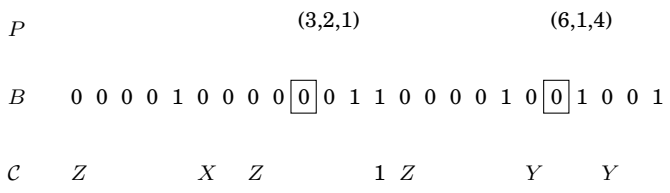
Table II lists the main properties of each collection. To give an idea of the compressibility ratio, we show the bits per cell (bpc) usage of their global CSA divided by $\log\sigma$.

For the implementation of the basic compact data structures, like bitmaps, wavelet trees, etc. we resort to library *libcds*, available at `http://libcds.recoded.cl`.

For grammar compression, we use our RePair implementation (`http://www.dcc.uchile.cl/gnavarro/software`), in particular the "balanced" variant. This variant implements a heuristic that does not guarantee balancedness, but always produced grammars of small height in our experiments.

For the implementation of RMQ data structures we resort to library *sdsl*, available at `http://www.uni-ulm.de/in/theo/research/sdsl`. As the structure is fast but its space is far from the optimal $2n$ bits, and it is used only by previous work, we conservatively assume it uses $2n$ bits, to account for possible future improvements.

We chose a very competitive CSA implementation, namely the *WT-FM-Index* available at *PizzaChili* (`http://pizzachili.dcc.uchile.cl/indexes/SSA`). The space and time of the global CSA is ignored in the experiments, as it is the same for all the solutions, but our choice is relevant for the CSA_d structures, where the chosen CSA poses a low constant-space overhead.



rule	string	ℓ	r
$X \to 00$	00	2	0
$Y \to X1$	001	3	1
$Z \to XY$	00001	5	1

Fig. 5. An example of our RePair representation of bitmaps with sampling period $s = 10$.

Our tests were run on an 4-core 8-processors Intel Xeon, 2GHz each, with 16GB RAM and 2MB cache. The operating system is Ubuntu $10.04.4$, kernel version $2.6.32-39$. Our code was compiled using GNU g++ version $4.4.3$ with optimization `-O3`.

We note that preliminary versions of our techniques have already been compared to the mmphf based techniques for document listing with frequencies [Belazzougui et al. 2012]. Their results show that even those early versions dominate the mmphf based solutions in space and time, except on collection `Proteins`, which is incompressible and thus our techniques do not apply. To avoid repeating experiments we omit comparing that solution in this paper, as the conclusion does not change with our improved variants: The mmphf based solutions are better in the worst case, but worse on compressible collections.

4. GRAMMAR COMPRESSION OF BITMAPS

Our first contribution is a compressed representation of bitmaps with support for *rank* and *select* queries, that takes advantage of repetitiveness. There are many representations of bitmaps with support for *rank* and *select* queries [Jacobson 1989; Munro 1996; Clark 1998]. Some of those use less space when the bitmap has few 1s [Okanohara and Sadakane 2007], and some when the bitmap has few or many 1s, or they are unevenly distributed [Raman et al. 2007]. However, none of them uses less space when the bitmap is repetitive. Although our motivation is to compress the document array in Section 5, this type of compression might also have independent interest.

We will use RePair compression (Section 2.4) to factor out the repetitions in the bitmap, thus our goal is to achieve efficient support for *rank* and *select* queries on a RePair compressed sequence.

4.1. RePair Compressed Bitmaps

Let us consider a bitmap $B[1, n]$, and its RePair representation: the compressed sequence $\mathcal{C}$ (containing both terminal and nonterminal symbols) and its dictionary $\mathcal{R}$. Let us consider a variant that generates a balanced grammar [Maruyama et al. 2012], of height $O(\log n)$. We define two functions, namely ℓ and r, to compute the length and the number of 1s of the string of terminals obtained when we expand a nonterminal. Let $\ell(c) = 1$ for terminals c, and for nonterminals let $\ell(Z) = \ell(X) + \ell(Y)$ if $Z \to XY \in \mathcal{R}$. Likewise, let $r(1) = 1$, $r(0) = 0$ and for non-terminals $Z \to XY \in \mathcal{R}$ let $r(Z) = r(X) + r(Y)$. For each nonterminal X added to the dictionary we store also the length $\ell(X)$ and the number of 1s, $r(X)$, of the string of terminals it expands to.

Additionally, we sample B at regular intervals s. Let $L(i) = 1 + \sum_{j=1}^{i-1} \ell(\mathcal{C}[j])$ be the starting position in B of the symbol $\mathcal{C}[i]$ when expanded. For each position $B[i \cdot s]$ we store $P[i] = (p, o, r)$, where p is the position in $\mathcal{C}$ of the symbol whose expansion will contain $B[i \cdot s]$, that is, $p = \max\{j,\ L(j) \leq i \cdot s\}$. The second component is the offset within that symbol, $o = i \cdot s - L(p)$, and the third is the rank up to that symbol, $r = rank_1(B, L(p) - 1)$. Figure 5 shows an example.

```
Rank(i)

j ← ⌊i/s⌋;
r ← P[j].r;
p' ← P[j].p;
l ← i − P[j].o;
while p' ≤ |C| do
    Z ← C[p'];
    if l + l(Z) < i then
        p' ← p' + 1;
        r ← r + r(Z);
        l ← l + l(Z);
    else
        return Expand(Z, l, r)
```

```
Expand(Z, l, r)

if l = i then
    return r
XY ← R[Z];
if l + l(X) ≥ i then
    return Expand(X);
else
    r ← r + r(X);
    l ← l + l(X);
    return Expand(Y);
```

Algorithm 1: Computing $rank_1(B, i) = Rank(i)$ with our grammar compressed bitmap. $Expand$ completes the computation within the nonterminal Z.

4.2. Solving Rank and Select queries

To answer $rank_1(B, i)$, we compute $i' = \lfloor i/s \rfloor$ and $P[i'] = (p, o, r)$. We then start from $\mathcal{C}[p]$ with position $l = L(p) = i - o$ and rank r. From position p we advance in $\mathcal{C}$ as long as $l \leq i$. Each symbol of $\mathcal{C}$ can be processed in constant time while l and r are updated, since we know $\ell(Z)$ and $r(Z)$ for any symbol $Z = \mathcal{C}[p']$ Finally we arrive at a position $p' \geq p$ so that $l = L(p') \leq i < L(p' + 1) = l + \ell(\mathcal{C}[p'])$. At this point we complete our computation by recursively expanding $\mathcal{C}[p'] = Z$. Let $Z \to XY \in \mathcal{R}$, then if $l + \ell(X) \leq i$ we expand X; otherwise we increase l by $\ell(X)$, r by $r(X)$, and expand Y. See Algorithm 1. As the grammar is balanced, the total time is $O(s + \log n)$.

For $select_1(B, j)$ we can obtain the same complexity. We first perform a binary search over the r values of P to find the interval that contains j. That is, we look for position t in P such as $r < j \leq r'$, where $P[t] = (p, o, r)$ and $P[t + 1| = (p', o', r')$. Then we sequentially traverse the block until exceeding the desired number of 1s, and finally expanding the last accessed symbol of $\mathcal{C}$.

4.3. Space Requirement

Let $R = |\mathcal{R}|$ be the number of rules in the grammar and $C = |\mathcal{C}|$ the length of the final array. Then a simple RePair compressor would require $(2R + C) \log R$ bits. Our representation requires $O(R \log n + C \log R + (n/s) \log n)$ bits, and the time for the operations is $O(s + \log n)$. The minimum interesting value for s is $\log n$, where the space is $O((R + C) \log n + n)$ bits and the time per operation is $O(\log n)$. We can reduce the $O(n)$ extra space to $o(n)$ by increasing s, which makes query times superlogarithmic.

However, we must remember that the original RePair algorithm stops when no pair appears twice, so that with each new rule there is a gain of (at least) one integer. This is true because for each non terminal the original algorithm stores two values (the left and the right side of the rule). On the other hand, our algorithm stores four values for each nonterminal, which leads us to stop replacements at an earlier stage.

4.4. In Practice

There are several ways to represent the dictionary $\mathcal{R}$ in compressed form. We choose one [González and Navarro 2007] that allows for random access to the rules. It regards $\mathcal{R}$ as a directed acyclic graph (DAG), where internal nodes are nonterminals and leaves are terminals, and each nonterminal points to the two nodes it rewrites as. The repre-

sentation corresponds to a DFS traversal of the DAG, so that the first time a node is found, it is expanded, but the next times we just write a reference to the place where it was first found. The representation consists of a sequence S_R and a bitmap S_B. A node is identified as a position in S_B, where a 1 denotes an internal node that is expanded. The two children of $S_B[i] = 1$ are written next to i, thus we obtain all the subtree by traversing $S_B[i \ldots]$ until we have seen more 0s than 1s. The 0s in S_B are either terminals or nonterminals that are found not for the first time. Those nonterminals are represented as positions in S_B, which must then be recursively expanded at decoding time. The identities associated to the positions $S_B[i] = 0$ are written in S_R, at the positions $S_R[rank_0(S_B, i)]$. This DAG representation takes, in good cases, as little as 50% of the space required by a plain array representation of $\mathcal{R}$ [González and Navarro 2007].

To reduce the $O(R \log n)$ space required to store ℓ and r, we will store $\ell(Z)$ and $r(Z)$ only for certain sampled nonterminals Z. When we need to calculate $\ell(Z')$ or $r(Z')$ for a nonterminal that is not sampled we simply expand it recursively until finding a sampled nonterminal (or a terminal). We studied two sampling policies:

Max Depth (MD). Given a parameter δ, we guarantee that no nonterminal in $\mathcal{C}$ will require expanding at depth more than δ to determine its length and number of 1s. That is, we expand each $\mathcal{C}[i]$ until depth δ or until reaching an already sampled nonterminal. Those nonterminals at depth δ are then sampled. We set up a bitmap $B_\delta[1, R]$ where each sampled nonterminal has a 1, and store $\ell(Z)$ and $r(Z)$ of marked nonterminal Z at an array $E[rank_1(B_\delta, Z)]$.

Short Terminals (ST). We fix a maximum number of bits m_l that we are going to spend for storing each nonterminal length and number of 1s. For every nonterminal whose length and number of 1s are in the interval $[0, 2^{m_l} - 2]$ we store them, and for those with greater values we use $2^{m_l} - 1$ as an escape value. With this heuristic we decide beforehand the exact extra space used to store the $r(Z)$ and $\ell(Z)$ values. To answer the queries we expand those nonterminals that are not sampled until we reach one that is sampled.

4.5. Experimental Results

We now analyze the behavior of our grammar compression technique over random bitmaps of various densities, that is, fraction of 1s (repetitive bitmaps will be considered in Section 5). For this sake we generate $1,000$ random and uniformly distributed bitmaps of length $n = 10^8$ with densities ranging from 1% to 15%. We compare our compression ratio with other compressed representations supporting *rank* and *select*, namely ***RRR*** [Raman et al. 2007], and ***SDArray*** [Okanohara and Sadakane 2007]. In this first experiment we intend to compare the compression techniques, independently of how much extra space we can use to speed up *rank* and *select queries*. Therefore, for now we do not use any extra space for the samplings. In particular, we store no $\ell(\cdot)$ nor $r(\cdot)$ data in our technique, and set $s = \infty$. We also include a theoretical line with the zero-order entropy of the sequence (H_0) as a reference.

Figure 6 shows the results. In these random bitmaps there is no expected repetitiveness, which is what our technique is designed to exploit. However, the space usage of our technique is competitive with the state of the art. The reason is that RePair reaches the high-order entropy [Navarro and Russo 2008], which in this case is the same as the zero-order entropy, just like the other two schemes.

In more detail, ***RRR*** requires $nH_0 + o(n)$ bits, and the $o(n)$ overhead is more significant where the density is smaller, therefore this technique becomes more appealing when the density is not that small. Instead, ***SDArray*** uses $nH_0 + O(\#1s)$ bits, which is closer to the entropy when the density is small. This is exactly the domain where this technique dominates. Finally, even though Navarro and Russo [2008] only gave

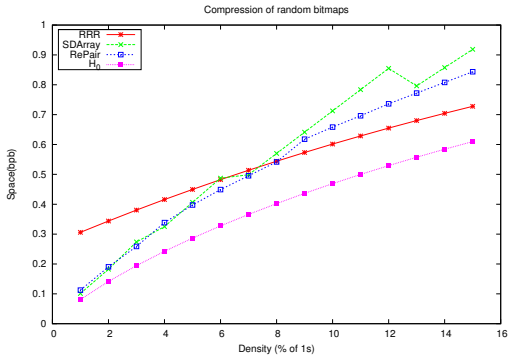


Fig. 6. Space usage, in bits per bit (bpb), of *RRR*, *SDArray* and RePair compressors for bitmaps with different densities. We disregard the space required for the samplings, as these can be decreased at will.

a bound of $2nH_k$ for the space usage of RePair, we can see in practice that RePair's overhead over the entropy is comparable with the overhead of the other techniques.

We now consider time to answer queries. Our RePair techniques can improve the time either by decreasing the sampling period (s), or by using more space to store the lengths and number of 1s of the rules. Because the sampling period s is orthogonal to our two approaches MD and ST (Section 4.4), we fixed different s values $(1024, 256, 128, 64)$ and for each such value we changed the parameters of techniques MD and ST. For MD, δ value was set to 0,1, 2,4,8, ..., h, where h is the height of the grammar. For ST, m_l was set to 0,2,4,6, ..., b, where b is the number of bits required to store the longest rule. For both techniques we plot the extreme scenarios where no rule is sampled, and where we sample all the rules. Figure 7 shows the different space-time trade-offs we obtained to answer *access* (i.e., compute $B[i]$), *rank* and *select* queries. We also include *RRR* (with sampling values 32, 64, and 128) and *SDArray*.

We note that whether or not a technique will dominate the other depends on how much space we are willing to spend on the samples. If we aim for the least space usage, then Max Depth offers the best time performance. If, instead, we are willing to spend more than the minimum space, then ST offers the best times for the same space. Note also that, in all cases, it is more effective to spend as much space as possible in sampling the rules rather than increasing the sampling parameter s. In particular, it is interesting to use ST and sample all the rules, because in this case the bitmap B_δ does not need to be stored, and the *rank* operation on it is also avoided.

We also note that the times for answering *access* and *rank* queries are fairly similar. This is because the number of expansions needed for computing a *rank* query is exactly the same to compute *access*, but during a *rank* computation we also keep count of the number of 1s. Operation *select* is only slightly slower. Note that, in any case, structures *RRR* and *SDArray* are one order of magnitude faster than our RePair compressed bitmaps. In the next sections we will consider this handicap and use RePair only when it impacts space the most and impacts time the least.

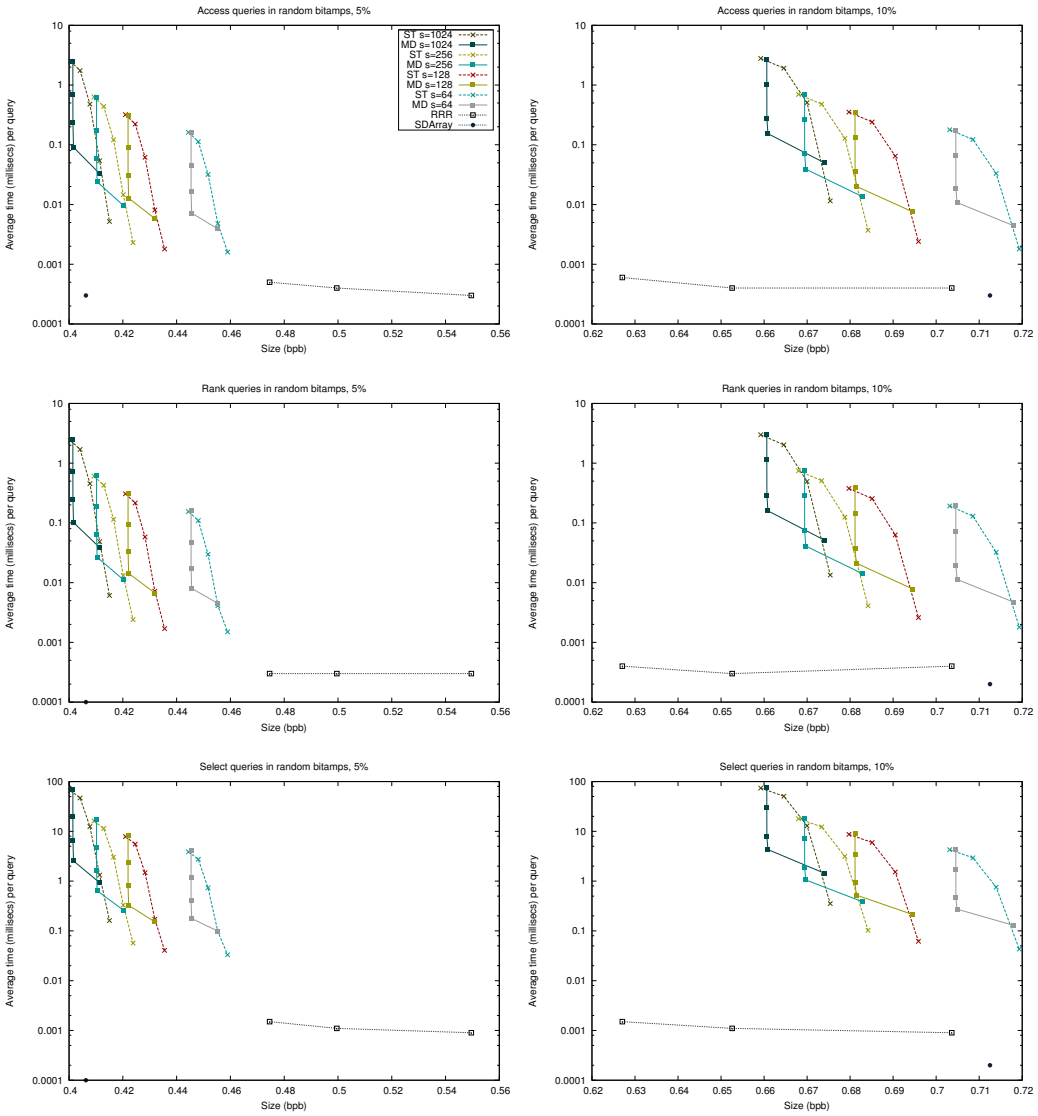


Fig. 7. Tradeoffs obtained for our different techniques to solve *access*, *rank* and *select* queries over random bitmaps with density 5% and 10%.

5. GRAMMAR COMPRESSION OF WAVELET TREES AND DOCUMENT ARRAYS

In this section we introduce new compressed representations of repetitive sequences with support for *rank* and *select* queries, and with applications to document retrieval. In particular, we use this data structure to represent the document array (Section 3.1).

Various studies have shown the practicality for document retrieval of representing the document array with a wavelet tree [Välimäki and Mäkinen 2007; Gagie et al. 2009; Culpepper et al. 2010]. The major drawback of these approaches is the space usage of the document array, which the wavelet tree does not help reduce. The existing compressed representations of wavelet trees reach the zero-order entropy of the sequence represented, or at best adapt to the distribution or runs of symbols [Navarro 2012]. In the case of the document array this yields little compression, because its zero-order entropy corresponds to the distribution of document lengths, and there are no known particularities in the symbol distribution.

5.1. RePair-Compressed Document Arrays

Gagie et al. [2013] showed that using wavelet trees to represent the document array is not strictly necessary: a representation that supports *access* and *rank* on the sequence is sufficient. This opens the door to using compressed sequence representations that exploit some specific regularities of document arrays. They described one such regularity (and the only known up to date).

A *quasi-repetition* in the suffix array $\mathsf{SA}[1,n]$ of T is an area $\mathsf{SA}[i..i+\ell]$ such that there is another area $\mathsf{SA}[i'..i'+\ell]$ such that $\mathsf{SA}[i+k] = \mathsf{SA}[i'+k]+1$ for $0 \le k \le \ell$. Let τ be the minimum number of quasi-repetitions in which SA can be partitioned. It is known that $\tau \le nH_k(T) + \sigma^k$ for any k [Mäkinen and Navarro 2005] (the upper bound is useful only when $H_k(T) < 1$).

González and Navarro [2007] proved that, if one differentially encodes the suffix array SA (so that the quasi-repetitions on SA become true repetitions on the differential version), and applies RePair compression on the differential version, the resulting grammar has size $R + C = O(\tau \log(n/\tau))$.

Gagie et al. [2013] noted that the document array contains almost the same repetitions of the differential suffix array. If $\mathsf{SA}[i] = \mathsf{SA}[i']+1$, then $\mathsf{DA}[i] = \mathsf{DA}[i']$, except when $\mathsf{SA}[i]$ points to the last symbol of document $\mathsf{DA}[i]$. As this occurs at most D times, they concluded that a RePair compression of DA achieves $R + C = O((\tau + D)\log(n/(\tau + D)))$. The formula suggests that the compressed representation of DA is smaller when the text is more compressible.

To use it for document retrieval, Gagie et al. [2013] proposed to compress $\mathsf{DA}[1,n]$ using RePair and add an extra index of size $o(n \log D)$ that answers *rank* queries on DA given only *access* to any $\mathsf{DA}[i]$ [Grossi et al. 2010, Thm. 5(a)]. While following that idea literally is unlikely to yield a good result in practice (in part because the $o(n \log D)$-bit component is only slightly sublinear and does not decrease as repetitiveness increases), we study a different alternative: we get back to using a wavelet tree for DA, but now using the representation of Section 4 for the node bitmaps. We show next that the repetitiveness of DA translates, at least partly, to the wavelet tree bitmaps. Thus we exploit repetitiveness while retaining the support for *rank* queries on DA.

5.2. RePair-Compressed Wavelet Trees

Given a (repetitive) sequence $S[1,n]$ over alphabet $[1,D]$, we build the wavelet tree of S and represent its bitmaps using the compressed format of Section 4. This yields *access*, *rank* and *select* support on S. We analyze next the resulting time and space.

Consider a RePair representation $(\mathcal{R}, \mathcal{C})$ of S, where the sizes of the components is R and C as before. Now take the top-level bitmap B of the wavelet tree. Bitmap B

can be regarded as the result of mapping the alphabet of S onto two symbols, 0 and 1. Thus, a grammar $(\mathcal{R}', \mathcal{C}')$ where the terminals are mapped accordingly, generates B. Since the number of rules in $\mathcal{R}'$ is still R and that of $\mathcal{C}'$ is C, the representation of B requires $O(R \log n + C \log R + (n/s) \log n)$ bits (this is of course pessimistic; many more repetitions could arise after the mapping).

The bitmaps stored at the left and right children of the root correspond to a partition of S into two subsequences S_1 and S_2. Given the grammar that represents S, we can obtain one that represents S_1, and another for S_2, by removing all the terminals in the right sides that do not belong to the proper subalphabet, and removing rules with right hands of length 0 or 1. Thus, at worst, the left and right side bitmaps can also be represented within $O(R \log n + C \log R)$ bits each, plus $O((n/s) \log n)$ for the whole level. Added over the D wavelet tree nodes, the overall space is no more than D times that of the RePair compression of S. Indeed, no rule of $\mathcal{R}$ can survive into both left and right children, which makes its total space $O(R \log n)$, but sequence $\mathcal{C}$ may keep the same length in both children, adding up to $O(DC \log n)$ in the worst case. The time for the operations, on the other hand, raises to $O((s + \log n) \log D)$.

Although this result is very pessimistic, and cannot distinguish the solution from using D bitmaps $B_c[1, n]$, where $B_c[i] = 1$ iff $S[i] = c$, it points out to the important practical fact that the repetitions exploited by RePair get cut by half as we descend one level of the wavelet tree, so that after descending some levels, no repetition structure can be identified. At this point RePair compression becomes ineffective. On the other hand, since RePair over a bitmap B uses $O(|B|H_0(B))$ bits [Navarro and Russo 2008], the space required for the complete wavelet tree of S is no worse than $O(nH_0(S))$ [Grossi et al. 2003].

5.3. In Practice

As D is likely to be large, we use a wavelet tree design without pointers, that concatenates all the bitmaps of the same wavelet tree level [Claude and Navarro 2008]. We apply the RePair representation from Section 4 to each of those $\log D$ levels. Therefore, we use one set of rules $\mathcal{R}$ per level.

As the repetitions that could be present in S get shorter when we move deeper in the wavelet tree, we evaluate at each level whether our RePair-based compression is actually better than an entropy-compressed representation [Raman et al. 2007] or even a plain one, and choose the one with the smallest space. The experiments in Section 4.5 show that computing *access* and *rank* on RePair-compressed bitmaps is in practice much slower than on alternative representations. Considering this, we use a space-time tradeoff parameter $0 < \alpha \leq 1$, so that we prefer RePair compression only when its size is α times that of the alternatives, or less.

Note that the algorithms that use wavelet trees on DA (Section 3.2) traverse many more nodes at deeper levels. Therefore, we have the fortunate effect that using RePair on the higher levels impacts much on the space, as repetitiveness is still high, and little on the time, as even when operations on RePair-compressed bitmaps are much slower, there are also much fewer operations on those bitmaps.

5.4. Experimental Results

Our experimental results will focus on our intended application, that is, the compression of the document array. Thus we use the collections described in Section 3.6

5.4.1. Compressing the Wavelet Tree. Considering our finding that repetitions degrade on deeper levels of the wavelet tree, we start by analyzing how the different bitmap compression techniques compared in Section 4.5 perform on successive levels of the

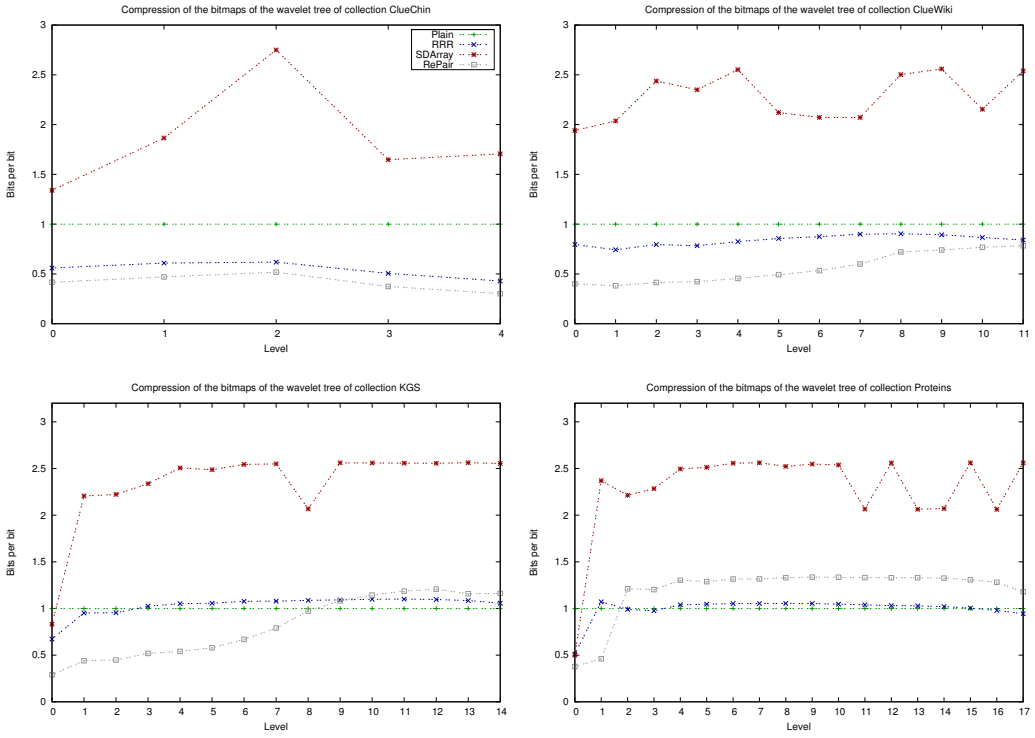


Fig. 8. Bits per bit achieved by the different bitmap compression methods at successive levels of the wavelet tree representation of the document arrays.

wavelet tree of DA. Figure 8 shows the space achieved. Again, we are not including any sampling information.

As expected, RePair dominates on the higher levels and degrades as the repetitions are broken, in the lower levels. Note also that no technique achieves much compression on `Proteins`, with the exception of RePair on the two first levels. This is also expected, since the entropy of this collection is very high (recall Table II). From now on we disregard technique *SDArray*, which does not perform well in this scenario.

5.4.2. Choosing the sampling technique for RePair. In Section 4.5 we compared techniques MD and ST to support *rank* and *select* queries on RePair-compressed bitmaps. We studied their performance over random bitmaps. We now repeat the experiments on the wavelet tree bitmaps, considering only operation *access* (document retrieval algorithms use only *access* and *rank* operations, and both perform similarly in these structures). Figure 9 shows the results on some levels: for each collection we choose the root bitmap and some deeper bitmap where RePair still achieves interesting space.

First, we note that our RePair-compressed bitmaps provide significant space advantages on various wavelet tree levels. Second, as on random bitmaps, technique MD yields the best results when using minimum space. When using more space, ST is faster for the same space.

5.4.3. Wavelet Trees for Document Listing with Frequencies. We now compare our wavelet tree representation of the document array with previous work: plain and statistical encoding of the wavelet tree, and Sadakane [2007] method based on individual CSAs (Section 3.3). As explained, alternative solutions [Sadakane 2007; Hon et al. 2009] for

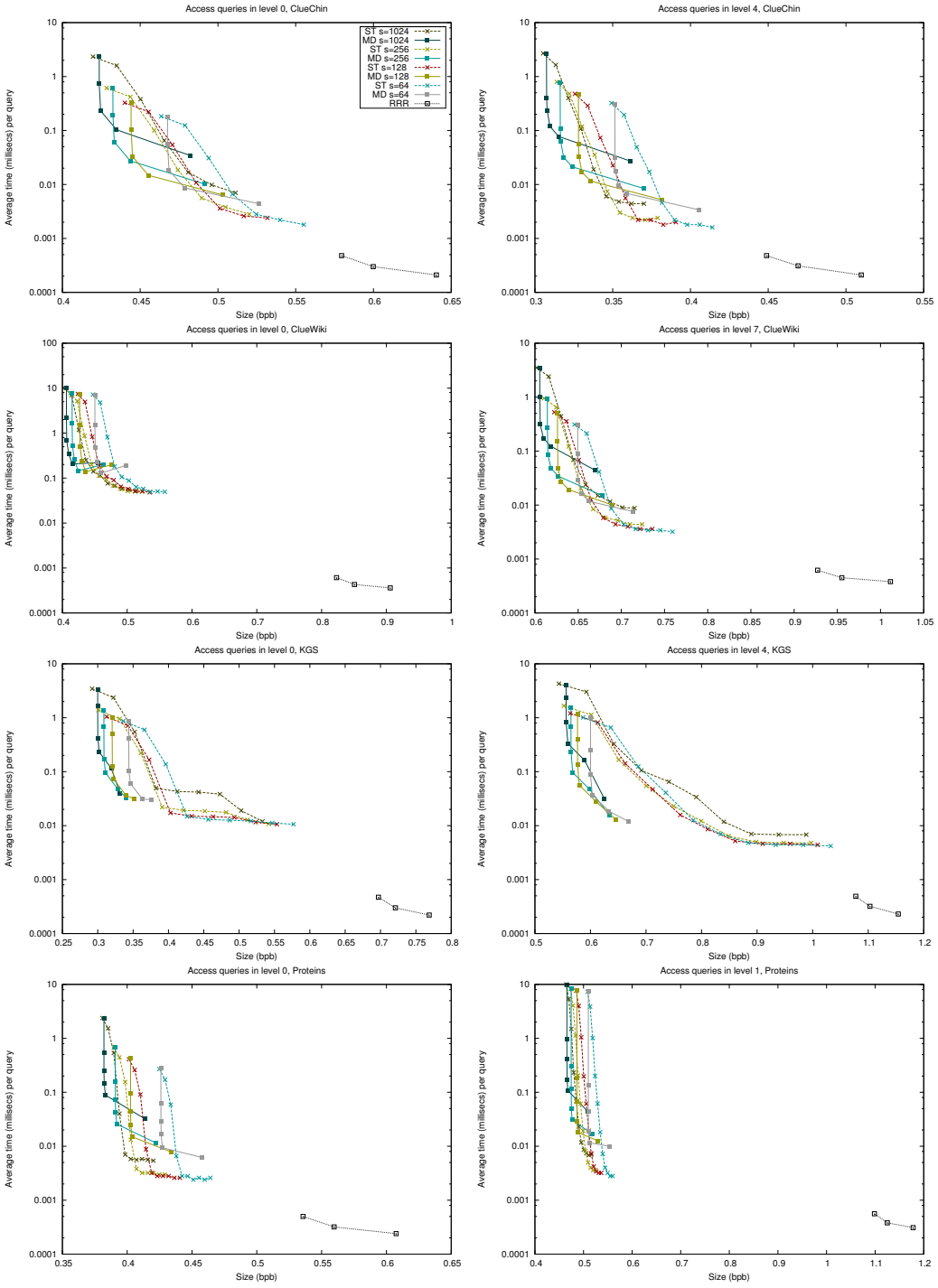


Fig. 9. Tradeoffs obtained for our different techniques to support *access*, over some wavelet tree levels of the document arrays of the collections.

the basic document listing problem are hardly improvable. They require very little extra space and are likely to perform similarly to wavelet trees in time. Therefore, our experiments focus on document listing with frequencies.

As the CSA search for P is common to all the approaches, we do not consider the time for this search nor the space for that global CSA, but only the extra space/time to support document retrieval once $[sp, ep]$ has been determined. We give the space usage in bits per text character (bpc).

Section 3.6 describes the CSA used to implement Sadakane's representation using one CSA per document. We encode the bitmap B using a plain representation because it is the fastest one, as well as an efficient RMQ data structure. Again, Section 3.6 describes those choices in more detail. We recall in particular that we charge only $2n$ bits to the RMQ data structure, to account for possible improvements. Similarly, we charge zero space for B, as it is sparse and could be compressed significantly. All those decisions play in favor of our implementation of Sadakane [2007].

Previous work by Culpepper et al. [2010] has demonstrated that the quantile approach [Gagie et al. 2009] is clearly preferable, in space and time, over previous ones based on wavelet trees [Välimäki and Mäkinen 2007]. They also showed that the quantile approach is in turn superseded by a DFS traversal, which avoids some redundant computations. (Section 3.2). Therefore, we carry out the DFS algorithm over a plain wavelet tree representation (*WT-Plain*), over one where the bitmaps are statistically compressed [Raman et al. 2007] (*WT-RRR*), and over our RePair-compressed ones.

As explained, our grammar compressed wavelet trees offer a space/time tradeoff depending on the α value (recall Section 5.3), which can be the same for all levels, or decreasing for the deeper levels (where one visits more nodes and thus being slower has a higher impact). Another space/time tradeoff is obtained with the sampling parameter s on each bitmap. Additionally, when deciding to increment the sampling values, they can be varied freely on each level, and one can expect that the impact on the query times may be more significant when the effort is done in the lower levels. For those reasons we plot a cloud of points with a number of combinations of α values and sampling parameters in order to determine our best scheme. We chose 10,000 random intervals of the form $[sp, ep]$, for interval sizes $ep - sp + 1$ from 100 to 100,000, and listed the distinct documents in the interval, with their frequencies.

Figure 10 shows the clouds of points obtained for interval sizes of 100 and 10,000. Among all the combinations we plotted, we highlight the points that dominate the space-time map. Among those dominating configurations we have selected four points that we are going to use for our combination called *WT-Alpha* in the following experiments. We will also include the variant using $\alpha = 1$, called *WT-RP*.

Figure 11 compares our variants with Sadakane [2007] (*SADA*) on collections `ClueChin` and `ClueWiki`. Even on `ClueChin`, with just 23 relatively large documents, the space overhead of indexing them separately makes *SADA* impractical (even with the generous assumptions on free bitmaps and cheap RMQs). It is also significantly slower. Considering this result, we have not attempted to implement the variant of Hon et al. [2009], which achieves $|\mathsf{CSA}| + o(n)$ extra space but is even slower. Its reduction in space (by, at best, $4n$ bits), would be insufficient to make it competitive.

Figure 12 compares the techniques on the other collections. To facilitate a visual comparison, we do not plot *SADA* on these, as it is still significantly slower and it requires more than 60 bpc on `KGS` and more than 90 bpc on `Proteins`, well out of bounds.

The results vary depending on the collection, but in general our compressed representation is able to reduce the space of the plain wavelet tree by a wide margin. The compressed size is 40% to 75% of the original wavelet tree size. The exception is `Proteins`, where the text is mostly incompressible and this translates into the incompressibility of the document array.

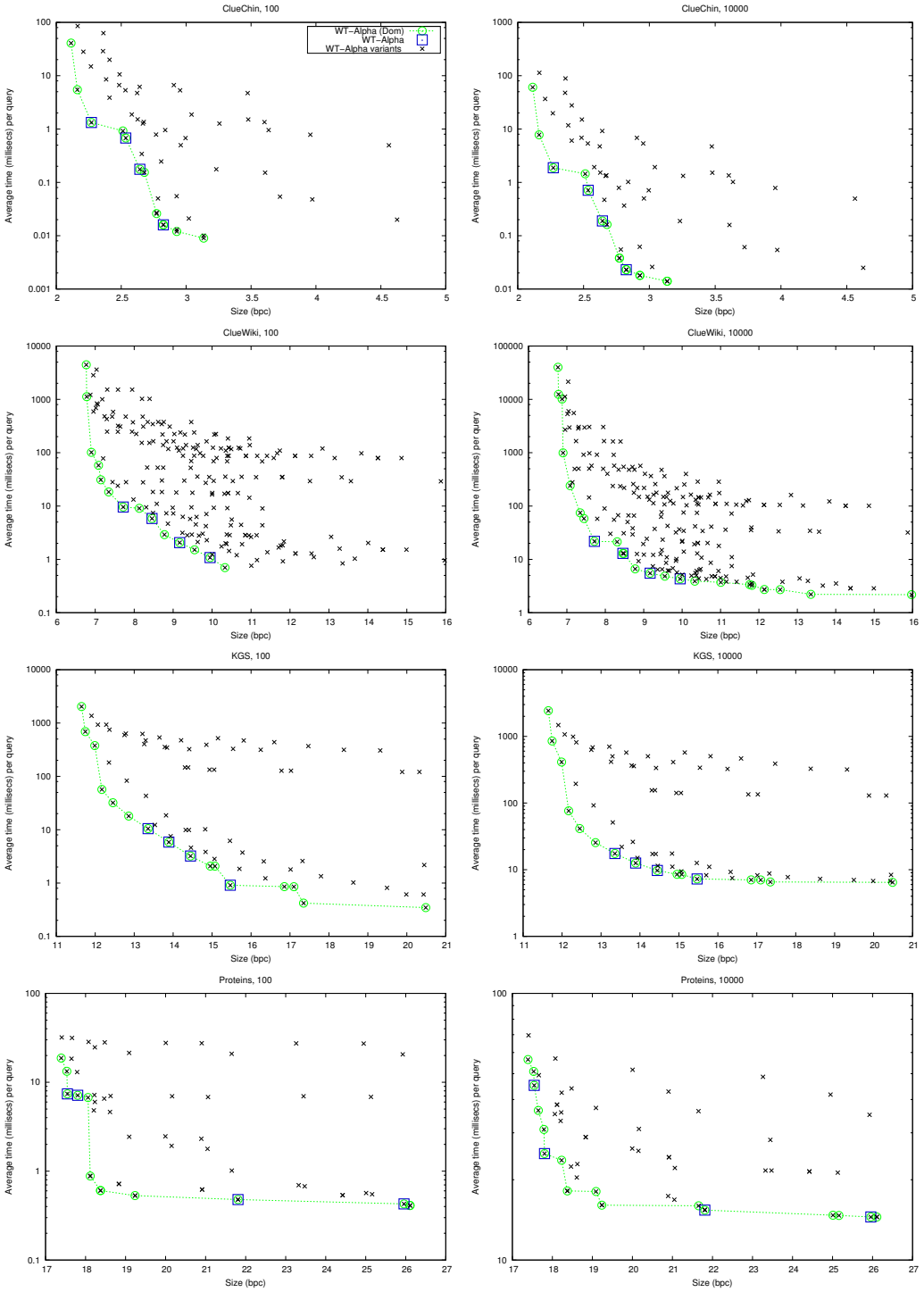


Fig. 10. Clouds of space-time tradeoffs for document listing with our techniques on document array intervals of length 100 and 10,000.

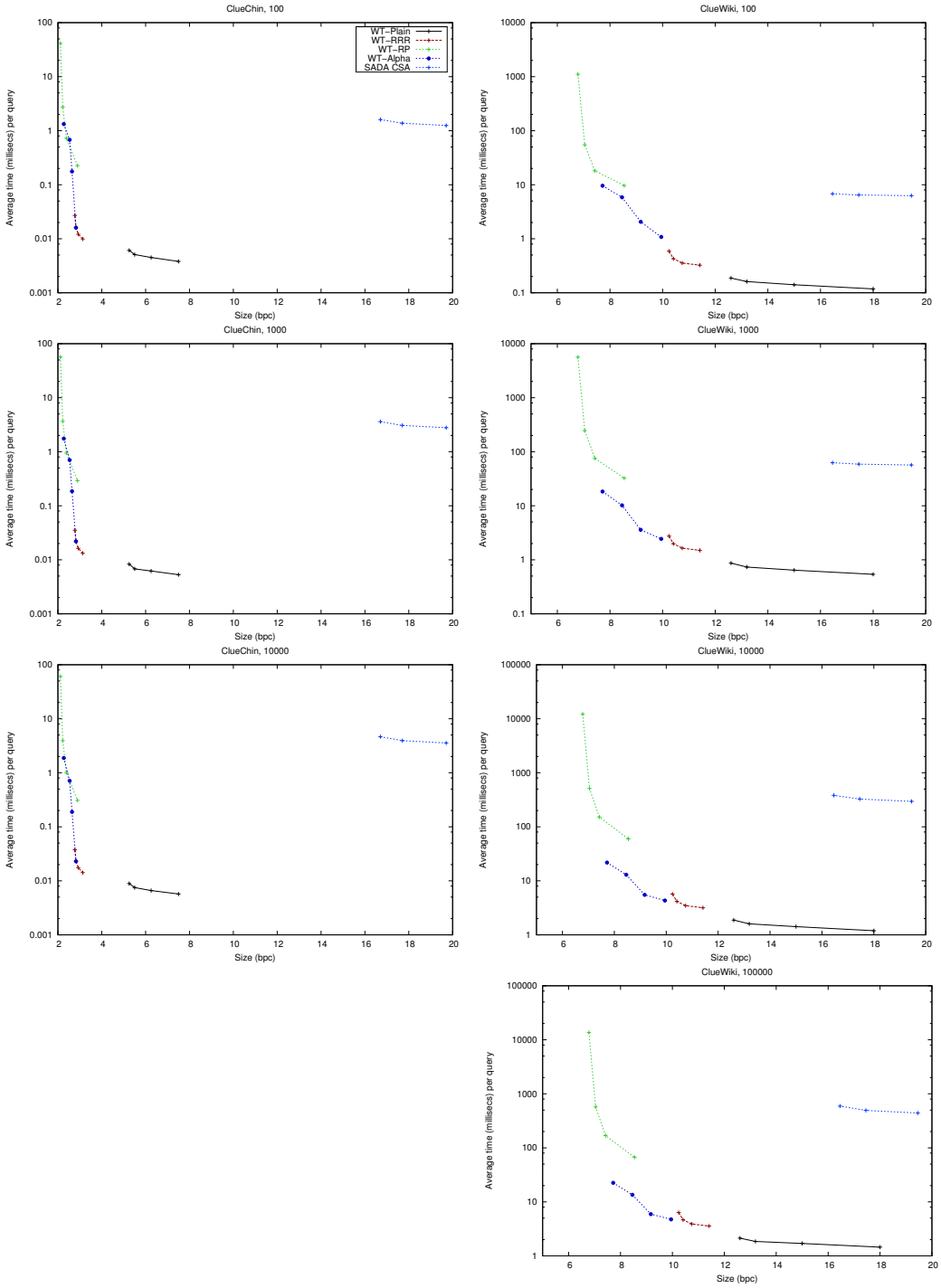


Fig. 11. Experiments on document listing with term frequencies, on ranges of lengths 100 to 100,000 on collection `ClueChin` (left) and `ClueWiki` (right).

While *WT-RP* is significantly slower than *WT-Plain* (up to 20 times slower in the most extreme case), the *WT-Alpha* versions provide useful tradeoffs. They achieve compression ratios of 50% to 80% and significantly reduce time gaps, to 7 times slower in the most extreme case. The answer time over the interval $[sp, ep]$ of length 10,000 is around 10-20 milliseconds. We note that our slowest version is still 10 times faster than *SADA*.

6. SPARSIFIED SUFFIX TREES AND TOP-*K* RETRIEVAL

In the previous section we have shown that the technique of Sadakane [2007], based on individual CSAs to obtain the frequency of individual documents, is not effective in practice, at least using existing CSA implementations. Hon et al. [2009] data structure for top-k document retrieval is built on the top of Sadakane's approach, therefore a straightforward implementation will suffer from the same lack of practicality.

In this section we propose various practical versions of Hon et al.'s $o(n)$-bit data structure for top-k queries, as well as efficient algorithms to use them on top of a wavelet tree representation of the document array instead of the original scheme, which uses individual CSAs. We carried out exhaustive experiments among them to find the best combination.

Our top-k algorithms combine the sparsified technique of Hon et al. [2009] with various of the techniques of Culpepper et al. [2010] to explore the remaining areas of the document array. We can regard the combination as either method boosting the other. Culpepper et al. boost Hon et al.'s method, while retaining its good worst-case complexities, as they find the extra occurrences more cleverly than by enumerating them all. Hon et al. boost plain Culpepper et al.'s method by having precomputed a large part of the range, and thus ensuring that only small intervals have to be handled.

As explained in Section 3.4, Gagie et al. [2013] showed that Hon et al.'s scheme can run on top of any document array representation able to compute *access* and *rank*, and the sequence representation by Golynski et al. [2006] is the fastest practical choice. We have also implemented this scheme and study its practicality.

6.1. Implementing Hon et al.'s Structure

The structure of Hon et al. [2009] is a sparse generalized suffix tree of T (*SGST*; "generalized" means it indexes D strings). It can be seen as a sampling of the suffix tree, with a sampling parameter g, which is sparse enough to make the sampled suffix tree require only $o(n)$ bits, but at the same time it guarantees that, for each possible interval $\mathsf{SA}[sp, ep]$, there exists a node whose corresponding interval spans inside the interval $[sp, ep]$ leaving at most $2g$ uncovered positions. For each sampled node they precompute a list with top-k most frequent documents and their frequencies. To answer top-k queries, they look for P in their sample, and then they compute the frequency of $O(g)$ uncovered documents using a technique inspired in Sadakane [2007].

This subsection focuses on practical implementations of this idea. In the next subsection we present new algorithms for top-k document retrieval.

6.1.1. Sparsified Generalized Suffix Trees (SGST). We call $l_i = \mathsf{SA}[i]$ the i-th suffix tree leaf. Given a value of k we define $g = k \cdot g'$, for a space/time tradeoff parameter g', and sample n/g leaves $l_1, l_{g+1}, l_{2g+1}, \ldots$, instead of sampling $2n/g$ leaves as in the theoretical proposal [Hon et al. 2009]. We mark internal SGST nodes $lca(l_1, l_{g+1}), lca(l_{g+1}, l_{2g+1}), \ldots$ The following lemma shows that this marking is lca-closed.

LEMMA 6.1. *Any* $v = lca(l_{ig+1}, l_{jg+1})$ *is also* $v = lca(l_{rg+1}, l_{(r+1)g+1})$ *for some* r.

PROOF. We use induction on $j - i$. Let us consider nodes $v_1 = lca(l_{ig+1}, l_{(i+1)g+1})$ and $v_2 = lca(l_{(i+1)g+1}, l_{(i+2)g+1})$. Then it holds $v_{12} = lca(v_1, v_2) = lca(l_{ig+1}, l_{(i+2)g+1})$: v_{12}

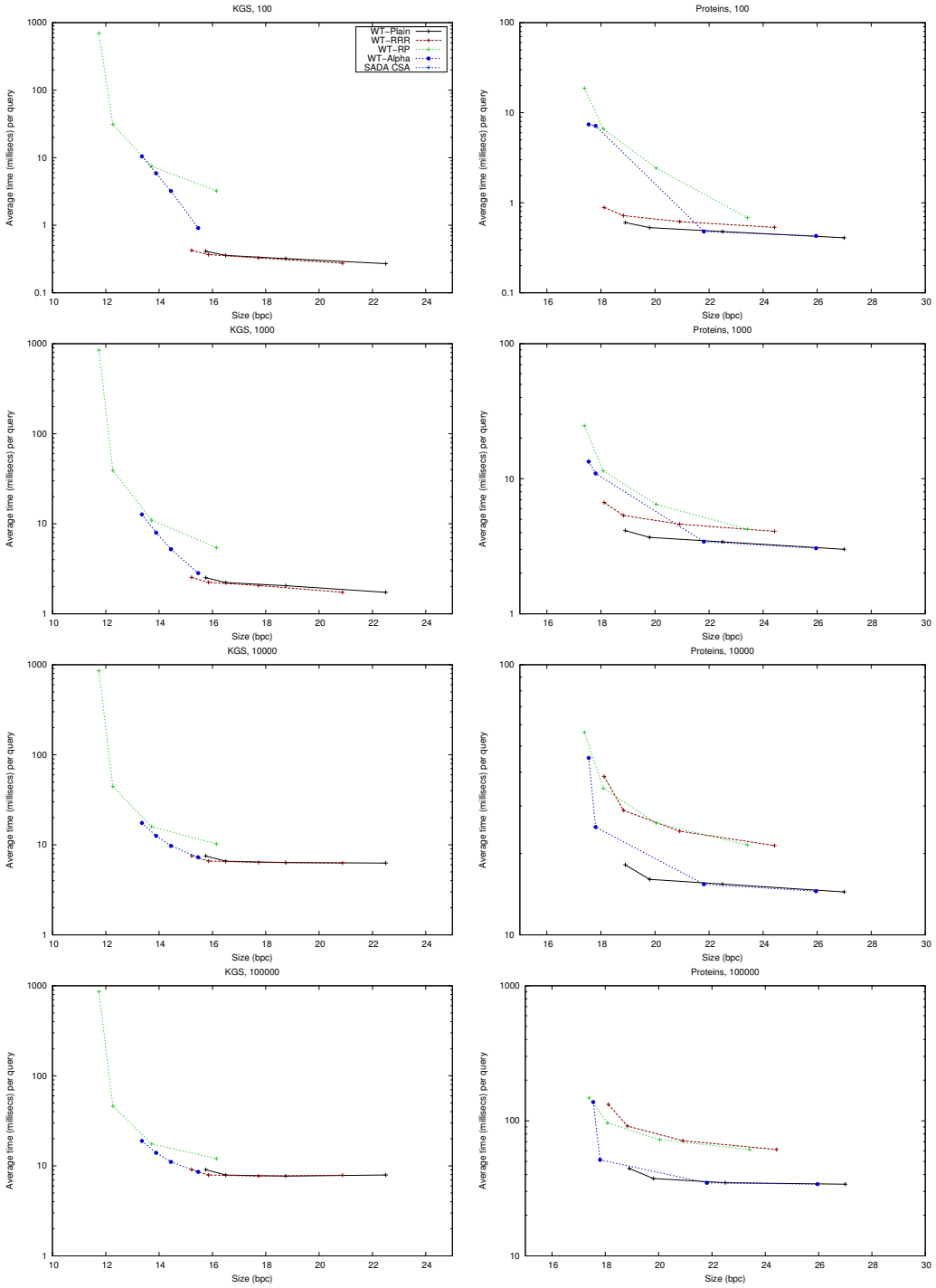


Fig. 12. Experiments on document listing with term frequencies, on ranges of lengths 100 to 100,000 on collection `KGS` (left) and `Proteins` (right).

is ancestor of both extremes, and if a node v^* is ancestor of both extremes, then it is also ancestor of $l_{(i+1)g+1}$, thus it is also ancestor of v_1 and v_2 and hence of v_{12}. Now, because both v_1 and v_2 are ancestors of $l_{(i+1)g}$, it follows that v_{12} is either v_1 or v_2. In general, consider $v_1 = lca(l_{ig+1}, l_{(j-1)g+1})$ and $v_2 = lca(l_{(j-1)g+1}, l_{jg+1})$. Then, similarily, $v = lca(v_1, v_2) = lca(l_{ig+1,jg+1})$ is either v_1 or v_2 as both are ancestors of $l_{(j-1)g+1}$. By the inductive hypothesis, v_1 is $lca(l_{rg+1}, l_{(r+1)g+1})$ for some $i \leq r \leq j-1$, whereas v_2 is also the lca of two consecutive samples. □

Therefore, those n/g marked nodes are sufficient. The required lca operations can be computed in $O(n/g)$ time [Bender and Farach-Colton 2000].

6.1.2. Redundancy among SGSTs. Gagie et al. [2013] point out that we need only $\log D$ sparsified trees τ_k, not $\log n$. What is less obvious is that there is also a great deal of redundancy among the τ_k trees, since the nodes of τ_{2k} are included in those of τ_k, and the $2k$ candidates stored in the nodes of τ_{2k} contain those in the corresponding nodes of τ_k. To factor out some of this redundancy we store only one tree τ, whose nodes are the same of τ_1, and record the *class* $c(v)$ of each node $v \in \tau$. This is $c(v) = \max\{k,\ v \in \tau_k\}$, and can be stored in $\log\log D$ bits. Each node $v \in \tau$ stores the top-$c(v)$ candidates corresponding to its interval, using $c(v)\log D$ bits, and their frequencies, using $c(v)\log n$ bits. All the canidates and frequencies of all the nodes are stored in a unique table, to which each node v stores a pointer. Each node v also stores its interval $[sp_v, ep_v]$, using $2\log n$ bits. Note that the class does not necessarily decrease monotonically in a root-to-leaf path of τ, thus we store the topologies of all the τ_k trees independently, to allow for their efficient traversal, for $k > 1$. Apart from topology information, each node of such τ_k trees contains just a pointer to the corresponding node in τ, using $\log|\tau|$ bits. All the information on intervals and candidates is thus factored in τ, saving space.

In our basic data structure, the topology of the trees τ and τ_k is represented using pointers of $\log|\tau|$ and $\log|\tau_k|$ bits, respectively. Those pointers are offsets in an array that stores the nodes.

To answer top-k queries, we find the range $\mathsf{SA}[sp, ep]$ using a CSA. Now we use the closest higher power of two of k, $k' = 2^{\lceil \log k \rceil}$. Then we find the locus in the appropriate tree $\tau_{k'}$ top-down, binary searching the intervals $[sp_v, ep_v]$ of the children v of the current node, and extracting those intervals using the pointers to τ. By the properties of the sampling [Hon et al. 2009] it follows that we will traverse, in this descent, nodes $u \in \tau_{k'}$ such that $[sp, ep] \subseteq [sp_u, ep_u]$, until reaching a node v where $[sp_v, ep_v] = [sp', ep'] \subseteq [sp, ep] \subseteq [sp' - g, ep' + g]$ (or reaching a leaf $u \in \tau_k$ such that $[sp, ep] \subseteq [sp_u, ep_u]$, in which case $ep - sp + 1 < 2g$). This v is the locus of P in $\tau_{k'}$, and we find it in time $O(|P|\log\sigma)$.

In practice, we can further reduce the space in exchange for possibly higher times. For example, the sequence of all precomputed top-k candidates can be Huffman-compressed, as there is much repetition in the sets, and values $[sp_v, ep_v]$ can be stored as $[sp_v, ep_v - sp_v]$, using DACs for the second components [Brisaboa et al. 2013], as many such differences will be small. Finally, as Gagie et al. [2013] pointed out, a major space reduction can be achieved by storing only the identifiers of the candidates, as their frequencies can be computed on the fly using *rank* on the wavelet tree of DA. All these variants are analyzed in Section 6.3.

6.1.3. Compressing Tree Topologies. The *SGST* uses sparsification to reach $o(n)$ bits, but the tree topology is still stored explicitly. More precisely, they store $O(n/\log^2 n)$ nodes, with their respective *F-list*, sp_v, ep_v values and pointers to the children.

We will study a variant called *Succinct SGST (SSGST)*, which uses a pointerless representation of the tree topologies. Although the tree operations are slightly slower than on a pointer-based representation, this slowdown occurs on a not too significant

part of the search process, and a succinct representation allows one to spend more space on structures with higher impact (e.g., reducing the sampling parameter g).

Arroyuelo et al. [2010] showed that, for the functionality it provides, the most promising succinct representation of trees is the so-called Level-Order Unary Degree Sequence (LOUDS) [Jacobson 1989]. As mentioned in Section 2.5, the theoretical space requirement of $2n+o(n)$ bits of space can be in practice as low as $2.1\,n$ bits to represent a tree of n nodes. LOUDS solves many operations in constant time (less than a microsecond in practice). In particular it allows fast navigation through labeled children.

We resort to their labeled tree implementation [Arroyuelo et al. 2010]. We encode the values sp_v and ep_v, pointers to τ (in τ_k), and pointers to the candidates in τ in a separate array, indexed by the LOUDS rank of the node v, managing them just as Arroyuelo et al. manage labels.

6.2. New Top-k Algorithms

Once the search for the locus of P is done, Hon et al.'s algorithm requires a brute-force scan of the uncovered leaves to obtain their frequencies (using individuals CSAs). When Gagie et al. [2013] showed that Hon et al.'s *SGST* may run on top of different structures, they also keep using this brute-force scanning. Instead, we run a combination of the algorithm by Hon et al. [2009] and those of Culpepper et al. [2010], over a wavelet tree representation of the document array $\mathsf{DA}[1,n]$. Culpepper et al. introduce, among others, a document listing method (DFS) and a Greedy top-k heuristic (recall Section 3.2). We adapt these to our particular top-k subproblem.

If the search for the locus of P ends at a leaf u that still contains the interval $[sp,ep]$, Hon et al. simply scan $\mathsf{SA}[sp,ep]$ by brute force and accumulate frequencies. We use instead Culpepper et al.'s Greedy algorithm, which is faster than a brute-force scanning.

When, instead, the locus of P is a node v where $[sp_v,ep_v]=[sp',ep']\subseteq[sp,ep]$, we start with the precomputed answer of the $k\leq k'$ most frequent documents in $[sp',ep']$, and update it to consider the subintervals $[sp,sp'-1]$ and $[ep'+1,ep]$. We use the wavelet tree of DA to solve the following problem: Given an interval $\mathsf{DA}[l,r]$, and two subintervals $[l_1,r_1]$ and $[l_2,r_2]$, enumerate all the distinct values in $[l_1,r_1]\cup[l_2,r_2]$ together with their frequencies in $[l,r]$. We propose two solutions, which can be seen as generalizations of heuristics proposed by Culpepper et al. [2010].

6.2.1. Restricted Depth-First Search. Our restricted DFS algorithm begins at the root of the wavelet tree and tracks down the intervals $[l,r]=[sp,ep]$, $[l_1,r_1]=[sp,sp'-1]$, and $[l_2,r_2]=[ep'+1,ep]$. More precisely, we count the number of 0s and 1s in B in ranges $[l_1,r_1]\cup[l_2,r_2]$, as well as in $[l,r]$, using a constant number of *rank* operations on B. If there are any 0s in $[l_1,r_1]\cup[l_2,r_2]$, we map all the intervals onto the left child of the node and proceed recursively from this node. Similarly, if there are any 1s in $[l_1,r_1]\cup[l_2,r_2]$, we continue on the right child of the node. When we reach a wavelet tree leaf we report the corresponding document, and the frequency is the length of the interval $[l,r]$ at the leaf. Figure 13 shows an example where we arrive at the leaves of documents 1, 2, 5 and 7, reporting frequencies 2, 2, 1 and 4, respectively.

When solving the problem in the context of top-k retrieval, we can prune some recursive calls. If, at some node, the size of the local interval $[l,r]$ is smaller than our current kth candidate to the answer, we stop exploring its subtree since it cannot contain competitive documents. In the worst case, the algorithm needs to reach the bottom of the wavelet tree for each distinct document, so the time required to obtain the frequencies is $O(g\log(D/g))$.

6.2.2. Restricted Greedy. Following the idea described by Culpepper et al. [2010], we can not only stop the traversal when $[l,r]$ is too small, but also prioritize the traversal of the nodes by their $[l,r]$ value. This may allow us to stop sooner.

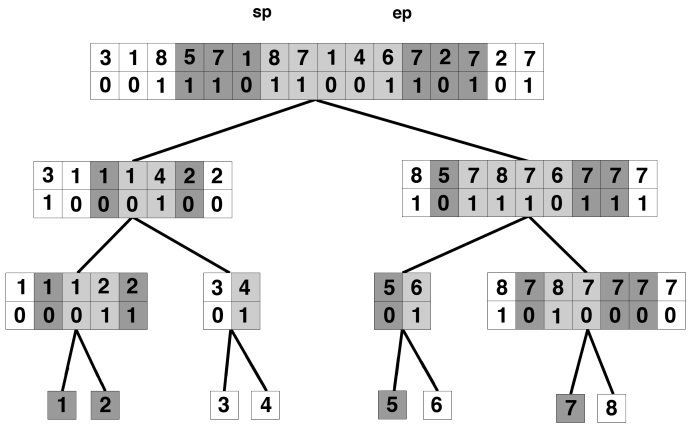


Fig. 13. Restricted DFS to obtain the frequencies of documents not covered by τ_k. Shaded regions show the interval $[sp, ep] = [4, 14]$ mapped to each wavelet tree node. Dark shaded intervals are the projections of the leaves not covered by $[sp', ep'] = [7, 11]$.

We keep a priority queue where we store the wavelet tree nodes yet to process, and their intervals $[l, r]$, $[l_1, r_1]$, and $[l_2, r_2]$. The priority queue begins with one element, the root. Iteratively, we remove the element with highest $r-l+1$ value from the queue. If it is a leaf, we report it. If it is not, we project the intervals into its left and right children, and insert each such children containing nonempty intervals $[l_1, r_1]$ or $[l_2, r_2]$ into the queue. As soon as the $r-l+1$ value of the element we extract from the queue is not larger than the kth frequency known at the moment, we can stop the whole process.

In the worst case this heuristic requires $O(g(\log(D/g) + \log g)) = O(g \log D)$ time.

6.2.3. Heaps for the k Most Frequent Candidates. Our two algorithms solve the query assuming that we can easily find, at any given moment, which is the kth best candidate known up to now. We use a min-heap data structure for this purpose. It is loaded with the top-k precomputed candidates corresponding to the interval $[sp', ep']$ stored in the ***F-List***. At each point, the top of the heap gives the kth known frequency in $O(1)$ time.

Each time the DFS or Greedy algorithms report a new element, we must check whether its frequency is larger than that of the kth current candidate. If it is, the top of the min-heap is replaced with the new frequency and the heap is reordered. If the heap does not yet contain k elements, the new frequency is simply inserted in it. Since the heap is always of size at most k, each reported candidate costs $O(\log k)$, which is dominated by the $O(\log D)$ time incurred by the Greedy method (DFS can be slightly better). There are also steps in both algorithms that do not yield any candidate, but $O(g \log D)$ is still an upper bound for all the costs.

A remaining issue is that we could find again, in our DFS or Greedy traversal, a document that was in the original top-k list, and thus possibly in the heap. This means that the document had been inserted with its frequency in $\mathsf{DA}[sp', ep']$, but since it appears further times in $\mathsf{DA}[sp, ep]$, we must now update its frequency, that is, increase it and restore the min-heap invariant. It is not hard to maintain a hash table with forward and backward pointers to the heap so that we can track the current candidate positions and replace their values. However, it is more practical to let those duplicate values coexist in the heap (which should then be of size $2k$ to avoid losing relevant

candidates) and at the end remove duplicates with lower frequencies. This adds just $O(g + k \log k)$ to the time complexity.

6.3. Experimental Results

We have run exhaustive experiments to determine the best alternatives. First we compare the different algorithms to answer top-k queries. Once we choose the best algorithm, we turn to study the performance of our different data structure variants. Finally, we compare our best choice with the related work. We extracted sets of 10,000 random substrings from each collection, of length 3 and 8, to act as search patterns. The time and space needed to perform the CSA search is orthogonal to all of the methods we present (and also the time is negligible, being at most 20 microseconds per query), thus we only consider the space and time to retrieve the top-k documents.

6.3.1. Evaluation of our Algorithms. We tested the different algorithms to find the top-k answers among the precomputed candidates and uncovered leaves (see Section 6.2):

Greedy. Our modified greedy algorithm.
DFS. Our modified depth-first-search algorithm.
Select. The brute-force selection procedure of the original proposal [Hon et al. 2009].

Because in this case the algorithms are orthogonal to the data structures for the sparsified trees, we run all the algorithms only on top of the straightforward implementation of Hon et al., which we will call *Ptrs*. For all the algorithms we use the best wavelet tree of Section 5, that is, the variant *WT-Alpha*, showing the four chosen points per curve. We consider three sampling steps, $g' = 200$, 400 and 800.

Figures 14 to 16 show the results. As expected, method *Greedy* is almost always better than *Select* (up to 80% better) and never worse than *DFS* (and up to 50% better), which confirms intuition. From here on we will use only the *Greedy* algorithm. Note, however, that if we wanted to compute top-k considering metrics more complicated than term frequency, then *Greedy* would not apply anymore (nor would *DFS*). In such a case we could still use method *Select*, whose times would remain similar.

6.3.2. Evaluation of our Data Structures. In this round of top-k experiments we compare our different implementations of *SSGSTs* (i.e., the trees τ_k, see Section 6.1) over a single implementation of wavelet tree (*WT-Alpha*), and using always method *Greedy*. We test the following variants:

Ptrs. Straightforward implementation of the original proposal [Hon et al. 2009].
LOUDS. Like *Ptrs* but using a LOUDS representation of the tree topologies.
LIGHT. Like *LOUDS* but storing the information of the nodes in a unique tree τ.
XLIGHT. Like *LIGHT* but not storing the frequencies of the top-k candidates.
HUFF. Like *LIGHT* but Huffman-compressing the candidate identifiers and encoding the $[sp_v, ep_v]$ intervals succinctly using DACs.

Figures 17 to 19 show the results. Using *LOUDS* representation instead of *Ptr* had almost no impact on the time, except on `ClueChin`, where all the methods are very fast anyway. This is because the time needed to find the locus is usually negligible compared with that to explore the uncovered leaves. Further costless space gains are obtained with variant *LIGHT*, which reduces the space significantly, especially for small g'. Variant *XLIGHT*, instead, slightly reduces the space of *LIGHT* at a noticeable cost in time that makes it not so interesting, except on `Proteins`. Time is impacted because k additional *rank* operations on the wavelet tree are needed. Space, instead, is reduced but it affects a low-overhead structure, so it does not impact much. Variant *HUFF*, instead, gains a little more space over *LIGHT* without a noticeable time penalty, and it dominates the space-time tradeoff map in almost all cases.

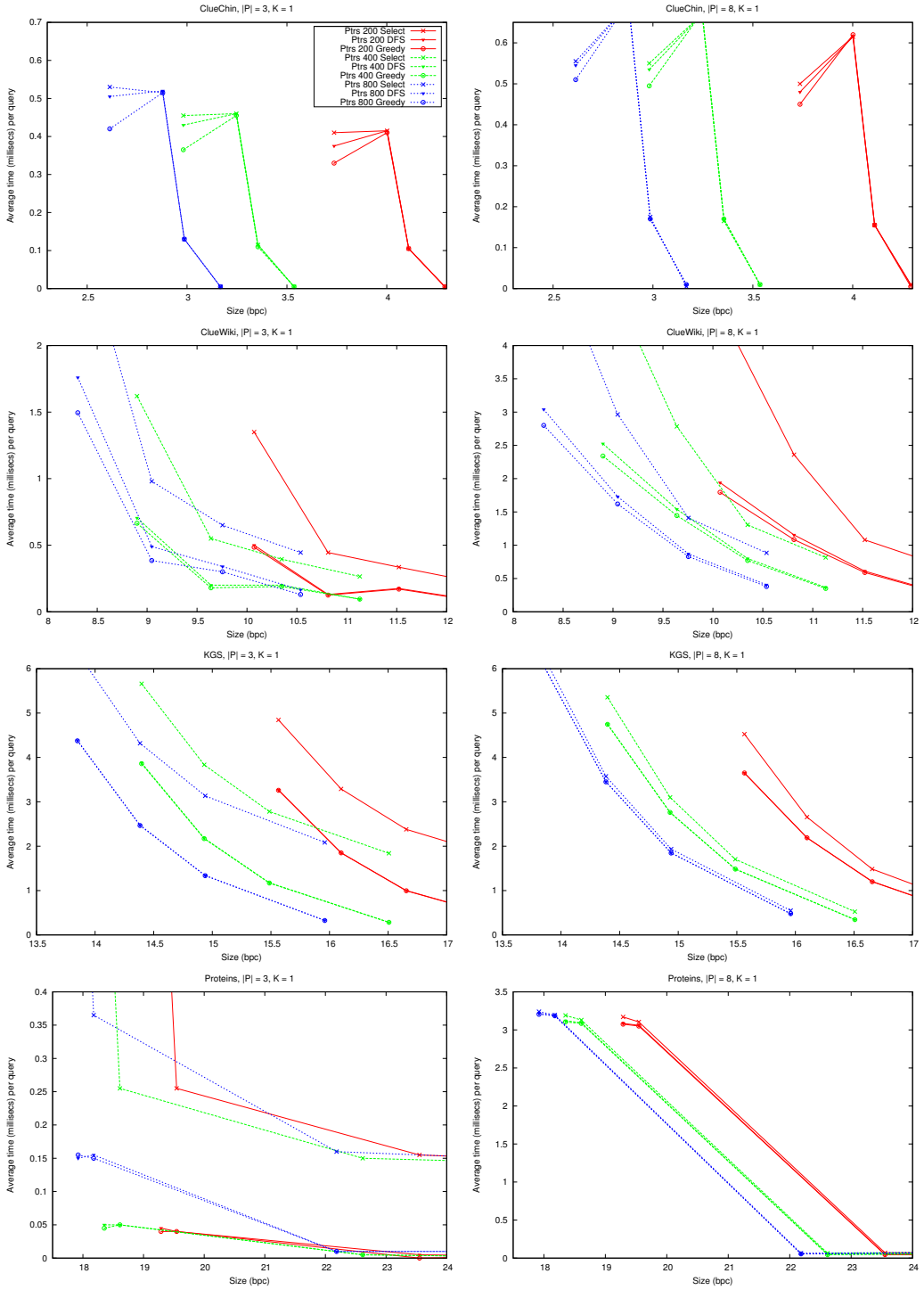


Fig. 14. Our different algorithms for top-10 queries. On the left for $|P| = 3$, on the right for $|P| = 8$.

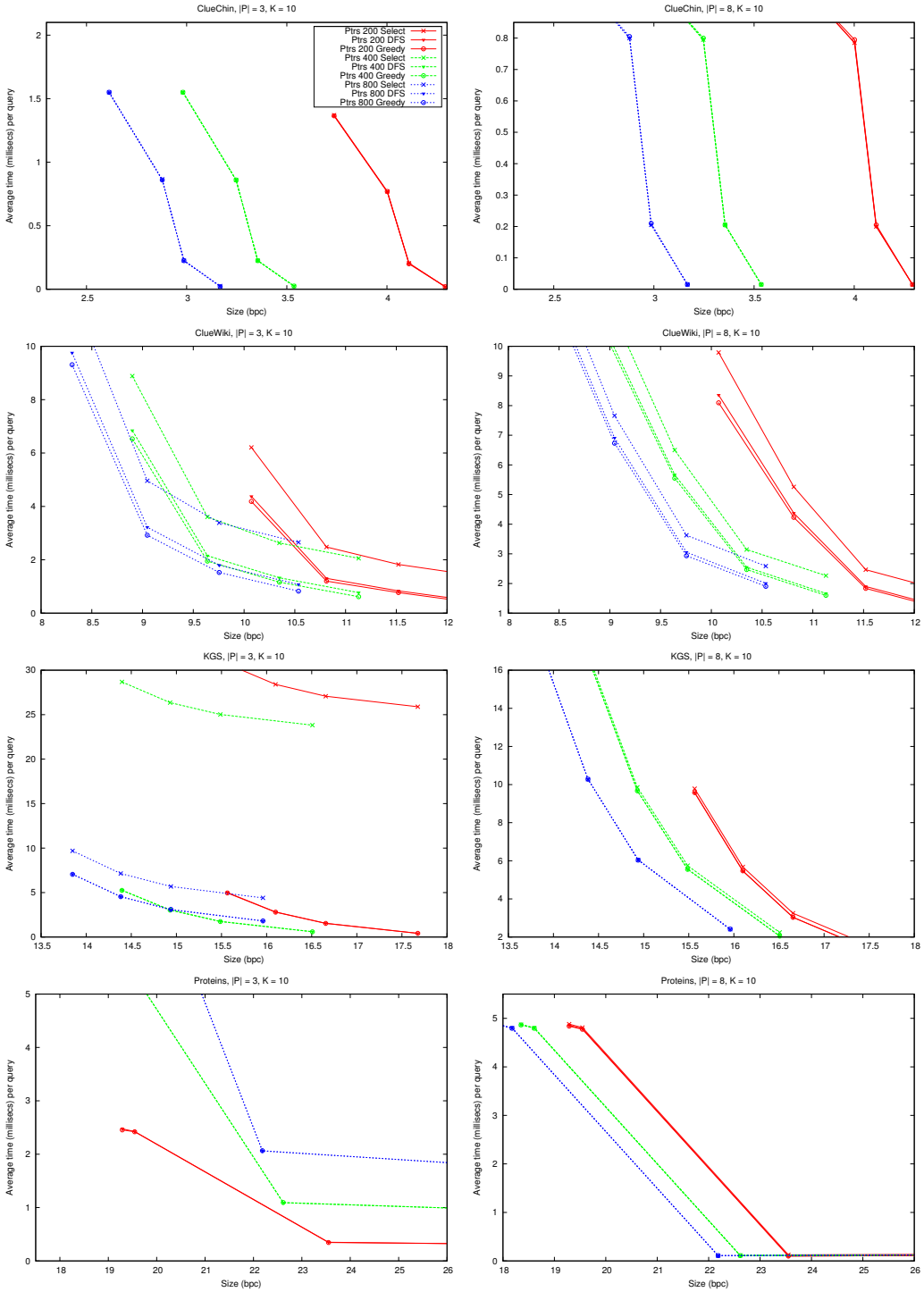


Fig. 15. Our different algorithms for top-10 queries. On the left for $|P| = 3$, on the right for $|P| = 8$.

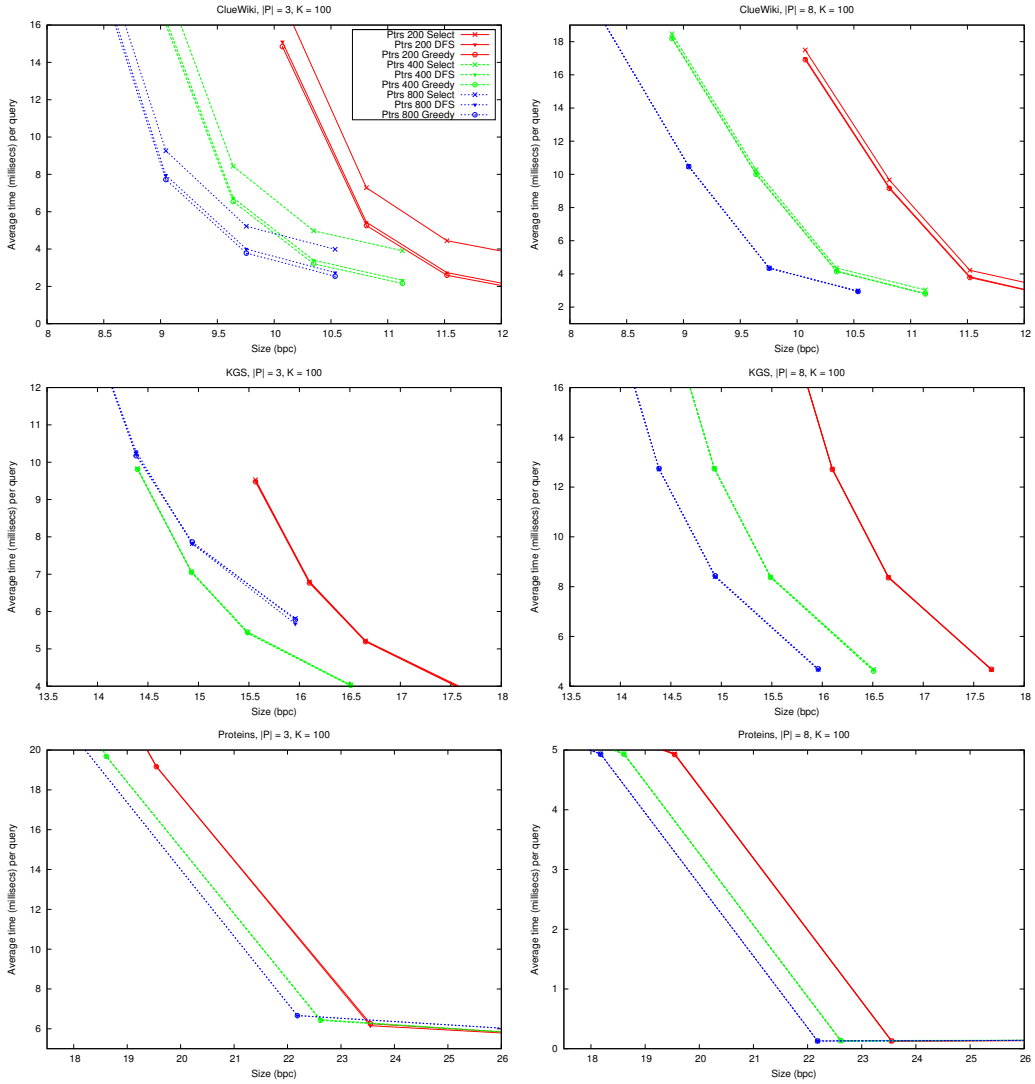


Fig. 16. Our different algorithms for top-100 queries. On the left for $|P| = 3$, on the right for $|P| = 8$.

It is interesting that the variant that does not include any structure on top of the wavelet tree, *WT-Alpha*, is much slower for small k, but it becomes competitive when k increases (more specifically, when the ratio between k and $ep - sp$ grows). This shows that, for less specific top-k queries, just running the Greedy algorithm without any extra structure may be the best option.

To compare with other techniques, we will use variant *HUFF*.

6.3.3. Comparison with Previous Work. Finally, we study the performance of our best solution compared with previous work to solve top-k queries.

The Greedy heuristic of Culpepper et al. [2010] is run over the different wavelet tree representations of the document array from Section 5: a plain one (*WT-Plain*, as in the original proposal [Culpepper et al. 2010]), an entropy-compressed one (*WT-RRR*), a

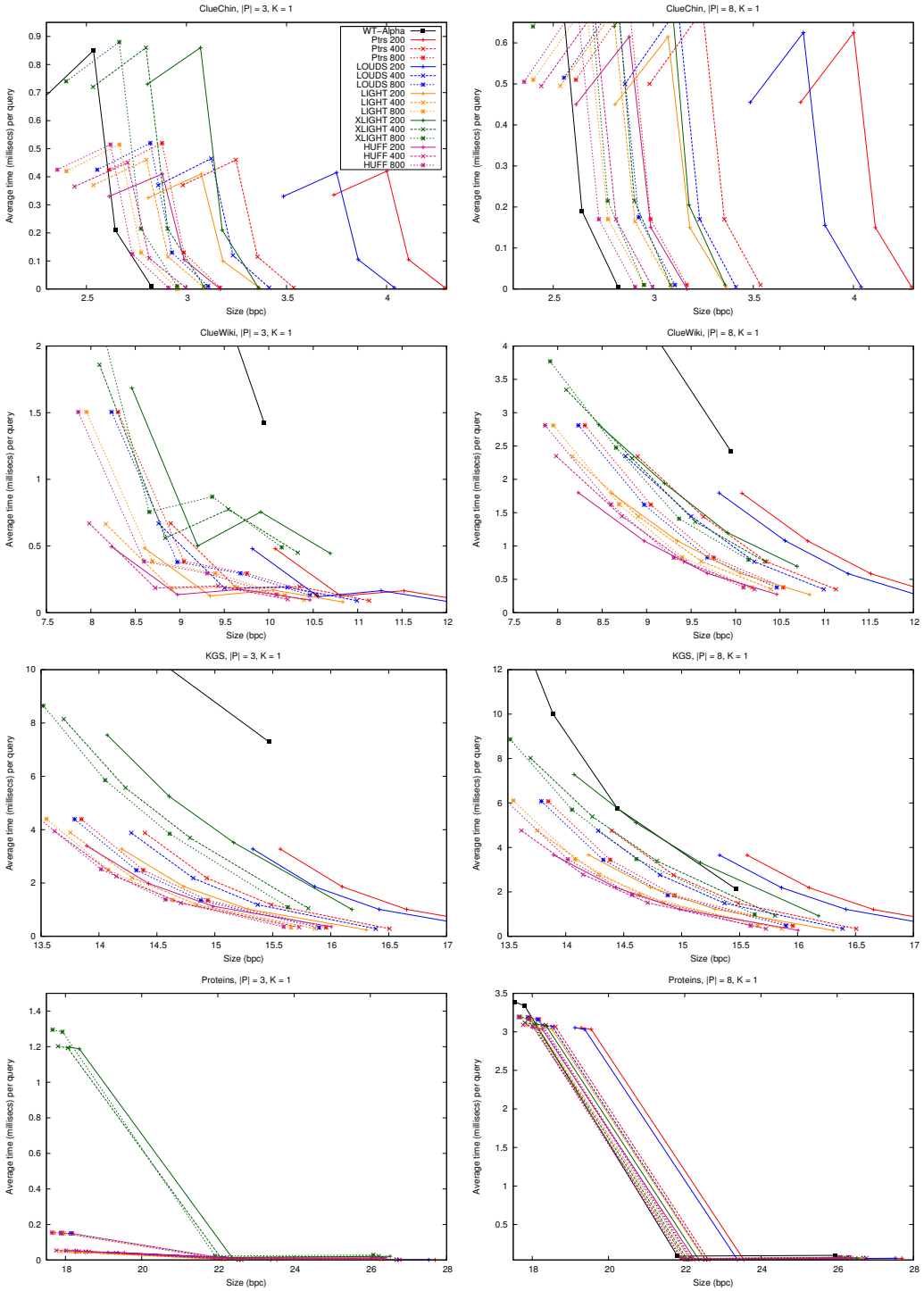


Fig. 17. Our different data structures for top-1 queries. On the left for $|P| = 3$, on the right for $|P| = 8$.

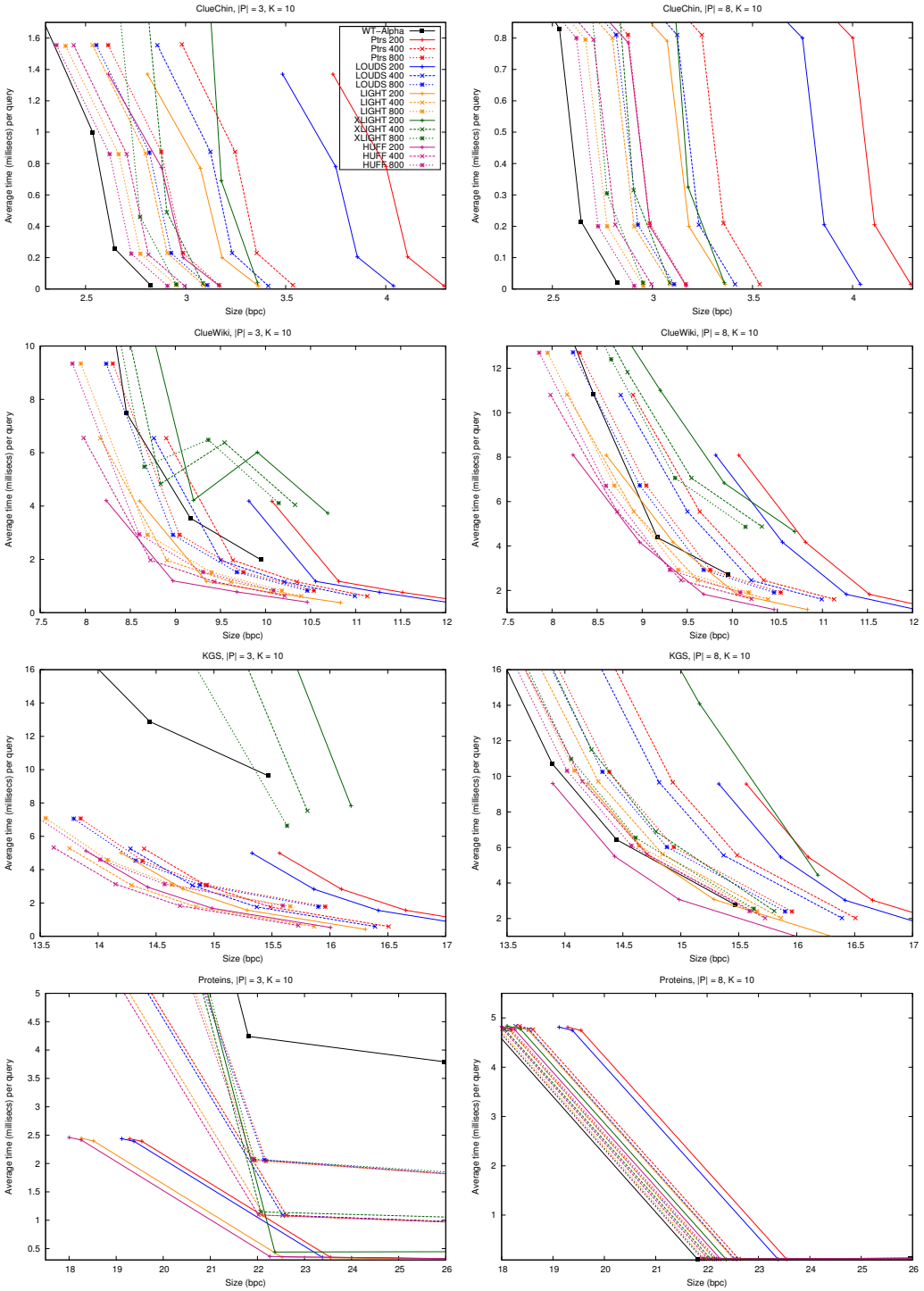


Fig. 18. Our different data structures for top-10 queries. On the left for $|P| = 3$, on the right for $|P| = 8$.

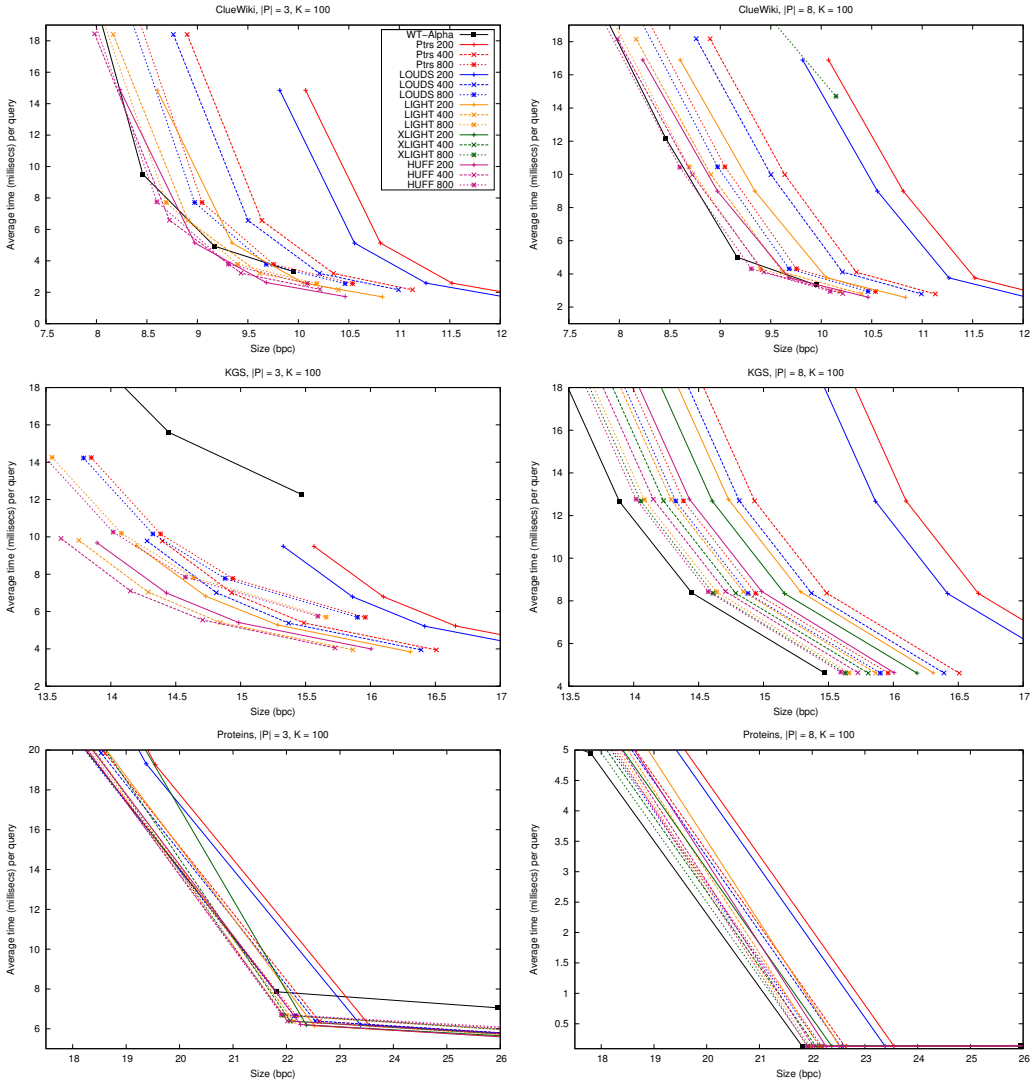


Fig. 19. Our different data structures for top-100 queries. On the left for $|P| = 3$, on the right for $|P| = 8$.

RePair-compressed one (*WT-RP*), and the hybrid that at each wavelet tree level chooses between plain, RePair, or entropy-based compression of the bitmaps (*WT-Alpha*).

We also combine *WT-Alpha* representation with our best implementation of Hon et al.'s structure (*WT-Alpha* + *HUFF* g'). Finally, we consider variant *Goly* + *HUFF* [Gagie et al. 2013; Hon et al. 2012], which runs the *rank*-based method (*Select*) on top of the fastest *rank*-capable sequence representation of the document array [Golynski et al. 2006]. This representation is faster than wavelet trees to compute *rank*, but does not support our more sophisticated traversal algorithms. It is not shown on `ClueChin` because its space is well out of bounds.

Figures 20 to 22 show the results. Our new structures dominate most of the space-time tradeoff map. *WT-Alpha* + *HUFF* is only dominated in space (in most cases only slightly) by *WT-RP*, which is however orders of magnitude slower. Only on `Proteins`,

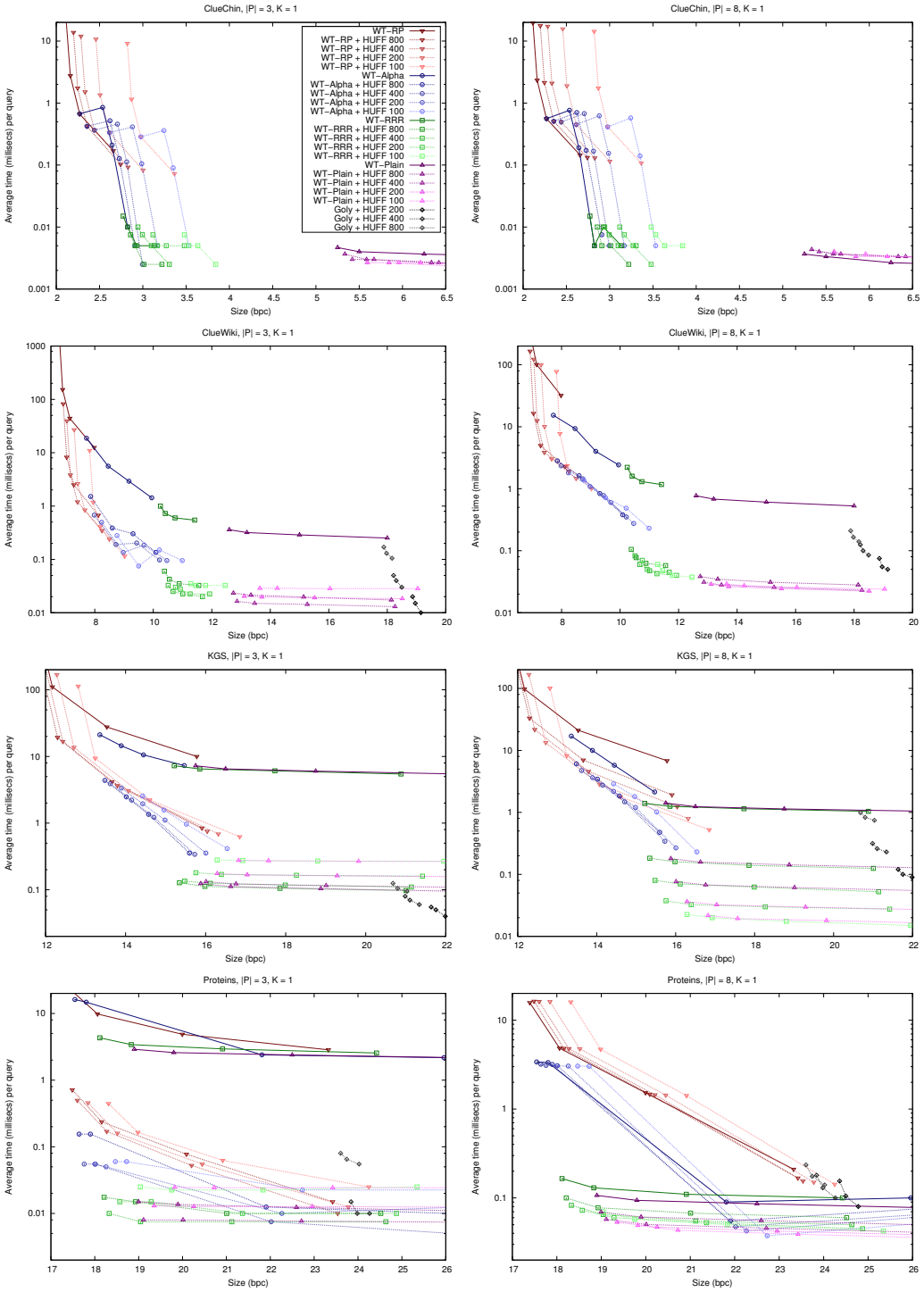


Fig. 20. Comparison with previous work for top-1 queries. On the left for $|P| = 3$, on the right for $|P| = 8$.

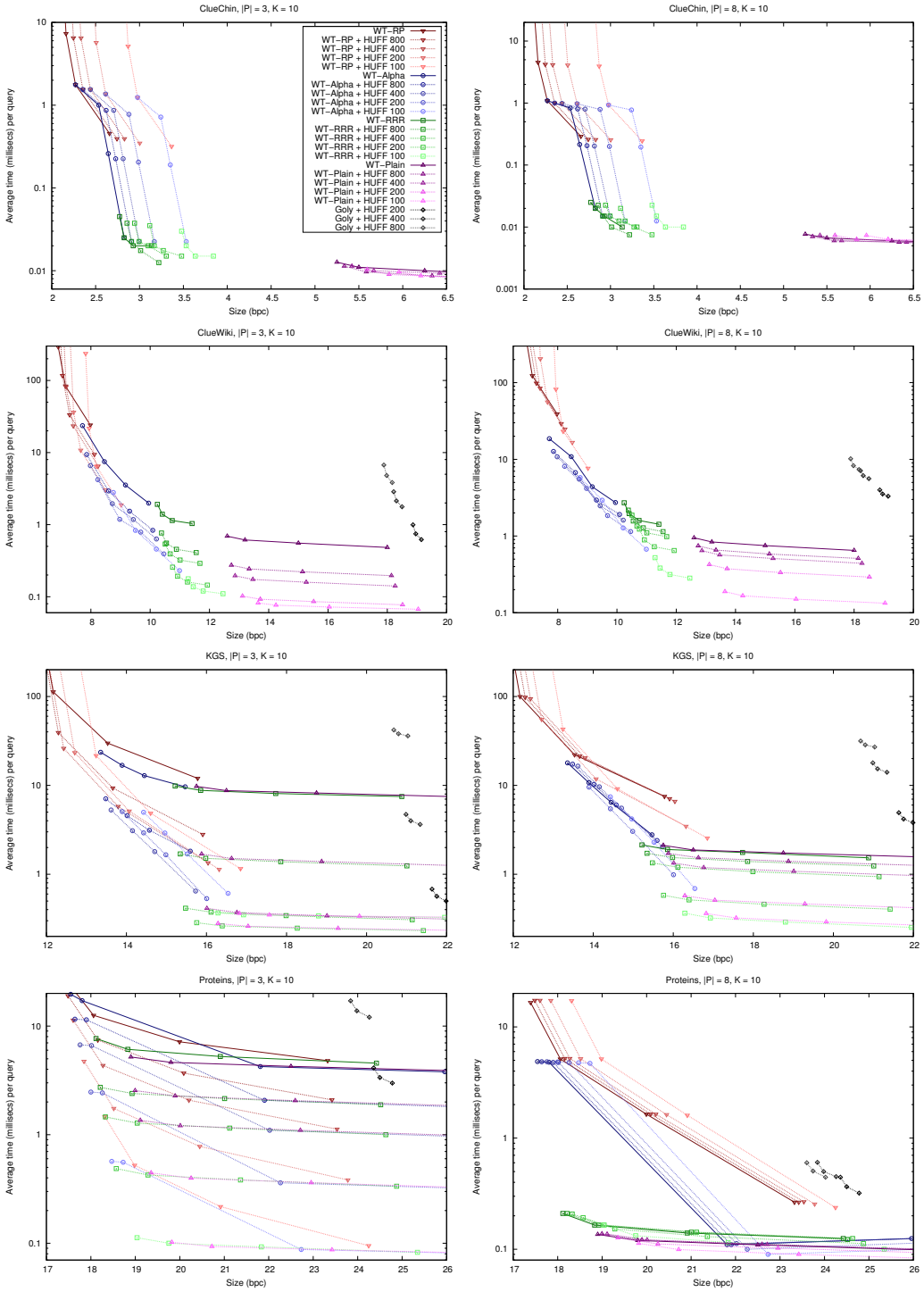


Fig. 21. Comparison with previous work for top-10 queries. On the left for $|P| = 3$, on the right for $|P| = 8$.

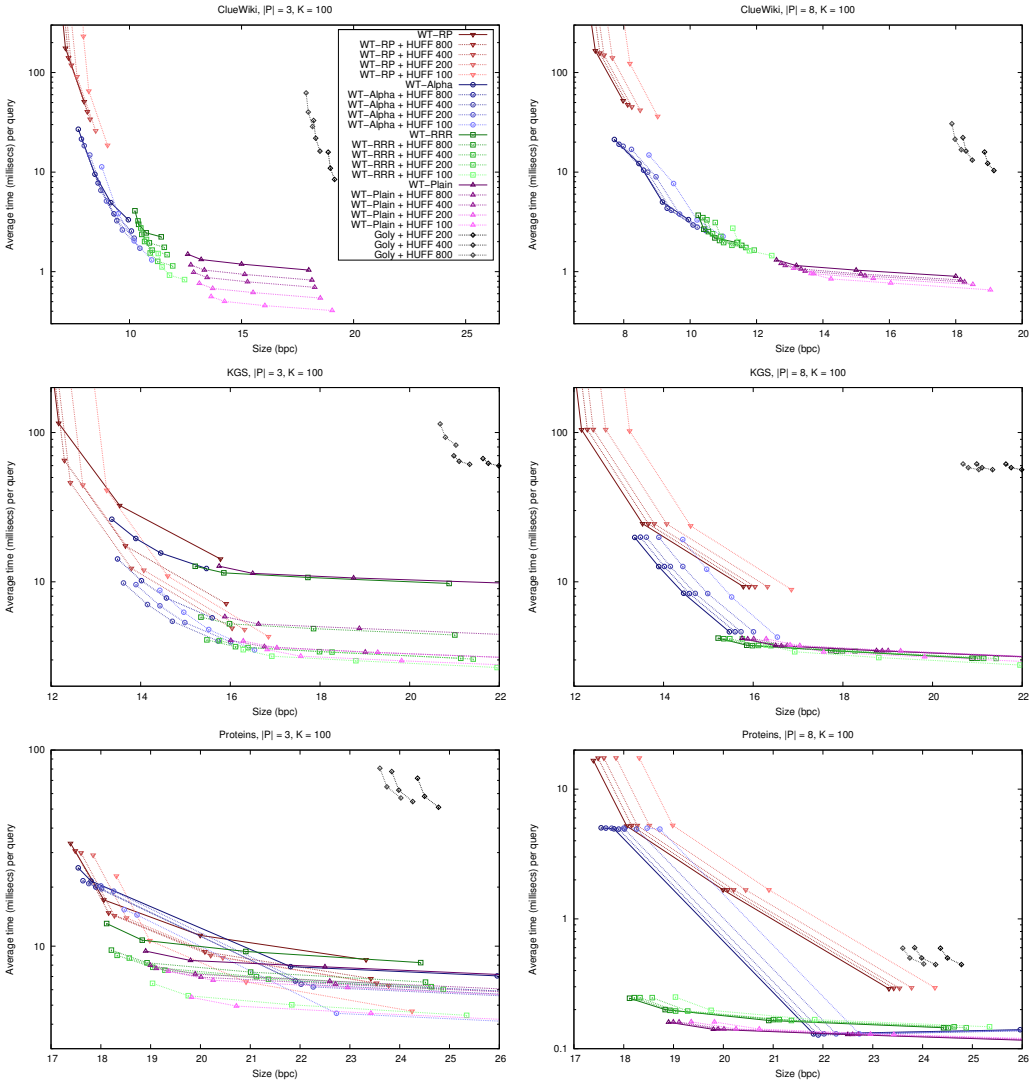


Fig. 22. Comparison with previous work for top-100 queries. On the left for $|P| = 3$, on the right for $|P| = 8$.

where compression does not work, our structures are in some cases dominated by previous work, *WT-Plain* and *WT-RRR*, especially for large k or small $ep-sp$. *Goly + HUFF* requires much space and is usually much slower than *WT-Plain*, which uses a slower sequence representation (the wavelet tree) but a smarter algorithm to traverse the block (our modified *Greedy*). We remind that, on `Proteins`, a structure not tested here performs much better [Belazzougui et al. 2012], whereas it is far from competitive on the compressible collections.

7. CONCLUSIONS AND FUTURE WORK

We have proposed the first compressed representation of the document array, a fundamental data structure to answer various general document retrieval queries, such as document listing with frequencies and top-k document retrieval. For this sake, we de-

veloped two novel compact data structures that achieve compression on generic repetitive sequences and might be of independent interest: (1) a grammar-compressed representation of bitmaps supporting *rank* and *select* queries, and (2) a compressed wavelet tree that extends the bitmap representation to sequences over general alphabets. We have shown that our technique reduces the space of a plain wavelet tree by up to half on real-life document arrays. Although our representation is significantly slower than a plain wavelet tree, we have engineered it to reach very competitive times on document retrieval queries while giving away just a small part of the space improvements.

We have also strengthened the performance of the document array for top-k queries by engineering the theoretical proposal of Hon et al. [2009]. We show that in practice this technique does not perform well if implemented on individual compressed suffix arrays (CSAs) as they propose (at least using current CSAs), but that it performs well over a wavelet tree representation of the document array. Our implementation removes various sources of redundancy and inefficiency of the original proposal (which are neglectable in an asymptotic analysis but relevant in practice), and involves various improvements in algorithmic and data structure design. This boosts the practical performance of the technique without giving up on its theoretical space and time guarantees. By combining it with our new compressed wavelet tree representations of the document array, queries are sped up by up to 10-fold for almost no extra space.

Our new techniques dominate the space-time tradeoff map of general document retrieval data structures, being outperformed by alternative techniques [Culpepper et al. 2010; Belazzougui et al. 2012] only on incompressible text collections, where the document array does not exhibit repetitiveness.

Various challenges remain, of course. Although we have managed to cut the space by half, our data structure still occupies at least 8–18 bits per text character (bpc), excluding our smallest collection, which uses less. This is in addition to the CSA, which using 5–6 bpc can already reproduce the text and thus contains sufficient information to answer any query. That is, the space of our structures is pure redundancy. Whether it is possible to find structures similar to CSAs, that use space close to that of the compressed text collection and in addition support document retrieval queries, is probably the most interesting open problem. We believe that more attention should be paid to the relation between text compressibility and the regularities manifest in data structures used for document retrieval. Advances on practical data structures that use less space and time in the worst case [Belazzougui et al. 2012; Hon et al. 2012] are also important. It is also interesting to design practical variants of optimal uncompressed data structures for top-k retrieval [Hon et al. 2009; Navarro and Nekrich 2012]. Some preliminary results [Konow and Navarro 2013] show that those solutions take some more space but can be orders of magnitude faster.[1]

Other recent advances can directly impact on the performance of our data structures. Claude and Navarro [2012] recently introduced the "wavelet matrix", which reorders the bitmaps of the wavelet tree so that fewer *rank*/*select* operations on the bitmaps are necessary to traverse it. This is very relevant for our RePair-compressed bitmaps, where those operations are very slow. We estimate that our times could be sped up by a factor of 2–3 with this technique. On the other hand, the impact of the reordering in the compressibility of the bitmaps is not yet well understood.

Another important challenge is to answer more powerful document retrieval queries. For example, we have used frequency as a simple relevance measure. In Information Retrieval, more sophisticated ones like *tf-idf* and *BM25* are used. Their formulas involve parameters like the *document size* and the *document frequency* (number of dis-

[1]A mistake in measuring space was made in that article, so the real space is about 50% higher, but their result is nevertheless relevant in proving the concept.

tinct documents in which the pattern appears). Techniques like Culpepper et al. [2010] do not immediately apply in such cases, but that of Hon et al. [2009] does, as it simply stores the precomputed top-k answers according to whatever relevance measure is used, and uses brute-force scanning of the uncovered cells. Similarly, our implementation of Hon et al.'s structure, using algorithm *Select* (instead of the more sophisticated ones derived from Culpepper et al. [2010]), could be used in a general scenario, without time penalties. More sophisticated "bag-of-word" queries involve several patterns, and the scores of the documents are added over the patterns that appear in them. There are many algorithms to handle bag-of-word queries in natural language scenarios [Baeza-Yates and Ribeiro-Neto 2011], many of which assume they have access to an inverted list of the documents where each word appears, in decreasing relevance order. Structures like those we have explored in this paper are able to emulate such lists for any arbitrary pattern (by solving the appropriate top-k query for that single pattern), and thus enable the implementation of any top-k algorithm for multiple words designed for inverted list structures, on general text collections. However, it is possible that data structures specifically designed for general text collections support better techniques natively, without emulating algorithms designed for other data structures.

Finally, scenarios where the collection undergoes changes over time and the structures must be updated, or where the structures must reside on disk, have received little attention and are useful and challenging.

REFERENCES

ANDERSSON, A., HAGERUP, T., NILSSON, S., AND RAMAN, R. 1995. Sorting in linear time? In *Proc. 27th Annual ACM Symposium on Theory of Computing (STOC)*. 427–436.

ARROYUELO, D., CÁNOVAS, R., NAVARRO, G., AND SADAKANE, K. 2010. Succinct trees in practice. In *Proc. 11th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 84–97.

BAEZA-YATES, R. AND RIBEIRO-NETO, B. 2011. *Modern Information Retrieval* 2nd Ed. Pearson Education.

BARBAY, J., GAGIE, T., NAVARRO, G., AND NEKRICH, Y. 2010. Alphabet partitioning for compressed rank/select and applications. In *Proc. 21st Annual International Symposium on Algorithms and Computation (ISAAC)*. LNCS 6507. 315–326 (part II).

BARBAY, J., HE, M., MUNRO, I., AND RAO, S. S. 2011. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Transactions on Algorithms 7,* 4, article 52.

BELAZZOUGUI, D., BOLDI, P., PAGH, R., AND VIGNA, S. 2009a. Monotone minimal perfect hashing: searching a sorted table with $o(1)$ accesses. In *Proc. 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 785–794.

BELAZZOUGUI, D., BOLDI, P., PAGH, R., AND VIGNA, S. 2009b. Theory and practise of monotone minimal perfect hashing. In *Proc. 10th Workshop on Algorithm Engineering and Experiments (ALENEX)*.

BELAZZOUGUI, D., NAVARRO, G., AND VALENZUELA, D. 2012. Improved compressed indexes for full-text document retrieval. *Journal of Discrete Algorithms 18*, 3–13.

BENDER, M. AND FARACH-COLTON, M. 2000. The LCA problem revisited. In *Proc. 2nd Latin American Symposium on Theoretical Informatics (LATIN)*. 88–94.

BENOIT, D., DEMAINE, E., MUNRO, J. I., RAMAN, R., RAMAN, V., AND RAO, S. S. 2005. Representing trees of higher degree. *Algorithmica 43,* 4, 275–292.

BRISABOA, N., LADRA, S., AND NAVARRO, G. 2013. DACs: Bringing direct access to variable-length codes. *Information Processing and Management 49,* 1, 392–404.

CLARK, D. 1998. Compact pat trees. Ph.D. thesis, University of Waterloo, Waterloo, Ont., Canada.

CLAUDE, F. AND NAVARRO, G. 2008. Practical rank/select queries over arbitrary sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 5280. 176–187.

CLAUDE, F. AND NAVARRO, G. 2012. The wavelet matrix. In *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 7608. 167–179.

CULPEPPER, S., NAVARRO, G., PUGLISI, S., AND TURPIN, A. 2010. Top-*k* ranked document search in general text databases. In *18th Annual European Symposium on Algorithms (ESA)*. LNCS 6347. 194–205 (part II).

FERRAGINA, P. AND MANZINI, G. 2005. Indexing compressed text. *Journal of the ACM 52,* 4, 552–581.

FERRAGINA, P., MANZINI, G., MÄKINEN, V., AND NAVARRO, G. 2007. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms 3,* 2, article 20.

FISCHER, J. AND HEUN, V. 2011. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing 40,* 2, 465–492.

GAGIE, T., KÄRKKÄINEN, J., NAVARRO, G., AND PUGLISI, S. 2013. Colored range queries and document retrieval. *Theoretical Computer Science 483*, 36–50.

GAGIE, T., NAVARRO, G., AND PUGLISI, S. 2012. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science 426-427*, 25–41.

GAGIE, T., PUGLISI, S., AND TURPIN, A. 2009. Range quantile queries: Another virtue of wavelet trees. In *Proc. 16th Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 5721. 1–6.

GEARY, R., RAHMAN, N., RAMAN, R., AND RAMAN, V. 2006. A simple optimal representation for balanced parentheses. *Theoretical Computer Science 368,* 3, 231–246.

GOLYNSKI, A., MUNRO, I., AND RAO, S. S. 2006. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 368–373.

GONZÁLEZ, R. AND NAVARRO, G. 2007. Compressed text indexes with fast locate. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*. LNCS 4580. 216–227.

GROSSI, R., GUPTA, A., AND VITTER, J. 2003. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 841–850.

GROSSI, R., ORLANDI, A., AND RAMAN, R. 2010. Optimal trade-offs for succinct string indexes. In *Proc. 37th International Colloquium on Algorithms, Languages and Programming (ICALP)*. 678–689.

GROSSI, R. AND VITTER, J. 2006. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing 35,* 2, 378–407.

HON, W.-K., SHAH, R., AND THANKACHAN, S. 2012. Towards an optimal space-and-query-time index for top-k document retrieval. In *Proc. 23rd Annual Symposium on Combinatorial Pattern Matching (CPM)*. LNCS 7354. 173–184.

HON, W.-K., SHAH, R., AND VITTER, J. 2009. Space-efficient framework for top-k string retrieval problems. In *Proc. 50th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*. 713–722.

JACOBSON, G. 1989. Space-efficient static trees and graphs. In *Proc. 30th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*. 549–554.

KONOW, R. AND NAVARRO, G. 2013. Faster compact top-k document retrieval. In *Proc. 23rd Data Compression Conference (DCC)*. 351–360.

LARSSON, N. AND MOFFAT, J. A. 2000. Offline dictionary-based compression. *Proceedings of the IEEE 88*, 1722–1732.

MÄKINEN, V. AND NAVARRO, G. 2005. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing 12,* 1, 40–66.

MÄKINEN, V. AND NAVARRO, G. 2007. Implicit compression boosting with applications to self-indexing. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*. LNCS 4726. 214–226.

MANBER, U. AND MYERS, G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing 22,* 5, 935–948.

MANZINI, G. 2001. An analysis of the Burrows-Wheeler transform. *Journal of the ACM 48,* 3, 407–430.

MARUYAMA, S., SAKAMOTO, H., AND TAKEDA, M. 2012. An online algorithm for lightweight grammar-based compression. *Algorithms 5,* 2, 214–235.

MUNRO, I. 1996. Tables. In *Proc. 15th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. LNCS 1180. 37–42.

MUNRO, I. AND RAMAN, V. 2002. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing 31,* 3, 762–776.

MUTHUKRISHNAN, S. 2002. Efficient algorithms for document retrieval problems. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 657–666.

NAVARRO, G. 2012. Wavelet trees for all. In *Proc. 23rd Annual Symposium on Combinatorial Pattern Matching (CPM)*. LNCS 7354. 2–26.

NAVARRO, G. AND MÄKINEN, V. 2007. Compressed full-text indexes. *ACM Computing Surveys 39,* 1, article 2.

NAVARRO, G. AND NEKRICH, Y. 2012. Top-k document retrieval in optimal time and linear space. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1066–1078.

NAVARRO, G. AND RUSSO, L. 2008. Re-pair achieves high-order entropy. In *Proc. 18th Data Compression Conference (DCC)*. 537.

OKANOHARA, D. AND SADAKANE, K. 2007. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*.

RAMAN, R., RAMAN, V., AND RAO, S. 2007. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms 3,* 4.

SADAKANE, K. 2003. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms 48,* 2, 294–313.

SADAKANE, K. 2007. Succinct data structures for flexible text retrieval systems. *Journal on Discrete Algorithms 5,* 1, 12–22.

SADAKANE, K. AND NAVARRO, G. 2010. Fully-functional succinct trees. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 134–149.

VÄLIMÄKI, N. AND MÄKINEN, V. 2007. Space-efficient algorithms for document retrieval. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*. LNCS 4580. 205–215.

WEINER, P. 1973. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory (SWAT)*. 1–11.

ZIV, J. AND LEMPEL, A. 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory 24,* 5, 530 – 536.