UNIVERSIDAD DE CHILE

FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

ESTRUCTURAS DE DATOS SUCINTAS PARA RECUPERACIÓN DE DOCUMENTOS

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

DANIEL ALEJANDRO VALENZUELA SERRA

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

MIEMBROS DE LA COMISIÓN:
JORGE PÉREZ ROJAS
BENJAMÍN BUSTOS CÁRDENAS
DIEGO ARROYUELO BILLIARDI

Este trabajo ha sido parcialmente financiado por los proyectos Fondecyt 1-080019, 1-110066, e Instituto de Dinámica Celular y Biotecnología (ICDB).

SANTIAGO DE CHILE
ENERO 2013

Resumen

La recuperación de documentos consiste en, dada una colección de documentos y un patrón de consulta, obtener los documentos más relevantes para la consulta. Cuando los documentos están disponibles con anterioridad a las consultas, es posible construir un índice que permita, al momento de realizar las consultas, obtener documentos relevantes en tiempo razonable. Contar con índices que resuelvan un problema como éste es fundamental en áreas como recuperación de la información, minería de datos y bioinformática, entre otros.

Cuando el texto que se indexa es lenguaje natural, la solución paradigmática corresponde al índice invertido. Sin embargo, los problemas de recuperación de documentos emergen también en escenarios en que el texto y los patrones de consulta pueden ser secuencias generales de caracteres, como lenguajes orientales, bases de datos multimedia, secuencias genómicas, etc. En estos escenarios los índices invertidos clásicos no se aplican con el mismo éxito. Si bien existen soluciones que requieren espacio lineal en este escenario de texto general, el espacio que utilizan es un problema importante: estas soluciones pueden utilizar más de 20 veces el espacio de la colección.

Esta tesis presenta nuevos algoritmos y estructuras de datos para resolver algunos problemas fundamentales para recuperación de documentos en colecciones de texto general, en espacio reducido. Más específicamente, se ofrecen nuevas soluciones al problema de *document listing* con frecuencias, y recuperación de los *top-k* documentos. Como subproducto, se desarrolló un nuevo esquema de compresión para bitmaps repetitivos que puede ser de interés por sí mismo.

También se presentan implementaciones de las nuevas propuestas, y de trabajos relacionados. Estudiamos nuestros algoritmos desde un punto de vista práctico y los comparamos con el estado del arte. Nuestros experimentos muestran que nuestras soluciones para *document listing* reducen el espacio de la mejor solución existente en un 40%, con un impacto mínimo en los tiempos de consulta.

Para recuperación de los top-k documentos, también se redujo el espacio de la mejor solución existente en un 40% en la práctica, manteniendo los tiempos de consulta. Así mismo, mejoramos el tiempo de esta solución hasta en un factor de 100, a expensas de usar un bit extra por carácter. Nuestras soluciones son capaces de retornar los top-10 a top-100 documentos en el orden de milisegundos. Nuestras nuevas soluciones dominan la mayor parte del mapa espacio-tiempo, apuntando a ser el estándar contra el cual comparar la investigación futura.

A Maggi.

Agradecimientos

Durante el desarrollo de este trabajo recibí el apoyo de muchas personas, que resultan dificiles de listar aquí. En primer lugar, quiero mencionar a mi profesor guía, Gonzalo Navarro. Agradecer su inconmensurable generosidad, paciencia, entrega y tremenda buena disposición para que este proceso se desarrollara de la mejor maner posible. Agradezco a los miembros de la comisión; Jorge Pérez, Benjamín Bustos y Diego Arroyuelo por sus comentarios luego de haber examinado cuidadosamente el borrador final de este documento.

Compañeros del programa de Magíster y del Grupo Miércoles de Algorítmos tuvieron también un rol muy especial. En particular, vaya un abrazo para Miguel Angel Martínez, Milton Inostroza, Francisco Claude, Roberto Konow, Rodrigo Cánovas y Rodrigo Paredes

Una mención muy importante también para Angélica Aguirre y Sandra Gaez, por la buena disposición para recibirme siempre en sus oficinas.

Finalmente, agradezco a mis padres por el apoyo y cariño brindado, no solamente durante este proceso, sino durante todos los años de mi vida.

University of Chile
Faculty of Physics and Mathematics
Graduate School

Succinct Data Structures for Document Retrieval

by

Daniel Valenzuela

Submitted to the University of Chile in fulfillment
of the thesis requirement to obtain the degree of
M.Sc. in Computer Science

Advisor :
GONZALO NAVARRO

Committee :
JORGE PÉREZ
BENJAMÍN BUSTOS
DIEGO ARROYUELO

This work is partially funded by Fondecyt projects 1-080019, 1-110066,
and Institute for Cell Dynamics and Biotechnology (ICDB).

Department of Computer Science - University of Chile
Santiago - Chile
January 2013

Abstract

Document retrieval consists in, given a collection of documents and a query pattern, obtaining documents relevant for the query. Having the documents available on advance allows one to build an index that obtains relevant documents within a reasonable amount of time. Indexes to solve such a fundamental problem are required in many fields, like Information Retrieval, data mining, bioinformatics, and so on.

When the text to be indexed is natural language, the paradigmatic solution is the inverted index. However, document retrieval problems arise also in scenarios where text and pattern can be general sequences of symbols, such as Oriental languages, multimedia databases, and genomic sequences. In those scenarios the classical inverted indexes cannot be successfully applied. Even though there exist linear-space solutions for this general text scenario, the space required is a serious concern in practice: those indexes may require more than 20 times the size of the collection.

This thesis introduces novel algorithms and data structures to solve some important document retrieval problems on general text collections in reduced space. More specifically, we provide new solutions for document listing with term frequencies and for top-k document retrieval. As a byproduct, we obtain a new compression scheme for repetitive bitmaps that might be of independent interest.

We implemented our proposals, as well as most relevant previous work, and studied their practicality. Our experiments show that our proposals for document listing reduce the space of the best previous solution by 40% with little impact on query times.

For top-k document retrieval our solutions allow one to reduce the space by 40% in practice, with no impact on the query time. In addition, we were able to reduce query times up to a factor of 100, as the cost of about 1 extra bit per character. Our solutions are able to retrieve from top-10 to top-100 documents within milliseconds. Our new combinations dominate most of the space/time tradeoff, aiming to become the reference standard with which subsequent research should compare.

Contents

Chapter 1

Introduction

Humankind is producing and collecting incredibly big amounts of data. While the Web is nowadays the paradigmatic example (over 20 billion pages conform the indexable Web), enormous repositories of data are arising in almost every area of human knowledge. Some examples are Web pages, genomic sequences, the data collected by the Large Hadron Collider, the increasingly detailed maps from the Earth, and astronomical data collected by telescopes, click-through data and query logs, among many others. Managing those amounts of data raises many challenges from many different perspectives. One of the main challenges is the problem of searching for certain patterns in this sea of data, obtaining meaningful results in a reasonable amount of time.

When the data to be searched is available beforehand, it is possible to build a data structure in a preprocessing phase. This additional data structure is used at query time to improve the speed and effectiveness in the process of answering queries. Such a data structure is called an *index*. The most naive index will pre-store the answer to every possible query. This, of course, would be very fast at answering the queries (just the time required to look at the table of answers) but, in most cases, extremely space-inefficient. 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 (for example, a sequential search over the visible Web would take months, while any decent search engine takes less than a second to answer). The challenge of indexing can be thought of as how to achieve relevant space-time trade-offs in between those extremes.

Data repositories can be of very different natures, and also the kind of queries that are expected may vary a lot depending on the context. Documents may have a well-defined structure (graphs, XML) or not (music and image repositories), and so on.

For the sake of this work we offer a very general definition: a document is a finite sequence of symbols over a given alphabet. That is, we regard documents simply as strings. Queries also may be defined in many different ways. In this work a query pattern is also defined as a given string. More formal definitions are given in Chapter 3.

1.1 Succinct Data Structures and Document Retrieval

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. Note that when the collections contain millions of documents, the output of the query could still be very big. In some scenarios, like those when the output is directly given to a user (think about Web search engines) it is also important to have a *ranked* document retrieval. That is, the documents in the output are sorted according to a certain *relevance* criterion. One of the most fundamental criteria for this work is the *term frequency*, which measures the relevance of a given document as the number of occurrences of the query pattern on the document [4]. Moreover, given a parameter k, one could ask only for the top-k ranked documents.

To date, the best known solution for this scenario is the inverted index. The essential idea of an inverted index is to store a vocabulary of text words and a list of occurrences for each vocabulary word. The list of occurrences stores the documents in which the word appears, plus some extra information depending on which queries are going to be answered. This technique has proven to be very effective, and due to Heaps's law [40] , the size of the vocabulary (and therefore the number of occurrence lists) is not that big. However, the inverted index relies on the assumption that the text is tokenizable into words, which is not true in many scenarios of interest. One such scenario is documents written in languages such as Chinese or Korean, where it is not easy to split words automatically. Search engines treat these texts as sequences of symbols, so that queries can retrieve any substring of the text. Even agglutinating languages such as Finnish or German pose problems to the inverted index approach. In other scenarios the data to be indexed do not even have a concept of word, yet document retrieval would be of interest: In the field of biotechnology, huge repositories of DNA sequences arise everyday. Looking for short patterns that appear frequently in those sequences (motifs), and finding similarities among given sequences is a critical task in many scientific contexts, like understanding diseases, establishing phylogeny, genome assembly, etc. In source code repositories it is important to look for functions making use of an expression or function call. Detecting copies of a video in a given database is important in the task of revealing plagiarism, and so on.

When the texts to be indexed are not easily tokenized into words, the most successful solutions are based on suffix trees and suffix arrays. Those structures support general queries over the text, meaning that they allow one to search for any arbitrary concatenation of symbols. A suffix array can be thought as a list of all the suffixes of the text stored in lexicographic order. Given a query pattern, with two binary searches it is possible to find an interval containing all the suffixes that begin with the query pattern. This interval corresponds to all the occurrences of the pattern in the text. The cost of this functionality is the high space consumption of suffix trees and suffix arrays: they can require up to 20 times the size of the original text. It is important to note that suffix arrays and suffix trees by themselves are not powerful enough to give an efficient solution to document retrieval problems. Suffix arrays and suffix trees give all the occurrences of the query pattern; however, this number of occurrences is typically much bigger than the number of documents where the pattern appears. Muthukrishnan, in a foundational work [56], introduced the document array, giving the first optimal solution for the document listing problem, and efficient solutions for many document retrieval problems. Muthukrishnan's approach is built on top of the suffix tree, therefore its space consumption is even higher.

On the other hand, in the last two decades we have witnessed a rich development of the field of succinct data structures [3, 26, 29, 38, 54, 58, 65, 66]. These aim to provide fast functionality while requiring space close to the information-theoretic lower bound.

The range of problems addressed with succinct data structures is very wide: rank and select over binary sequences [19, 54, 65], succinct representations of trees with rich functionality [2, 15, 45], graph representation of social networks [41], and so on. In particular, compressed suffix arrays enable the same functionality of classical suffix arrays, requiring as little space as that of the compressed text. They also allow one to reconstruct the text, therefore they can be seen as a replacement of the text itself. For this reason, they are also called *self-indexes*.

Self-indexes have achieved certain maturity, exhibiting results very close to the theoretical bounds both in theory and in practice. However, the functionality they provide is mainly focused on the pattern matching problem. Only in the recent years we have seen the first efforts to develop succinct data structures for document retrieval problems, revealing that there is a lot of room for improvement.

On the one hand, only relatively simple document listing problems are being addressed, where no document relevance information is retrieved. On the other hand, the solutions are far from being optimal. There are solutions for retrieving the top-k documents, for instance, that requires twice the minimal space, but we are going to show that their implementation is not practical. There are more practical solutions in practice, but they require much more than the minimum space and have no time worst-case-time guarantees. Our main contributions help to reduce all those gaps.

1.2 Outline and Contributions

This thesis introduces novel algorithms and data structures to solve some important document retrieval problems on general text collections on reduced space. More specifically, we provide new solutions for the document listing with term frequencies and the top-k document retrieval problems. As a byproduct, we obtain new compression schemas for bitmaps and sequences that might be of independent interest.

We implemented our proposals, as well as some relevant previous work, and studied their practicality. Our experiments show that our proposals for document listing reduce the space of the best previous solution by 40% with little impact on query times.

For top-k document retrieval our solutions allow to reduce the space by a 40% with no impact on the query time. Also we were able to reduce query times up to a factor of 100, at the cost of about 1 extra bit per character, and we presented combinations that reduced query time and space requirement simultaneously.

The outline of the thesis is as follows:

- Chapter 2 gives basic definitions, introduces the fundamental topics in the field, and reviews the related work.
- In Chapter 3 we introduce document retrieval concepts, provide definitions for the main problems addressed in the thesis, and review the previous approaches. Finally, we present the document collections and experimental environment to be used throughout the thesis.
- In Chapter 4 we propose a compressed representation of binary sequences that exploit repetitiveness and is capable of answering *rank* and *select* queries.
- In Chapter 5 we propose compressed wavelet trees based on the new sequence representation of Chapter 4, and analyze how those wavelet trees perform in a document retrieval scenario.
- Chapter 6 proposes a practical version of a previous theoretical work of Hon et al. [43] for top-k document retrieval. We also propose novel algorithms that improve its practical performance. This structure, in combination with the solution of Chapter 5, dominates almost all the space/time tradeoff.
- Chapter 7 presents our implementation of a different approach for document listing, based on Monotone Minimal Perfect Hash Functions.
- Finally, in Chapter 8 we discuss our contributions and possible directions for future work.

The work of Chapters 4 and 5 was published in the Symposium of Experimental Algorithms (SEA) 2011 [60]. The work of Chapter 6 was published in the Symposium of Experimental Algorithms (SEA) 2012 [62]. Both articles [60, 62] were invited and submitted to a Special Issue of the Journal of Experimental Algorithms. The work of Chapter 7 will appear in a Special Issue of the Journal of Discrete Algorithms [13].

Chapter 2

Related Work

2.1 Entropy of a Text

Consider a sequence of characters $T[1,n]$ over an alphabet $\Sigma = \{1,\ldots,\sigma\}$ and let n_c be the number of times character c appears in T. The empirical zero-order entropy is defined as follows: [1]

$$H_0(T) = \sum_{c\in\Sigma, n_c>0} \frac{n_c}{n} \log \frac{n}{n_c}.$$

The value $nH_0(T)$ is the minimum size in bits that can be achieved to encode T using a compressor that assigns a unique code to each symbol in Σ.

However, the compression can be improved if codes are assigned depending on the context. We can define the empirical k-th order entropy as [52]:

$$H_k(T) = \sum_{s\in\Sigma^k, T^s\neq\varepsilon} \frac{|T^s|}{n} H_0(T^s),$$

where T^s is the string obtained if we concatenate each character that appears followed by the context s in T. The empirical k-th order entropy of a text T is a lower bound for the numbers of bits per symbol required to encode T using any algorithm that encodes a symbol considering the context defined by the following k symbols.

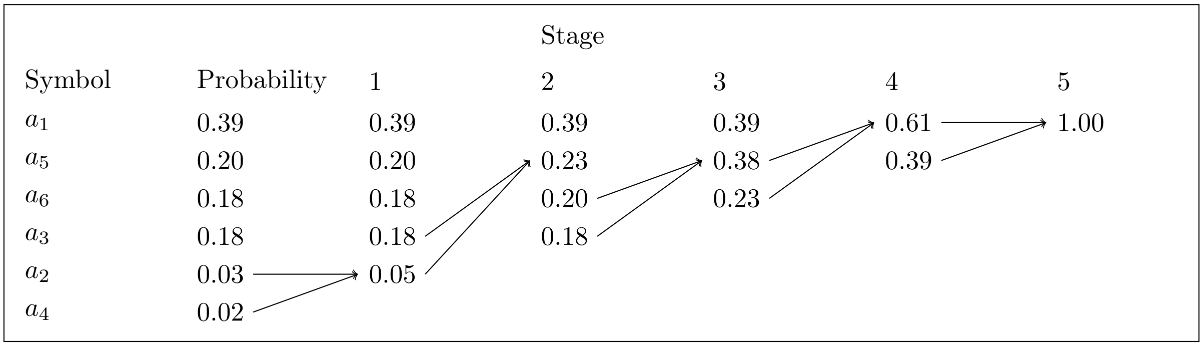
2.2 Huffman Coding

Huffman [44] is one of the most popular coding schemes used to compress to zero-order entropy. It assigns a variable-length code to each symbol of the original sequence, minimizing the length of the encoded sequence. Huffman codes are *prefix free* codes, meaning that there

[1] In this thesis we will assume logarithms are to the base 2 by default.

is no code that is a prefix of another code. This property allows the encoded sequence to be decompressed unambiguously without need of any end marker between codewords. The main idea behind Huffman coding is to assign shorter codes to more frequent symbols and longer codes to the less frequent ones.

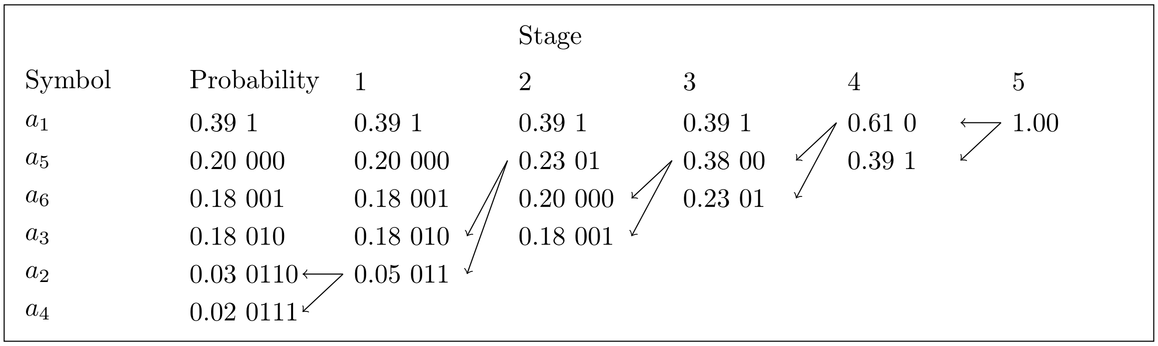
The algorithm first computes the frequency of each symbol and then sorts the symbols by frequency. A virtual symbol is then created to replace the two least frequent symbols. The frequency of the new symbol is the sum of the frequencies of those two symbols that compose it. The procedure is applied repeatedly (to original and virtual symbols) until the number of symbols is reduced to one. Figure 2.1 shows an example.



				Stage		
Symbol	Probability	1	2	3	4	5
a_1	0.39	0.39	0.39	0.39	0.61	1.00
a_5	0.20	0.20	0.23	0.38	0.39	
a_6	0.18	0.18	0.20	0.23		
a_3	0.18	0.18	0.18			
a_2	0.03	0.05				
a_4	0.02					

Figure 2.1: Procedure to obtain the virtual symbols. At every stage the arrows show the two least frequent symbols being replaced by a virtual symbol.

The codes assigned to the symbols are sequences of 0s and 1s. The process to obtain the symbols is explained as follows: The virtual symbol with probability 1.0 will be encoded with the empty string. Then, for each virtual symbol, we compute recursively the code of the composing symbols, making the following expansion: we append a 0 to the current code to obtain the code of the first composing symbol, and we append a 1 to the current code to obtain the code of the second composing symbol. When there is no virtual symbol left to be expanded, we have computed the codes of all the original symbols. Figure 2.2 shows the expansion for our example.



				Stage		
Symbol	Probability	1	2	3	4	5
a_1	0.39 1	0.39 1	0.39 1	0.39 1	0.61 0	1.00
a_5	0.20 000	0.20 000	0.23 01	0.38 00	0.39 1	
a_6	0.18 001	0.18 001	0.20 000	0.23 01		
a_3	0.18 010	0.18 010	0.18 001			
a_2	0.03 0110	0.05 011				
a_4	0.02 0111					

Figure 2.2: Procedure to obtain Huffman codes for the original symbols. Virtual symbols are recursively expanded into their composing symbols. In the process, a 0 is appended to the code of the first composing symbol and a 1 is appended to the code of the second composing symbol.

2.3 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.

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. In the following sections we review the most relevant techniques for our work.

2.3.1 Binary Rank and Select

Consider a binary sequence $B[1,n]$. The classic solution [19, 54] is built upon the plain sequence, requiring $o(n)$ additional bits. Generally, $rank_1$ and $select_1$ are considered the default *rank* and *select* queries.

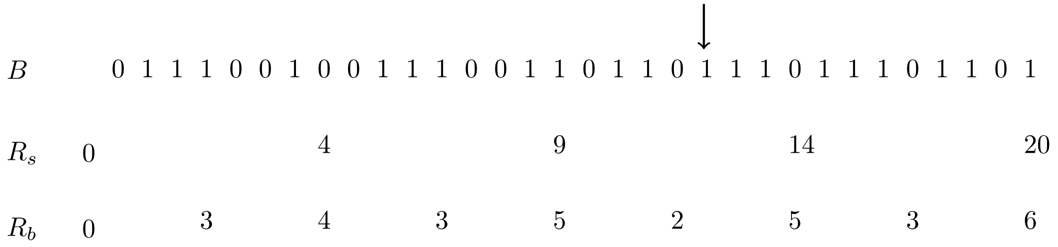
Let us consider the solution for $rank_1$ ($rank_0$ can be directly computed as $rank_0 = i - rank_1$): The idea is based on a two-level dictionary that stores the answers at regular spaced positions plus a small table containing the answer for every sequence that is short enough.

Let us divide B into blocks of size $b = \lfloor \log(n)/2 \rfloor$ and consider also superblocks of size $s = b\lfloor \log n \rfloor$. We build an array R_s that stores the *rank* value at the beginning of each superblock. More precisely, $R_s[j] = rank_1(B, j \times s),\ j = 0 \ldots \lfloor n/s \rfloor$. R_s requires $O(n/\log n)$ bits, because it contains $n/s = O(n/\log^2 n)$ elements of size $\log n$ bits.

We also need an array R_b for the blocks. For each block we will store the relative *rank* with respect to the beginning of the corresponding superblock. More precisely, for each block k contained in a superblock $j = \lfloor k/b \rfloor$, $k = 0 \ldots \lfloor n/b \rfloor$ we store $R_b[k] = rank(B, k \times b) - rank(B, j \times s)$. R_b requires $(n/b)\log s = O(n \log\log n/\log n)$ bits of space.

Finally, a small table R_p stores the *rank* values for any binary sequence of size b. Formally, $R_p[S,i] = rank(S,i)$, for every binary sequence S of size b, $0 \leq i < b$. R_p requires $O(2^b \times b \times \log(b)) = O(\sqrt{n} \log n \log\log n)$ bits. Figure 2.3 shows an example of a *rank* calculation using those data structures.

The idea is similar for answering *select* queries, but the data structures and algorithms are more complicated. To provide the satisfactory performance the samplings need to be regular in the space $[1,n]$ of possible arguments for $select(B,j)$. It is also possible to answer a *select* query by doing a binary search within the *rank* values [36].



$$
\begin{aligned}
rank(B, 21) &= R_s[3] + R_b[6] + R_p[1110, 1] \\
&= 9 + 2 + 1 = 12
\end{aligned}
$$

R_p	1	2	3	4
0000	0	0	0	0
0001	0	0	0	1
⋮	⋮	⋮	⋮	⋮
1110	1	2	3	3
1111	1	2	3	4

Figure 2.3: An example of the additional structures to answer *rank* using $n + o(n)$ bits.

2.3.2 Compressed Binary Rank and Select

Raman, Raman and Rao [65] showed that is also possible to represent B in a compressed form using $nH_0(B)$ bits, with some extra data structures using $o(n)$ bits, and still answer *rank* and *select* queries in constant time.

The main idea is to divide B into blocks of size $b = \lfloor \log(n)/2 \rfloor$. Each block $I = B_{bi+1,bi+b}$ will be represented using a pair (c_i, o_i), where c_i represents the class the block belongs to, and o_i indicates which of the elements of the class the block corresponds to. Each class c is the collection of all the sequences of size b that have c bits with value 1. For example, if $b = 4$, class 0 is $\{0000\}$, class 1 is $\{0001, 0010, 0100, 1000\}$, class 2 is $\{0011, 0101, 0110, 1001, 1010, 1100, \ldots\}$ and class 4 is $\{1111\}$. We need $\lceil \log(b+1) \rceil$ bits to store each value c_i, because there are $b+1$ classes of sequences of size b. To represent o_i it is important to note that class c_i has $\binom{b}{c_i}$ elements, so we require $\lceil \log \binom{b}{c_i} \rceil$ bits to represent each o_i. In order to represent the sequence B we use $\lceil n/b \rceil$ (c_i, o_i) pairs. The total space of the c_i terms is $\Sigma_{i=0}^{\lceil n/b \rceil} \lceil \log(b+1) \rceil = O(n \log(b)/b) = O(n \log\log n / \log n) = o(n)$ bits. The space required to represent the o_i terms is given by [64]:

$$
\begin{aligned}
\left\lceil \log \binom{b}{c_1} \right\rceil + \ldots + \left\lceil \log \binom{b}{c_{\lceil n/b \rceil}} \right\rceil &< \log\left(\binom{b}{c_1} \times \ldots \times \binom{b}{c_{\lceil n/b \rceil}} \right) + n/b \\
&\leq \log \binom{n}{c_1 + \ldots + c_{\lceil n/b \rceil}} + n/b \\
&= \log \binom{n}{m} + n/b \\
&= nH_0(B) + O(n/\log n).
\end{aligned}
$$

Where $m = c_1 + \ldots + c_{\lceil n/b \rceil}$ is the number of 1s in the sequence. In order to answer *rank* and *select* queries we will need additional structures in a similar manner as in the previous section. We will use the same arrays R_s and R_b. Because pairs (c_i, o_i) have variable lengths, this time we will need to build two additional arrays, namely $Rpos_s$ and $Rpos_b$, pointing to the position in the compressed representation of B where each superblock ($Rpos_s$) and each block ($Rpos_b$) begins. These add extra $o(n)$ bits.

The table R_p of the previous section is not useful anymore because we cannot access it through the chunks of the original sequence. This time we construct an analogous table, which will be accessible through (c_i, o_i) pairs. Data structures for *select* are somewhat more complicated but the idea is still the same.

In this way, it is possible to represent a binary sequence B occupying $nH_0(B) + o(n)$ bits and answering *rank* and *select* queries in constant time.

2.3.3 General Sequences: Wavelet Tree

Although there are many solutions for the *rank* and *select* problem on general sequences [6,7, 29,35,38], we will focus on one of the most versatile and useful, namely the wavelet tree [38]. Let us consider a sequence $T = a_1 a_2 \ldots a_n$ over an alphabet Σ.

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 support 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.

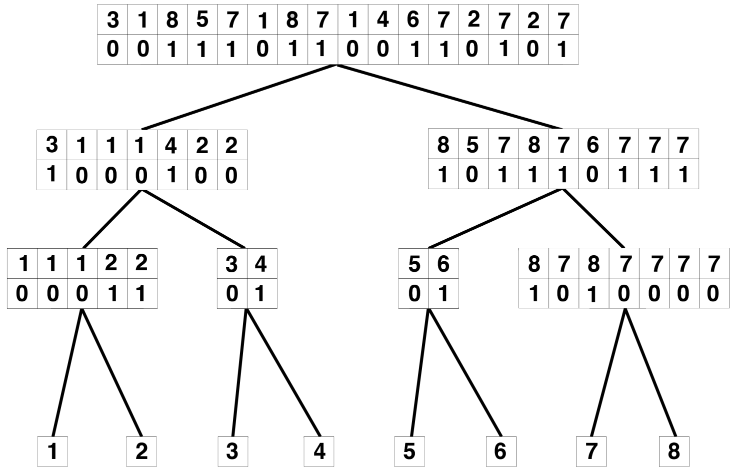


Figure 2.4: Wavelet tree of the sequence $T = 3185718714672727$. We show the mapped sequences in the nodes only for clarity, the wavelet tree only stores the bitmaps.

2.4 Directly Addressable Codes (DAC)

There are many variable-length codes available in the literature to represent sequences in compressed form, like γ-codes, δ-codes, Huffman codes, among many others [76]. When variable-length codes are used, a common problem that arises is that is not possible to access directly the i-th encoded element. This issue is very common for compressed data structures, and the typical solution is to make a regular sampling and store the position of the samples in the encoded sequence. Therefore, decompression from the last sampling suffices to access the i-th element.

Directly Addressable Codes (DAC) [16] is a variant of Vbytes codes [75] that uses variable-length codes reordered so as to allow accessing the i-th element without need of any sampling.

Vbytes codes a number n splitting the $\lfloor \log n + 1 \rfloor$ bits of its binary representation into blocks of b bits. Each block is stored in a chunk of $b + 1$ bits. The extra bit is a flag whose value is 0 in the chunk that contains the least significative bits, and 1 in the others. For instance, if $n = 25 = 11001$ and $b = 3$, then Vbytes needs two chunks, and the representation is $1011, 0001$.

Directly addressable codes consider a sequence of numbers $T[1, n]$, and compute the Vbytes code of each number. The most significative blocks are stored together in an array A_1. The extra bit used as a flag is stored separately in a bitmap B_1 capable of answering *rank* queries. The remaining chunks are stored similarly in arrays A_i and bitmaps B_i keeping together the i-th chunks of the numbers that have them.

2.5 Grammar Compression of Sequences: RePair

Grammar compression techniques replace the original sequence by a context-free grammar that only 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-Hard [18]. However, there are very efficient algorithms that run in linear time and asymptotically achieve the entropy of the sequence, like LZ77 [77], LZ78 [78], and RePair [48] among others.

We focus on RePair [48] because it has proven to fit very well in the field of succinct data structures [37]. RePair is an off-line dictionary-based compressor that achieves high-order compression, taking advantage of repetitiveness and allowing fast random access [48, 61].

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.

Given a sequence T over and alphabet of size σ, a more formal description is to begin with an empty dictionary $\mathcal{R}$ and do the following:

1. Identify 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. Create a new symbol A, add a new rule to the dictionary, $\mathcal{R}(A) \rightarrow ab$, and replace every occurrence of ab in T with A.

3. Repeat 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 $\sigma' = \sigma + |\mathcal{R}|$. Using the proper data structures to account for the frequencies of the pairs the process runs in $O(n)$ time and requires $O(n)$ space [48].

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 $O(|\mathcal{C}[j]|)$.

2.6 Succinct Representation of Trees

How to succinctly encode a tree has been subject to much study [2, 7, 15, 45, 69]. 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 [34, 45, 55, 65, 69], Depth-First Unary Degree Sequence (DFUDS) [15], or Level-Order Unary Degree Sequence (LOUDS) [45].

Arroyuelo et al. [2] 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 it solves many operations in constant time (less than a microsecond 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. Each node with c children is encoded by writing its arity in unary, that is, 1^c0 is appended 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, where the labels are indexed by the *rank* of the node in the bitstring.

2.7 Range Minimum Queries

Range minimum queries (RMQ) are useful in the field of succinct data structures [30, 68]. 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: the first one, called systematic, meaning that the sequence A is available at query time, and the non-systematic setting, meaning that A is no longer available during query time.

Sadakane [68] gave the first known solution for the non-systematic setting, requiring $4n + o(n)$ bits and answering the queries in $O(1)$ time. This solution is based on a succinct representation of the Cartesian tree [73] of A. Later, Fischer and Heun. [30] offered a solution requiring the optimal $2n + o(n)$ bits, and answering the RMQ queries in $O(1)$ time. This is not achieved using the Cartesian tree, but instead they define a data structure called *2d-Min-Heap*, which is at the core of their solution.

Each node of the *2d-Min-Heap* represents a position on the array, and thus corresponds to the value stored in this position. The *2d-Min-Heap* satisfies two heap-like properties: (i) the value corresponding to every node is smaller than the value corresponding to its children, and (ii) the value corresponding to every node is smaller than the value corresponding to every right sibling. Provided a succinct representation of the *2d-Min-Heap*, Fischer and Heun showed how to answer RMQ queries in optimal $O(1)$ time.

2.8 Suffix Trees

The suffix tree is a classic full-text index introduced in 1973 by Weiner [74], providing a powerful data structure that requires optimal space in the non-compressed sense, and supports the counting of the occurrences of an arbitrary pattern P in optimal $O(|P|)$ time, as well as to 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 2.5 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:

i 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;

ii we read all the characters of P and end up at a tree node, which is 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), then all the answers are in the subtree of the *locus* of P; or

iii 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.

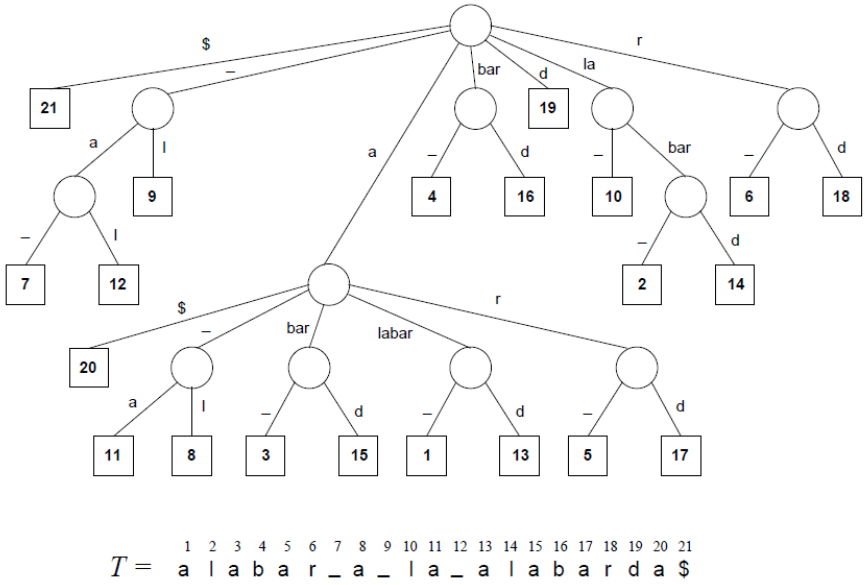


Figure 2.5: Suffix tree of the text “alabar_a_la_alabarda$” [58].

2.9 Suffix Arrays

The suffix array [51] 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 $T[1,n] = t_1t_2 \dots t_n$ 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 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. Algorithm 1 shows the pseudo-code, which takes $O(|P| \log n)$ time to find the interval. Figure 2.6 shows an example.

```
$sp \leftarrow 1$;
$st \leftarrow n + 1$;
while $sp < st$ do
    $s \leftarrow \lfloor (sp + st)/2 \rfloor$;
    if $P > T_{\mathsf{SA}[s],\mathsf{SA}[s]+m-1}$ then
        $sp \leftarrow s + 1$;
    else
        $st \leftarrow s$;
$ep \leftarrow sp - 1$;
$et \leftarrow n$;
while $ep < et$ do
    $e \leftarrow \lceil (ep + et)/2 \rceil$;
    if $P = T_{\mathsf{SA}[e],\mathsf{SA}[e]+m-1}$ then
        $ep \leftarrow e$;
    else
        $et \leftarrow e - 1$;
return $[sp, ep]$
```

Algorithm 1: Binary search used to find the interval associated with the pattern P in the suffix array SA of T. There are $ep - sp + 1$ occurrences of P.

2.10 The Burrows-Wheeler Transform (BWT)

The Burrows-Wheeler transform (BWT) of a text T is a reversible transformation of T which is usually more easily compressible than T itself.

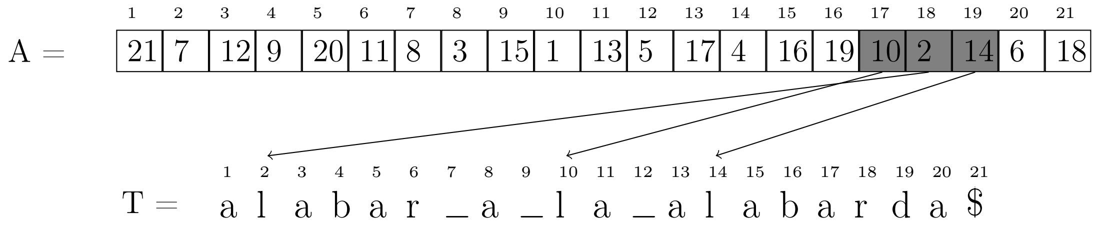


Figure 2.6: Suffix array of the text "alabar_a_la_alabarda$". The shaded interval corresponds to the suffixes that begin with "la".

Let us consider the suffix array of T, $\mathsf{SA}[1,n]$. The BWT of T, T^{bwt}, is defined as follows: when $\mathsf{SA}[i] \neq 1$ $t_i^{bwt} = t_{SA[i]-1}$, otherwise $t_i^{bwt} = t_n = \$$. In other words, the BWT can be obtained beginning with an empty sequence, to which we will append the character preceding each suffix we found in the suffix array.

Another view of the BWT emerges from a matrix M that contains the sequence $t_{i,n}t_{1,i-1}$ in the i-th row. If we sort the rows of the matrix in lexicographical order, the last column of the resulting matrix is the BWT of T. This can be seen in Figure 2.7

In order to obtain the inverse of the BWT it will be necessary to compute functions that allow us to map the last column of M to the first one. For this purpose, let us define F as the first column of M, and $L = T^{bwt}$ the last one. Some useful functions in this chapter are C_{BWT} and Occ, which are defined as follows: $C_{BWT}(c)$ is the number of occurrences in T of characters alphabetically smaller than c; and $Occ(c,i)$ is the number of occurrences of character c in $L_{1,i}$, that is, $Occ(c,i) = rank_c(L,i) = rank_c(T^{BWT},i)$. Now we are ready to define a function called *LF-mapping*, such that $LF(i)$ is the position in F where L_i appears. It has been proved [17,26] that LF can be computed as follows: $LF(i) = C_{BWT}(L_i) + Occ(L_i, i)$.

It is straightforward to obtain the original text from T^{bwt} using the *LF-mapping*: We know that the last character of T is \$, and $F_1 = \$$ because \$ is smaller than any other character. By construction of M, the character that precedes F_i in the text is L_i, thus we know that the character preceding \$ in the text is L_1. In the example shown in Figure 2.7, this is an a. Now we compute $LF(1)$ in order to obtain the position of that a in F and we will obtain that the preceding character in T is $L_{LF(1)}$. In general, we consider $T[n] = \$$ and $s = 1$ and then for each $k = n-1 \ldots 1$, we do $s \leftarrow LF(s)$, $T[k] \leftarrow T^{bwt}[s]$.

2.11 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.

```
                                         F                    L = T^BWT
alabar_a_la_alabarda$                    $alabar_a_la_alabarda
labar_a_la_alabarda$a                    _a_la_alabarda$alabar
abar_a_la_alabarda$al                    _alabarda$alabar_a_la
bar_a_la_alabarda$ala                    _la_alabarda$alabar_a
ar_a_la_alabarda$alab                    a$alabar_a_la_alabard
r_a_la_alabarda$alaba                    a_alabarda$alabar_a_l
_a_la_alabarda$alabar                    a_la_alabarda$alabar_
a_la_alabarda$alabar_                    abar_a_la_alabarda$al
_la_alabarda$alabar_a                    abarda$alabar_a_la_al
la_alabarda$alabar_a_                    alabar_a_la_alabarda$
a_alabarda$alabar_a_l      ⟶             alabarda$alabar_a_la_
_alabarda$alabar_a_la                    ar_a_la_alabarda$alab
alabarda$alabar_a_la_                    arda$alabar_a_la_alab
labarda$alabar_a_la_a                    bar_a_la_alabarda$ala
abarda$alabar_a_la_al                    barda$alabar_a_la_ala
barda$alabar_a_la_ala                    da$alabar_a_la_alabar
arda$alabar_a_la_alab                    la_alabarda$alabar_a_
rda$alabar_a_la_alaba                    labar_a_la_alabarda$a
da$alabar_a_la_alabar                    labarda$alabar_a_la_a
a$alabar_a_la_alabard                    r_a_la_alabarda$alaba
$alabar_a_la_alabarda                    rda$alabar_a_la_alaba
```

Figure 2.7: Matrix M to obtain the BWT of the text $T =$ "alabar_a_la_alabarda$". Note that the last column $L =$ "araadl_ll$_bbaar_aaaa" is T^{bwt} and the first one, F, corresponds to the characters pointed by the suffix array of the text.

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 the space requirement 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*.

There are mainly three families of self-indexes, namely the FM-Index [26, 29, 50, 58], the Compressed Suffix Array [38, 39, 66, 67], and the LZ-Index [3, 27, 46, 57]. For every family there are many variations, and there have been many works [20, 22, 24, 25, 58] showing that the choice of the best self-index, both in theory and in practice, will depend on the scenario (which kind of data is going to be indexed, which kind of query we will use more often, etc.).

For some document retrieval applications of self-indexes we will need not only to answer the queries defined above, but also need to simulate the suffix array; that is, to compute $\mathsf{SA}[i]$ and its inverse, $\mathsf{SA}^{-1}[i]$, efficiently. We will focus on those families of self-indexes that are capable of such computation, namely the Compressed Suffix Array and the FM-Index. Table 2.1 briefly sketches their space and time requirements. We will refer to any of the members of those two families as CSA.

Index	Size	t_{SA}	$search(P)$
CSA [66]	$\frac{1}{\varepsilon}n(H_0+1)+o(n\log\sigma)$	$O(\log^{\varepsilon} n)$	$O(\lvert P\rvert\log n)$
CSA [38]	$(1+\frac{1}{\varepsilon})nH_k+o(n\log\sigma)$	$O(\log^{\varepsilon} n)$	$O(\lvert P\rvert\log\sigma+\text{polylog}(n))$
FM-Index [29]	$nH_k+o(n\log\sigma)$	$O(\log^{1+\varepsilon} n)$	$O(\lvert P\rvert\frac{\log\sigma}{\log\log n})$

Table 2.1: Summary of some of the main self-indexes that will be useful for this thesis, 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. ε is any constant greater than 0.

2.11.1 FM-Index and Backward Search

The FM-Index is the common name for a family of self-indexes whose core is the Burrows-Wheeler Transform (BWT), which can be used to find the occurrences of the pattern and to recover the text itself. The BWT can be compressed using many techniques. Depending on the choice made to represent the BWT, different space-time trade-offs emerge [26, 29, 50, 58].

Backward Search and Burrows-Wheeler Transform

We have shown in Section 2.9 how to find the occurrences of an arbitrary pattern $P = p_1p_2\dots p_m$ in a text T using the suffix array of T, SA. Backward search allows us to do the same, but this time using a different approach. The first step is to find the interval $[sp_m, ep_m]$ in SA pointing to every suffix beginning with the last character of P, p_m.

The function C_{BWT}, defined in Section 2.10, allows us to find this interval using the following identity: $[sp_m, ep_m] = [C_{BWT}(p_m)+1, C_{BWT}(p_m+1)]$. Now, given $[sp_m, ep_m]$, we need to find $[sp_{m-1}, ep_{m-1}]$, the interval in the suffix array pointing to those suffixes that begin with $p_{m-1}p_m$. Note that $[sp_{m-1}, ep_{m-1}]$ is a subinterval of $[C_{BWT}(p_{m-1})+1, C_{BWT}(p_{m-1}+1)]$.

In general, given $[sp_{i+1}, ep_{i+1}]$, we need to find $[sp_i, ep_i]$. The key tool for this purpose is the LF function defined in Section 2.10. We will use the fact that the occurrences of p_i in $L[sp_{i+1}, ep_{i+1}]$ appear contiguously in F, preserving their relative order. In order to find the new interval we would like to find the first and the last occurrence of p_i in $L[sp_{i+1}, ep_{i+1}]$. Thus, we are looking for b and e such that $L_b = p_i$ and $L_e = p_i$ are the first and the last occurrence of p_i in $L[sp_{i+1}, ep_{i+1}]$, respectively. Then, $LF(b)$ and $LF(e)$ are the first and the last row in F that begins with p_i followed by $p_{i+1} \dots p_m$, that is, $sp_i = LF(b)$ and $ep_i = LF(e)$. Let us show that we do not need to know the precise values of b and e:

$$\begin{aligned} LF(b) &= C_{BWT}(L_b) + rank_{L_b}(T^{bwt}, b) && (2.1)\\ &= C_{BWT}(p_i) + rank_{p_i}(T^{bwt}, b) && (2.2)\\ &= C_{BWT}(p_i) + rank_{p_i}(T^{bwt}, sp_{i+1} - 1) + 1. && (2.3) \end{aligned}$$

Equation (2.1) is a consequence of the properties shown in Section 2.10. Equation (2.2) holds because the character that appears at the position b of T^{bwt} is precisely the character that we are looking for, p_i. Finally, Equation (2.3) holds because b is the first occurrence of p_i in the interval. Analogously, for $LF(e)$ we know that the occurrences of p_i up to the position e are the same up to the position ep_{i+1} because by definition e is the last position in L the range $[sp_{i+1}, ep_{i+1}]$ of an occurrence in p_i. Therefore, it holds

$$LF(e) = C_{BWT}(p_i) + rank_{p_i}(T^{bwt}, ep_{i+1}). \qquad (2.4)$$

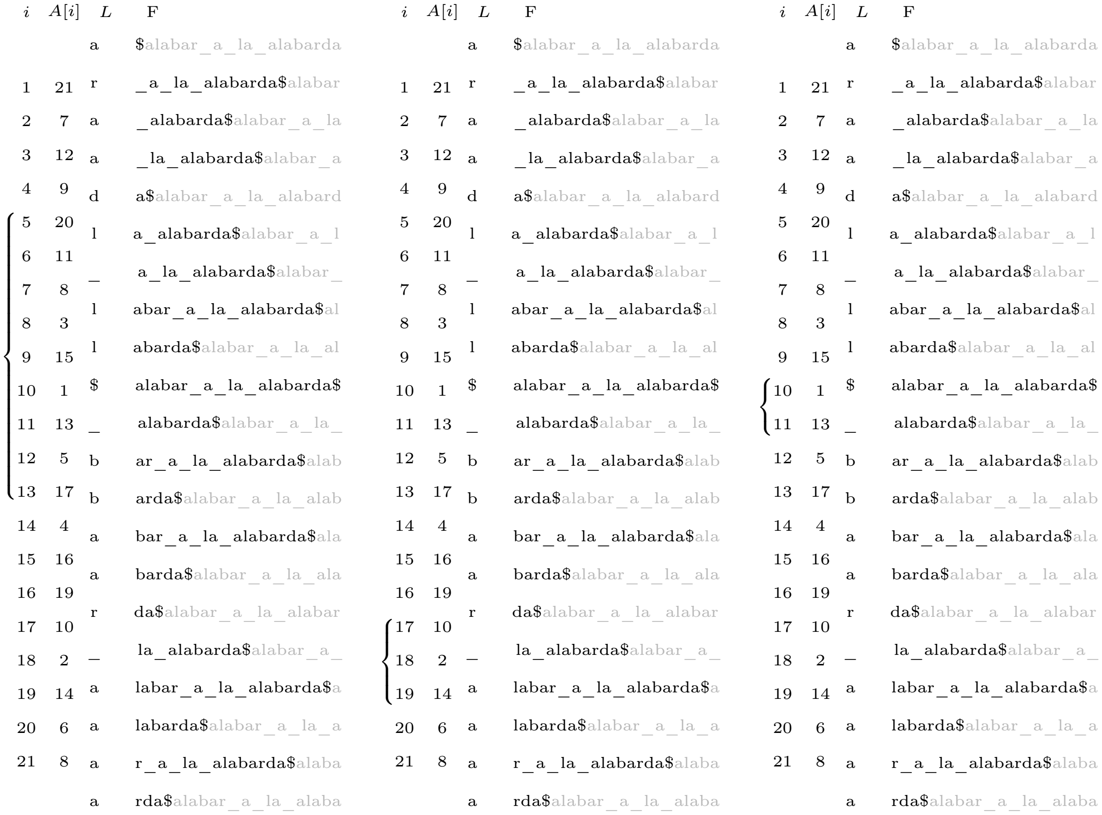
The pseudo code of the backward search is shown in Algorithm 2. Figure 2.8 shows an example.

```
$sp \leftarrow C_{BWT}(p_m) + 1$;
$ep \leftarrow C_{BWT}(p_m + 1)$;
for $i \leftarrow |P| - 1$ to 1 do
    $sp \leftarrow C_{BWT}(p_i)+ rank_{p_i}(T^{BWT},p_i,sp -1) + 1$;
    $ep \leftarrow C_{BWT}(p_i)+ rank_{p_i}(T^{BWT},p_i,ep )$;
    if $sp > ep$ then
        return $\phi$
return $[sp, ep]$
```

Algorithm 2: Backward search of suffixes that begin with $P_{1,m}$.

Encoding the Burrows-Wheeler Transform

If we store C_{BWT} as a plain array, it requires $\sigma \log n$ bits. The search time of Algorithm 2 is dominated by the time required to calculate $2m$ times the function $rank_{p_i}(T^{bwt}, c)$. There



i	A[i]	L	F
1	21	a	$alabar_a_la_alabarda
2	7	r	_a_la_alabarda$alabar
3	12	a	_alabarda$alabar_a_la
4	9	a	_la_alabarda$alabar_a
5	20	d	a$alabar_a_la_alabard
6	11	l	a_alabarda$alabar_a_l
7	8	_	a_la_alabarda$alabar_
8	3	l	abar_a_la_alabarda$al
9	15	l	abarda$alabar_a_la_al
10	1	$	alabar_a_la_alabarda$
11	13	_	alabarda$alabar_a_la_
12	5	b	ar_a_la_alabarda$alab
13	17	b	arda$alabar_a_la_alab
14	4	a	bar_a_la_alabarda$ala
15	16	a	barda$alabar_a_la_ala
16	19	r	da$alabar_a_la_alabar
17	10	_	la_alabarda$alabar_a_
18	2	a	labar_a_la_alabarda$a
19	14	a	labarda$alabar_a_la_a
20	6	a	r_a_la_alabarda$alaba
21	8	a	rda$alabar_a_la_alaba

i	A[i]	L	F
1	21	a	$alabar_a_la_alabarda
2	7	r	_a_la_alabarda$alabar
3	12	a	_alabarda$alabar_a_la
4	9	a	_la_alabarda$alabar_a
5	20	d	a$alabar_a_la_alabard
6	11	l	a_alabarda$alabar_a_l
7	8	_	a_la_alabarda$alabar_
8	3	l	abar_a_la_alabarda$al
9	15	l	abarda$alabar_a_la_al
10	1	$	alabar_a_la_alabarda$
11	13	_	alabarda$alabar_a_la_
12	5	b	ar_a_la_alabarda$alab
13	17	b	arda$alabar_a_la_alab
14	4	a	bar_a_la_alabarda$ala
15	16	a	barda$alabar_a_la_ala
16	19	r	da$alabar_a_la_alabar
17	10	_	la_alabarda$alabar_a_
18	2	a	labar_a_la_alabarda$a
19	14	a	labarda$alabar_a_la_a
20	6	a	r_a_la_alabarda$alaba
21	8	a	rda$alabar_a_la_alaba

i	A[i]	L	F
1	21	a	$alabar_a_la_alabarda
2	7	r	_a_la_alabarda$alabar
3	12	a	_alabarda$alabar_a_la
4	9	a	_la_alabarda$alabar_a
5	20	d	a$alabar_a_la_alabard
6	11	l	a_alabarda$alabar_a_l
7	8	_	a_la_alabarda$alabar_
8	3	l	abar_a_la_alabarda$al
9	15	l	abarda$alabar_a_la_al
10	1	$	alabar_a_la_alabarda$
11	13	_	alabarda$alabar_a_la_
12	5	b	ar_a_la_alabarda$alab
13	17	b	arda$alabar_a_la_alab
14	4	a	bar_a_la_alabarda$ala
15	16	a	barda$alabar_a_la_ala
16	19	r	da$alabar_a_la_alabar
17	10	_	la_alabarda$alabar_a_
18	2	a	labar_a_la_alabarda$a
19	14	a	labarda$alabar_a_la_a
20	6	a	r_a_la_alabarda$alaba
21	8	a	rda$alabar_a_la_alaba

Figure 2.8: Backward search of the pattern “ala” in the text “alabar_a_la_alabarda$” using the BWT.

are several proposals of encoding techniques for the BWT that allow one to compute *rank* of characters quickly [26,28,29,49,50,58]. One of the most successful encoding schemes for that purpose is the wavelet tree. The wavelet tree was used by Ferragina et al. [28] to develop a variant of the FM-Index, the *Alphabet Friendly FM-Index* [28], and also by Mäkinen and Navarro to develop the *Run Length FM-Index* (RL-FMI) [49]. Both indexes count the occurrences of a pattern in time $O(m \log \sigma)$ and use space proportional to the k-th order entropy of the text.

Later, Mäkinen and Navarro [50] showed that using a wavelet tree that encodes its bitmaps using zero-order entropy [65] is enough to encode T^{bwt} using $nH_k(T) + o(n \log \sigma)$ bits, for any $k \leq \alpha \log \sigma$ and constant $\alpha \leq 1$, without any further refinement. Later, Claude and Navarro showed the practicality of this scheme [21].

2.11.2 CSA based on Function Ψ

A different line of research [38,39,66,67] is based on the function Ψ, which is the inverse of the LF function used in the FM-Index (see Sections 2.10 and 2.11.1). Function Ψ maps the suffix $t_{\mathsf{SA}[i],n}$ to the suffix $t_{SA[i]+1,n}$ inside SA.

Grossi and Vitter's [39] representation of the suffix array reduces the space requirement from $n \log n$ to $O(n \log \sigma)$ bits. However, to find the occurrences of a pattern P they still need access to the text T. Sadakane [66,67] proposes a representation that allows one to efficiently find the occurrences P without accessing T at all. The search algorithm is carried out in a way very similar to the binary search algorithm shown in Section 2.9, emulating the access to SA and T with the compressed structures instead of accessing the original ones.

The main data structures required by the CSA are the function Ψ, the array C_{BWT} defined in Section 2.10, and samplings of the suffix array and inverse suffix array. Sadakane proposed a hierarchy to store the data structures using bitmaps to signal which values are stored in the next level. For the sake of clarity, we present the CSA in a more practical fashion, closer to the actual implementation of Sadakane.

Given the suffix array SA of T, the function Ψ is defined so that $\mathsf{SA}[\Psi(i)] = SA[i] + 1$, if $\mathsf{SA}[i] < n$. When $\mathsf{SA}[i] = n$ it is defined $\mathsf{SA}[\Psi(i)] = 1$, so Ψ is actually a permutation. This definition of Ψ allows one to traverse (virtually) the text from left to right. This is done by jumping in the suffix array domain from the position that points to t_i to the position that points to t_{i+1}. Because T is not stored we simulate its access on the suffix array, and we need a way to know which is the corresponding character in the text. That is, given a position i, we need to know the character $T[SA[i]]$. The function C_{BWT} (recall Section 2.10) is useful for this purpose. If $t_{SA[i]} = c$, then $C_{BWT}(c) < i \leq C_{BWT}(c+1)$. However, instead of storing C_{BWT} itself Sadakane uses a bitmap $newF[1,n]$ such that $newF[1 + C[i]] = 1$ for every $i \in \{1, \ldots, \sigma\}$, and an array $S[1,\sigma]$ that stores in $S[j]$ th j-th different character (in lexicographic order) appearing in T. With these structures we can compute the desired character c in constant time with the following formula: $c = S[rank(newF, i)]$. Finally, samples of the suffix array an its inverse are stored in arrays SA' and SA'^{-1} respectively. The sampling is regular on the text: They mark one text position out of $\log^{1+\varepsilon}$n for a given constant $\varepsilon > 0$, and collect the SA values pointing to those positions in the array SA'. The corresponding inverse SA^{-1} values are also collected in SA'^{-1}. In order to find out if a given value $\mathsf{SA}[i]$ is sampled or not, they store a bitmap $mark[1, n]$ such that $mark[i] = 1$ if and only if the $\mathsf{SA}[i]$ value is stored in SA'.

In order to reconstruct a substring of length l starting from position i, $T[i, i + l - 1]$ we need to find the rightmost sample of the text before position i. This position is $\lfloor i/\log^{1+\varepsilon} n\rfloor$. Using the (sampled) inverse suffix array it is possible to find the position in the suffix array that points to this sample. That is $p = SA'^{-1}[\lfloor i/\log^{1+\varepsilon} n\rfloor]$. Then, we iteratively apply function Ψ up to reaching the position p' in the suffix array that points to $T[i]$. That is $p' = \Psi^r(p)$, where $r = i - \lfloor i/\log^{1+\varepsilon} n\rfloor \log^{1+\varepsilon} n$.

To obtain the first character of the substring we use $t_i = S[rank(newF, p')]$. Then we move (virtually) to the right with $p'' = \Psi(p')$ so we obtain $t_{i+1} = S[rank(newF, p'')]$. After repeating this procedure l times we obtain the desired substring $T[i, i + l - 1]$. The time is $l + \log^{1+\varepsilon} n$ times the cost of calculating Ψ and the *rank* functions. If we have constant time access for both we are able to retrieve any substring of length l in $O(l + \log^{1+\varepsilon} n)$ time.

Given a pattern P, the search algorithm is essentially the same shown in Section 2.9: Two binary searches are made on the suffix array, comparing the current position with the pattern. Those binary searches give us the begin and the end of the $[sp, ep]$ interval.

Given a position s in the suffix array domain, to find the character of the text corresponding to $t_{SA[s]}$, we compute $t_{A[s]} = S[rank(newF, s)]$ in constant time. In this way we can extract as many characters of the suffix as needed to complete the lexicographic comparison with P. Therefore, the desired interval $[sp, ep]$ is obtained in time $O(|P| \log n)$.

To locate each of the occurrences the procedure is as follows: given a position p in $\mathsf{SA}[sp, ep]$ we apply Ψ recursively until we find the next position $p' = \Psi^r(p)$ marked in *mark*. Then, $\mathsf{SA}[p] = SA[\Psi^r(p)] - r = SA'[rank(mark, \Psi^r(p))] - r$. Therefore, locating the $occ = ep - sp + 1$ occurrences of P in T requires $O(m \log n + occ \log^{1+\varepsilon} n)$ time.

The samplings and marking array require $O(n/\log^{\varepsilon} n)$ bits. The structures used to emulate C_{BWT}, that is, $newF$ and S, require $n + o(n)$ and $\sigma \log \sigma$ bits, respectively.

The most space-consuming structure is Ψ. If we stored it in plain form it would require $n \log n$ bits. Grossi and Vitter [39] showed that Ψ is monotonically increasing in the areas of SA that point to suffixes starting with the same character, and therefore it can be encoded in reduced space. One possibility [67] is to use Elias-δ codes to represent Ψ. This requires $nH_0(T) + O(n \log \log \sigma)$ bits, and supports the computation of $\Psi(i)$ in constant time. In this way, the CSA requires $nH_0(T) + O(n \log \log \sigma)$ bits of space. Grossi, Gupta and Vitter [38] reduced the space to $(1+\frac{1}{\varepsilon})nH_k + o(n \log \sigma)$, however the time required for searching a pattern raises to $O(|P| \log \sigma + \text{polylog}(n))$.

2.12 Information Retrieval Concepts

In this section we sketch some basic concepts of Information Retrieval. In particular we present the Inverted Index, which is the main data structure behind most information retrieval systems.

Inverted indexes have been very successful for *natural languages*. This success is closely related with the structure of the inverted index itself: it can be roughly explained as a set of lists that store the occurrences of every different word that appears in the collection. Therefore, the inverted index approach requires that the documents are tokenizable into words. Words are referred to as *index terms* because the process of tokenization is not straightforward, and deciding which substrings are going to be considered as retrievable units (i.e., conceptually meaningful) is also part of the indexing process.

2.12.1 Inverted Indexes

The inverted index (or inverted file) is probably the most important data structure for document retrieval in information retrieval systems. It has proven to work very well for document collections of natural language. The inverted index is well suited to answer word queries, bag-of-word queries, proximity queries, and so on.

The inverted index stores mainly two data structures: (1) The *vocabulary* V (also called *lexicon* or *dictionary*), which stores all the different words that exist in the collection and (2) the *occurrences* (also called *posting lists* or *posting file*), which stores for each word of the vocabulary the list of documents where this word occurs.

The inverted index we have presented so far is not powerful enough to answer more sophisticated queries. It will only retrieve the documents in which a word appears

Usually the inverted lists are enriched with extra information, such as the positions in the document where the word appears (to enable, for example, proximity queries). If no such information is provided, the model is usually referred as *bag of words*. In order to return documents most likely to be relevant for the user, some measure of relevance can be stored within each document in the inverted lists. The inverted lists are usually sorted in decreasing order according to a relevance measure, thus enabling fast answers to queries.

2.12.2 Answering queries

When the query pattern is formed only by one word, the inverted index simply looks for the inverted list corresponding to the query word, and retrieves those documents.

When the queries are composed by two or more words we need to consider, at least, two scenarios: conjunctive (AND operator) and disjunctive (OR operator) queries. When a conjunctive query is performed, the documents in the output must contain all the query patterns. When a disjunctive query is performed, the documents to be delivered contain at least one of the query terms.

In order to answer conjunctive queries, the inverted lists of all the query terms are obtained. Then, the intersection of those lists needs to be computed. There are many algorithms to compute such kind of intersections efficiently, depending on the nature of the lists [4,5,8,9,71]. To answer disjunctive queries the answer is obtained computing the union of the lists of all the query terms.

If the lists are sorted according to a given relevance measure, answering a top-k query is not difficult when the pattern is formed only by one query term: once the proper list is found, the first k documents are retrieved. To answer top-k queries for multiple-word queries the intersection and union algorithms become more complex [4].

2.12.3 Relevance Measures

The existence of a *ranking* among the documents is critical for information retrieval systems. For instance, a ranking is needed to define the notion of top-k queries. Because information systems aim to retrieve *meaningful* information, the *quality* of such a ranking is also a very important matter. There are many relevance measures available in the literature. We will briefly review the most fundamentals measures.

Term frequency is probably the simplest relevance measure, and it is just the number of times that a given word w_i appears in certain document j, $TF_{i,j}$. Several variants of this measure have been proposed, like normalized term frequency, which is $TF_{i,j}/|d_j|$, where $|d_j|$ is the length of the document j.

Plain term frequency has some drawbacks on queries formed by various words. Consider a word w_i that appears in every document of the collection, and it appears the same number of times in every one of them (think about articles or prepositions). Then consider another word w_j that only appears in one document, and it appears the same number of times as w_i. If we rank the documents for a query following only *term frequency* we will assign the same rank to both results. A measure that aims to overcome this problem is the *Inverse Document Frequency* $IDF_i = \log \frac{|D|}{DF_i}$, where DF_i is the total number of documents containing the term w_i and $|D|$ is the number of documents. *IDF* attempts to measure how common or rare the term w_i is across all documents. This is combined with term frequency, into a relevance measure that has become very popular, namely $TF \times IDF = TF_{i,j} \times IDF_i$.

There are other relevance measures that are more sophisticated and arise from different probabilistic models, such as BM25. These are usually variants of the $TF \times IDF$ measure. In some contexts static ranks are also used (alone or as a part of the relevance formula). A popular example is PageRank. The measures we have presented so far correspond to the vectorial model, however there are other less popular approaches, such as latent semantic indexing (LSI), probabilistic models, etc. [4].

Despite all the variations, the contributions of this thesis are focused on one-word queries.

In such scenario, *IDF* is irrelevant and we will use *TF* as the relevance criterion.

2.12.4 Heaps' Law

A very important issue for inverted indexes is the number of different words that appear in the document collection, that is, the size of the *vocabulary* V. In information retrieval the standard model to predict the growth of the vocabulary size in natural language texts is the assumption of Heaps' Law [4,40]. Heaps' Law states that the size of the vocabulary of a text of n words is given by $|V| = Kn^{\beta} = O(n^{\beta})$, where K and β depend on the particular text. K is typically between 10 and 100, and β is a positive number smaller than one [4]. Therefore, the size of the vocabulary is sub-linear in the size of the text.

Heaps' law is exactly what makes inverted indexes feasible: term indexes are the basic token of queries, and it is possible to precompute and store the answer for every different token in linear space. When a query is made of many tokens the task at query time is to obtain the desired results using the precomputed results for each term. However, if we want to enable queries in which the basic terms (or words) are arbitrary substrings of the collection, Heaps' Law does not apply. Instead, the size of the vocabulary becomes quadratic and the approach is not feasible anymore.

Chapter 3

Full Text Document Retrieval

The self-indexes described in Chapter 2 are built over a text T to solve *pattern matching* queries, that is, to count and locate the occurrences of an arbitrary pattern P. A more interesting query for information retrieval would consist in, given a pattern P, finding the documents where P appears (all of them, or the most relevant ones). Although relevance can be defined in many ways, most of the relevance criteria are based on the *term frequency* (*TF*), that is, the number of occurrences of P in a given document.

From now on we will consider a collection of D documents $\mathcal{D} = \{d_1, d_2, \ldots, d_D\}$, $\sum_1^D |d_i| = n$, and we will call T their concatenation. We will assume that every document i ends with a unique end-marker $\$_i$, satisfying $\$_i < \$_j$ iff $i < j$ and $\$_i < c, \forall c \in \Sigma$. For an arbitrary pattern P, the following queries are defined:

- *Document Listing(P):* Return the ndoc documents that contain P.
- *Document Listing(P) with frequencies:* Return the ndoc 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. That is, the k documents with highest $TF_{P,d}$ value.

For the context of natural language collections and word queries, the inverted index (Section 2.12.1) will solve these queries very efficiently. However, when the query pattern may be an arbitrary concatenation of characters, the inverted index approach is not useful. In the following sections we cover the previous work that addresses these queries in the scenario of general sequences.

3.1 Non-Compressed Approach

Muthukrishnan [56] proposed the first solution to the document listing problem that is optimal-time and linear-space. His solution is built upon Weiner's suffix tree (see Section 2.8), and introduces a new data structure for document retrieval problems: the so-called *document array*.

Mutukrishnan'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 ndoc, the number of different documents where P occurs. An optimal algorithm would answer the query in $O(|P| + \mathsf{ndoc})$ time.

For this purpose, Muthukrishnan [56] 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 2.7). The space required by the C array and the RMQ structure is $O(n \log n)$ bits.

To understand Mutukrishnan's [56] 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 [1]. The total time is $O(|P| + \mathsf{ndoc})$, which is optimal for document listing.

An optimal solution to top-k document retrieval has required more work. Hon et al. [43] 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 [59] 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.

[1] Using the standard word random access model (RAM); the computer word size w is assumed to be such that $\log n = O(w)$.

3.2 Wavelet Tree Based Approach

Although the space of Mutukrishnan's solution is linear in the input size (i.e., $O(n)$ machine words) it is still impractical on a collection that requires only $n \log \sigma$ bits. There have been various approaches to reduce the space of this solution. The suffix tree or suffix array of T can be easily replaced by any of the self-indexes presented in Section 2.11. As stated in Section 2.11, we will refer to any of those indexes as a compressed suffix array (CSA). All the algorithms presented in this section will use the 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 representation of the arrays DA and C, and how to answer document retrieval queries given the $[sp, ep]$ interval. The rest of this chapter surveys the main techniques for document listing and top-k queries.

3.2.1 Emulating Muthukrishnan's algorithm

Mäkinen and Välimäki [72] were the first in using a wavelet tree (Section 2.3.3) 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 2.3.1), 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 $O(search(P) + \mathsf{ndoc} \log D)$. The wavelet tree also allows them to compute the frequency of P in 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 algorithms.

3.2.2 Document Listing with TF via Range Quantile Queries

Later, Gagie et al. [33] showed that the wavelet tree is 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 continues 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 $ep = rank_1(root, ep)$ and recurses to the right. When a leaf is reached, its corresponding value is the kth smallest value in the subinterval, therefore it is returned.

To solve the document listing problem, Gagie et al. considered the wavelet tree of the document array, and successive applications of the range quantile query. 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 $O(\mathsf{ndoc} \log D)$.

3.2.3 Document Listing with *TF* via DFS

Culpepper et al. [23] pointed out that the essence of Gagie et al.'s algorithm was a depth-first search, and by recasting the implementation the performance was improved to $O(\mathsf{ndoc} \log \frac{D}{\mathsf{ndoc}})$ time. They also made a practical improvement over Gagie et al. algorithm that obtains the frequencies for free.

The algorithm begins in the bitmap corresponding to the root of the wavelet tree, B_{root}, 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 recurses 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.

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 ndoc times) while the DFS visits them at most twice, avoiding multiple visits to some nodes. The DFS traversal requires $O(\mathsf{ndoc} \log \frac{D}{\mathsf{ndoc}})$ time to list all the documents in the interval [32].

Figure 3.1 shows an example.

3.2.4 Top-k via Quantile Probing

Culpepper et al. [23] explored different heuristics to solve the top-k document retrieval problem using a wavelet tree. *Quantile probing* is based on the observation that in a sorted array $X[1, \ldots, m]$, if there exists a $d \in X$ with frequency larger than f, 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 finishes.

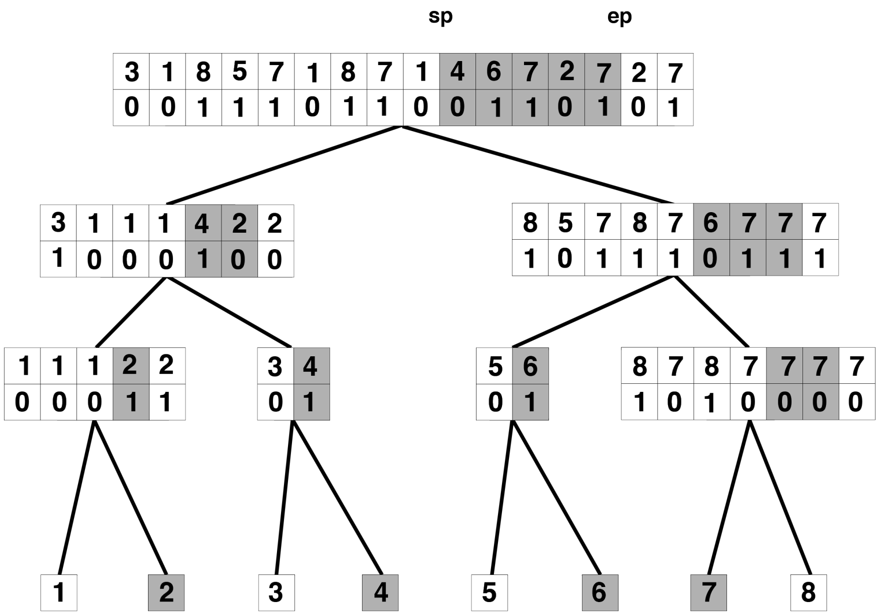


Figure 3.1: Wavelet tree representation of a document array and depth-first search traversal to perform document listing with frequencies. Shaded regions show the intervals $[sp, ep]$ mapped in each node.

3.2.5 Top-k via Greedy Traversal

Among the top-k techniques presented by Culpepper et al. [23] using a wavelet tree, their so-called "greedy traversal" was the best in practice.

The key idea is that, if a document has a high $TF_{P,d}$ value, then the number of bits used in the bitvectors on the path to that document leaf in the wavelet tree should also be large.

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. Thus, the documents will be reported in decreasing TF order until having output k documents. In this way, depending on the distribution of frequencies of the documents, the algorithm can avoid having to inspect a large part of the tree that would be explored if a depth-first search had been performed. Although very good in practice, there are no time guarantees for this algorithm beyond the DFS worst-case complexity, $O(\mathsf{ndoc} \log(D/\mathsf{ndoc}))$.

3.3 CSA Based Approach

A parallel development started with Sadakane [68]. 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 2.3.1). 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 (see Section 2.3.2).

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 adresses the same problem using just $2n + o(n)$ bits [30], see Section 2.7). Therefore, the overall space required for document listing was that of the CSA plus $O(n)$ bits. The time is $O(search(P) + \mathsf{ndoc} \cdot t_{\mathsf{SA}})$, 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. [43] showed how to reduce the extra space to just $o(n)$ by sparsifying the RMQ structure, impacting the time with a factor $\log^{\varepsilon} n$ for listing each document, for some constant $0 < \varepsilon < 1$. To accomplish this, instead of building the RMQ structure over C, Hon et al. build the RMQ structure over a sampled version of C, C^*. They consider chunks of size $b = \log^{\varepsilon} n$ 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 $O(search(P) + \mathsf{ndoc} t_{\mathsf{SA}} \log^{\varepsilon} n)$.

3.3.1 Document Listing with TF via Individual CSAs

Sadakane [68] 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 behind his solution 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 looking for the pattern in the corresponding CSA of the document d, CSA_d. This would take $O(\mathsf{ndoc}(search(P) + t_{\mathsf{SA}}))$ 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. To enumerate the rightmost indices j of all distinct values in $\mathsf{DA}[sp, ep]$, Sadakane uses range maximum queries over C', which are solved by using range minimum queries on the negative values $-C'[i]$. As for C, C' is not stored but only the RMQ structure over it.

Using these RMQ structures, for each document d in the output, Sadakane finds the leftmost l_d and the rightmost r_d indices in $[sp, ep]$ where $\mathsf{DA}[l_d] = \mathsf{DA}[r_d] = d$ (a sorting step of time $O(\mathsf{ndoc} \log\log \mathsf{ndoc})$ is needed to match the same documents d in both lists).

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 $z = select(B, d)$. 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 $O(t_{\mathsf{SA}})$. Figure 3.2 illustrates the mapping process.

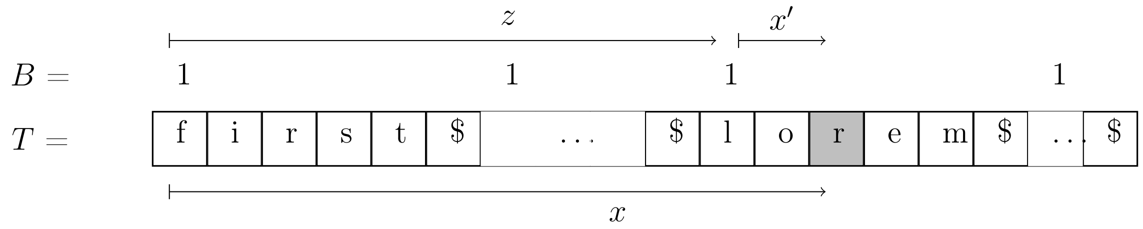


Figure 3.2: Illustration of the relationship among global and document's CSAs and how the mapping is performed. SA_d points to the positions in document d and SA points to positions in T, the concatenation of documents. If $x' = SA_d[j]$, and $x = SA[j']$ refer to the same suffix, and $z = select(B, d)$, then it holds that $x = z + x'$.

Thus, Sadakane solved the document listing with frequencies problem in $O(search(P) + \mathsf{ndoc}(t_{\mathsf{SA}} + \log\log \mathsf{ndoc}))$ time (resorting to a sorting algorithm of Anderssen et al. [1]). The space needed is $2|\mathsf{CSA}| + O(n)$ bits.

3.3.2 Top-k via sparsification

Hon et al. [43] 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 precomputing some top-k answers 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. In the first pass they mark n/g nodes, and 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 $g = \Theta(k \log^{2+\varepsilon} n)$, for an arbitrary $\varepsilon > 0$, the space of the τ_k tree is $O(n/\log^{1+\varepsilon})$. Adding the space required by the τ_k trees for $k = 1, 2, 4, 8, 16, \ldots$, the total amount of space is $O(n/\log^{\varepsilon} n) = o(n)$ 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 $d = rank_1(B, \mathsf{SA}[i])$). 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 notice that, for any position j in CSA_d, it is possible to identify the position j' in CSA such that $\mathsf{SA}_d[j]$ and $\mathsf{SA}[j']$ refer to the same suffix of document d. This is done via SA and SA^{-1} operations on both the global CSA and in CSA_d in $O(t_{\mathsf{SA}})$ time (and also constant number of *rank* end *select* operations on the bit-vector B) as follows: The position in T where T_d begins is simply $z = select(B, d)$. Then the respective position in T is given by $x = z + \mathsf{CSA}_d(j)$. Finally, $j' = \mathsf{CSA}^{-1}(x)$. Figure 3.2 illustrates the process.

Also they observe that the suffixes of document d have the same relative rank in SA and SA_d. Thus, for each position i in $[sp, L) \cup (R, ep]$, they first find $\mathsf{SA}[i]$ and thus its corresponding document d. Then they perform an exponential search using CSA and CSA_d to find the range $[sp_d, ep_d]$ in SA_d that corresponds to those suffixes of d stored within $[sp, ep]$ in SA. 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 $[sp, ep]$.

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 [43]. They find the frequencies of these documents, each costing $O(t_{\mathsf{SA}} \ \log n)$ time, so the total time taken by this is $O(t_{\mathsf{SA}} \ k \log^{3+\varepsilon} n)$. 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.'s [43] data structure requires $2|\mathsf{CSA}| + D\log(n/D) + o(n)$ bits and retrieves the k most frequent documents in $O(search(P) + k \ t_{\mathsf{SA}} \ \log^{3+\varepsilon} n)$ time, for any $\varepsilon > 0$.

3.4 Hybrid Approach for top-k Document Retrieval

Gagie et al. [31] pointed out that in fact 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 $O(\log D)$ time too. This solves operation (2) with $rank_{\mathsf{DA}[i]}(\mathsf{DA}, ep) - rank_{\mathsf{DA}[i]}(\mathsf{DA}, sp-1)$. With the obvious disadvantage of the considerable extra space to represent DA, this solution improves on Hon et al.'s (Section 3.3.2) replacing $t_{\mathsf{SA}} \log n$ by $\log D$ in the query time. They show many other combinations that solve (1) and (2). One of the fastest uses Golynski et al.'s representation [35] of DA, which within the same $n \log D + o(n \log D)$ bits improves $t_{\mathsf{SA}} \log n$ to $\log \log D$ in the time of Section 3.3.2. Very recently, Hon et al. [42] 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 $O(search(P) + k(\log k + (\log \log n)^{2+\varepsilon}))$. Although promising, this structure has not yet been implemented.

3.5 Monotone Minimal Perfect Hash Functions

A recent work by Belazzougui and Navarro [12] presented a new approach to document retrieval based on monotone minimal perfect hash functions (mmphf) [10, 11]. 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 2.3.1. One variant of mmphf data structure over B 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 $O(\log \log(n/m))$ [10].

Belazzougui and Navarro's approach for document listing with frequencies is similar to the method of Sadakane [68] (see Section 3.3.1). They use the CSA of the whole collection, T, the RMQ structures over the C and C' arrays, but instead of the individual CSAs, they make use of mmphfs.

For each document d they store 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 $\sum_d O(n_d \log \log(n/n_d))$ bits, which is $O(n(\log H_0(\mathsf{DA})+1)) = O(n \log\log D)$ bits.

In order to answer the queries, they proceed in the same way as Sadakane does (cf. Section 3.3.1), 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, for each document j with leftmost and rightmost occurrences l_i and r_i, its frequency is computed as $TF = f_j(r_i) - f_j(l_i) + 1$ in constant time.

Let us now analyze their other variant, where the mmphfs use $O(n_d \log\log\log \frac{n}{n_d})$ bits. By the log-sum inequality these add up to $O(n \log\log\log D)$ bits. The time to query f_d is $O(\log\log \frac{n}{n_d})$. To achieve the $O(\log\log D)$ time in the worst case, they use constant-time mmphfs when $\frac{n}{n_d} > D \log\log D$. This implies that on those arrays they spend $O(n_d \log\log \frac{n}{n_d}) = O(\frac{n}{D \log\log D} \log\log D) = O(n/D)$ bits, as it is increasing with n_d and $n_d < \frac{n}{D \log\log D}$. Adding over all documents d, they have at most $O(n)$ bits.

In summary, Belazzougui and Navarro [12] offer two solutions, both on top of the CSA of the entire collection and the two RMQ structures of Sadakane [68]. The first one uses $O(n \log\log D)$ bits, and solves document listing with frequencies in time $O(search(P) + \mathsf{ndoc} t_{\mathsf{SA}})$. The second one requires $O(n \log\log\log D)$ bits and time $O(search(P) + \mathsf{ndoc}(t_{\mathsf{SA}} + \log\log D))$. The $\log D$ in the space complexities can be replaced by $\log(H_0(\mathsf{DA})) + 1$.

Furthermore, they also offer a solution for top-k document retrieval. Despite Gagie et al. [31] (Section 3.4) pointed that Hon et al. solution may run on top of many data structures, mmphfs are not able to compute arbitrary frequencies. Their solution enriches the τ_k trees of Hon et al., so they give enough information to answer the top-k queries with the mmphf instead of the individual 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 $O(search(P) + t_{\mathsf{SA}} k \log k \log^{1+\varepsilon} n)$ for any $\varepsilon > 0$. They also make several other improvements over previous work, yet these are mostly theoretical.

3.6 Experimental Setup

In this section we present the experimental setup that is going to be used in the rest of the document. 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 (`http://boston.lti.cs.cmu.edu/ Data/clueweb09`), a collection of Chinese Web pages.

`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 3.1 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 $\log \sigma$.

Collection	Documents	Characters	Doc. length (avg.)	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	$6.02/6.57 = 0.92$

Table 3.1: Collections

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

For grammar compression, we use an implementation (`www.dcc.uchile.cl/gnavarro/software`). that, although does not guarantee balancedness, has always produced a grammar of very 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`.

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.

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`.

Chapter 4

Grammar Compression of Bitmaps

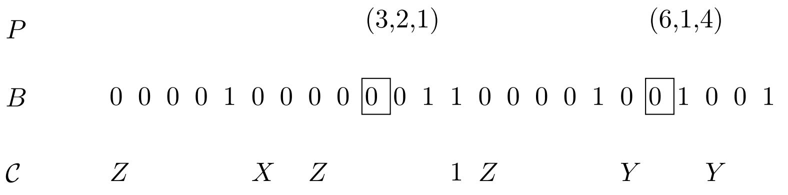
The first contribution of this work is a compressed representation of bitmaps with support for *rank* and *select* queries, that takes advantage of repetitiveness. There are many representations of bitmaps with support for *rank* and *select* queries [19,45,54,63,65]. Some of those use less space when the bitmap has few 1s [63], and some when the bitmap has few or many 1s, or they are unevenly distributed [65]. However, none of them uses less space when the bitmap is repetitive. Although our motivation is to handle arbitrary sequences supporting *rank* and *select* (see Chapter 5), this type of compression might also have independent interest.

We focus on the algorithm RePair, explained in Section 2.5. RePair factors out repetitions, 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 [53, 70], 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 $Z \rightarrow XY \in \mathcal{R}$. Likewise, let $r(1) = 1$, $r(0) = 0$ and for non-terminals $Z \rightarrow XY \in \mathcal{R}$ let $r(Z) = r(X) + r(Y)$. For each nonterminal Z added to the dictionary we store also the length $\ell(Z)$ and the number of 1s, $r(Z)$, 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 4.1 shows an example.


P
(3,2,1)
(6,1,4)
B
0 0 0 0 1 0 0 0 0 $\boxed{0}$ 0 1 1 0 0 0 0 1 0 $\boxed{0}$ 1 0 0 1
$\mathcal{C}$
Z X Z 1 Z Y Y

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

Figure 4.1: An example of our RePair representation of bitmaps with sampling period $s = 10$.

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. As the grammar is balanced, the total time is $O(s + \log n)$. Algorithm 3 shows the pseudo-code.

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 expand 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 we achieve space $O((R + C) \log n + n)$ bits and $O(\log n)$ time for the operations. We can reduce the $O(n)$ extra space to $o(n)$ by increasing s, which impacts query times and makes them superlogarithmic.

We must remind 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

```
$Rank(i)$

$j \leftarrow \lfloor i/s \rfloor$;
$r \leftarrow P[j].r$;
$p' \leftarrow P[j].p$;
$l \leftarrow i - P[j].o$;
while $p' \leq |\mathcal{C}|$ do
    $Z \leftarrow \mathcal{C}[p']$;
    if $l + l(Z) < i$ then
        $p' \leftarrow p' + 1$;
        $r \leftarrow r + r(Z)$;
        $l \leftarrow l + l(Z)$;
    else
        return $Expand(Z, l, r)$
```

```
$Expand(Z, l, r)$

if $l = i$ then
    return $r$
$XY \leftarrow \mathcal{R}[Z]$;
if $l + l(X) \geq i$ then
    return $Expand(X)$;
else
    $r \leftarrow r + r(X)$;
    $l \leftarrow l + l(X)$;
    return $Expand(Y)$;
```

Algorithm 3: Computing $rank_1(B, i)$ with our grammar compressed bitmap.

stop the replacement procedure at an earlier stage.

Navarro and Russo [61] showed that RePair achieves high-order entropy, and it is easy to prove that their bound still holds for our scheme supporting *rank* and *select*. Given that each iteration of RePair reduces the size of the structures, we analyze the size when no pair appears more than $b = \log^2 n$ times. This is achieved in at most $n/(b+1)$ RePair steps, so there are $O(n/\log^2 n)$ rules. Therefore, when we add a new rule $z \rightarrow xy \in \mathcal{R}$ to the dictionary we are able to store not only the right side, but also the $r(z)$ and $\ell(z)$ values and still achieve $o(n)$ bits for the dictionary. To analyze the size of $\mathcal{C}$ in this stage, Navarro and Russo considered the parsing of the sequence $B = expand(\mathcal{C}[1]\mathcal{C}[2])expand(\mathcal{C}[3]\mathcal{C}[4]) \ldots expand(\mathcal{C}[C-1]\mathcal{C}[C])$ of $t = C/2$ strings that do not appear more than b times. Resorting to a theorem of Korasaju and Manzini [47], which states that in such a case it holds that $t \log t \leq nH_k(B) + t\log(n/t) + t\log(b) + \Theta(t(1+k))$, they proved that the space required to store $\mathcal{C}$ is at most $2nH_k(B) + o(n)$ bits for any $k = o(\log n)$. In our case, as in theirs, the size of the dictionary is $o(n)$ so the total space of our representation is bounded by $2nH_k(B)$.

4.4 In Practice

There are several ways to represent the dictionary $\mathcal{R}$ in compressed form. We choose one [37] that allows for random access to the rules. It represents $\mathcal{R}$ in the form of a directed acyclic graph (DAG) as 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 and a 0 a leaf. 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 associated with positions in S_R (that is, $S_B[i] = 0$ is associated to $S_R[rank_0(S_B, i)]$). Those leaf symbols are either terminals or nonterminals. Nonterminals are represented as positions in S_B, which must then be recursively expanded. This DAG representation takes, in good cases, as little as 50% of the space required by a plain array representation of $\mathcal{R}$ [37].

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): Another heuristic consists in fixing 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 Chapter 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 Raman, Raman and Rao (RRR) [65], explained in Section 2.3.2, and a practical approach of Okanohara and Sadakane (SDArray) [63]. For this experiment we intend to compare the compression techniques, therefore 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 4.2 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 [61], which in this case is the same as the zero-order entropy, just like the other two schemes [63, 65].

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. On the other hand, SDArray [63] uses $nH_0 + O(\#1s)$ bits, which is also close to the entropy, especially when the density is small. This is exactly the domain where this technique dominates. Finally, even though Navarro and Russo [61] only gave a bound of $2nH_k$ for the space usage of RePair, we can see in practice that RePair's overhead over the zero-order 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 (see 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 both techniques we plot the extreme scenarios where no space is advocated to sample the rules, and where the space is maximal. For MD, δ value was set to 0,1,2,4,8, $\ldots$, h, where h is the height of the grammar. For ST, m_l was set to 0,2,4,6, $\ldots$, b, where b is the number of bits required to store the longest rule. Figures 4.3 and 4.4 show the different space-time trade-offs we obtained to answer *access*, *rank* and *select* queries. We also include RRR (with sampling values 32, 64, and 128) and SDArray as a reference.

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 space-time trade-offs. 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 at least one order of magnitude faster than our RePair compressed bitmaps. In the next chapters we will consider this handicap and use RePair only when it impacts space the most and impacts time the least.

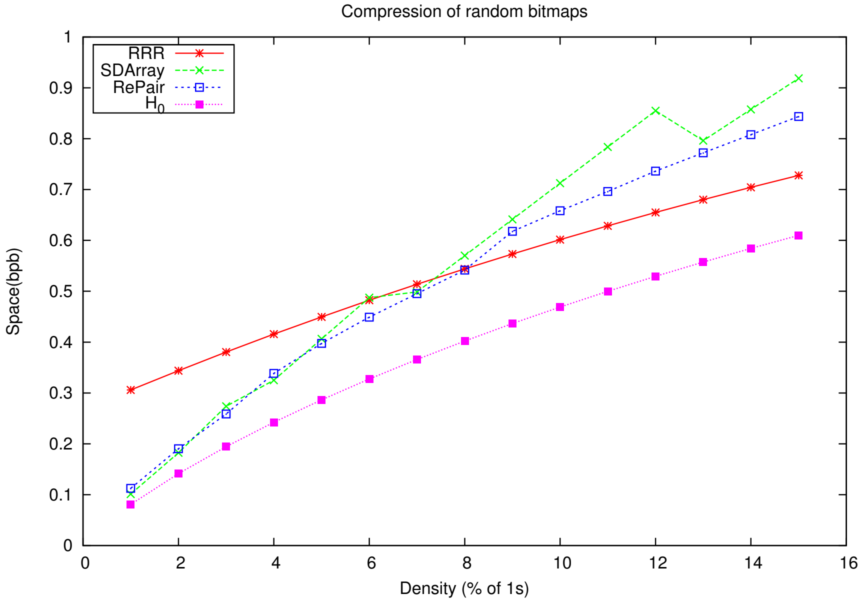


Figure 4.2: Compression ratio in bits per bit (bpb) of RRR, SDArray and RePair compressors for bitmaps with different densities. We disregard the space required for the samplings, as these can be decreased at will.

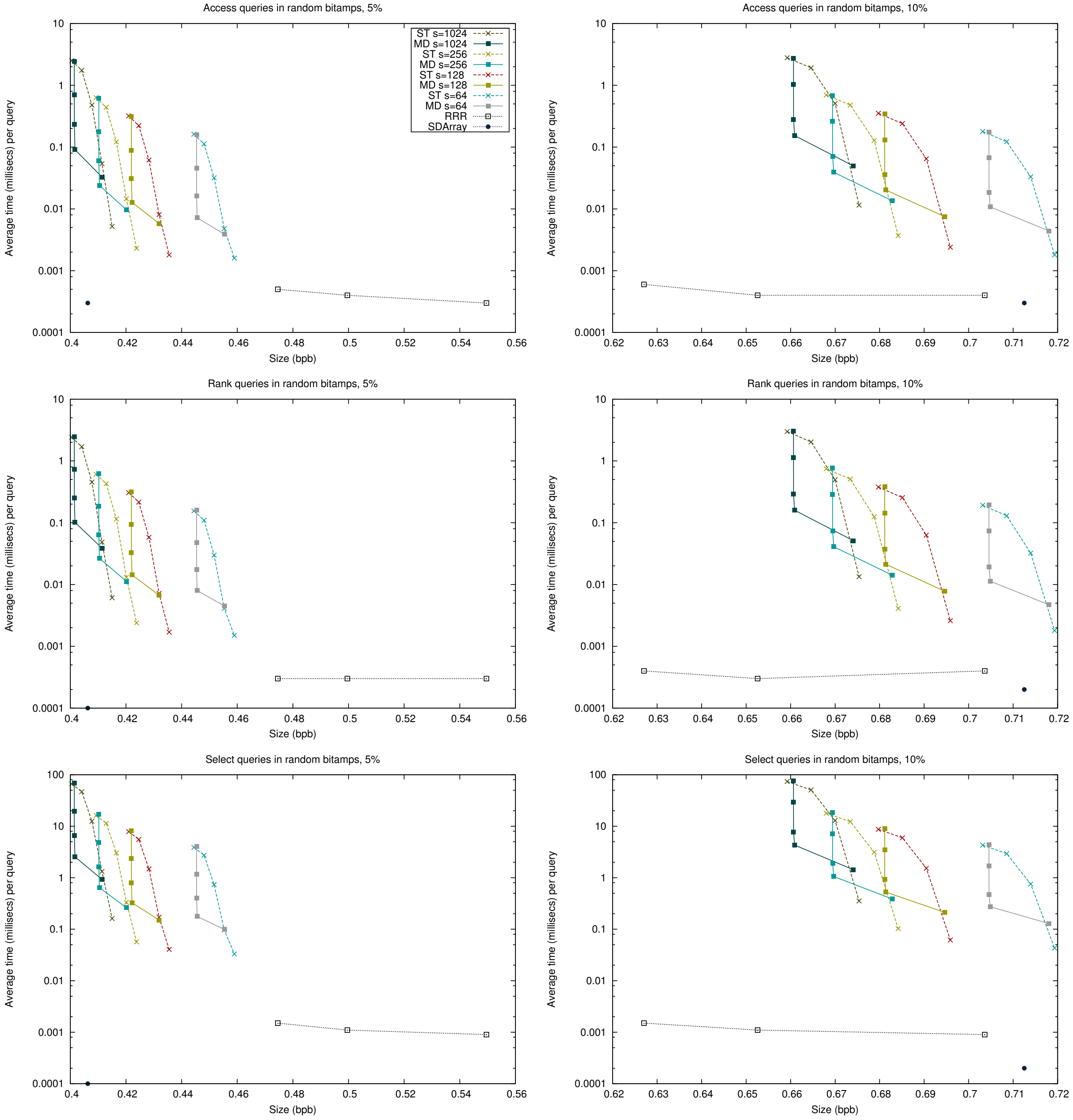


Figure 4.3: Tradeoffs obtained for our different techniques to solve *access*, *rank* and *select* queries over random bitmaps with density 5% and 10%.

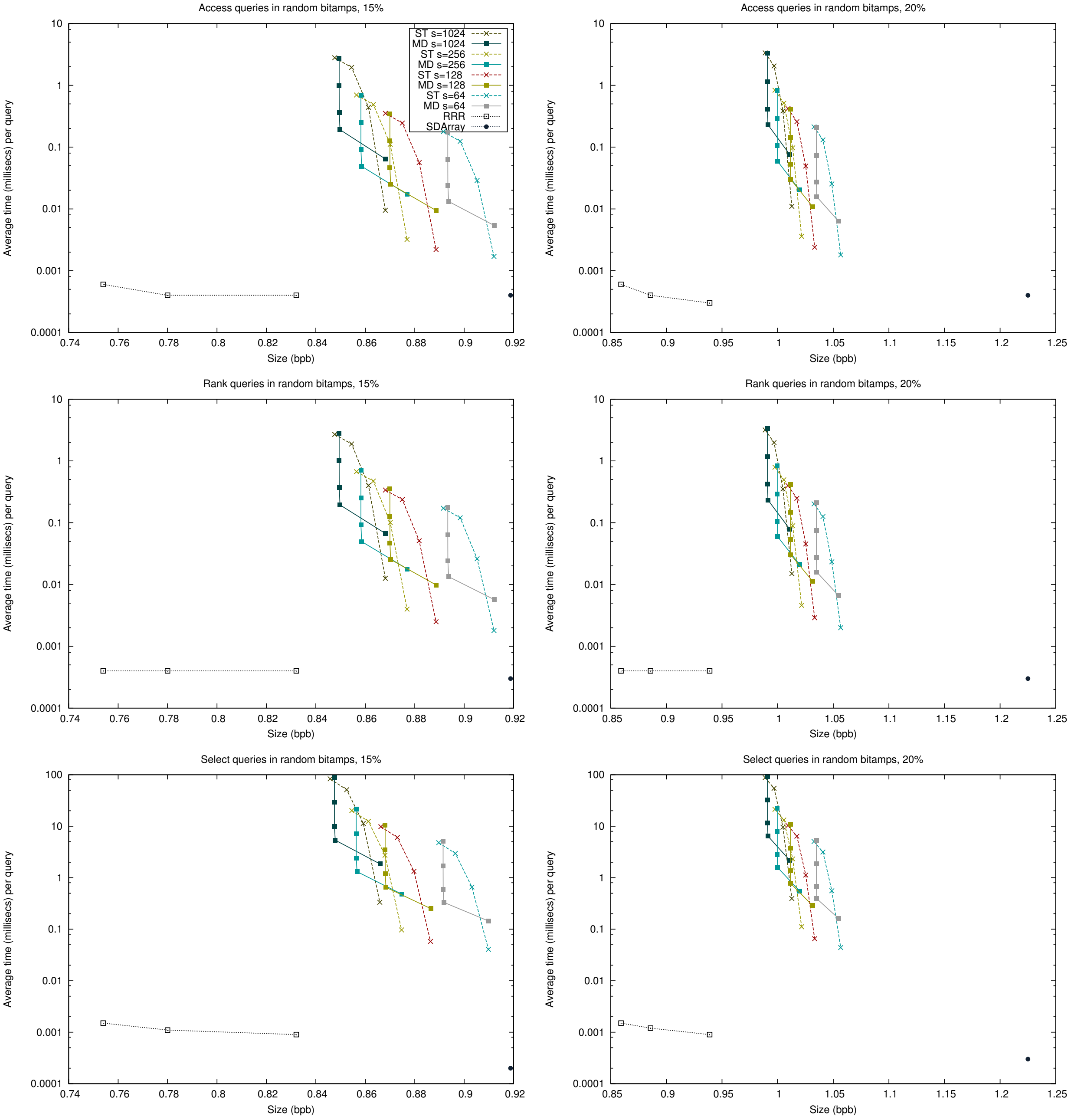


Figure 4.4: Tradeoffs obtained for our different techniques to solve *access*, *rank* and *select* queries over random bitmaps with density 15% and 20%.

Chapter 5

Grammar Compression of Wavelet Trees and Document Arrays

In this chapter we present new compressed representations of sequences with support for *rank* and *select* queries and with applications to document retrieval.

Given a sequence $S[1,n]$ over alphabet $[1,D]$, we build the wavelet tree of S and represent its bitmaps using the compressed format of Chapter 4, with the aim of exploiting the repetitions in S. Our intention is to apply this data structure to the representation of the document array (Section 3.1)

Various studies have shown the practicality for document retrieval of representing the document array with a wavelet tree [23,31,33,72]. The major drawback of these approaches is the space usage of the document array, which the wavelet tree does not help to reduce. The existing compressed representations of wavelet trees achieve the zero-order entropy of the sequence represented. In the case of the document array this yields little compression, because its zero-order entropy corresponds to the distribution of document lengths. However, we will show that the document array contains self-repetitions that can be exploited for compression [37].

5.1 General Analysis

Consider a RePair representation $(\mathcal{R},\mathcal{C})$ of S, where the sizes of the components is R and C as in Section 4.1. 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. The time for the operations raises to $O((s + \log n) \log D)$.

This result does not look alphabet-friendly, and actually the upper bounds are no better than applying the method described in Chapter 4 on D bitmaps $B_c[1, n]$, where $B_c[1] = 1$ iff $S[i] = c$. Thus the analysis gives only a (very) pessimistic upper bound. Still, one can expect 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, we showed that our RePair technique over a bitmap B uses $O(|B|H_0(B))$ bits, hence, using this technique over each level, the space required for the complete wavelet tree of S is $O(nH_0(S))$ [38].

5.2 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 [21]. We apply the RePair representation from Chapter 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 [65] 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. Therefore, 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.

5.3 Compressing the Document Array

The document array $\mathsf{DA}[1, n]$ is not generally compressible in terms of statistical entropy. Consider for instance the situation when all the documents have the same length. Then, the zero-order entropy of the document array will be maximal, no matter the compressibility of the documents. However, we show now that it contains repetitions that are related to the statistical high-order entropy of the text $T[1,n]$.

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 $0 \leq k \leq \ell$. Let τ be the minimum number of quasi-repetitions in which SA can be partitioned. It is known that $\tau \leq nH_k(T) + \sigma^k$ for any k [49] (the upper bound is useful only when $H_k(T) < 1$).

González and Navarro [37] 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. [31] 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.

This theoretical result had not been verified in practice. Moreover, the situation is more complicated because we do not need to represent DA, but the wavelet tree of DA, in order to support the various document retrieval tasks. As explained, RePair compression degrades in the lower levels of the wavelet tree.

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 the different techniques for representing bitmaps on each level of the wavelet tree of the document array. Figures 5.1 and 5.2 show the space achieved. Again, we are not storing any sampling information.

As explained in Section 5.3, we expect more repetitiveness on the higher levels of the wavelet tree, and the experimental results confirm it: our technique dominates on the higher levels. The wavelet tree breaks those repetitions when moving to the lower levels, which explains the deterioration of our compression ratio. It is also worth noting that no technique achieves much compression on the Proteins collection, with the exception of RePair on the first two levels. This is also expected, because the entropy of this collection is very high (recall Table 3.1). From now on we disregard technique SDArray, which is not useful in this scenario.

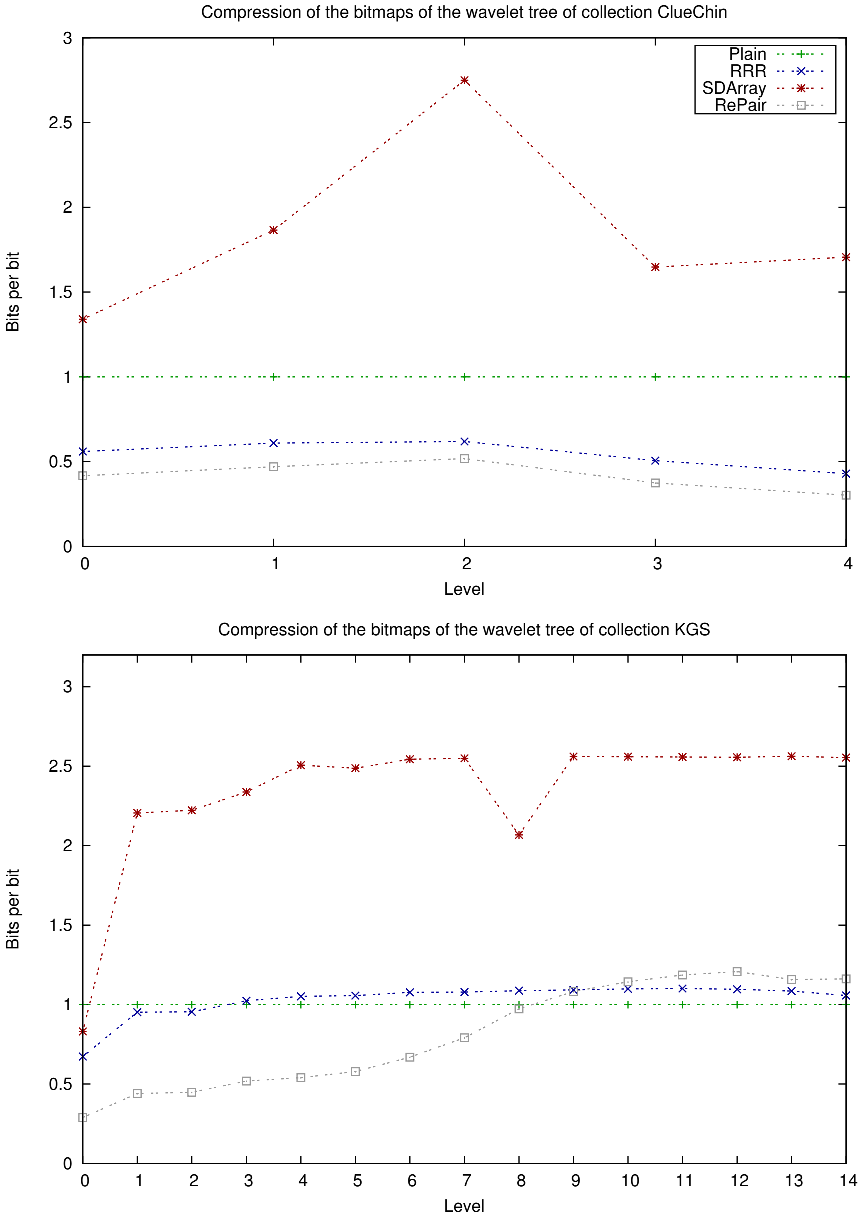


Figure 5.1: Bits per bit achieved by the different bitmap compression methods on the bitmaps of the wavelet tree representation of the document array of collections ClueChin and KGS.

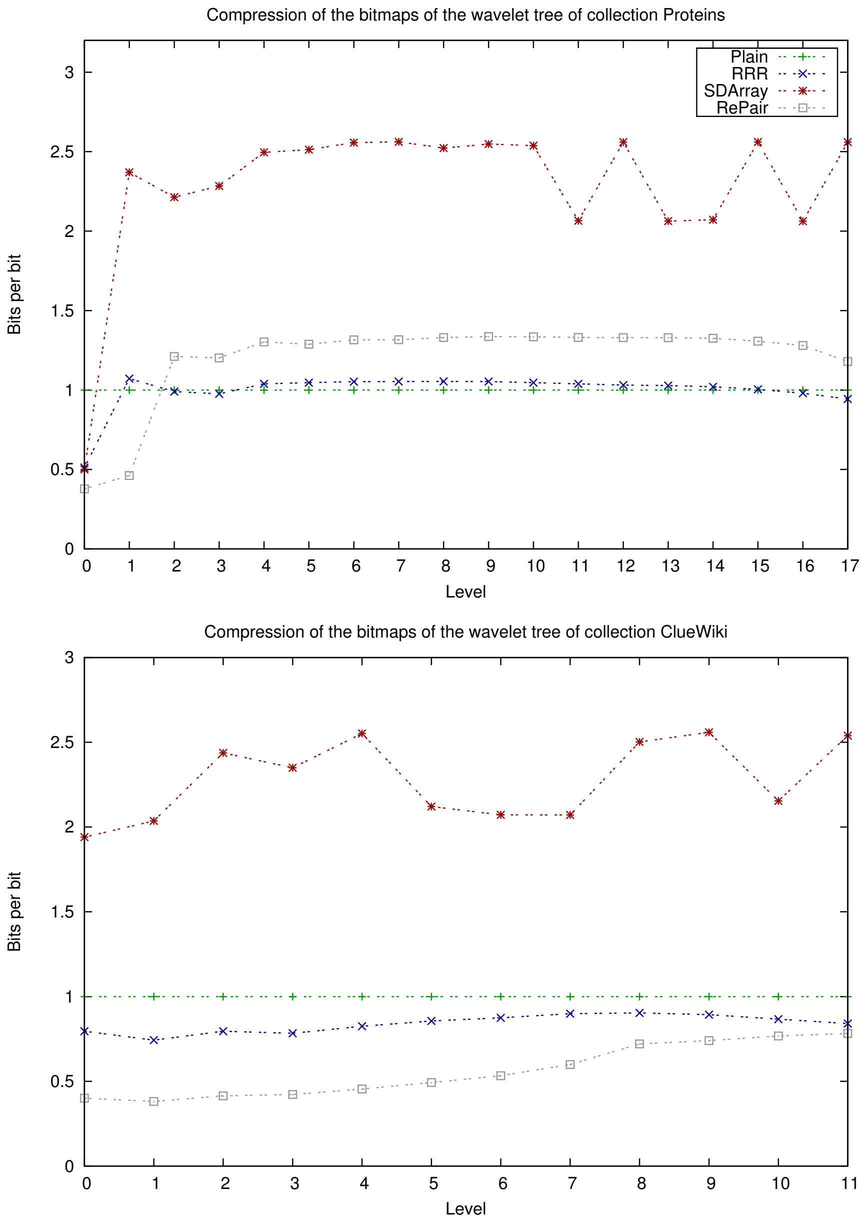


Figure 5.2: Bits per bit achieved by the different bitmap compression methods on the bitmaps of the wavelet tree representation of the document array of collections Proteins and ClueWiki.

5.4.2 Choosing the sampling technique for RePair

In Section 4.5 we presented two different techniques to support *rank* and *select* queries on our compressed representation of bitmaps. We studied their performance over random bitmaps, and the results showed that the best performing technique depends on the context.

Therefore, we carried out the same experiments of Section 4.5, obtaining space-time tradeoffs for our two techniques for representing bitmaps, now over the most relevant levels of the wavelet tree. That is, those levels in which the experiments on bitmap compression (Figures 5.1 and 5.2) show that RePair compresses better than RRR and requires less than one bit per bit. Figures 5.3 to 5.8 show the results.

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

In this section we compare our wavelet tree representation of the document array with previous work: plain and statistical encoding of the wavelet tree, and Sadakane's [68] method based on individual CSAs (Section 3.3). As explained, alternative solutions [43, 68] for 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, and also use an efficient RMQ data structure. Again, Section 3.6 describes those choices in more detail. For the space we charge only $2n$ bits (the lower bound) for each RMQ structure and zero for B, to account for possible future space reductions.

Previous work by Culpepper et al. [23] has demonstrated that the quantile approach [33] is clearly preferable, in space and time, over previous ones based on wavelet trees [72]. They also showed that the quantile approach is in turn superseded by a DFS traversal, which avoids some redundant computations (see Section 3.2.3). Therefore, we carry out the DFS algorithm over a plain wavelet tree representation (*WT-Plain*), over one where the bitmaps are statistically compressed [65] (*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 5.2), 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).

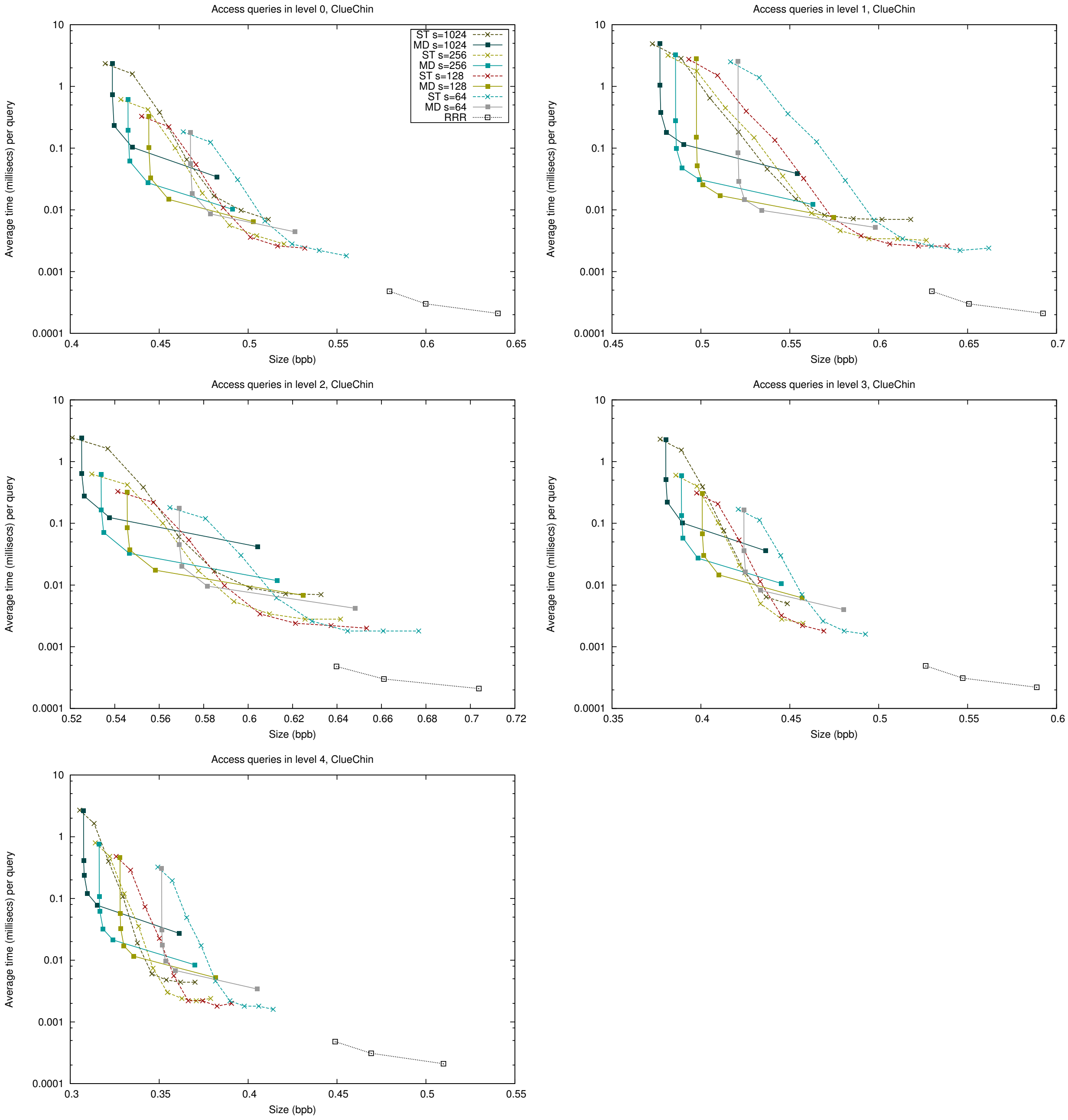


Figure 5.3: Tradeoffs obtained for our different techniques to support *access*, over the wavelet tree planes of the document array of the collection ClueChin.

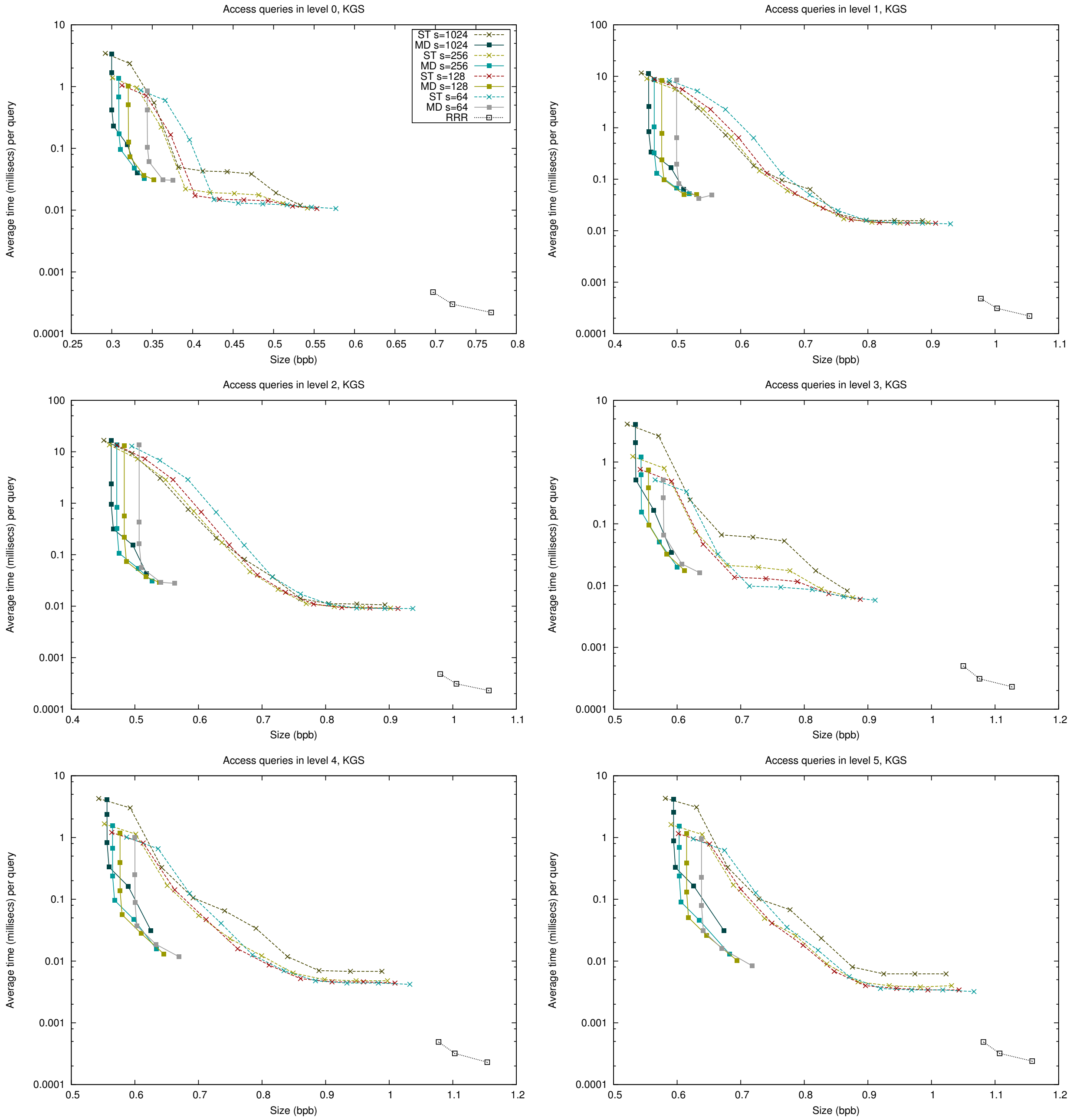


Figure 5.4: Tradeoffs obtained for our different techniques to support *access*, over the wavelet tree planes of the document array of the collection KGS.

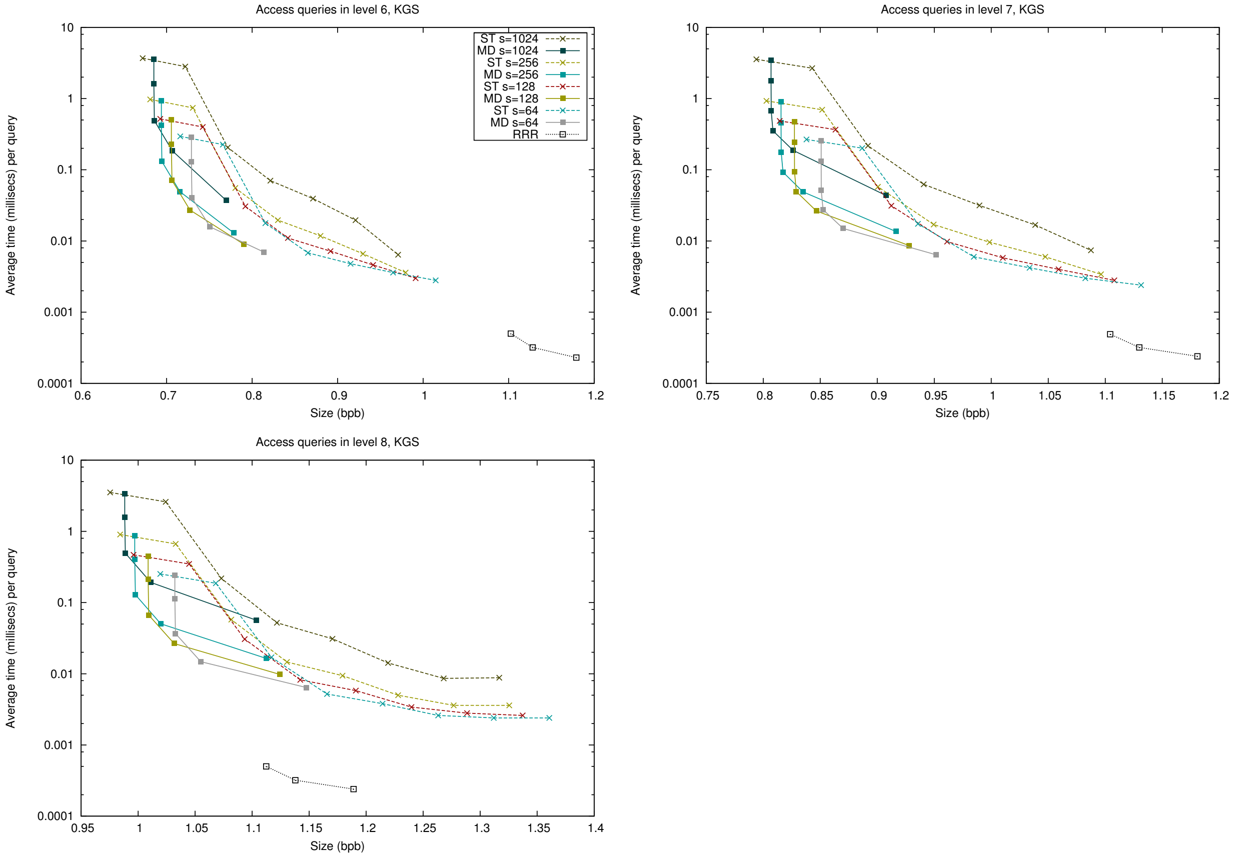


Figure 5.5: Tradeoffs obtained for our different techniques to support *access*, over the wavelet tree planes of the document array of the collection KGS.

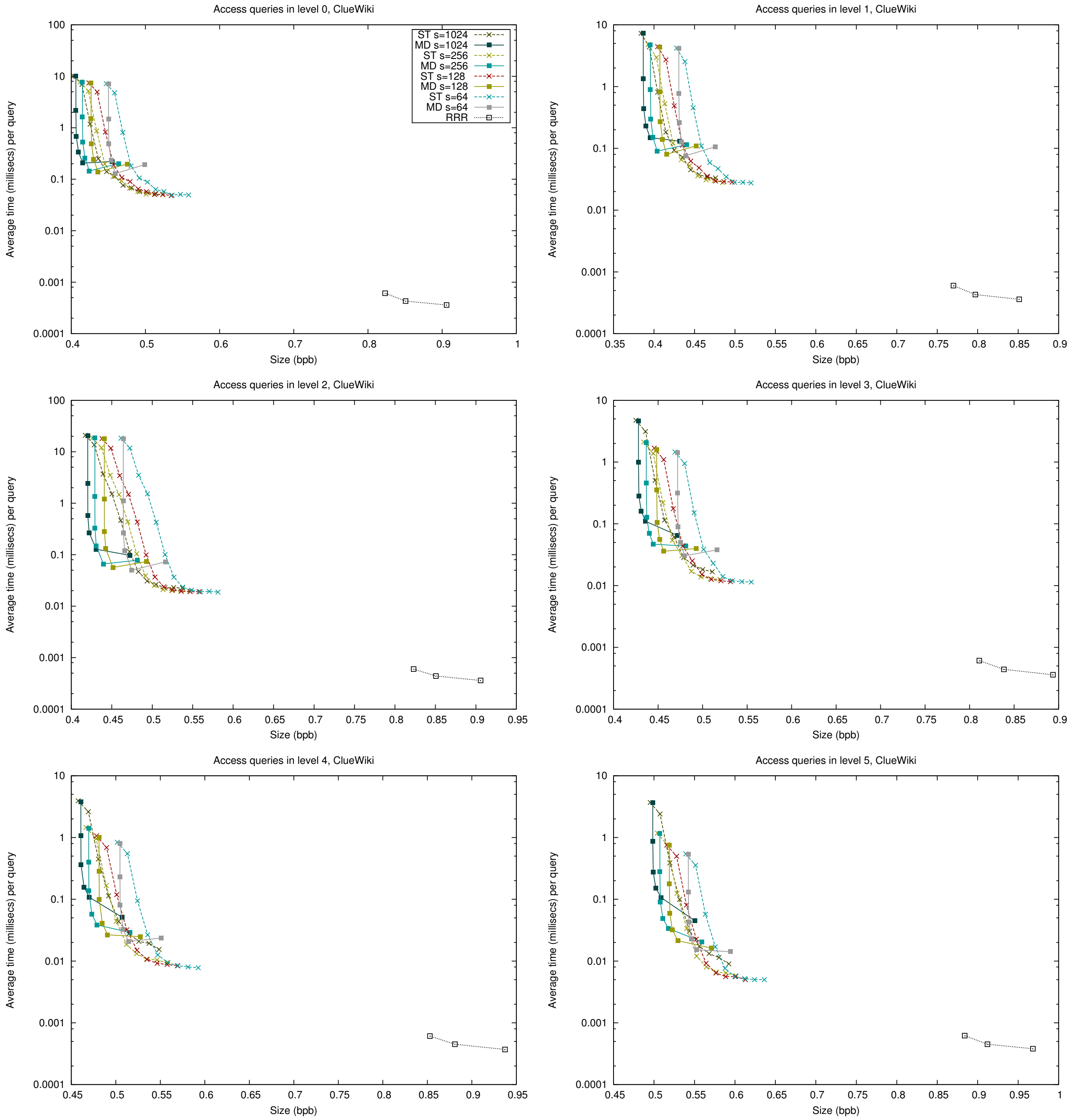


Figure 5.6: Tradeoffs obtained for our different techniques to support *access*, over the wavelet tree planes of the document array of the collection ClueWiki.

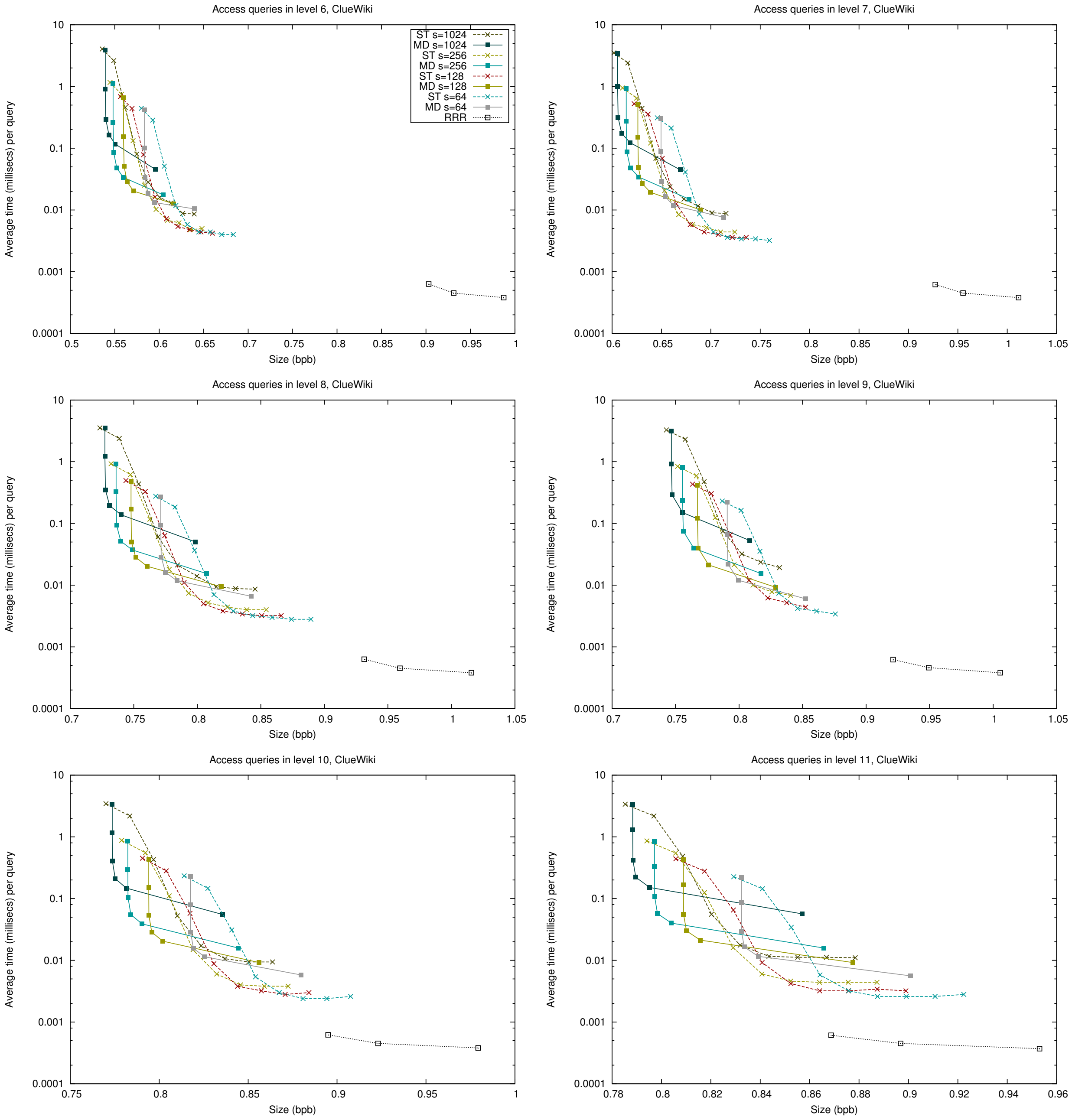


Figure 5.7: Tradeoffs obtained for our different techniques to support *access*, over the wavelet tree planes of the document array of the collection ClueWiki.

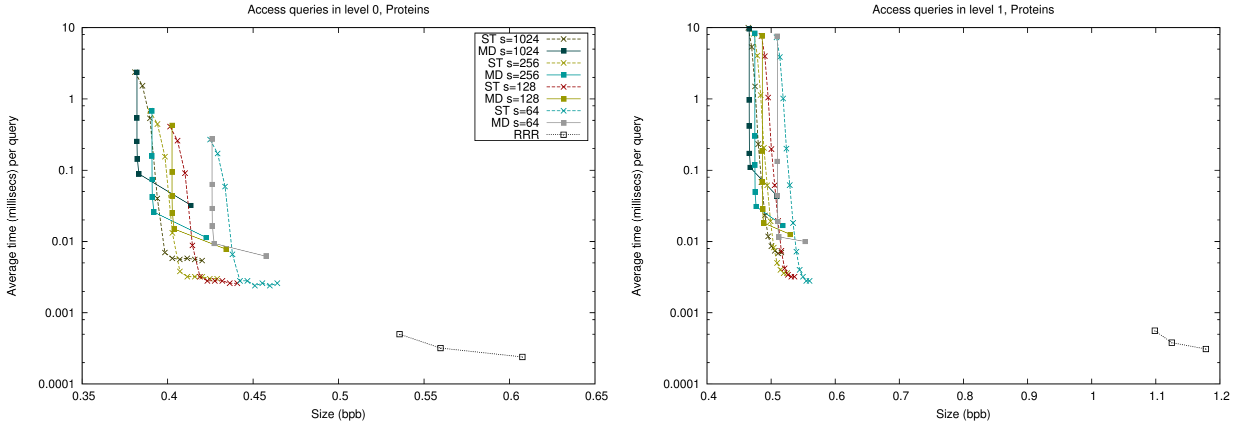


Figure 5.8: Tradeoffs obtained for our different techniques to support *access*, over the wavelet tree planes of the document array of the collection Proteins.

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 all of the above reasons we will 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.

Figures 5.9 to 5.12 show the average query time for the different collections and interval sizes. 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. The detailed specification of those choices is shown in Tables 5.1 to 5.4. We will also include the variant using $\alpha = 1$, called *WT-RP*.

In the upcoming experiments we will compare our technique with previous work, showing two grammar-compressed alternatives, the variant with $\alpha = 1$ (*WT-RP*) and the best-performing alternative with $\alpha < 1$ (*WT-Alpha*).

Figures 5.13 to 5.16 show the results of our techniques in the context of the related work.

The space overhead of indexing the documents separately makes Sadakane's approach impractical (even with the generous assumptions on extra bitmaps and RMQs). Even on `ClueChin` and `ClueWiki` with relatively large documents Sadakane's solution requires more space than all our techniques, and is also slower. In collections KGS and Proteins we keep it out of the plots, as it requires more than 60 bpc.

The results are different depending on the type of 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.

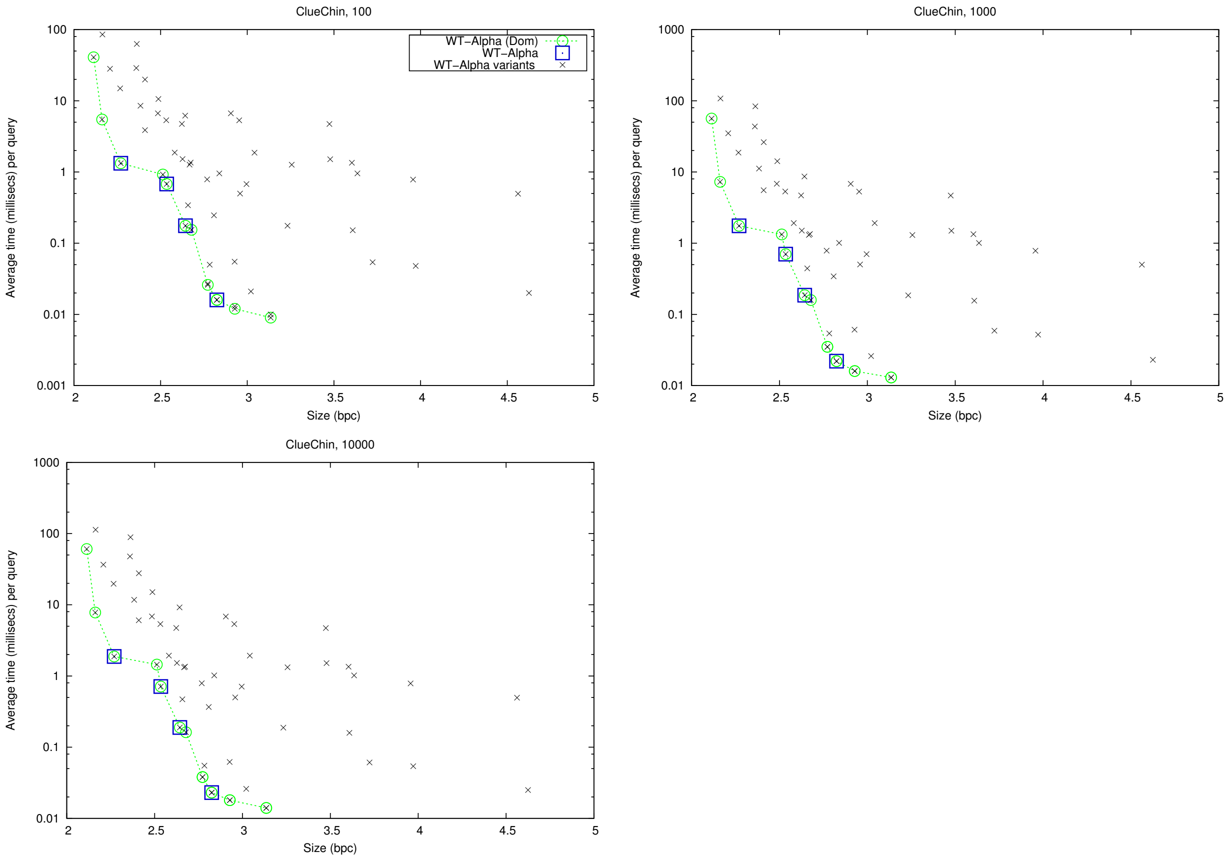


Figure 5.9: Clouds of space-time tradeoffs for document listing with our techniques on document array intervals of length from 100 to 10,000 on collection ClueChin

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* CSA.

Planes	Size	Technique	Sampling	MD(δ)
0	Smallest	RePair	128	1
1		RePair	128	1
2		RePair	128	1
3		RePair	128	1
4		RePair	128	1
0	Small	RePair	256	3
1		RePair	256	3
2		RRR	128	-
3		RRR	128	-
4		RRR	128	-
0	Medium	RePair	128	1
1		RePair	128	1
2		RRR	64	-
3		RRR	64	-
4		RRR	64	-
0	Large	RRR	128	-
1		RRR	128	-
2		RRR	128	-
3		RRR	128	-
4		RRR	128	-

Table 5.1: *WT-Alpha* configuration, collection ClueChin.

Planes	Size	Technique	Sampling	MD(δ)
0 to 4	Smallest	RePair	256	3
5-14		Plain	10*32	-
0 to 3	Small	RePair	256	3
4 to 14		Plain	10*32	-
0 to 2	Medium	RePair	256	3
3 to 14		Plain	10*32	-
0	Large	RePair	256	3
1 to 2		RRR	128	-
3 to 14		Plain	10*32	-

Table 5.2: *WT-Alpha* configuration, collection KGS.

Planes	Size	Technique	Sampling	MD(δ)
0 to 7	Smallest	RePair	128	1
8 to 11		RRR	64	-
0 to 7	Small	RePair	64	0
8 to 11		RRR	32	-
0 to 3	Medium	RePair	128	1
4 to 11		RRR	64	-
0 to 1	Large	RePair	128	1
2 to 11		RRR	64	-

Table 5.3: *WT-Alpha* configuration, collection ClueWiki.

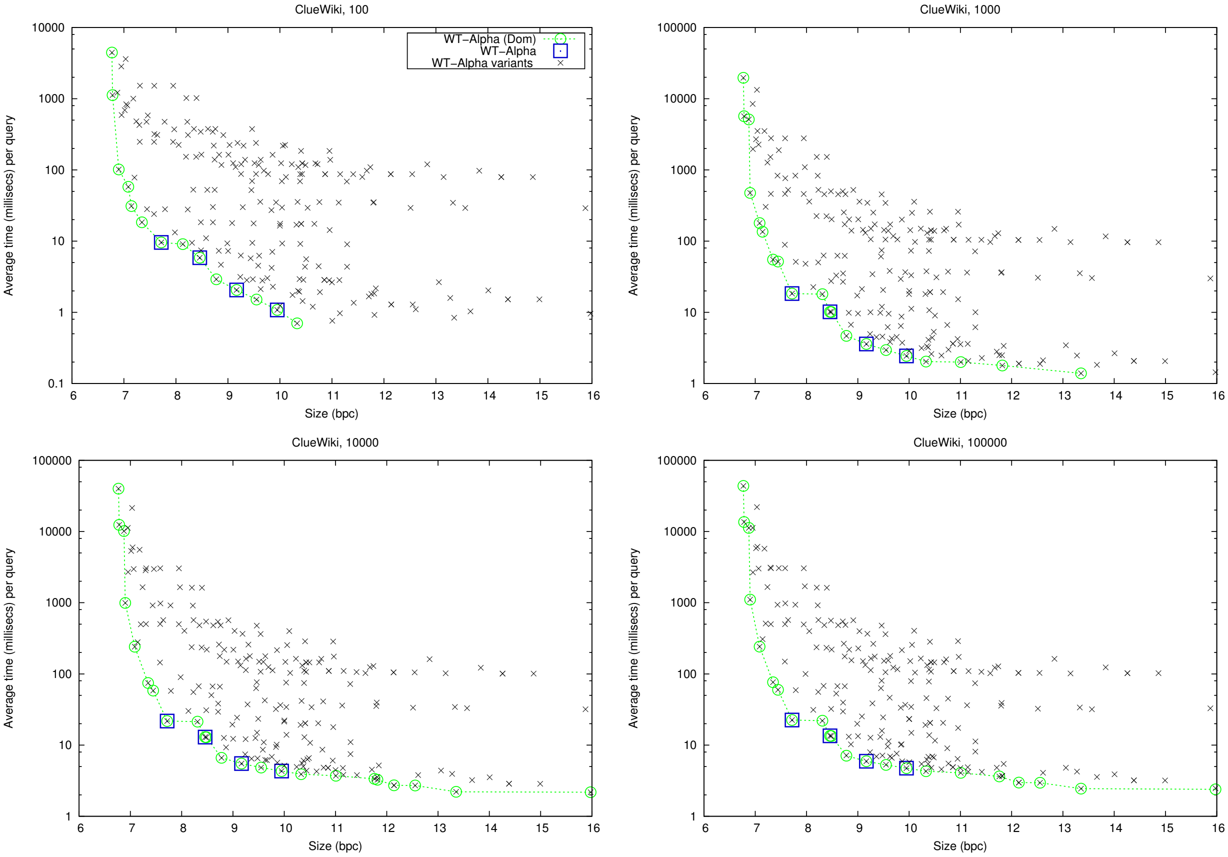


Figure 5.10: Clouds of space-time tradeoffs for document listing with our techniques on document array intervals of length from 100 to 10,000 on collection ClueWiki

Planes	Size	Technique	Sampling	MD(δ)
0		RRR	256	-
1		RePair	1024	1
2 to 3	Smallest	RRR	256	-
4 to 15		Plain	20*32	-
16 to 17		RRR	256	-
0		RRR	256	-
1	Small	RePair	1024	1
2 to 17		Plain	20*32	-
0	Medium	RRR	64	-
1 to 17		Plain	4*32	-
0	Large	RePair	64	0
1 to 17		Plain	2*32	-

Table 5.4: *WT-Alpha* configuration, collection Proteins.

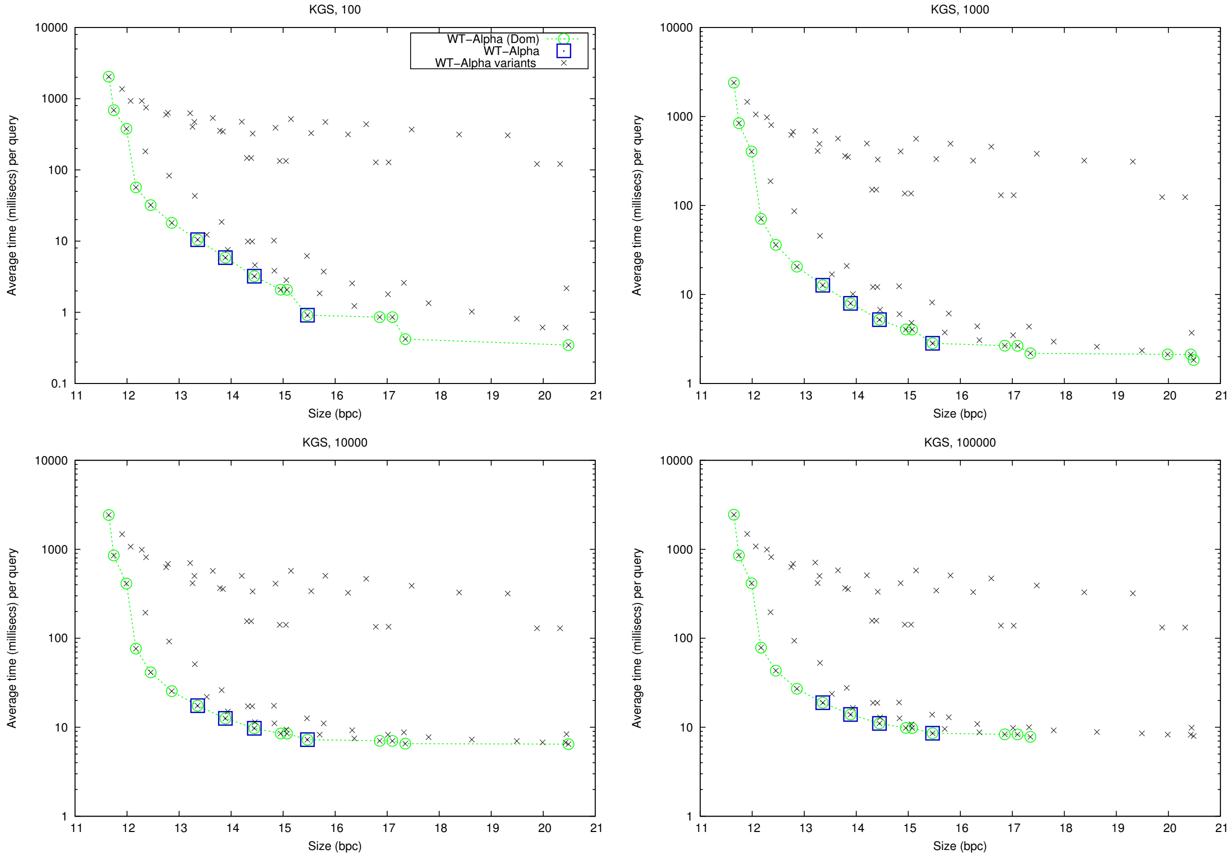


Figure 5.11: Clouds of space-time tradeoffs for document listing with our techniques on document array intervals of length from 100 to 10,000 on collection KGS

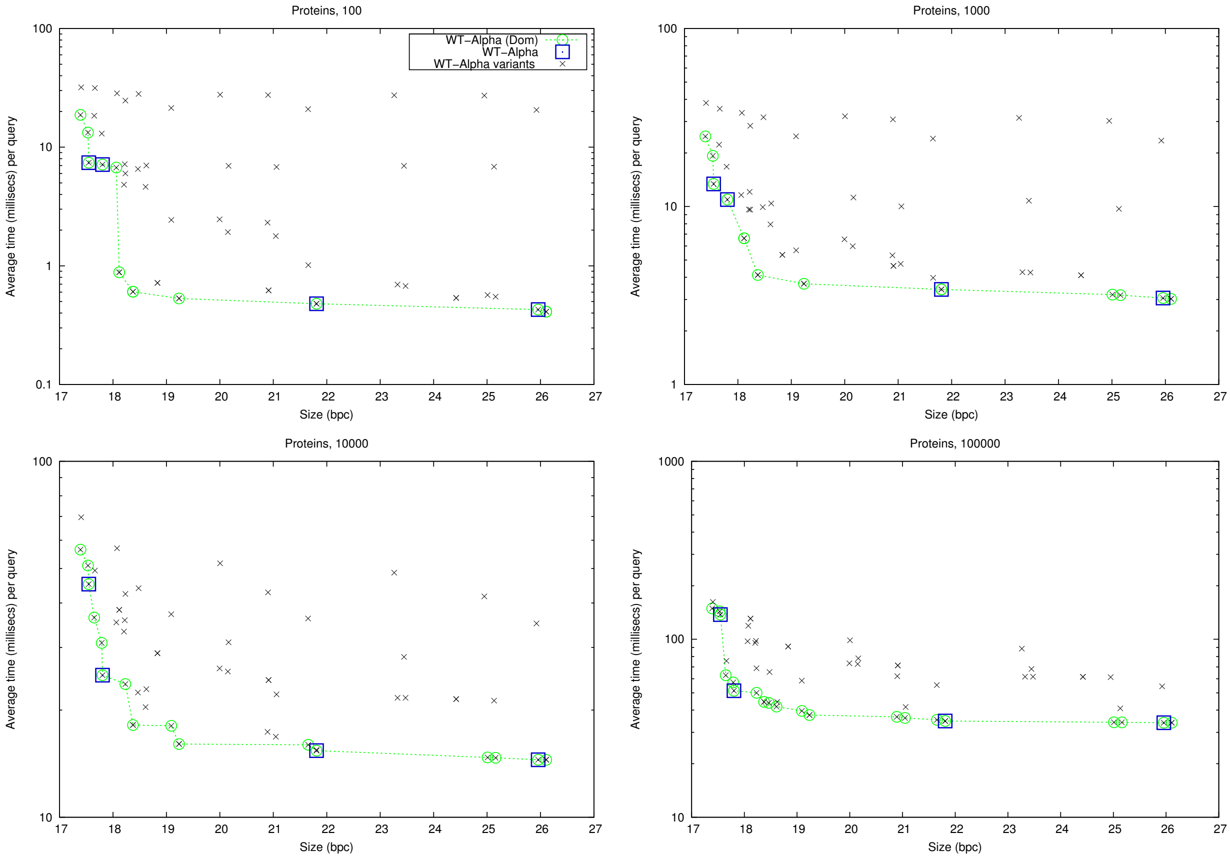


Figure 5.12: Clouds of space-time tradeoffs for document listing with our techniques on document array intervals of length from 100 to 10,000 on collection Proteins

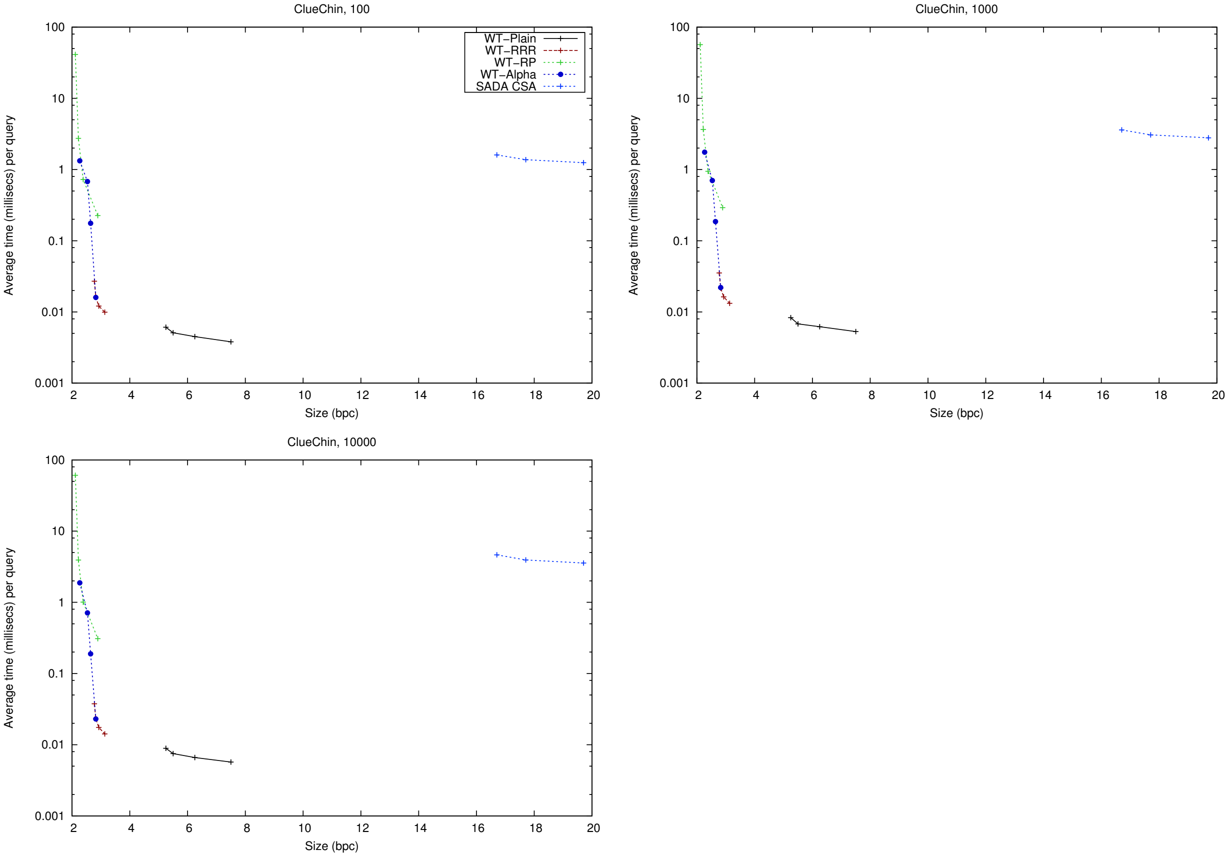


Figure 5.13: Experiments for document listing with term frequencies, collection ClueChin.

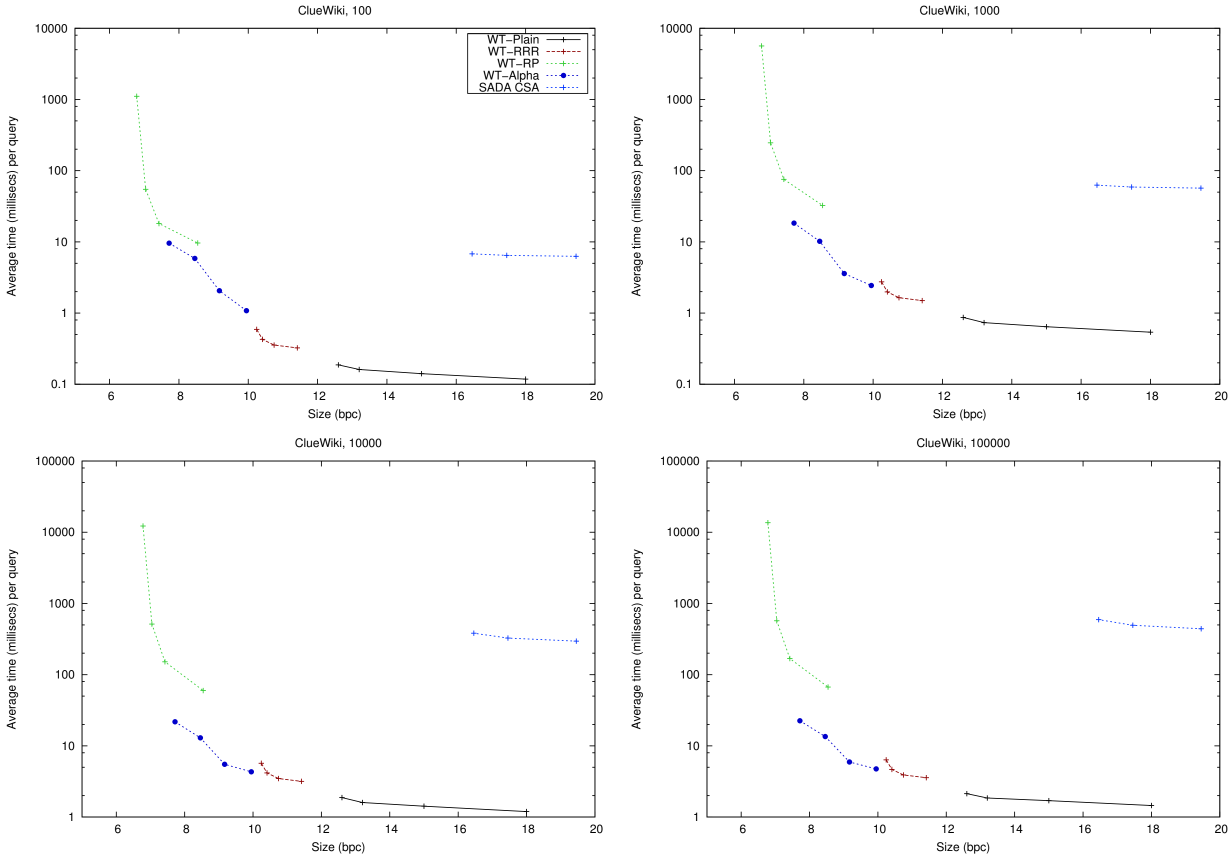


Figure 5.14: Experiments for document listing with term frequencies, collection ClueWiki.

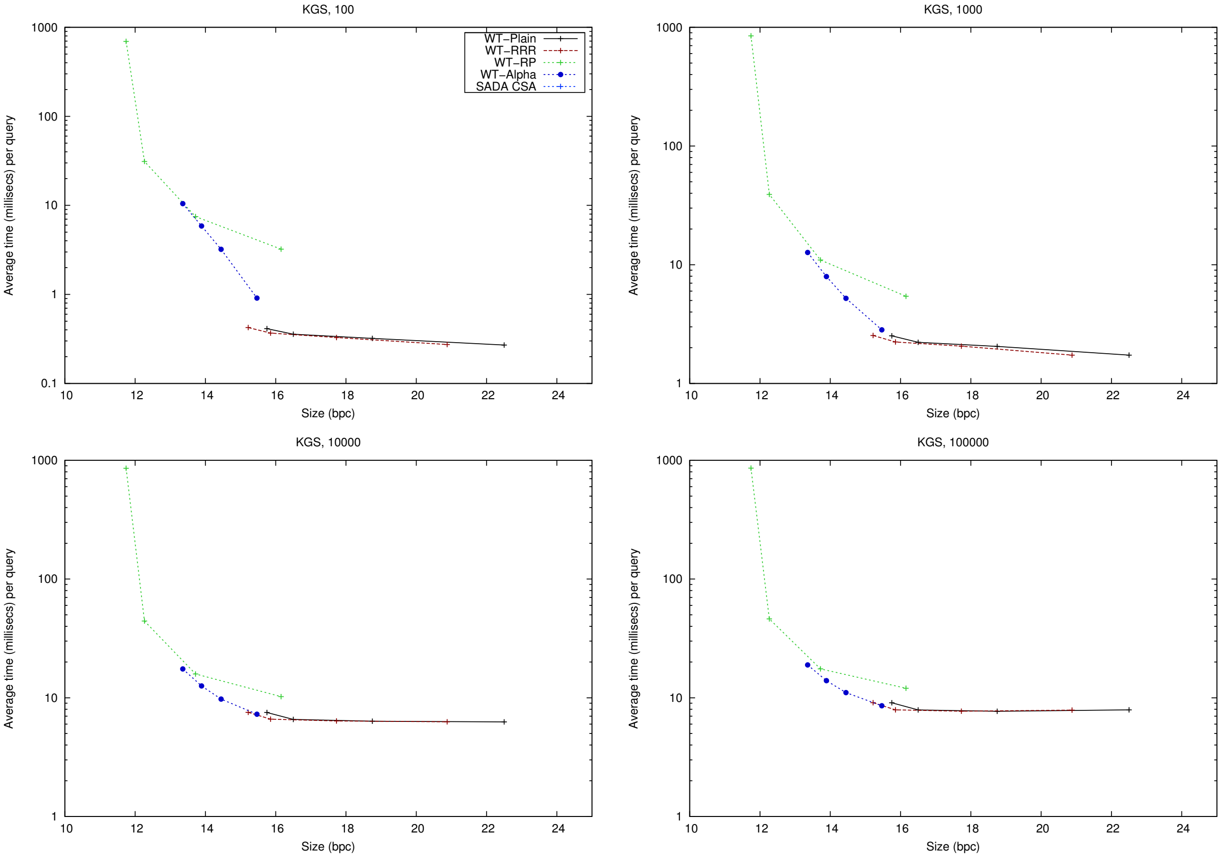


Figure 5.15: Experiments for document listing with term frequencies, collection KGS.

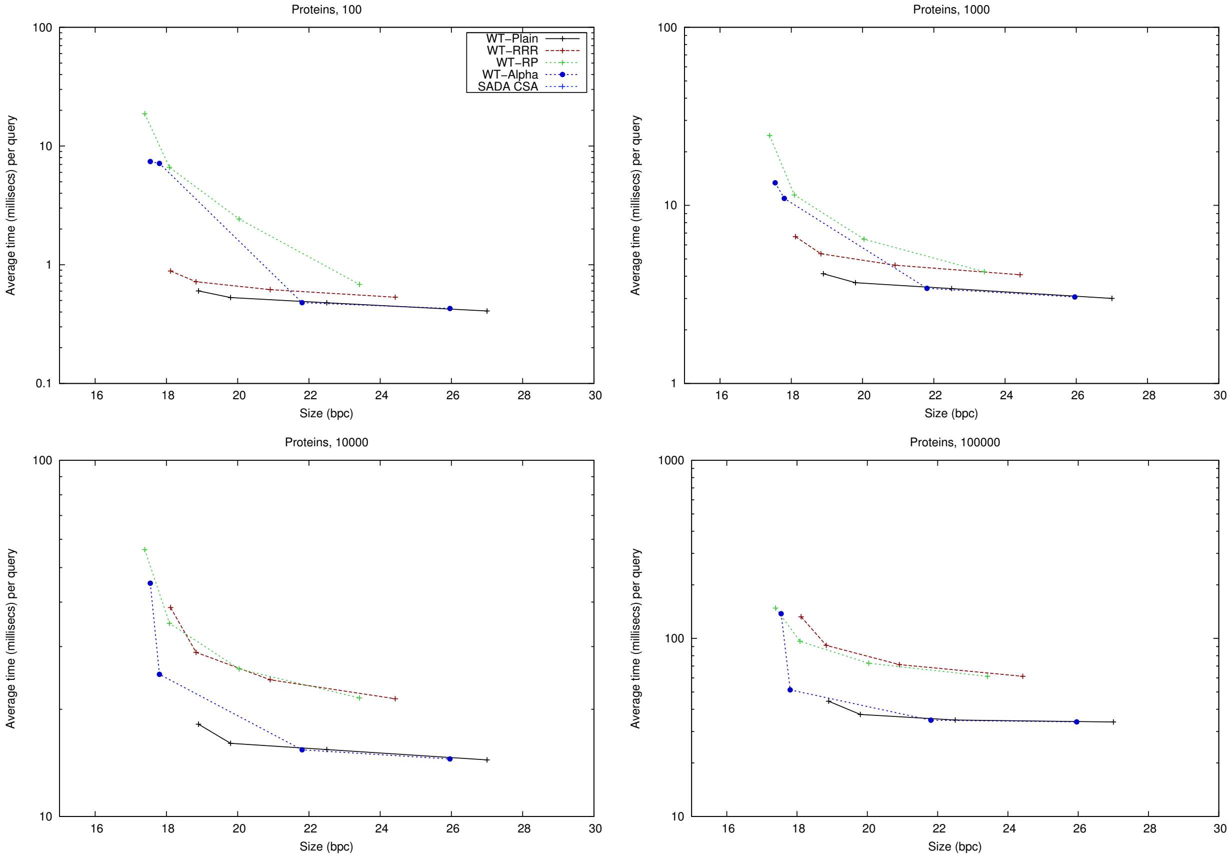


Figure 5.16: Experiments for document listing with term frequencies, collection Proteins.

Chapter 6

Sparsified Suffix Trees and Top-k Retrieval

In the previous chapter we have shown that the technique of Sadakane [68], 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.'s [43] data structure for top-k document retrieval (Section 3.3.2) is built on the top of Sadakane's approach, therefore a straightforward implementation will suffer from the same lack of practicality.

In this chapter we propose various practical versions of Hon et al.'s $o(n)$-bit data structures 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. [43] with various of the techniques of Culpepper et al. [23] 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. [31] showed that Hon et al.'s succinct scheme can actually run on top of any data structure able to (1) telling 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 $\mathsf{SA}[sp, ep]$ (see Section 3.4). They gave many combinations that solve (1) and (2), being Golynski et al.'s representation [35] of the document array the fastest choice in theory. We have also implemented this scheme and studied its practicality.

6.1 Implementing Hon et al.'s Succinct Structure

The succinct structure of Hon et al. [43] 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, and 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 by Sadakane's solution [68]. See Section 3.3.2 for details.

This section focuses on practical implementations of this idea. In the next section 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 [43]. We mark internal SGST nodes $lca(l_1, l_{g+1}), lca(l_{g+1}, l_{2g+1}), \ldots$. It is not difficult to prove that any $v = lca(l_{ig+1}, l_{jg+1})$ is also $v = lca(l_{rg+1}, l_{(r+1)g+1})$ for some r. We prove it by induction: Let us consider nodes $v_1 = lca(l_{ig+1}, l_{(i+1)g+1})$ and $v_2 = lca(l_{(i+1)g+1}, l_{(i+2)g+1})$. Because v_1 and v_2 are both ancestors of $l_{(i+1)g}$, it follows that $v_{12} = lca(v_1, v_2)$ is either v_1 or v_2. By closure property of the lca operation, it follows that $v_{12} = lca(l_{ig+1}, l_{(i+2)g+1})$. 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. Then the inductive thesis holds.

Therefore with these n/g SGST nodes we already have a set of nodes that is closed under lca operation and no further nodes are required. These nodes can be computed in linear time [14]. Gagie et al. [31] point out that we need only $\log D$ sparsified trees τ_k, not $\log n$ as in the original proposal.

6.1.2 Level-Ordered Unary Degree Sequence (LOUDS) GST

SGST is built up on the idea of sparsification to achieve $o(n)$ bits, but the data is still stored explicitly. More precisely, they store $O(n/\log^2 n)$ nodes within their respective *F-list*, L_v, R_v values and pointers to the children.

Our first data structure 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. [2] showed that, for the functionality it provides, the most promising succinct representation of trees is the so-called Level-Order Unary Degree Sequence (LOUDS) [45]. As mentioned in Section 2.6, the theoretical space requirement of $2n + o(n)$ bits can be in practice as little 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 trees implementation [2]. We encode the values L_v and R_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. [2] manage labels.

6.1.3 Other Sources of Redundancy

We note that there is also a great deal of redundancy in the $\log D$ trees τ_k, 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 candidates 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.

In our second data structure, the topology of the trees τ and τ_k is represented using pointers of $\log |\tau|$ and $\log |\tau_k|$ bits, respectively.

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 [43] 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 [16], as many such differences will be small. Also, as Gagie et al. pointed out [31], a major space reduction can be achieved by storing only the identifiers of the candidates, and their frequencies can be computed on the fly using *rank* on the wavelet tree of DA. These variants are analyzed in Section 6.3.

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. [31] 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. [43] and those of Culpepper et al. [23], 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 always 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. [23].

6.2.1 Restricted Depth-First Search

Note that any interval $\mathsf{DA}[i, j]$ can be projected into the left child of the root as

$$[i_0, j_0] \quad = \quad [rank_0(B, i{-}1){+}1, rank_0(B, j)],$$

and into its right child as

$$[i_1, j_1] \quad = \quad [rank_1(B, i{-}1){+}1, rank_1(B, j)].$$

where B is the root bitmap. Those can then be projected recursively into other wavelet tree nodes.

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 zeros and ones 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 zeros in $[l_1, r_1] \cup [l_2, r_2]$, we map all the intervals into the left child of the node and proceed recursively from this node. Similarly, if there are any ones in $[l_1, r_1] \cup [l_2, r_2]$, we continue on the right child of the node.

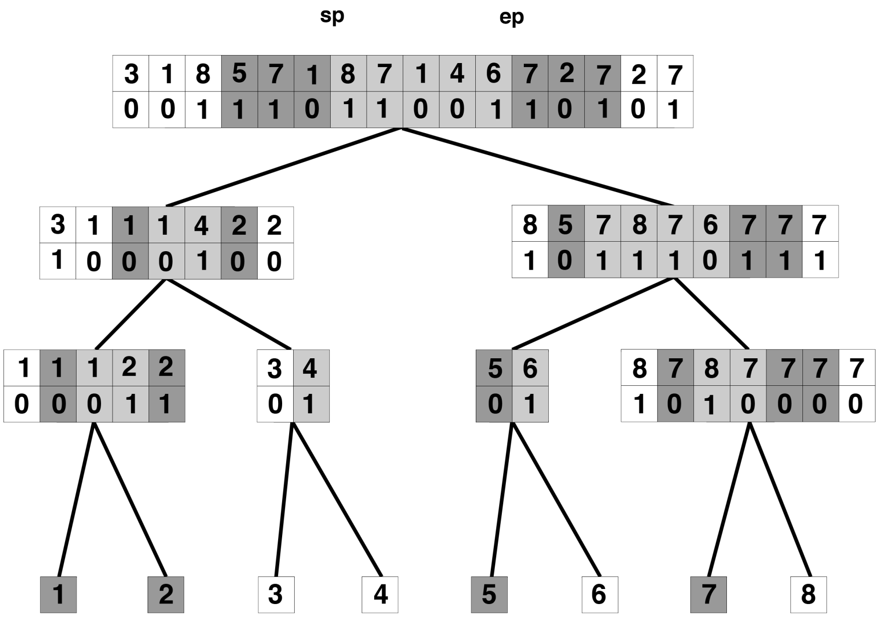


Figure 6.1: Restricted DFS to obtain the frequencies of documents not covered by τ_k. Shaded regions show the interval $[sp, ep] = [4, 14]$ mapped to each wavelet tree node. Dark shaded intervals are the projections of the leaves not covered by $[sp', ep'] = [7, 11]$.

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 6.1 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., 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.

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 constant time.

Given that the Greedy algorithm stops when it reaches a wavelet tree node where $r-l+1$ is not larger than the kth known frequency, it follows that each time the algorithm reports a new candidate, that reported candidate has a higher frequency value than our previous kth known candidate. Thus we replace the top of our heap with the reported candidate and reorder the heap (which is always of size k, or less until we find k distinct elements in $\mathsf{DA}[sp, ep]$). Therefore each reported candidate costs $O(\log D + \log k)$ time (there are also steps that do not yield any result, but the overall upper bound is still $O(g(\log D + \log k))$). The DFS algorithm does not report the documents in a particular order, so we need to check if a reported document has a higher frequency than the minimum of the heap. If it occurs, then we replace the top of the heap with the reported candidate and reorder the heap. The overall bound is still $O(g(\log D + \log k))$.

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, for the small k values used in practice (say, tens or at most hundreds), it is more practical to scan the heap for each new candidate to insert than to maintain all those pointers upon all operations.

6.3 Experimental Results

In this section we 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 [43].

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. [43], 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 6.2 to 6.5 show the results. As expected, method *Greedy* is 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, *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 [43].

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.

We consider sampling steps of 200, 400 and 800 for g'. For each value of g, we obtain a curve with various sampling steps for the *rank* computations on the wavelet tree bitmaps. Figures 6.6 to 6.9 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

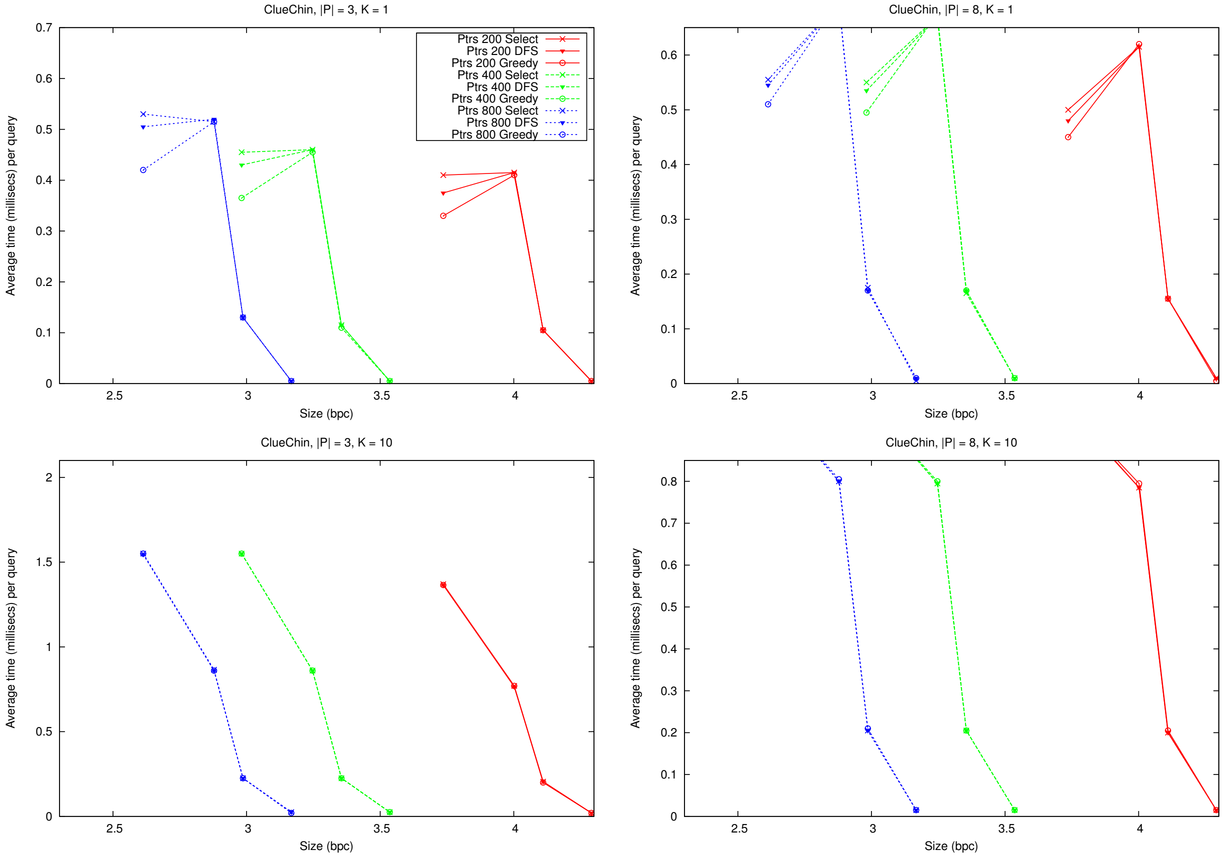


Figure 6.2: Our different algorithms for top-1, and top-10 queries on the collection ClueChin. On the left for $|P| = 3$, on the right for $|P| = 8$.

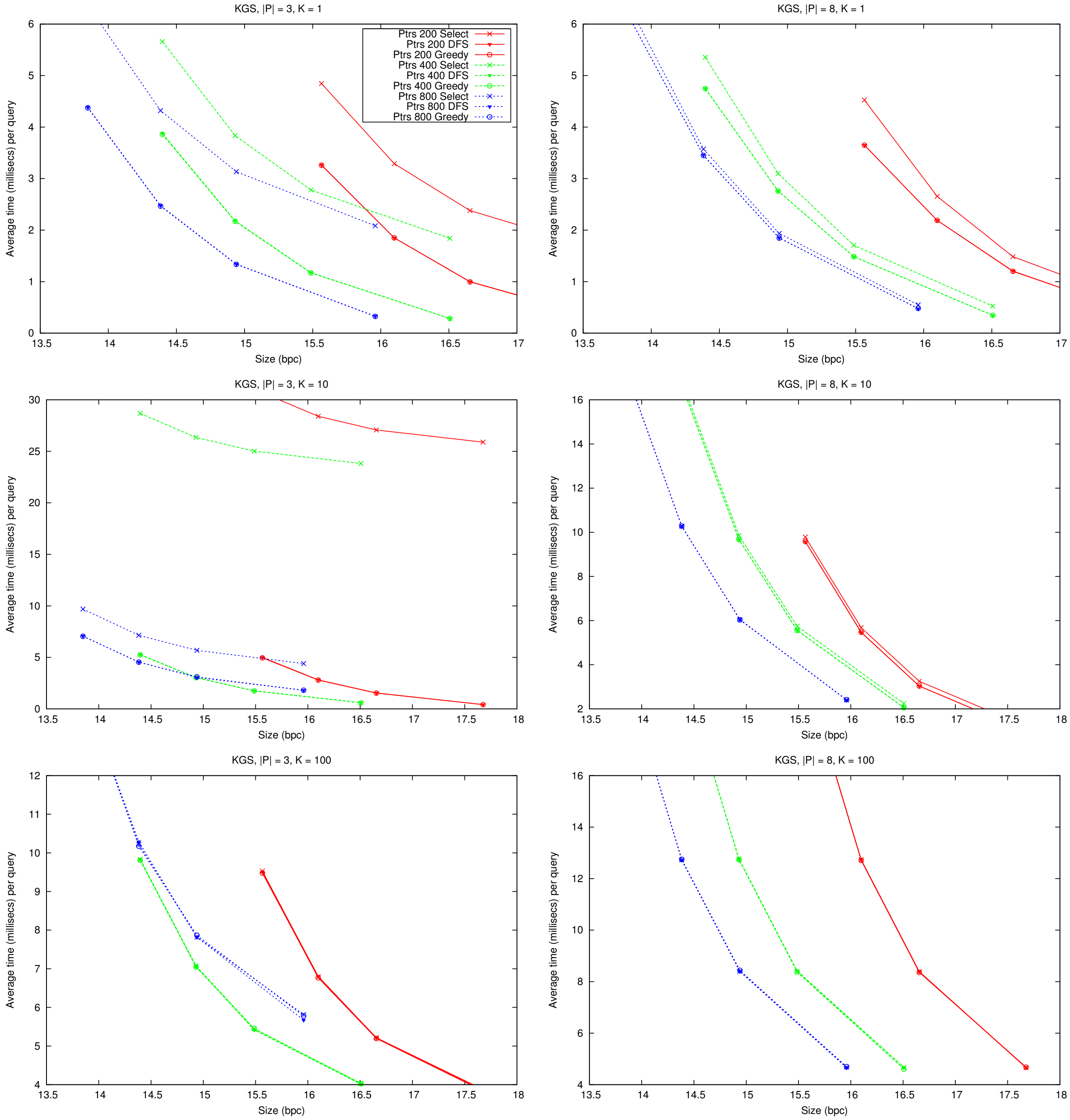


Figure 6.3: Our different algorithms for top-1, top-10 and top-100 queries on the collection KGS. On the left for pattern length $|P| = 3$, and on the right for $|P| = 8$.

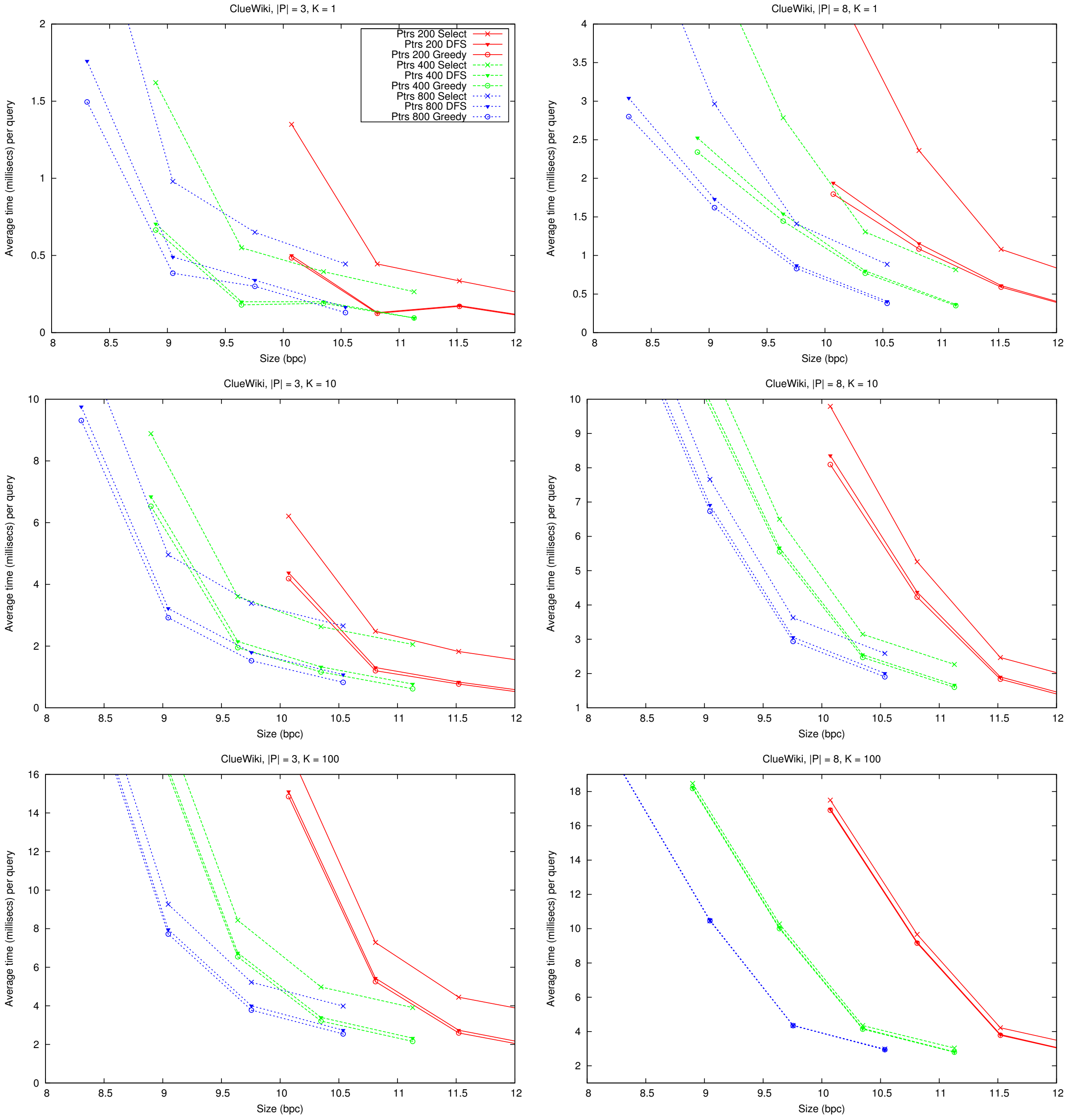


Figure 6.4: Our different algorithms for top-1, top-10 and top-100 queries on the collection ClueWiki. On the left for pattern length $|P| = 3$, and on the right for $|P| = 8$.

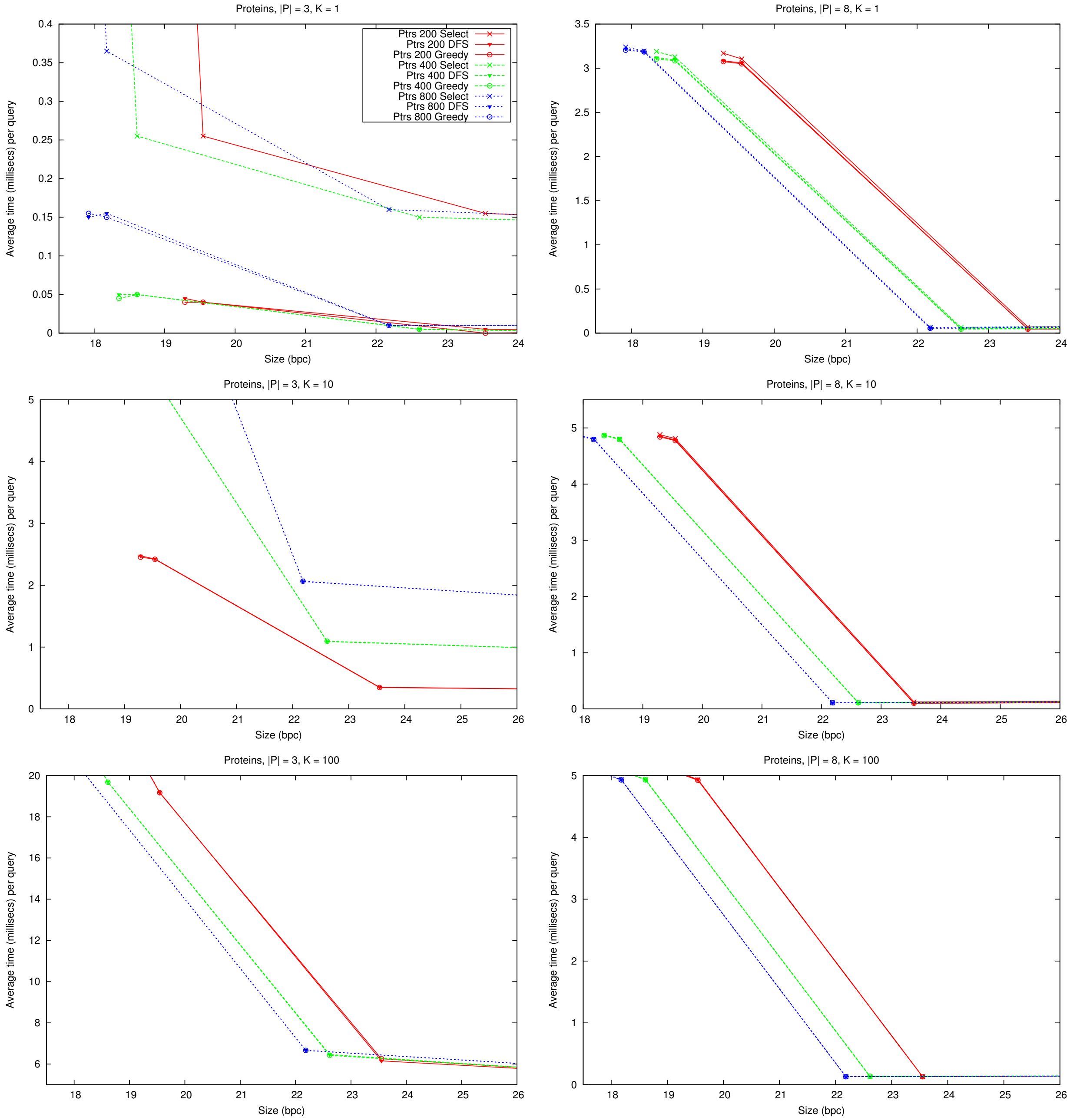


Figure 6.5: Our different algorithms for top-1, top-10 and top-100 queries on the collection Proteins. On the left for pattern length $|P| = 3$, and on the right for $|P| = 8$.

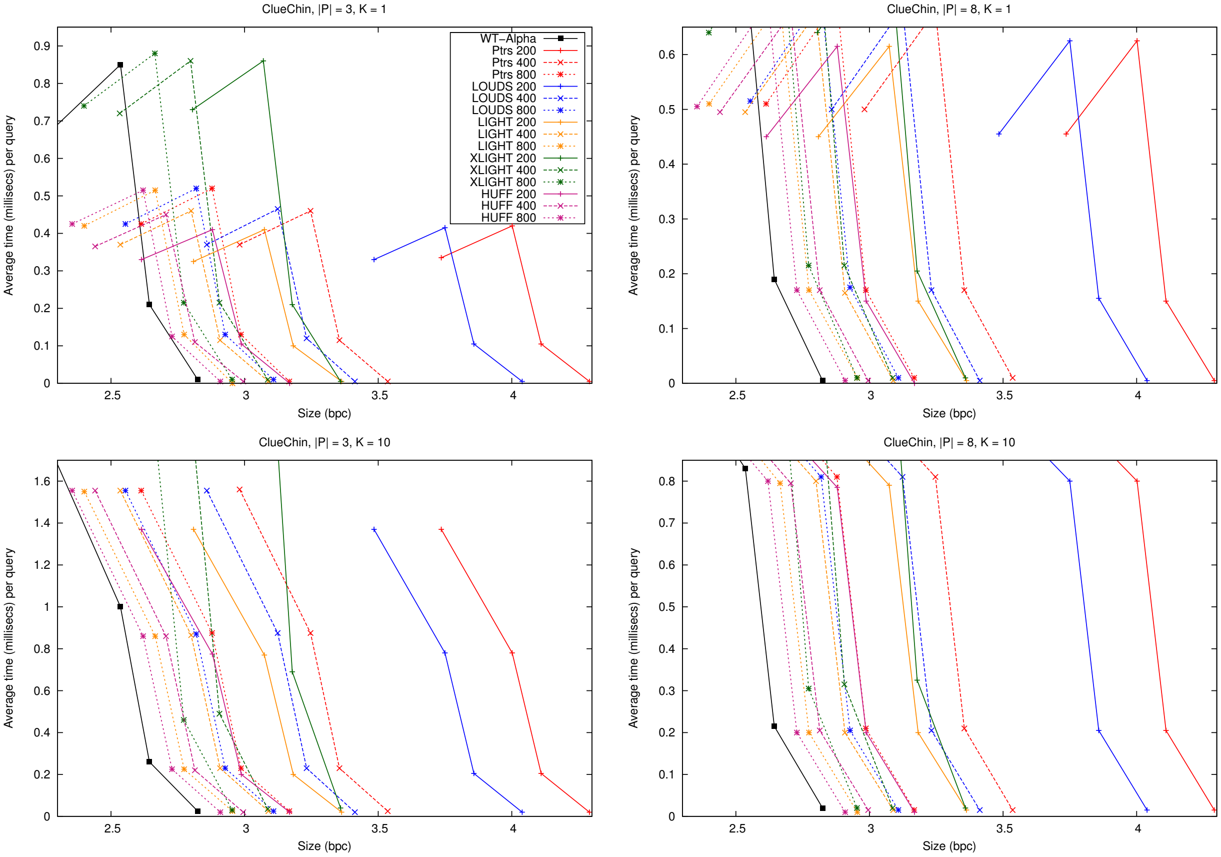


Figure 6.6: Our different data structures for top-1 and top-10 queries on the collection ClueChin. On the left for pattern length $|P| = 3$, and on the right for $|P| = 8$.

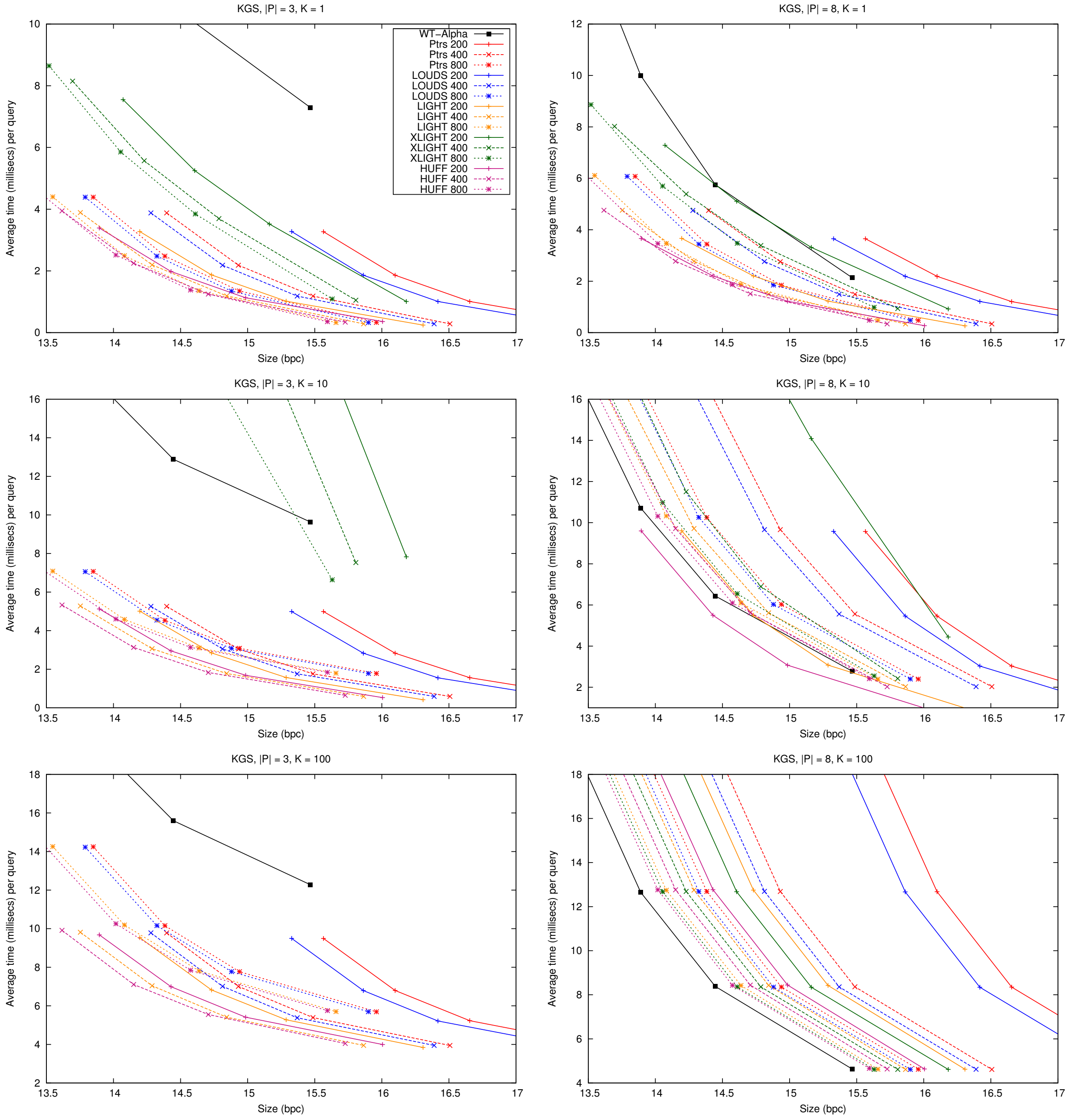


Figure 6.7: Our different data structures for top-1,top-10 and top-100 queries on the collection KGS. On the left for pattern length $|P| = 3$, and on the right for $|P| = 8$.

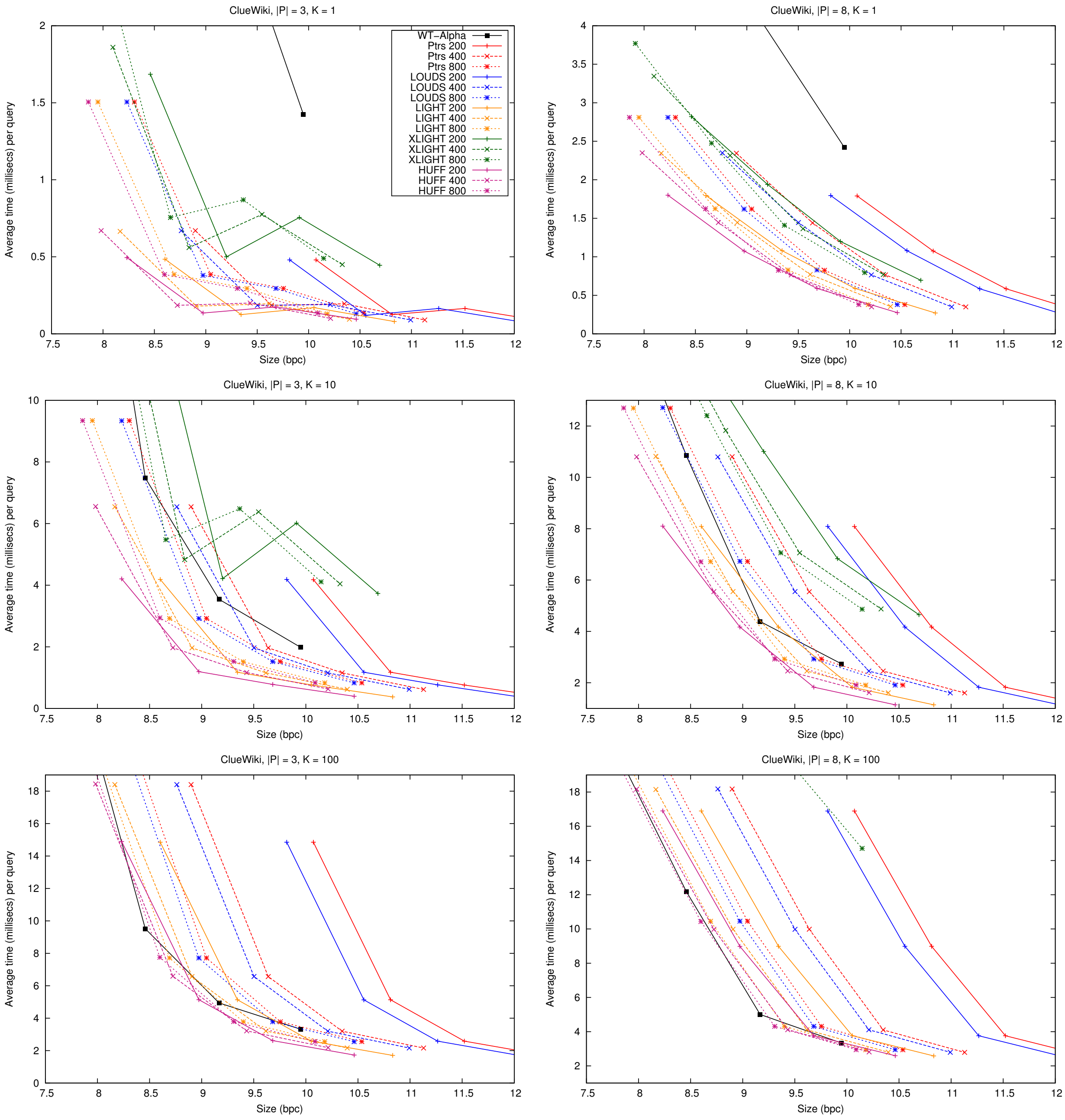


Figure 6.8: Our different data structures for top-1,top-10 and top-100 queries on the collection ClueWiki. On the left for pattern length $|P| = 3$, and on the right for $|P| = 8$.

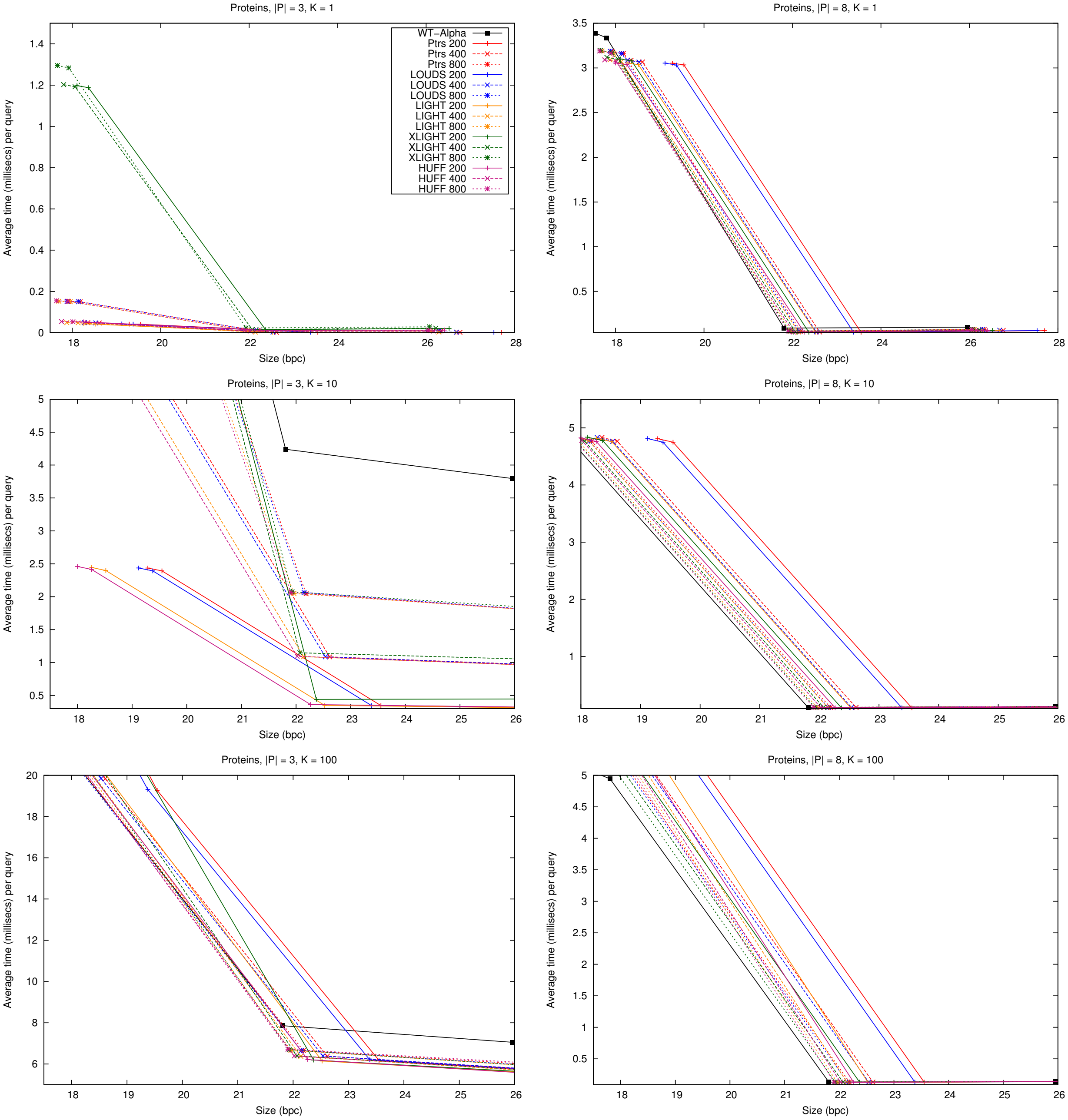


Figure 6.9: Our different data structures for top-1,top-10 and top-100 queries on the collection Proteins. On the left for pattern length $|P| = 3$, and on the right for $|P| = 8$.

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, 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. 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.

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. In various cases a sparser sampling dominates a denser one in space and time, whereas in others a denser sampling makes the structure faster. 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. [23] is run over the different wavelet tree representations of the document array from Chapter 5: a plain one (*WT-Plain*, as in the original proposal) [23], an entropy-compressed one (*WT-RRR*), a 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 those wavelet tree representations with our best implementation of Hon et al.'s structure (suffixing the previous names with *+SSGST*). We also consider variant *Goly+SSGST* [31, 42], which runs the *rank*-based method (*Select*) on top of the fastest *rank*-capable sequence representation of the document array [35]. This representation is faster than wavelet trees to compute *rank*, but does not support our more sophisticated traversal algorithms.

Figures 6.10 to 6.13 show the results. *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`, 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*).

It is important to remark that, although in some cases *WT-RP* is the fastest alternative (e.g. for high values of k), our variant *WT-Alpha + HUFF* includes the wavelet tree, so it supports Culpepper et.al.'s original algorithms. Thus in a real-world application it would be interesting to store the τ_k only up to certain value, and when a top-k query for higher values of k is required, it can be answered using the original algorithms.

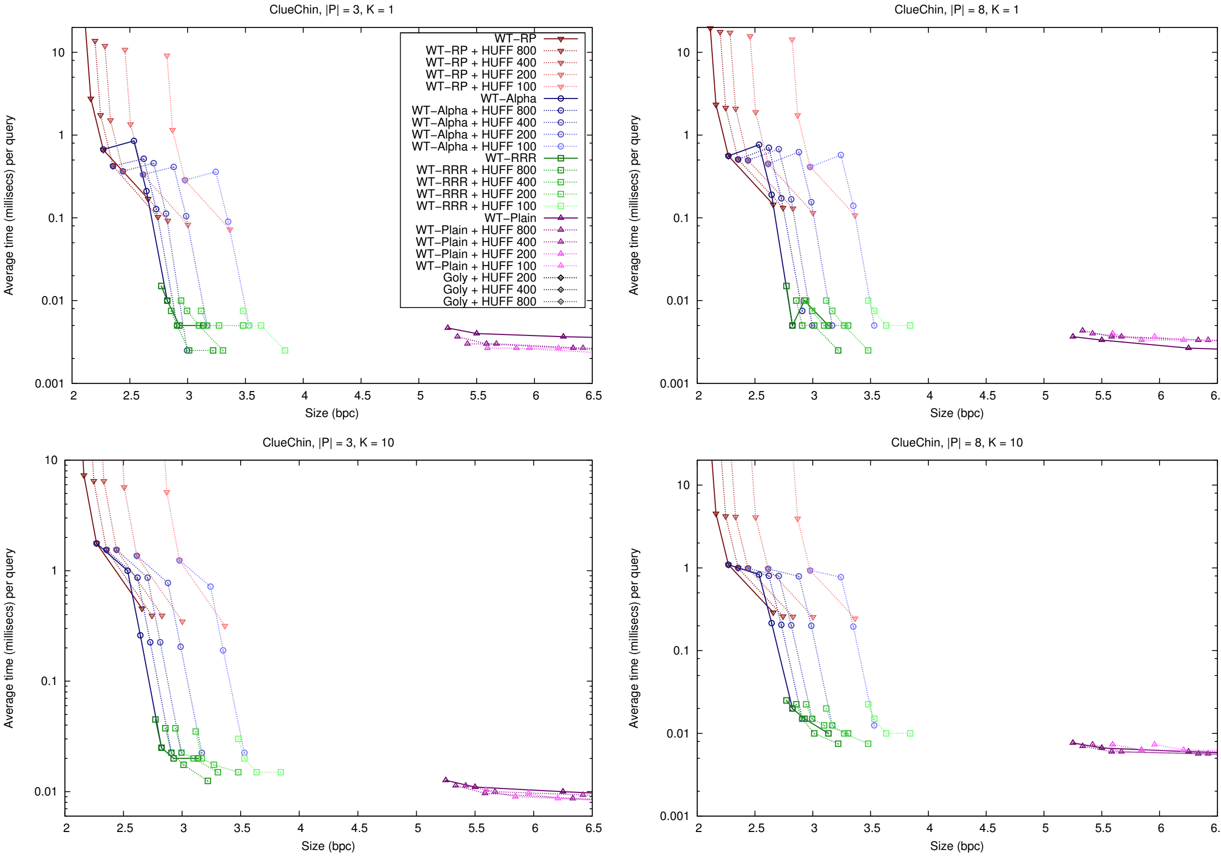


Figure 6.10: Comparison with previous work for top-1 and top-10 queries on the collection ClueChin. On the left for $|P| = 3$, on the right for $|P| = 8$.

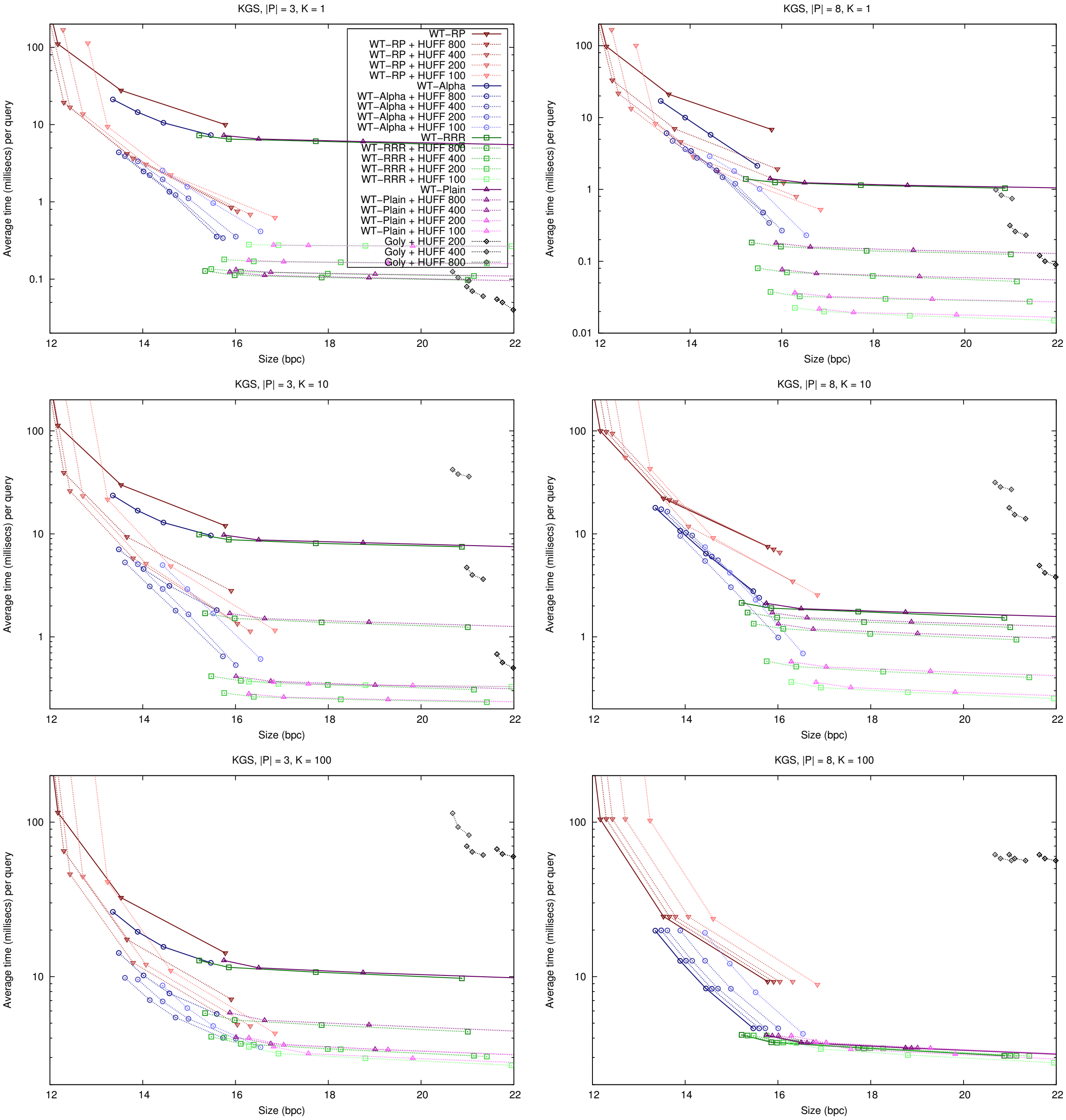


Figure 6.11: Comparison with previous work for top-1, top-10 and top-100 queries on the collection KGS. On the left for $|P| = 3$, on the right for $|P| = 8$.

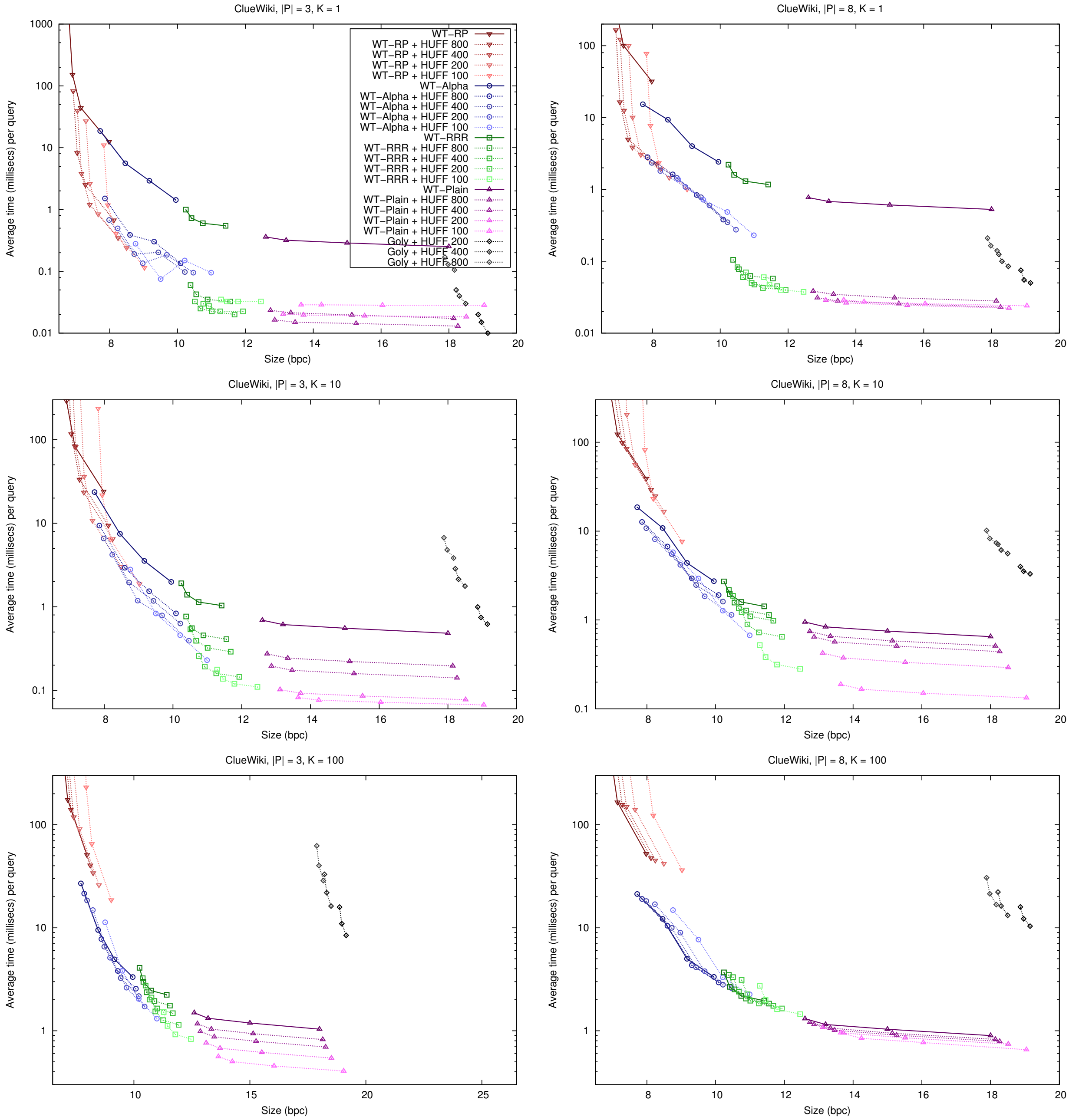


Figure 6.12: Comparison with previous work for top-1, top-10 and top-100 queries on the collection ClueWiki. On the left for $|P| = 3$, on the right for $|P| = 8$.

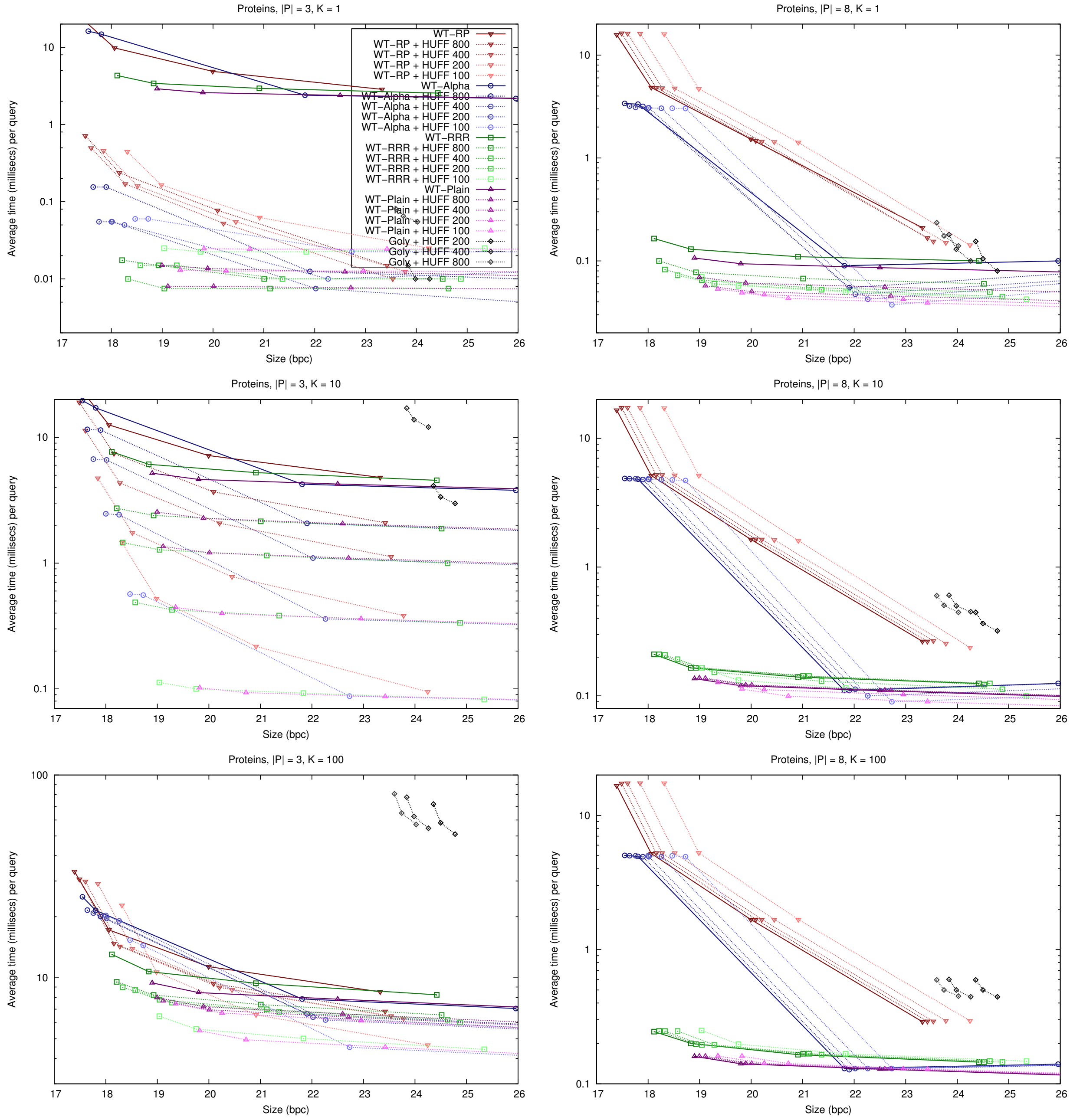


Figure 6.13: Comparison with previous work for top-1, top-10 and top-100 queries on the collection Proteins. On the left for $|P| = 3$, on the right for $|P| = 8$.

Chapter 7

Mmphf Approach for Document Listing

In this chapter we present our implementation of a different approach to solve the problem of document listing with frequencies. This approach was recently stated by Belazzougui and Navarro [12] and is based on Monotone Minimal Perfect Hash Functions (mmphfs) [10, 11] (see Section 3.5).

We implemented their idea for document listing with frequencies, in order to explore the practical potential of mmphfs. In Chapter 5 we showed that the approaches based on individuals CSAs were not efficient in practice, despite their theoretical soundness. In Chapters 5 and 6 we advocated for exploring the practicality of the wavelet tree representation of the document array. However, both of those approaches store redundant information that poses serious space overheads, both in theory and in practice. Mmphf is another approach, which have a potential high-impact in practice, because it aims to get rid of the aforementioned redundancy.

7.1 In Practiece

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

For the mmphf-based method, we use a practical mmphf implementation by Belazzougui et al. (space-optimized since its original publication [11]). Section 3.6 describes the CSA and RMQ we used to implement mmphf approach in detail.

We implement the algorithm as described in Section 3.5, except that we do a naive sorting of the leftmost and rightmost occurrences.

Note that the CSA is used by all the techniques to obtain *sp* and *ep*, but in addition this method uses it to compute suffix array cell contents. To carry out this task the CSA makes use of a further sampling, whose space cost will be charged (only) to our data structure. Note this is not totally fair because, in a scenario where one wants to carry out document retrieval and pattern locating queries, we would use the same sampling structure for both activities. On the other hand, its impact is relatively low.

7.2 Experimental Results

Table 7.1 gives the space of the different substructures that make up this solution. It is interesting that spaces are roughly equal and basically independent of, say, how compressible the collection is. Note also that the space is basically independent of the number of documents in the collection. This is in contrast to the $O(n \log D)$ space of wavelet-tree-based solutions, and suggests that our scheme could compare better on much larger test collections.

Collection	RMQs	B	mmphf	CSA	Total
ClueChin	5.90	0.34	5.39	$21/z$	$11.63{+}22/z$
ClueWiki	5.84	0.32	4.67	$28/z$	$10.83{+}26/z$
KGS	5.82	0.34	3.14	$25/z$	$9.30{+}25/z$
Proteins	5.81	0.36	3.56	$26/z$	$9.73{+}26/z$

Table 7.1: Space breakdown, in bpc, of our scheme for the three collections. Value z is the sampling step chosen to support access to the CSA cells. The total space includes *two* RMQ structures, B, mmphf, and the CSA sampling.

As for times, we note that each occurrence we report requires to compute 4 RMQ queries, 2 accesses to B and to the CSA, and 0 or 2 mmphfs (this can be zero in the case the leftmost and rightmost positions are the same, so we know that the frequency is 1 without invoking the mmphf). Our queries are made of 10,000 random intervals of the form $[sp, ep]$, for interval sizes $ep - sp + 1$ from 100 to 100,000, and for every length we listed the distinct documents in the interval, with their frequencies. We made a first experiment replacing the CSA with a plain suffix array (where t_{SA} corresponds to simply accessing a cell). The times were 12.8 msec on ClueWiki and 18.4 msec on KGS and Proteins. These are significantly lower than the times we will see on CSAs, which shows that the time performance is sharply dominated by parameter z. Due to the design of Sadakane's CSA (and most implemented CSAs, in fact), the time t_{SA} is essentially linear on z. This gives our space/time tradeoff.

We compare the mmphfs approach with the four wavelet tree variants of Chapter 5, namely *WT-Plain*, *WT-RP*, *WT-RRR*, and *WT-Alpha*. In their case, space-time tradeoffs are obtained by varying various samplings. They are stretched up to essentially the minimum space they can possibly use.

Figures 7.1 to 7.4 give the space/time results. When we compare to *WT-Plain*, which is the basic wavelet-tree-based theoretical solution, the mmphf-based technique makes good on its theoretical promise of using less space (more clearly on collections Proteins and KGS, where D is sufficiently large). The wavelet tree uses 12.5 to 19 bpc, depending on the number of documents in the collection. The mmphf technique uses, in our experiments, as little as 12 bpc. On the other hand, the time $O(t_{\mathsf{SA}})$ spent for each document reported turns out to be, in practice, much higher than the $O(\log D)$ used to access the wavelet tree. The space/time tradeoff obtained is likely to keep improving on collections with even more documents, as the space and time of wavelet trees grow with $\log D$, whereas the mmphf solution has a time independent of D, and a space that depends log-logarithmically (or less) on D.

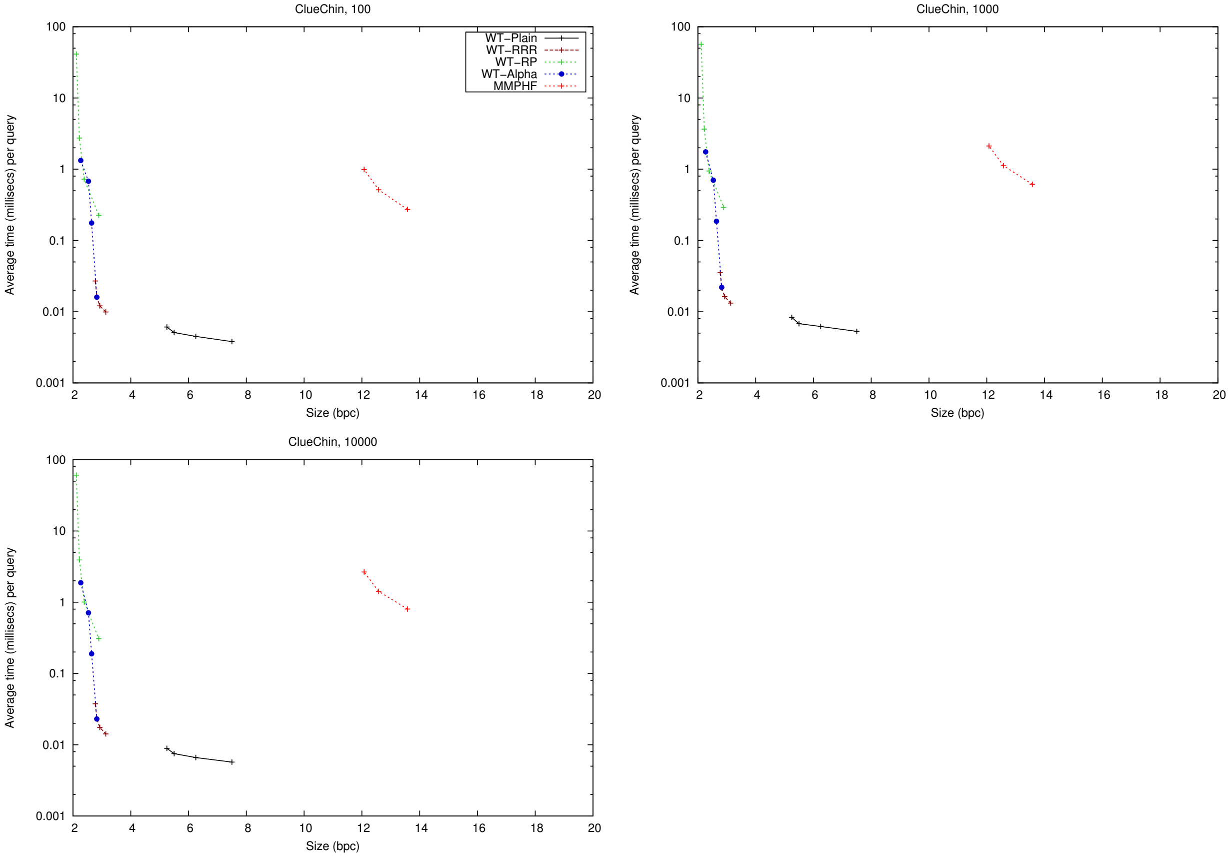


Figure 7.1: Experiments for document listing with term frequencies, collection ClueChin.

When we consider the practical improvements to compress wavelet trees (Chapter 5), however, we have that these offer more attractive tradeoffs on collections ClueChin, ClueWiki and KGS, whereas on Proteins our attempts to compress has a very limited effect, as seen in Section 5.4.

The mmphf technique stands as a robust alternative whose performance is very easy to predict, independently of the characteristics of the collection. We notice that this technique is more sensitive to the length of the $[sp, ep]$ interval than the solutions presented in Chapter 5. Therefore, the mmphf technique is interesting for uncommon queries, that will produce shorter $[sp, ep]$ intervals. On large collections that are resilient to the known techniques to compress the wavelet tree, the mmphf-based solution offers a relevant space/time tradeoff. Moreover, the technique is likely to be more scalable, as explained, and its times benefit directly from any improvement in access times to Compressed Suffix Arrays.

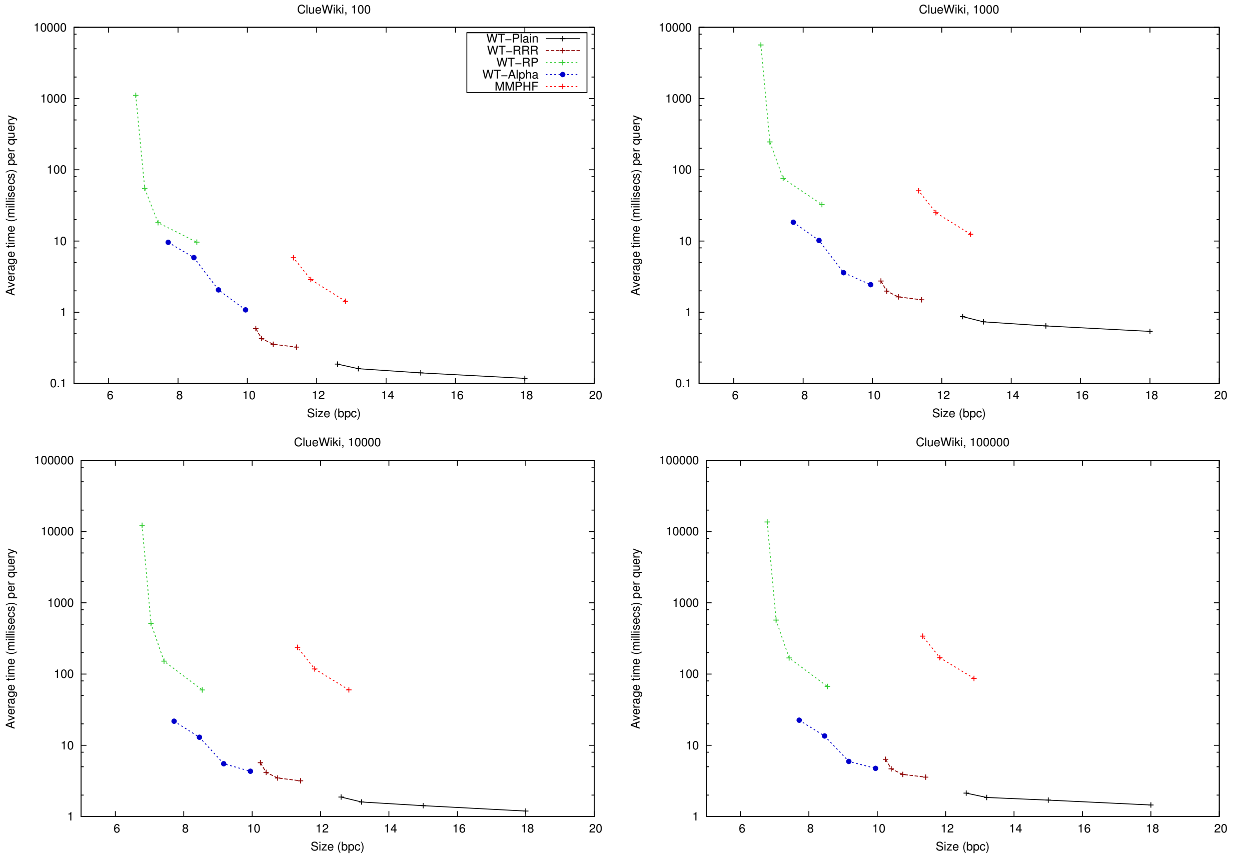


Figure 7.2: Experiments for document listing with term frequencies, collection ClueWiki.

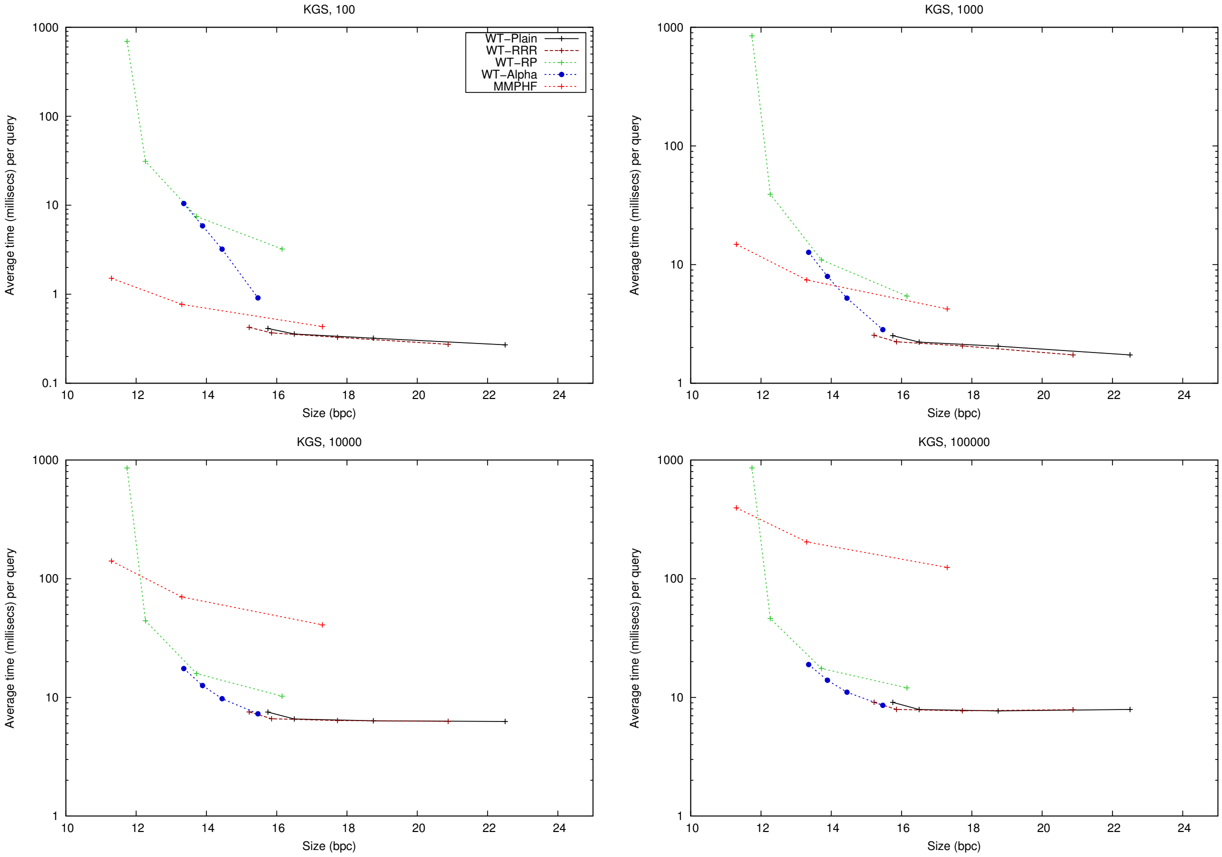


Figure 7.3: Experiments for document listing with term frequencies, collection KGS.

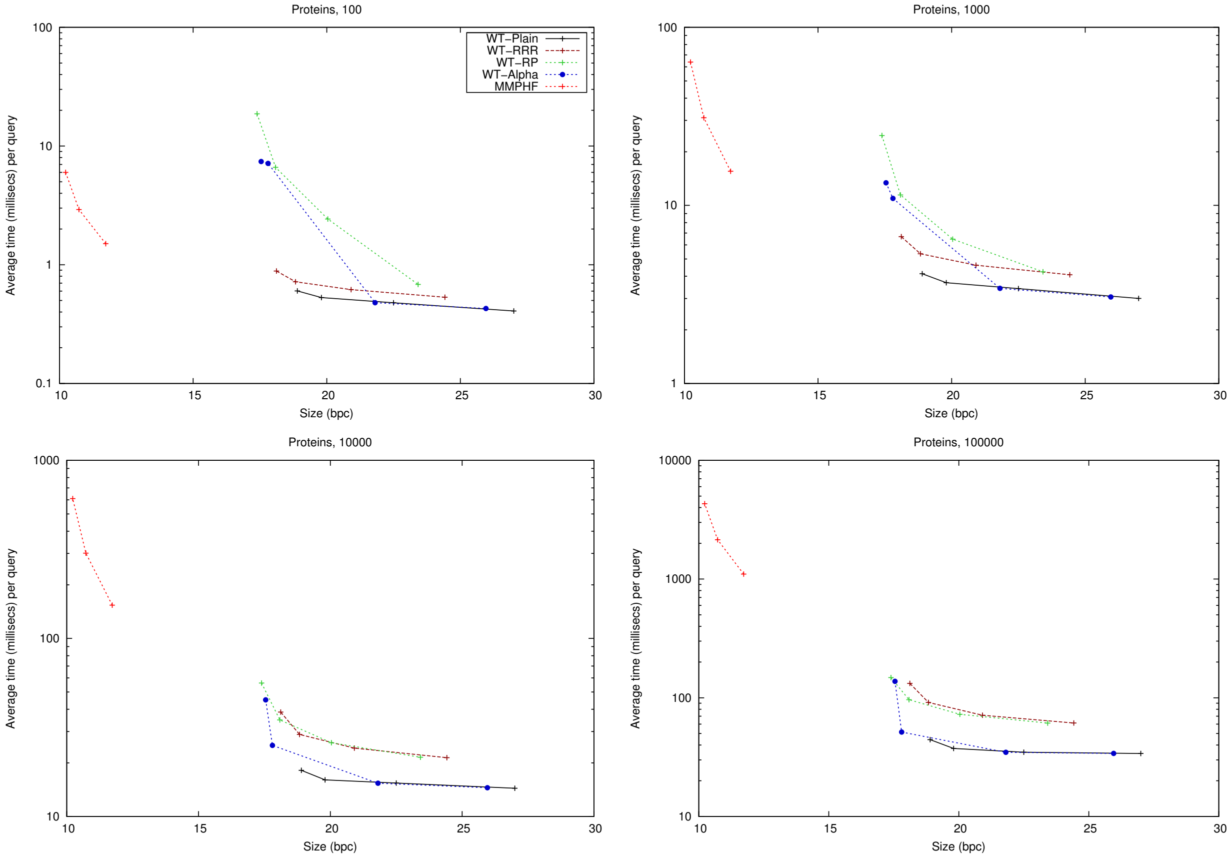


Figure 7.4: Experiments for document listing with term frequencies, collection Proteins.

Chapter 8

Conclusions

This work has introduced new algorithms and data structures to solve some important document retrieval problems. We have improved the state of the art for document listing and top-k document retrieval, and as byproduct, we have introduced new compression schemes for relevant data structures.

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 developed 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 to answer document listing with frequencies, as well as for retrieving the top-k documents.

We also implemented the theoretical proposal of Hon et al. [43]. First 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). We developed new algorithms that allowed this structure to run on top of a unique wavelet tree instead of individual CSAs, and showed that this technique performs very well in practice. Our implementation of this data structure removes various sources of redundancy and inefficiency of the original proposal (which are neglectable in an asymptotic analysis but relevant in practice).

While Hon et al.'s original proposal, that runs over CSAs, has worst-case guarantees, in practice it requires 2-3 times the space required by the plain solution, while being 10 times slower to answer the queries. On the other hand, the original algorithms over wavelet trees were the best in practice, but had no worst-case guarantees. Our new algorithms have the best of both worlds: They retain the theoretical bounds of Hon et al. and perform very well in practice.

Our implementation of Hon et al., running in combination with the best wavelet tree variants, dominate most of the space-time map.

We also explored the practicality of a different approach, based on monotone minimal perfect hash functions (mmphf). Even though this approach was not the best performing in most collections, we verified that in practice it behaves as theoretically expected: the space overhead shows a mild dependence on the nature of the collection. The same happens regarding to the time needed to solve the document listing with frequencies problem. Therefore, this technique has a great potential to scale very well for use with larger collections. On collection *Proteins*, which is not compressible, our previous techniques did not improve over previous work, however the mmphf techinque offered much better performance.

8.1 Future Work

Term frequency is probably the simplest relevance measure. In Information Retrieval, more sophisticated ones like BM25 are used. Such a formula involves the sizes of the documents, and thus techniques like Culpeppper et al.'s [23] do not immediately apply. However, Hon et al.'s [43] does, by simply storing the precomputed top-k answers according to BM25 and using their brute-force method for the uncovered cells, instead of our restricted greedy or DFS methods. The times would be very similar to the variant we called *Select* in Section 6.3.1.

Sadakane [68] showed how to efficiently compute *document frequencies* (i.e., in how many documents does a pattern appear), in constant time and using just $2n + o(n)$ bits. With term frequency, these two measures are sufficient to compute the popular tf-idf score. Note, however, that as long as queries are formed by a single term, the top-k sorting is the same as given by term frequency alone. Document frequency makes a difference on *bag-of-word* queries, which involve several terms. Structures like those we have explored in this paper are able to emulate a (virtual) inverted list, sorted by decreasing term frequency, for any pattern, and thus enable the implementation of any top-k algorithm for bags of words designed for inverted indexes. However, it is possible that extensions of the heuristics of Culpepper et al. [23] support better techniques natively on the wavelet trees.

Bibliography

[1] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *Proc. 27th Annual ACM Symposium on Theory of Computing (STOC)*, pages 427–436, 1995.

[2] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. 11th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 84–97, 2010.

[3] D. Arroyuelo, G. Navarro, and K. Sadakane. Stronger Lempel-Ziv based compressed text indexing. *Algorithmica*, 62(1):54–101, 2012.

[4] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Pearson Education, 2nd edition, 2011.

[5] R. Baeza-Yates and A. Salinger. Fast intersection algorithms for sorted sequences. In *Algorithms and Applications*, pages 45–61. 2010.

[6] J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet partitioning for compressed rank/select and applications. In *Proc. 21st Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS 6507, pages 315–326 (part II), 2010.

[7] J. Barbay, M. He, I. Munro, and S. Srinivasa Rao. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Transactions on Algorithms*, 7(4):52, 2011.

[8] Jérémy Barbay and Claire Kenyon. Alternation and redundancy analysis of the intersection problem. *ACM Transactions on Algorithms*, 4(1), 2008.

[9] Jérémy Barbay, Alejandro López-Ortiz, Tyler Lu, and Alejandro Salinger. An experimental investigation of set intersection algorithms for text searching. *ACM Journal of Experimental Algorithmics*, 14, 2009.

[10] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Monotone minimal perfect hashing: searching a sorted table with $o(1)$ accesses. In *Proc. 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 785–794, 2009.

[11] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Theory and practise of monotone minimal perfect hashing. In *Proc. 10th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2009.

[12] D. Belazzougui and G. Navarro. Improved compressed indexes for full-text document retrieval. In *Proc. 18th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 7024, pages 386–397, 2011.

[13] D. Belazzougui, G. Navarro, and D. Valenzuela. Improved compressed indexes for full-text document retrieval. *Journal of Discrete Algorithms*, 2012. To appear.

[14] M. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. 2nd Latin American Symposium on Theoretical Informatics (LATIN)*, pages 88–94, 2000.

[15] D. Benoit, E.D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[16] N. Brisaboa, S. Ladra, and G. Navarro. Directly addressable variable-length codes. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5721, pages 122–130, 2009.

[17] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[18] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[19] D. Clark. *Compact pat trees.* PhD thesis, University of Waterloo, Waterloo, Ont., Canada, 1998.

[20] F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro. Compressed q-gram indexing for highly repetitive biological sequences. In *Proc. 10th IEEE Conference on Bioinformatics and Bioengineering (BIBE)*, pages 86–91, 2010.

[21] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187, 2008.

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

[23] S. Culpepper, G. Navarro, S. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In *18th Annual European Symposium on Algorithms (ESA)*, LNCS 6347, pages 194–205 (part II), 2010.

[24] A. Fariña, N. Brisaboa, G. Navarro, F. Claude, A. Places, and E. Rodríguez. Word-based self-indexes for natural language text. *ACM Transactions on Information Systems (TOIS)*, 30(1):article 1, 2012.

[25] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13:article 12, 2009.

[26] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.

[27] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[28] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proc. 11th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 3246, pages 150–160, 2004.

[29] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.

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

[31] T. Gagie, G. Navarro, and S. Puglisi. Colored range queries and document retrieval. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 6393, pages 67–81, 2010.

[32] T. Gagie, G. Navarro, and S.J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426-427:25–41, 2012.

[33] T. Gagie, S. J. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proc. 16th Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5721, pages 1–6, 2009.

[34] R. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368(3):231–246, 2006.

[35] A. Golynski, I. Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

[36] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Posters Proc. of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, Greece, 2005. CTI Press and Ellinika Grammata.

[37] R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 216–227, 2007.

[38] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[39] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2006.

[40] H. S. Heaps. *Information Retrieval: Computational and Theoretical Aspects*. Academic Press, 1978.

[41] C. Hernández and G. Navarro. Compressed representation of web and social networks via dense subgraphs. In *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 7608, pages 264–276, 2012.

[42] W.-K. Hon, R. Shah, and S. Thankachan. 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, pages 173–184, 2012.

[43] W.-K. Hon, R. Shah, and J. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. 50th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 713–722, 2009.

[44] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E.*, 40(9):1098–1101, 1952.

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

[46] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP)*, pages 141–155. Carleton University Press, 1996.

[47] R. Kosaraju and Giovanni Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29:893–911, 1999.

[48] N.J. Larsson and J. A. Moffat. Offline dictionary-based compression. *Proceedings of the IEEE*, 88:1722–1732, 2000.

[49] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

[50] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 214–226, 2007.

[51] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[52] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[53] S. Maruyama, H. Sakamoto, and M. Takeda. An online algorithm for lightweight grammar-based compression. *Algorithms*, 5(2):214–235, 2012.

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

[55] I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2002.

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

[57] G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms*, 2(1):87–114, 2004.

[58] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[59] G. Navarro and Y. Nekrich. Top-k document retrieval in optimal time and linear space. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1066–1078, 2012.

[60] G. Navarro, S. Puglisi, and D. Valenzuela. Practical compressed document retrieval. In *Proc. 10th International Symposium on Experimental Algorithms (SEA)*, LNCS 6630, pages 193–205, 2011.

[61] G. Navarro and L. Russo. Re-pair achieves high-order entropy. In *Proc. 18th Data Compression Conference (DCC)*, page 537, 2008.

[62] G. Navarro and D. Valenzuela. Space-efficient top-k document retrieval. In *Proc. 11th International Symposium on Experimental Algorithms (SEA)*, LNCS, 2012.

[63] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[64] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

[65] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.

[66] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 225–232, 2002.

[67] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[68] K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.

[69] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 134–149, 2010.

[70] H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *Journal on Discrete Algorithms*, 3:416–430, 2005.

[71] P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 71–83.

[72] N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 205–215, 2007.

[73] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23:229–239, 1980.

[74] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[75] H. Williams and J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42:193–201, 1999.

[76] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes.* Morgan Kaufmann, 2nd edition, 1999.

[77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530 – 536, 1978.