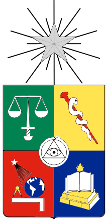

UNIVERSITY OF CHILE
FACULTY OF PHYSICAL AND MATHEMATICAL SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

ÍNDICES COMPRIMIDOS PARA LA RECUPERACIÓN DE DOCUMENTOS

THESIS SUBMITTED IN FULFILMENT OF THE THESIS REQUIREMENTS FOR THE DEGREE OF PH.D. IN COMPUTER SCIENCE

HÉCTOR RICARDO FERRADA ESCOBAR

ADVISOR:
GONZALO NAVARRO BADINO

COMMITTEE:
DIEGO ARROYUELO BILLIARDI
BENJAMÍN BUSTOS CÁRDENAS
KUNIHIKO SADAKANE

This work has been partially funded by CONICYT Ph.D Scholarship Chile; Fondecyt Grant 1-140976; Millennium Nucleus for Information and Coordination in Networks, and Basal Center for Biotechnology and Bioengineering..

SANTIAGO OF CHILE
2016

Resumen

Document Retrieval (DR) apunta a la recuperarción eficiente de documentos relevantes de una colección, para las consultas del usuario. Una variante que surge como desafío es cuando los documentos provienen de una gran coleción de textos arbitrarios. Este escenario ocurre con colecciones de secuencias de ADN o proteínas, repositorios de software, secuencias multimedia e idiomas del Lejano Oriente, entre otros entornos. Varias estructuras de datos comprimidas para DR han sido desarrolladas a fin de hacer frente a este desafío, ofreciendo diferentes complejidades en tiempo/espacio. Sin embargo, en la práctica las propuestas con el mejor rendimiento en tiempo, requieren a su vez de demasiado espacio extra.

Esta tesis innova tres aspectos: (1) construímos índices para DR en base a la compresión Lempel-Ziv 1978 (LZ78) en lugar de arreglos de sufijos; (2) manipulamos colecciones altamente repetitivas en base a la compresión Lempel-Ziv 1977 (LZ77); (3) comenzamos a estudiar cómo entregar respuestas aproximadas en dicho escenario de DR, lo cual es una práctica común en textos de lenguaje natural.

Nuestra principal contribución es un nuevo enfoque para DR basado en la compresión de datos LZ78, ofreciendo estructuras que resuelven los dos problemas fundamentales del campo de DR: *Document Listing* (DL) y *Top-k Retrieval*. Nuestros nuevos índices ofrecen desempeño competitivo en tiempo/espacio en ambos casos. Además nuestras propuestas también entregan respuestas aproximadas, ahorrando considerable espacio y/o tiempo comparado con cualquier otra estructura que entregue una respuesta completa a alguno de estos problemas.

También diseñamos una estructura que indexa colecciones de texto altamente repetitivo y resuelve el problema de DL, basada en la compresión LZ77. Este el primer intento dirigido a resolver un problema de DR utilizando compresión de datos LZ77, que además es el mejor esquema de compresión para dichas colecciones.

Por otro lado, relizamos mejoras sobre estructuras de datos básicas utilizadas en DR. Presentamos un diseño alternativo a la mejor solución teórica para *Range Minimum Queries*, manteniendo sus buenas complejidades en términos de espacio utilizado y tiempo de consulta. Logramos una formula más sencilla obteniendo como resultado la implementación más rápida y compacta conocida hasta hoy.

Además implementamos varias propuestas teóricas promisorias para el arreglo de sufijos, de las cuales no existen implementaciones previas. Finalmente, diseñamos e implementamos un índice de texto comprimido para colecciones altamente repetitivas que resuelve el *pattern matching*, el cual se basa en la compresión LZ77, y que además es la base para nuestro índice sobre el LZ77 para DR.

Abstract

Document Retrieval (DR) aims at efficiently retrieving the documents from a collection that are relevant to user queries. A challenging variant arises when the documents are arbitrary strings and the collection is large. This scenario arises in DNA or protein sequence collections, software repositories, multimedia sequences, East Asian languages, and others. Several DR compressed data structures have been developed to face this challenge, offering different space/time complexities. However, in practice the proposals with the best time performance require too much extra space.

This thesis innovates in three aspects: (1) we build on Lempel-Ziv 1978 (LZ78) compression, instead of suffix arrays, to build DR indices; (2) we build on Lempel-Ziv 1977 (LZ77) compression to handle highly repetitive collections; (3) we start the study of approximate answers in this DR scenario, which is common in DR on natural language texts.

In this aspect, our main contribution is a new approach to DR based on LZ78 data compression, offering structures to solve the two most fundamental problems in the DR field: Document Listing (DL) and Top-k Retrieval. Our novel indices offer a competitive space/time tradeoff for both situations. Besides, our proposals are also capable of retrieving approximate answers, saving a lot of space and/or time compared with any structure that returns the full answer for any of these problems.

Our second main contribution is the design of a structure for indexing highly repetitive text collections that solves the DL problem, which is built on the LZ77 parsing. This is the first attempt to solve DR problems using LZ77 data compression, which is the best compression scheme for such collections.

On the other hand, we improve on basic data structures used, among others, in DR. We present an alternative design to the best theoretical Range Minimum Queries solution, maintaining its good complexities in space usage and query time. We obtain a simpler formula that leads to the fastest and most compact practical implementation to date.

We also implemented various promising theoretical proposals for compressed suffix arrays, for which no previous implementations existed. Finally, we design and implement a compressed text index for highly repetitive collections that solves pattern matching, which is based on the LZ77 compression, and which is the basis for our LZ77-based DR index.

Dedicated to my God and father of my lord Jesus Christ
and my lovely family.

Acknowledgements

First of all I want to thank my God for all his unconditional love in my Lord Jesus Christ, not only during my studies, but also in every time of my life. Today more than ever before, I am convinced that *Apart from him I can do nothing* (John 15:5), and that *I can do all things through him who gives me strength* (Philippians 4:13). All that I am and everything that I have achieved, including this Academic Grade, is only for his abundant grace on me. I love you God of my life.

I want also to thank my brothers in Christ of my little congregation. Thanks to all for your prayers each time that I needed during this challenge. God bless you so much!.

I could not believe when I was accepted in this great house of studies. The Department of Computer Science of the University of Chile gave me the opportunity to discover my real vocation. Here I received all what was necessary for dedicating exclusively to my studies and research. Thanks to all the members of this department.

I remember when asked Professor Navarro about his field and research group, and he gave me an extensive and passionate talk about all the research lines in which they inquired. I knew in that moment that I want to work with him in algorithms. Then, I would like Thanks to my advisor, Gonzalo Navarro, very very much for allowing me to be guided by him in these years. I know that it was not easy to adapt to my special way of “to do science”. I started with several and big weaknesses, from my poor English to my little knowledge in Algorithmics. All those things should have exhausted his patience many times. However he not only supported me, but also he personally got me a scholarship when I needed and also funded me the travel several times to present our work. I then hope that God will bless him for all his help. I thank CONICYT for its economic support during most part of my Ph.D. And also to Millennium Nucleus for Information and Coordination in Networks and Basal Center for Biotechnology and Bioengineering for funding me several travels to conferences and helping me with additional scholarships in the last part of my stay in the program.

Also, here I knew wonderful people that taught me a bit more out of the academic world. I thank Angélica Aguirre and Sandra Gáez, who have been of great help in the department. Never received one “NO” as a response when I asked them for any type of help. Of course, I want to thank all my classmates of my Ph.D program. Thanks Guys for every moment shared together and God bless you so much!.

My biggest motivation has been my lovely family. Thank to my parents for all that they gave me in my life. And also my gratitude to my grandmother and my sisters for their concern and care, and especially to my nieces and little nephew for their gladness and affection. All of you are in my heart and in my prayers every time.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Improving Current DR Solutions . . . 4
1.2.1 Design of Data Structures to Use in DR Solutions . . . 4
1.2.2 Design of Compressed Indexes to solve DR Problems . . . 4
1.3 Thesis Statement . . . 5
1.3.1 Thesis Contributions . . . 5
1.4 Thesis Structure . . . 8

2 Basic Background **9**
2.1 Text Compression . . . 9
2.2 Measures of Compressibility . . . 11
2.3 Bitvectors . . . 13
2.4 Fundamental Text Indexes . . . 13
2.5 Other Useful Data Structures . . . 15
2.5.1 Wavelet Trees . . . 15
2.5.2 Succinct Tree Representations . . . 17
2.5.3 Cartesian Tree . . . 18
2.5.4 Lowest Common Ancestor and Range Minimum Query . . . 18
2.6 Compressed Text Indexes Based on the SA . . . 19
2.6.1 The Compressed Suffix Array . . . 20
2.6.2 The FMI Family . . . 21
2.6.3 The Locally Compressed Suffix Array . . . 22
2.7 LZ-Based Compressors . . . 24
2.7.1 LZ77 Compression . . . 24
2.7.2 LZ78 Compression . . . 25
2.8 The LZ-Index . . . 26
2.8.1 The Basic Structure . . . 26

3 Document Retrieval Review **29**
3.1 Document Listing . . . 29
3.2 Top-k Retrieval . . . 32
3.3 Document Listing in Repetitive Texts . . . 34

4 Contributions in Text Indexing **38**
4.1 Structures for Compressed Suffix Arrays . . . 38

4.1.1 Elias-Fano Coding . . . 39
4.1.2 The Suffix Array of Grossi and Vitter . . . 39
4.1.3 The Suffix Array of Rao . . . 42
4.1.4 Experimental Results . . . 43
4.2 Hybrid Indexing on Repetitive Datasets . . . 52
4.2.1 Hybrid Indexing . . . 52
4.2.2 Implementation . . . 54
4.3 Experiments . . . 55
4.4 Conclusions . . . 58

5 Improved Range Minimum Queries 59
5.1 State of the Art . . . 60
5.2 A Simplified Implementation . . . 64
5.2.1 Construction . . . 65
5.3 Implementing Balanced Parentheses . . . 66
5.4 Experimental Results . . . 67
5.5 Conclusions . . . 69

6 An LZ-based Index for Document Listing 71
6.1 Structure . . . 72
6.2 Queries . . . 74
6.3 Implementation . . . 76
6.3.1 Experimental Results . . . 77
6.4 Conclusions . . . 83

7 An LZ-based Index for Top-k Retrieval 85
7.1 Structure . . . 86
7.2 Queries . . . 86
7.3 Improving the Quality . . . 88
7.4 Experimental Results . . . 89
7.4.1 Space study . . . 89
7.4.2 Space/time tradeoffs . . . 89
7.4.3 Quality . . . 93
7.5 Conclusions . . . 96

8 An LZ77 Based Index for Document Listing 98
8.1 A Document Listing Approach Based on the Hybrid-Index . . . 98
8.1.1 The Structure for Primary Matches . . . 99
8.1.2 The Structure for Secondary Matches . . . 99
8.1.3 The Document Listing Algorithm . . . 104
8.1.4 Reducing the Size of the Inverted Lists . . . 105
8.2 Including Frequencies . . . 106
8.3 Conclusions . . . 107

9 Conclusions and Further Research 108

References 111

List of Tables

2.1 Time-space complexities of most popular CSAs. The time construction for these indexes is $O(n \log \sigma)$. 21

2.2 Time-space complexity of main self-indexes of the FM-Index family. The time construction for these indexes is generally $O(n \log \sigma)$. 22

4.1 Main characteristics for the texts considered in the experiments with the indexes. We show the entropy of order 0 and 4, and also the real compressibility for these texts, using the best-known compressors: gzip (option -9), bzip2 (option -9) and PPMDi (option -1 9). 43

4.2 Sizes in MB of the uncompressed files, the files compressed with 7zip and the three indexes: the LZ77-Index of Kreft and Navarro (with default values), the FM-Indexes with SA-FMI-sampling in 32 and 256, and the hybrid indexes with maximum patterns lengths M in 50 and 100, with SA-FMI-sampling in 32 and 256 in the internal FM-Index for the filtered text. Between parentheses are the parameter values for the FMIs and the hybrid indexes. 56

6.1 Main characteristics of the text collections. 78

6.2 Space breakdown of the main components in our LZ-DLIndex structure, with values in bpc. For RevTrie and Doc columns the space is the sum of the components detailed below them (bpc values in italics). The Range columns does not include the RMQ structures to speed up the index. The percentages refer to the total size of the index. The column (/|LZ78|) indicates the ratio of the total size over |LZ78|, and the last column, in turn, gives also (n/n'). . 79

6.3 Number of occurrences of each type, for pattern lengths $m = 6$ and $m = 10$. Under each number, we give the percentages of the documents output. For the three types of occurrences these refer to ndoc, and for column ndoc this refers to D. 80

8.1 New points in the grid G for the three phrases of our example at Figure 8.2. 102

List of Figures

2.1 A *Trie* on the strings: *she$\$_1$*, *sees$\$_2$* and *cheese$\$_3$*, and also the GST and GSA for the text: *she$\$_1$sees$\$_2$cheese$\$_3$*. . . . 14
2.2 The *Wavelet Tree* for the sequence *S="she_sees_cheese$"* over the alphabet $\Sigma = \{\$, _, c, \mathrm{e}, h, s\}$. . . . 15
2.3 A wavelet tree showing the nodes that cover a leaves range. . . . 16
2.4 A *Cartesian Tree* on a input array 18
2.5 The *2d-Min-Heap* data structure on the array A and the BPS, DFUDS and LOUDS succinct representations of the tree. . . . 20
2.6 The resulting phrases after applying the LZ77 parsing on a text and the two types of occurrences in the parsed text 24
2.7 The resulting phrases when applying the LZ78 parsing of a collection with 3 texts. . . . 25
2.8 The three types of occurrences according to how they span blocks (or phrases). 26
2.9 The structures to report occurrences of type 1. . . . 26
2.10 The scheme to report the occurrences of type 2. . . . 27

3.1 The generalized suffix tree a text and the arrays E and C that form the structure of Muthukrisman to solve DL queries. . . . 30

4.1 GVCSA with $t = 2$ levels of decomposition for a suffix array. . . . 40
4.2 RaoCSA with $t = 1$ level of decomposition and for $l = 4$ for a suffix array. . . 43
4.3 Space/time tradeoffs for accessing one cell using various options for (t, l) for GVCSA. . . . 45
4.4 Various options for (t, l) for RaoCSA. On the left we show the basic scheme, on the right our improvement using wavelet trees. . . . 46
4.5 Various options for (t, l) for our improvement in RaoCSA using runs and wavelet trees. . . . 47
4.6 Time-space tradeoffs to access one cell. On the left, basic GVCSA versus the version with runs, for all the texts. On the right, the best variants of RaoCSA. 48
4.7 Construction time and space for the different indexes on each text. . . . 50
4.8 Time/space tradeoffs to access one random cell for the different indexes on each text. . . . 51
4.9 The Basic scheme to find secondary matches. . . . 54
4.10 Index sizes for prefixes of cere of 100, 200, 300 and 400 MB. . . . 56
4.11 Average query times for the different indexes to locate occurrences with patterns of different lengths. . . . 57

4.12 Index sizes and locate query time for the Hybrid-Index against the LZ77-Index. 58

5.1 An example array (top right) and its Cartesian tree (left). 61
5.2 The same arrangement of Figure 5.1, now on the DFUDS representation of the Cartesian tree. 63
5.3 The general tree (at middle) derived from the example Cartesian tree. 64
5.4 Query space and time on random arrays, for ranges of size 10,000, comparing the standard with our new implementations. 68
5.5 Query space and time on random arrays, for ranges of size 10,000. 68
5.6 Query time on random arrays, for ranges of increasing size and two values of n. 69
5.7 Query time on pseudo-sorted arrays, $n = 10^6$ and ranges of size 10,000. . . . 70

6.1 The structures to report documents for occurrences of type 1. 73
6.2 The scheme to report the occurrences of type 2 using RMQ structures in each level of the wavelet tree of Range. 76
6.3 Space/time comparison for pattern length $m = 6$. 81
6.4 Space/time comparison for pattern length $m = 10$. 82
6.5 Fraction of the real answer of our LZ-DLIndex for real queries, as a function of the prefix size of TodoCLin GB, for words and phrases of two words. 84

7.1 The main data structures of our approximate top-k index. 88
7.2 Space breakdown of our structures for different g values (g is the x-axis). . . 90
7.3 Space/time comparison for pattern length $m = 6$ (left) and $m = 10$ (right). Space (bpc) is the x-axis. 91
7.4 Space/time comparison for pattern length $m = 6$ (left) and $m = 10$ (right). Space (bpc) is the x-axis. 92
7.5 Recall of our approximate top-k solution, as a function of the fraction of the answer (x-axis). 94
7.6 Quality of our approximate top-k solution, as a function of the pattern length, for top-10 (left) and top-100 (right). 95
7.7 Fraction of the real answer found by LZ-AppTopK for real queries, as a function of the prefix size of TodoCL for words (left) and phrases of two words (right). 95
7.8 Fraction of the real answer found by LZ-TopkApp as a function of the prefix size of TREC, for arbitrary patterns of lengths 6 and 10, in top-10 and top-100. 96

8.1 The basic scheme with non-overlapped phrase sources. 100
8.2 An example with non-overlapped phrase sources distributed in two documents. 101
8.3 An example with several overlapped phrase sources in a document d_U. 103

Chapter 1

Introduction

This chapter describes the scope of this thesis, the open problems in the field and the concrete contributions of our research. First of all we give the context in which we describe the challenges that have motivated this study. The chapter ends summarizing each one of the results obtained by this research.

1.1 Motivation

This thesis belongs to the *Document Retrieval* (DR) field, which deals with the study of procedures to represent text collections and offers search functionality in order to efficiently locate the documents that better satisfy an information need at query time.

Several of the most important challenges in Computer Science are related to finding efficient algorithms to solve *Search Problems*. Independently of the scope of these algorithms, many of them have as their main goal to retrieve specific information from a digital data base. In research fields such as *Text Searching* and *Document Retrieval*, these kind of algorithms operate on sequences of symbols or strings. Search problems can be as simple as string matching (i.e., find occurrences of a string) or as complex as ranked document retrieval (i.e., find documents most relevant to a query). Besides, the search tasks become more complex because we usually have to handle large data sets. We are then faced with additional problems related to the storage and representation of massive amounts of data, which implies that not only we need efficient search algorithms on them, but also to build compressed representations to maintain memory usage under control.

A classic example is a *Web Search Engine*, where a user needs to know where a specific information is located on the "*World Wide Web*". In the process, the user gives a brief description (typically a short string) about what he/she needs and waits for the output. Behind the user interface, an *Information Retrieval System* is working to search for the required information in the data representation.

As we have illustrated with Web searches, there are many other classical environments

where DR takes great relevance. In software repositories, a frequent task for developers is to find where an object is mentioned in their source code files; for instance with user function calls. Hence, pieces of source code are treated as documents and a DR framework is built as a part of the development environment. In music collections, we also find various tasks related to *Music Information Retrieval* (MIR). In MIDI sequence analysis[1], one of the most relevant concerns is to locate the occurrences of *a theme* in a *piece of music*. The theme can be a melody or a sequence of notes called a *musical pattern*. The music is simply a song file in MIDI format, a set of symbolically encoded notes that form the musical sequences representing each document, which is obtained from a digital-to-symbolic conversion of audio data. Bioinformatics is another research area where DR solutions are often sought [8]. Advances in DNA sequencing have produced databases of thousands of human genomes, which implies additional problems related to data storage and how to retrieve pieces of sequences from it. The challenge is again to build small representations for these big biological sequences and to offer methods to carry out efficient searches on them. These sequences must be well-compressed in data structures that allow us to filter biological documents without the need to decompress the whole representation. In DNA sequencing, a popular problem is to list all the *genes* where a *DNA marker* appears, where the sequence is composed only of base pairs from the set {A, C, T, G}. Another frequent task, related with protein sequences, is to find all the proteins where an *amino acid* sequence appears frequently.

The Inverted Index [5] is the most widely used data structure to solve DR problems when the texts can be split into words. It is very similar to a book index, where for a set of pre-determined words, we store for each word a list of all the documents that contain it. In order to answer a DR query, where queries are sets of words, the inverted index finds the lists of documents where each query word appears. After that, it must solve operations for sets such as union, intersection or differences between the retrieved lists. The type of operation depends on the problem to solve, and other variables are included to build the final answer, such as scores for each document or weighting documents according to query word frequencies. However, this approach is not easily applicable to human languages such as Chinese, Korean, Thai, and other Asian languages, because these texts have no delimiters to mark word boundaries. The same problem happens with agglutinating languages as Hungarian, Turkish or Finnish, where sentences are concatenated into words. Another example is the biological sequence analysis on DNA sequences, where as we mentioned the alphabet is a set composed of only four characters without any delimiter. There are also many applications where inverted indexes cannot be applied because the concept of word does not exist: source and binary codes in software repositories, MIDI files, or any other multimedia database. Consequently, the indexes of general string collections must be more general than inverted indexes.

In this given context, an elementary and closely related problem (widely studied in text indexing [88]) is *Pattern Matching*. It aims to locate all the positions where a given arbitrary string, called the *search pattern*, occurs in a text given beforehand. The *Suffix Tree* (ST) [116] is the most popular data structure used to solve this problem in optimal time and linear space. For a given text $T_{1..n}$ and a search pattern $p_{1..m}$ that matches occ times in T,

[1]MIDI is an acronym for the Musical Instrument Digital Interface, and has taken on multiple meanings as the data in a Standard MIDI File (SMF). That standard describes the format designed to work with MIDI hardware devices [110].

the ST requires $O(n \log n)$ bits[2] and solves a query for any $p_{1..m}$ in $O(m+\mathsf{occ})$ time. Instead, DR problems aim to find which texts (or documents) in the collection satisfy a well-defined relation with the query pattern, and they are much less developed than Pattern Matching.

The most elementary problem in DR is *Document Listing* (DL), which consists in finding all documents that contain at least one match of p. Although solving DL using only the suffix tree of the whole text collection (called *Generalized Suffix Tree* (GST)) is possible, it can also be inefficient. This is because the number of occurrences of P in the collection can be much higher than the number of documents where P appears.

Muthukrishnan [84] added some data structures to the GST that maintain the size complexity in $O(n \log n)$ bits and solve DL in optimal time. After this work several *succinct/compressed indexes*[3] for DL have appeared trying to reduce the space requirements of Muthukrishnan's structure, both in theory and in practice. However, these smaller structures do not maintain the optimal query time.

The highest-level task in DR is the so-called *Top-k Retrieval*. The objective is to determine the k most relevant documents for the queried pattern. This relevance is a predefined criterion that may depend on the document itself (e.g., *PageRank* in Web searches) and/or on the occurrences of the pattern in the document (the most used one is the term frequency). Something similar to DL happened with the progress of top-k solutions. Hon et al. [58] gave a useful framework to solve top-k in linear space (i.e., $O(n \log n)$ bits), which is based on the GST, achieving a query time near to the optimal. Navarro and Nekrich [89] reduced the time to optimal. Their proposal also uses linear space, but the constants involved in the size complexities are very large. Subsequent works tried to reduce the size of their proposal, but their time is not optimal.

We focus on solutions for DR on general string collections, aiming at obtaining low space and query time. Below we list the formal definitions for the two fundamental DR problems that we address in this thesis.

Fundamental Document Retrieval Problems

Suppose that we are given a collection ∇ of documents $d_1, d_2, ..., d_D$ of total length $\sum_{j=1}^{D} |d_j| = n$, which must be preprocessed to build an index for ∇, such that later we can efficiently support on-line queries for any pattern string $p_{1..m}$[4]. On these assumptions, the following are two of the most fundamental problems in DR:

Document Listing *(DL).* List all the documents of ∇ that contain p as a substring. If in addition we give the number of occurrences of p in each document reported, the problem is called *DL with Frequencies.*

[2] All the complexities that we describe in this thesis are based in the word RAM model, which can perform any arithmetic operation on $\omega = \Omega(\log n)$-bit integers in constant time.

[3] We say that an index is succinct when it provides fast search functionalities in a space proportional to that of the text itself (say, two times the text size). If the size is proportional to that of the compressed text then it is called a compressed index [88] (see Section 2.2).

[4] In this document we use indistinctly $t_{x..y}$ or $t[x..y]$ to indicate the substring of t from x to y.

Top-k Document Retrieval *(Top-k).* The goal is to list the k most *relevant* documents of ∇ for a given string query p, in decreasing order of relevance. We define that the document d_i is more relevant than document d_j for a pattern p if, and only if, $w(d_i, p) > w(d_j, P)$, where $w(d, p)$ is a function that assigns a numeric weight to the document d given the string p. In this thesis we use the *term frequency score* $tf(P, d)$ as function $w(d, p)$, which is the number of times that p appears in d.

1.2 Improving Current DR Solutions

1.2.1 Design of Data Structures to Use in DR Solutions

A practical way to obtain improvements in the field is to develop intermediate tools that can be used as a part of the data structures that solve DR problems. There are a several of those useful structures commonly used by DR solutions. Among them, a *Full-Text Index* is usually necessary to carry out searches for patterns in the collection of files. For instance, the popular compressed structure of Sadakane, which solves DL [106], uses a *Compressed Suffix Array* (see Section 2.6.1) to obtain the interval of all the suffixes that contain the pattern as a substring. Therefore, an improved full-text index has a positive impact on most DR solutions. This thesis offers (Chapter 4) various implementations of compressed full text indexes that had not been implemented before, although they do not outperform the best current variants.

Another useful structure in DR is the *Document Array* (DA), introduced by Muthukrishnan [84], which requires of $n\lceil\log D\rceil$ bits in plain form[5]. Today there are several proposals to represent it succinctly, or alternative structures that can replace its functionality on particular scenarios [114, 46, 45, 91]. We tried various approaches to compress the DA, but had no good results.

Yet another useful structure used in DR indexes, generally associated with the document array, is one that can locate the position of the smallest element inside an array interval. Such data structure is called *Range Minimum Query* (RMQ). Fisher and Heun [40] gave a method to build, from an input array of length n, an RMQ structure that requires only $2n + o(n)$ bits, and can answer queries in constant time. Any improvement in the implementation of RMQ structures impacts on most DL and Top-k solutions. This thesis offers an RMQ implementation using $2.2n$ bits that answers queries in a few microseconds (Chapter 5), outperforming all the previous alternatives.

1.2.2 Design of Compressed Indexes to solve DR Problems

Another approach is to directly design improved indexes to solve DR problems. Most current solutions [87] build on (possibly compressed) suffix arrays or trees (the only exception is

[5]In this thesis, we denoted $\log x$ (or just $\lg x$) to the logarithm in base two of x, unless that we specify another base.

the grammar-based DL index of Claude and Munro [23]). The main reason is that these contributions are based in the pioneer DL optimal-time structure of Muthukrishnan [84], which is built on the suffix tree. Therefore, subsequent proposals try to reduce the size of that structure, as well as augment it in order to solve Top-k retrieval. In this thesis we focus on brand new approaches, based on full text indexes that build on LZ77 [117] or LZ78 [118] compression. In Chapter 6 we adapt such a full-text index for LZ78 [85] to perform DL, and in Chapter 7 we extend it to Top-k. The results are very competitive DR indexes that, in addition, are much more efficient to deliver partial or approximate answers, which are usually tolerated in DR scenarios. This offers a new research path that had not been explored much. On highly repetitive collections, LZ77 is much stronger than LZ78. In Chapter 8 we introduce another approach to build a DL index based on the LZ77 parser, and also show how to retrieve the frequencies in the output documents. However, this work has not been implemented yet; we plan to do it after this thesis work.

1.3 Thesis Statement

This thesis is focused on developing theoretical and/or practical contributions to solve fundamental problems in Document Retrieval more efficiently, in terms of time and/or space.

We design and implement new algorithms to build compressed data structures that: (i) can be used as part of DR solutions, or (ii) index general collections of symbol sequences to support DR queries. We obtain various solutions with better space and/or time performance than the state of the art.

1.3.1 Thesis Contributions

The contributions of this thesis have been divided into separate chapters, which are related with different topics:

Chapter 4. *Contributions in Text Indexing.* This chapter describes various implementations of text indexes. These indexes perform pattern matching, which is frequently a preliminary process in document retrieval queries.

Section 4.1. *Structures for Compressed Suffix Arrays.* It describes two implementations for different theoretical proposals of compressed suffix arrays [54, 55, 100], for which no previous implementations are known despite their good theoretical guarantees. We show experimentally that these proposals, in practice, do not perform better than the current and popular implementations in the field, both in query time and in required space. Although it is a negative result, clearly establishing this fact is valuable for researchers and practitioners.

This result has been included as a part of an article appeared in the ACM Journal of Experimental Algorithmics (2014) [51].

Section 4.2. *Hybrid Indexing on Repetitive Datasets.* It introduces a simple technique for reducing the size of conventional indexes on highly repetitive texts. Given upper bounds on pattern lengths and edit distances, we preprocess the text with LZ77 to obtain a filtered text, for which we store a conventional index. Later, given a query, we find all matches in the filtered text, then use their positions and the structure of the LZ77 parse to find all matches in the original text. Our experiments show this significantly reduces space and query times.

This result was published in the Philosophical Transactions of the Royal Society A (2014) [29].

Chapter 5. *Improved Range Minimum Queries.* Fischer and Heun [40] proposed the first Range Minimum Query (RMQ) data structure on an array $A[1, n]$ that uses $2n + o(n)$ bits and answers queries in $O(1)$ time without accessing A. Their scheme converts the Cartesian Tree of A into a general tree, which is represented using DFUDS [16]. We show that, (i) by using BP representation [83] instead of DFUDS, the formula becomes simpler since border conditions are eliminated; (ii) for the BP representation, the *Range Min-Max Tree* [107] with only minimum values is sufficient to implement the formula and it significantly reduces the space requirements. This leads to the fastest and most compact practical implementation to date, which uses around $2.2n$ bits and takes 1–4 microseconds per query.

This result was published in the Proceedings of the 26th Data Compression Conference (DCC 2016) [32]. It won the Capocelli Prize (best student paper award).

Chapter 6. *An LZ-based Index for Document Listing.* It describes the first attempt to solve the DL problem using an LZ78 compressed index of the text collections. We show that the resulting solution is very fast to output most of the output documents, taking more time for the final ones. This makes this index particularly useful for interactive scenarios or when listing some documents is sufficient. Yet, it also offers a competitive space/time tradeoff when returning the full answers.

This result was published in the Proceedings of the 20th International Symposium on String Processing and Information Retrieval (SPIRE 2013) [30].

Chapter 7. *An LZ-based Index for Top-k Retrieval.* It introduces a top-k retrieval index for general string collections, which is based on the index described in the previous chapter. Our implementations achieve competitive space/time tradeoffs compared to existing solutions, dominating a significant part of the space/time tradeoff map. The approximate variant of our index (LZ-TopkApp) is orders of magnitude faster, and uses much less space, than previous work. Typically it uses 4–7 bits per symbol and returns each result in about 1–5 microseconds. We show that the quality of its answers improves asymptotically with the size of the collection, reaching over 90% of the accumulated term frequency of the real answer already for patters of length ≤ 8 on rather small collection, and improving for larger ones.

This result was published in the Proceedings of the 21st International Symposium on String Processing and Information Retrieval (SPIRE 2014) [31].

Chapter 8. *LZ77 Based Index for Document Retrieval.* This is a preliminary work based on the LZ77 parsing, where we detail how we can adapt the structure described in Section 4.2 for the Hybrid-Index, which solves the pattern matching problem, to solve document listing queries and also to retrieve term frequencies. The implementation and refinement of these structures is left as future work, which we expect to continue after this PhD thesis.

Additionally to the detailed contributions, this thesis work has been presented at the *Encuentro de tesistas - Jornadas Chilenas de Computación (ET-JCC 2014)*. We also expect other two publications. The first one is the extended version of the main work described in Chapters 6 and 7, which has been submitted to the journal *Information and Computation*. The other publication is the extended version of our paper *Improved Range Minimum Queries* [32] described in Chapter 5, which has been invited to a special issue in the *Journal of Discrete Algorithms*.

Practical Contributions

As practical contributions of this thesis, there are several implementations available in public repositories:

- The Compressed Suffix Array implementations of Grossi and Vitter (GVCSA) and of Rao (RaoSA), from Section 4.1.
  `http://pizzachili.dcc.uchile.cl/additionalSuffixArrays.html`
- The implementation of the Hybrid-Index to solve pattern matching, as a result of Section 4.2.
  `https://www.cs.helsinki.fi/u/gagie/hybrid/`
- The libraries to build an RMQ compressed structure, from Chapter 5.
  `https://github.com/hferrada/rmq.git`
  `https://github.com/hferrada/rmqFischerDFUDS.git`
- A compressed LZ-based index to solve DL, from Chapter 6.
  `https://github.com/hferrada/LZ-DLIndex.git`
- The implementations of Sadakane's index for DL, from Chapter 6.
  `https://github.com/hferrada/Sada-DLIndex.git`
- A compressed LZ-based approximation for top-k queries, from Chapter 7.
  `https://github.com/hferrada/LZ-AppTopK.git`
- A compressed LZ-based index to solve top-k (full answers), from Chapter 7.
  `https://github.com/hferrada/LZ-Topk.git`

1.4 Thesis Structure

Chapter 2 gives the basic concepts that are necessary to understand the rest of this document. Chapter 3 details the most important solutions for the DR problems addressed in this thesis.

The content specifically related with the detailed contributions is organized in parts:

First part. It comprises Chapters 4 and 5, which focus on intermediate tools that can be used as a part of compressed structures to solve DR problems. Chapter 4 presents the study of full text indexes and Chapter 5 details how to improve the current way to compute Range Minimum Queries.

Second part. It comprises Chapters 6 and 7. These detail the LZ-based indexes to solve document listing and top-k retrieval problems, respectively.

Third part. It comprises Chapter 8. It describes an LZ77-based index for DL useful when the documents are highly repetitive.

Chapter 9 summarizes the conclusions obtained from this research, it discusses the impacts and describes possible future work directions.

Chapter 2

Basic Background

We have adopted the standard word random access model (RAM) as the computation model, where if n is the maximum size of the problem, the basic operations between words of maximum size $O(\log n)$, or reading/writing $O(\log n)$ bits of memory, can be done in constant time.

Throughout this thesis we consider the text $T = d_1\$_1 d_2\$_2..d_D\$_D$ as a concatenation of D text documents. Each document d_i is a finite sequence built over an alphabet Σ of size σ (Σ is a totally ordered set of symbols), such that $\$_1 <\$_2 < .. <\$_D$ and $\$_D < c$ for any character c of Σ. Then, for any pair of strings a, b of Σ^* and any $i, j \leq D$, it is possible to determine which word $a\$_i$ or $b\$_j$ is the lexicographically lowest. This yields a total order $\preceq$ between suffixes of documents.

2.1 Text Compression

In text processing, we usually have to handle several texts, facing the problem of how to store them. The simplest way is to store the whole collection explicitly without resizing the data. This is ideal only when the total amount of data is small and we have enough resources to save them. However, in many practical cases the explicit storage is prohibitive given the space restrictions –at least in main memory. The solution then is to look for techniques that can be applied on the texts to reduce their size, which is called *Text Compression* (TC) [14]. The objective of TC is to obtain an equivalent structure for the input text, called *a representation*, which is smaller than the input text, does not lose information, and can be used instead of the original data.

Salomon [108] defines *Data Compression* (DC) as the process of converting an input data stream (the source stream or the original raw data) into another data stream (the output, the bitstream, or the compressed stream) that has a smaller size. Accordingly, the process of DC is promoted by two important motivations. The first one is related to the limitations of the memory architecture. No matter how big the available space is to handle data, there will always be a time when this resource cannot offer more storage. DC aims at delaying

that moment. The second one is in relation to the transfer rate by communication channels. The larger the amount of data that will be transferred, the longer the time of the transfer. DC also aims to reduce this time.

A more complex scenario

The above considers only the problem of storing large volumes of texts, without regarding the access time to the compressed data nor how to quickly find within it any information when it is needed. Taking into account that the time needed to access secondary storage is orders of magnitude slower than manipulating data in main memory, it is convenient to build smaller representations that avoid using secondary memory. Furthermore, searching and locating text are one of the activities most frequently associated with text collections. These considerations have motivated the recent trend of designing data structures for indexing the text and accelerating the search tasks. These are called *Text Indexes*, which require generally more space than the data indexed. For that reason, the most useful indexes are those that require (at most) space proportional to what is needed to store the text collection. Even more desirable is that the indexes implement TC and do not need to decompress the whole index in order to carry out text searches.

Through this thesis we will introduce different levels of compressibility in structures that index the data. We refer to a *Full Text Index* as an index enabled to carry out text searches much faster than sequentially searching the collection. In the field there is a standard nomenclature used to classify indexes, which considers the structure size and its dependency with the indexed text. The next paragraph, extracted from the survey of G. Navarro [88], defines a hierarchy of three compression levels[1]:

"*A succinct index is an index that provides fast search functionality using a space proportional to that of the text itself (say, two times the text size). A stronger concept is that of a compressed index, which takes advantage of the regularities of the text to operate in space proportional to that of the compressed text. An even more powerful concept is that of a self-index, which is a compressed index that, in addition to providing search functionality, contains enough information to efficiently reproduce any text substring. A self-index can therefore replace the text.*"

Whatever the level of compressibility of a structure, we refer to it as a *Compact Data Structure* if its space requirements are (at least) in proportion to the data representation while it supports useful operations without the need to decompress the whole structure. In particular, our objective is to search directly in the compressed DR index (in our compact data structure) instead of decompressing it.

[1]In other cases, succinct denotes an index of size $|data| + o(|data|)$, whereas an index of size $O(|data|)$ is called compact.

2.2 Measures of Compressibility

A key point of our interest when designing structures for DR is in relation to the representation of text documents. When building a full text index, it is necessary to represent the entire input collection in a data structure that offers search functionalities over it. In particular, we aim to build a representation as small as the compressed data itself, which supports fast DR capabilities on the documents.

To compress the space we must take advantage of the regularities of the data. Under this context, we need tools to measure how good is the compression we achieved. The *Information Theory* [111, 14, 4] offers an accurate way to quantify information with metrics that answer the question: "how much information is included in a piece of data?". Using it, we can estimate how good a representation like an index is, with respect to the information contained in the original data.

The *Kolmogorov-Chaitin complexity* (KCC) [66] defines the complexity of an object, like a binary string, as the number in bits of the shortest program that generates it. In simple words it is the length of the shortest program that can list, print, or write in a text file the original sequence. This measure tells us which is the best result in terms of space resources that we could get from building a representation of the object. Therefore, if the KCC of a bitstring S is as long as the sequence itself, for instance in a random bitstring, we can say that S is incompressible. On the other hand, when the KCC is shorter than the sequence, for instance in $S' = 110110110...$, such that a loop in a program can write 110 until obtaining S', we say that S' can be compressed.

Chaitin [20] studied the shape of binary strings, observing that the common bitstrings used in practice, which represent text, images and sound are placed, on average, between S and S' (the kind of bitstrings indicated in the previous paragraph). He showed that most of these sequences are neither incompressible (not random), nor repetitive sequences like S'. The conclusion is that it is possible to obtain different levels of compression for general bit-sequences. Considering that we need represent text documents, one of the challenges in this research is to find how to get compression in full text indexes for DR.

Although the concept of KCC is clear, it is not computable, so we need a "bridge" between theory and practise. For that we consider the *Entropy* of Shannon [111], which in information theory is closely related to KCC but not so general. The entropy of an object x, $\mathcal{H}(x)$, is a quantity number that measures the average uncertainty of x. It is the smallest number of bits required, on average, to identify an object from a set (the length of a *code* for x). It then gives the average lower bound, in bits, to encode each object.

The codelength that we give to each symbol is crucial to obtain a smaller representation of the data. The *Worst-Case entropy*, denoted as $\mathcal{H}_{WC}$, is used when there is no other option than to assign codes with identical length to all the symbols. $\mathcal{H}_{WC}$ is the shortest possible codelength to univocally identify each element from a set source $\mathcal{U}$:

$$\mathcal{H}_{WC}(\mathcal{U}) = \lg |\mathcal{U}|.$$

A better situation is when we can handle codes with variable lengths. Suppose that there is a small percentage of symbols in $\mathcal{U}$ that are very frequent in a sequence S. If we assign shorter codes to these highly frequent elements, though the less frequent ones are longer, we could save space by rewriting S with these encodings.

Shannon gave a formula to compute the entropy when we have the occurrence probabilities of the symbols, that is, a symbol u appears in the sequence with probability $Pr(u)$. He defined the *entropy for a probability distribution* Pr, where $Pr : \mathcal{U} \rightarrow [0.0, 1.0]$ as:

$$\mathcal{H}(Pr) = \sum_{u \in \mathcal{U}} Pr(u) \cdot \lg \frac{1}{Pr(u)}$$

Empirical Entropy

The *Empirical entropy* is a compressibility measure for symbol sequences that lower bounds the performance of certain types of compressors without assuming the sequence comes from a particular distribution. This is useful because we usually do not know the probability of the occurrence of symbols. The value H_k corresponds to the *k-th Order Empirical Entropy* defined for finite texts [79]. It provides a lower bound to the number of bits needed to compress T using any compressor that encodes each character considering only the context of k characters that follow it in T.

The *zero-order empirical entropy* of $T_{1..n}$, where its symbols come from the set Σ, is defined as:

$$H_0(T) = \sum_{\omega \in \Sigma, n_\omega > 0} \frac{n_\omega}{n} \log \frac{n}{n_\omega},$$

where n_ω is the number of occurrences of character ω in $T_{1..n}$.

The *k-order empirical entropy* of $T_{1..n}$ is defined as:

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

where ε denotes the empty string and T^s is the subsequence of T formed by all the characters that occur followed by the context s in T. In order to have a context for the last k characters of T, we pad T with k characters "\$" (in addition to $T_n = \$$). More precisely, if the occurrences of s in $T_{2,n}\$^k$ start at positions $p_1, p_2, ...$, then $T^s = T_{p1-1} T_{p2-1} ....$

For example, let $s =$*carretera*. We have $H_0 = 2\frac{1}{9} \log 9 + 2\frac{2}{9} \log \frac{9}{2} + \frac{1}{3} \log 3 \simeq 2.197$. For $H_1(s)$, we have $c^s = a$, $a^s = r$, $r^s = rea$, $\mathrm{e}^s = tr$ and $t^s = \mathrm{e}$; with $H_0(a) = H_0(r) = H_0(\mathrm{e}) = 0$, $H_0(rea) = 1.585$ and $H_0(tr) = 1$. Then $H_1(s) = \frac{1}{9}(H_0(a) + H_0(r) + H_0(\mathrm{e}) + 3H_0(rea) + 2H_0(tr)) \simeq 0.751$.

2.3 Bitvectors

A *bitvector* is one of the most fundamental data structures in text indexing. It is a representation of a bit sequence equipped with additional functionality, for the purpose of quickly answering queries about the bits stored in relation to any sequence's prefix. It replaces the original sequence S, by allowing the retrieval of the bit value at any position in S. Formally, a bitvector $B_{1..n}$ is an explicit or compressed representation of a sequence of n bits, which supports the following operations in constant time (or very close to it) on the RAM model:

- $access(B, i)$ returns the bit at position i, for any $1 \leq \mathrm{i} \leq n$.
- $rank_b(B, i)$ returns the numbers of bits $b \in \{0, 1\}$ up to position i, for any $1 \leq \mathrm{i} \leq n$.
- $select_b(B, i)$ returns the position of the ith bit $b \in \{0, 1\}$, for any $1 \leq \mathrm{i} \leq rank_b(B, n)$.

In the current state of the art we find uncompressed and compressed solutions to build bitvectors. The best uncompressed proposals use $o(n)$ extra bits to compute the three previous operations in constant time (see for instance [60, 82, 22]). For compressed bitvectors, the best solutions [99, 96, 6] typically require $nH_0(B) + o(n)$. The key to saving size is to consider blocks instead of individual bits to build the representation. Additionally, small tables are used during rank and select operations to compute the answer for single blocks in constant time.

2.4 Fundamental Text Indexes

A basic and simple data structure that can be used as a general index for a text builds on the *Digital Tree* or *Trie* [42, 65]. This structure organizes and stores a set of strings in a tree, so that the search of a string $p_{1..m}$ can be performed efficiently in time proportional to the length m. Each node represents a different prefix of the set, where the root node represents the empty string ε, and each edge stores a symbol c used to guide the search. The construction of this tree can be done in time proportional to the total length of the strings to be stored. The search is performed from the root node down through the tree, consuming the pattern $p_{1..m}$ symbol by symbol, and selecting the correct edge labeled with the symbol read. Additional data structures are necessary to retrieve the correct child during the search in constant time, for example a perfect hash table. If we only store the children ordered in an array, we then must implement a binary search in each internal node, increasing the search time to $O(m \log \sigma)$, where σ is the alphabet size.

The *Suffix Tree* (ST) was proposed by Weiner [116]. It is a *digital trie* of all the suffixes of a text, where unary paths (sequences of intermediate nodes with a unique child) are replaced by a single edge labeled with the concatenation of all the symbols of the replaced edges. The children of an intermediate node are placed from left to right in lexicographical order. The leaves represent each of the n suffixes of the text $T_{1..n}$, and they store the position of their corresponding suffix.

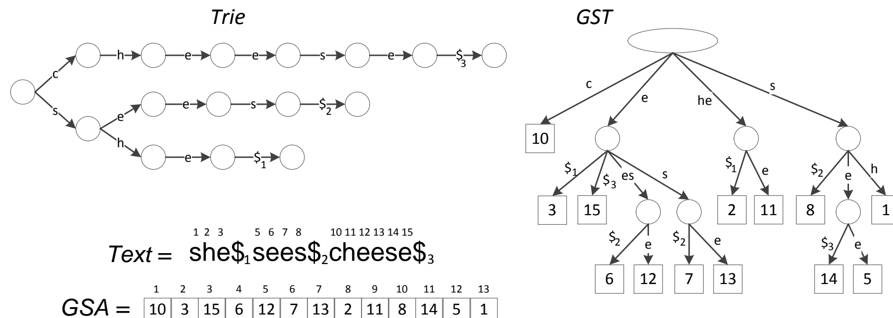

Figure 2.1: A *Trie* on the strings: *she*$\$_1$, *sees*$\$_2$ and *cheese*$\$_3$, and also the GST and GSA for the text: *she*$\$_1$*sees*$\$_2$*cheese*$\$_3$. For simplicity, in GST and GSA the corresponding suffixes of symbols $\$_\text{i}$ have been omitted.

A *Generalized Suffix Tree* (GST) is a suffix tree built on a concatenation of all text documents of the collection, $T_{1..n} = \text{d}_1\$_1\text{d}_2\$_2...\text{d}_D\$_D$. As every occurrence of p in T is a prefix of some suffix, pattern matching is performed similarly to the trie, but taking into account that now the edges could have more than one symbol. If at some point the path given by p cannot be followed, then p does not occur in T. Otherwise, we will consume all the symbols of p, arriving at a node v. Then, all the leaves of the subtree rooted at v are the *occ* occurrences sought. With this structure we solve the pattern matching problem in optimal time, $O(m + occ)$, but it is necessary to access the text when we reach a leaf x and the number of symbols from the root to x is lower than m. Nevertheless, this structure requires $O(n \log n)$ bits of space to be stored, which in practice is 10-20 times the text size.

The *Suffix Array* (SA) [77] is a structure that reduces the size of the suffix tree, but supports fewer functionalities. The SA consists of an array of integers that represents a permutation of the n suffixes of $T_{1..n}$, which are lexicographically ordered from left to right in the array. Due to this ordering, all the suffixes that start with the same prefix of length m are in consecutive positions on the array. The SA can be used to delimit the segment of all the suffixes starting with $p_{1..m}$ by two binary searches (for these searches, as in the ST, it is necessary to access the text). Therefore, the time complexity with this structure is $O(m \log n)$, and the data structure requires exactly $n \lceil \log n \rceil$ bits of space. That time can be reduced to $O(m + \log n)$ by adding some structures. Like the GST, when the text is a concatenation of several text documents, the SA is called *Generalized Suffix Array* (GSA). The suffix array can be created from the suffix tree, placing all of the tree leaves in the same order from left to right. However, it is preferable to build it directly by ordering the suffixes with any efficient algorithm, some of which build the SA for $T_{1..n}$ in linear time [98]. Figure 2.1 shows an example of these structures.

These fundamental text indexes require much space, that is, $\Theta(n \log n)$, which is more than a few times the collection itself, hence these are not compact. Section 2.6 covers compressed text indexes, most of which are based on the SA and frequently used in DR.

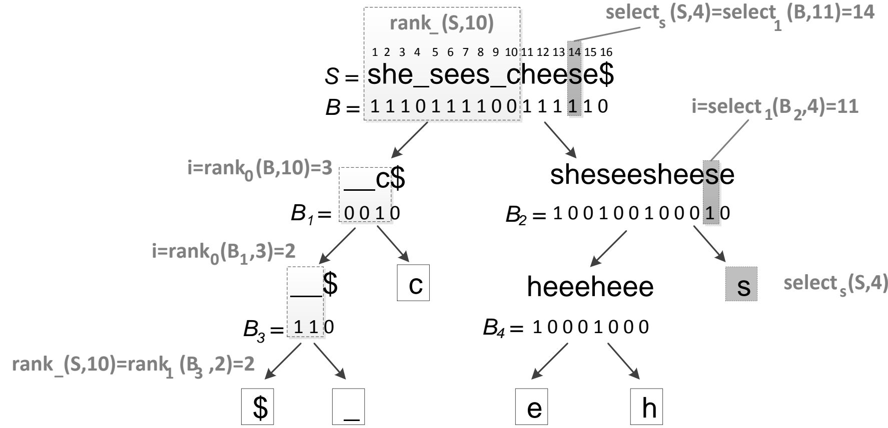

Figure 2.2: The *Wavelet Tree* for the sequence *S="she_sees_cheese$"* over the alphabet $\Sigma = \{\$, _, c, \mathrm{e}, h, s\}$. For example in the root node, the bits of B are set to 0 for the symbols set $\{\$, _, c\}$, whose characters belong to the lower half of the alphabet. For computing $rank_(S, 10) = 2$, where $i = 10$, the figure shows the re-computation of the i values in each step as we go down, using by rank operations relative to the previous bitvector B. For $select_s(S, 4) = 14$, where $i = 4$, we illustrate how to recalculate the new values of i, by select operations, while climbing the tree from the leaf of the symbol s. The internal rank/select operations are always computed on the bitvector B of the node recently visited in the path.

2.5 Other Useful Data Structures

2.5.1 Wavelet Trees

According to Grossi et al. [52], the Wavelet Tree (WT) for a symbol sequence $S_{1..n}$ over an alphabet Σ of size σ is a binary tree, which is able to compute $S[i]$, $rank_c(S, i)$ and $select_c(S, i)$[2], $\forall\ c \in \Sigma$. The classical version of WT consists of a balanced binary tree that computes access, rank and select queries in $O(\log \sigma)$ time. It requires $(1 + o(1))n \log \sigma$ bits, thus it is a succinct text index, and can be constructed in $O(n \log \sigma)$ time. Each internal node represents a subsequence of characters of S. For this purpose the node stores an uncompressed bitvector that supports rank/select. The binary string at the root node contains n bits for representing the complete sequence $S_{1..n}$, and handles all symbols of Σ. Any internal node v, located at depth i, for $0 \le i \le \lceil \log \sigma \rceil$, handles a subset of $\sigma' = \lceil \sigma / 2^{\mathrm{i}} \rceil$ symbols. Then each bit of its bit-sequence is set to 0 when the corresponding character is among the $\lceil \sigma'/2 \rceil$ lower symbols with respect to the node i, otherwise the bit is set to 1. Consequently, its left and right children represent the symbols of the lower half (bits in 0) and upper half (bits in 1) of v, respectively. According to this model, the tree contains n bits in each of the $O(\log \sigma)$ levels, which are used to guide the search. Also, the WT has σ leaves located in order from left to right; these correspond to the distinct symbols of Σ (see an example in Figure 2.2).

For computing access, that is, $S[i]$, we move down in the tree until we reach the correct

[2] In Section 2.3 we defined *rank* and *select* on bit sequences (i.e, $|\Sigma| = 2$), here we refer to these same operations, but now restricted to symbol sequences with alphabets of any size.

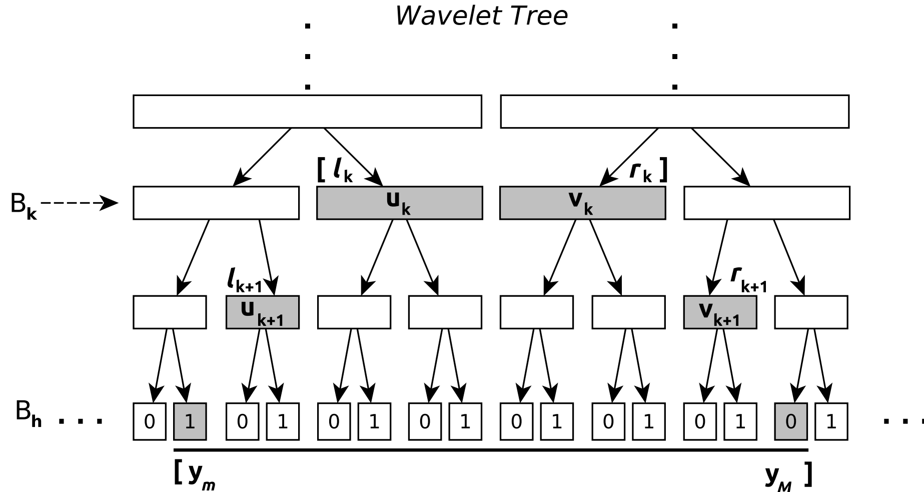

Figure 2.3: A wavelet tree where we shadowed the $O(\log n)$ nodes that cover the leaves range $[y_m, y_M]$. The range is covered with at most two maximal nodes per level. It is sufficient to map an original range $[x_m, x_M]$ from the root to those $O(\log n)$ nodes to find all the points in $[x_m, x_M] \times [y_m, y_M]$. Then those points can be reported one by one, or their total amount can be counted in time $O(\log n)$.

leaf and report its symbol. We go down to the left if the ith bit is 0 and to the right if it is 1, and recompute in each step the value of the index i, depending on whether we go down to the left or right child. We recalculate $i = rank_0(B, i)$ on the left (or $rank_1$ on the right), where B is the bitvector of the current internal node at this step. To compute $rank_c(S, i)$, we update in each step the value of the relative index i as in operation access. Here, we move down to the left if c is in the first half of the subset of symbols handled by the current node, otherwise we go down to the right. When we reach the leaf c, the current value of i is the answer. For computing $select_c(S, i)$, we start from the leaf corresponding to the symbol c, at position i and go up to the root. Each time we go up, we update the index with $i = select_1(B, i)$ if we go up from the right child, or $select_0$ from the left child. Figure 2.2 shows an example of rank/select operations.

Wavelet trees can also represent an $n \times n$ grid with n points, one per column: $(1, y_1), (2, y_2), \ldots, (n, y_n)$, by regarding the points as a sequence $S_{1..n} = y_1 y_2 \ldots y_n$ on the alphabet $[1..n]$. The wavelet tree takes $n \lg n + o(n \log n)$ bits to represent the points, and can retrieve the t points in any rectangle $[x_m, x_M] \times [y_m, y_M]$ in time $O((t+1) \log n)$, as follows. We start at the root bitvector B_v with the range $[x_m, x_M]$. Then we go to the left child with the new range $[rank_0(B_v, x_m - 1) + 1, rank_0(B_v, x_M)]$, *and* to the right child with the range $[rank_1(B_v, x_m - 1) + 1, rank_1(B_v, x_M)]$. We stop the recursion at any node v where either the range is empty or there is no intersection between the sub-alphabet of $[1..n]$ handled by v and the range $[y_m, y_M]$. When we reach a leaf, we report its corresponding y value (we can report the x value as well, by going upwards as for *select*). Since the range $[y_m, y_M]$ is covered by $O(\log n)$ wavelet tree nodes, it is possible to count the number t of points in a rectangle in $O(\log n)$ time, by adding up $x_M - x_m + 1$ on those nodes that cover $[y_m, y_M]$, instead of tracking all their points up to the leaves. Figure 2.3 exemplifies the process.

2.5.2 Succinct Tree Representations

Ordinal Trees and *Cardinal Trees* are the major families of data structures for trees. The first are rooted trees and their nodes can have any degree. They are also called *ordered trees*, because the order of the children of any node is significant. In the second family of trees, each node reserves k fixed slots for its children, and in every node its slots can be occupied or free. Though the cardinal trees are very important (a binary tree is an example with $k = 2$), we focus only on the succinct representation of ordinal trees, because these data structures are the most used ones in the field of document retrieval.

Observe that we need $\Theta(\log n)$ bits to store a pointer to a node of a tree T with n nodes. So a pointer-based tree representation needs $\Theta(n \log n)$ bits. However, there are only $4^n/\Theta(n^{3/2})$ trees with n nodes, so $\log(4^n/\Theta(n^{3/2})) = 2n - \Theta(\log n) \leq 2n$ bits should be sufficient according to the information theory lower bound. Mainly, there are four approaches for the succinct representation of ordinal trees:

1. *Balanced Parenthesis Sequences (BPS)*. BPS was proposed by Munro and Raman [83], where an ordinal tree is encoded in $2n$ bits by performing a depth-first traversal of T (i.e., the nodes are listed in preorder). This encoding is formed as follows. We append an opening parenthesis each time we arrive at a new node, and a closing parenthesis when we definitively leave the node in the traversal. Thus we use a pair of parentheses "( )" for each node in the resulting sequence $P[1..2n]$.
2. *Depth First Unary Degree Sequences (DFUDS)*. This scheme was proposed by Benoit et al. [16]. In order to build this sequence P, with the same depth-first traversal of the tree, we encode the arity of each node in unary (for a node of degree d, we put d opening parentheses plus one closing parenthesis). As a result we obtain a sequence of $2n$ balanced parentheses.
3. *Level Order Unary Degree Sequences (LOUDS)*. The idea of Jacobson [60] is to build a sequence of length $2n$ by a level-order traversal of the tree. For each level in increasing order, we encode the arities of all the nodes from left to right in unary codes, as for DFUDS.
4. *Tree Covering (TrC)*. In this approach, given by Geary et al. [47], the representation of T also occupies $2n$ bits. The method consists of a tree decomposition into a set of connected *mini-trees*, each of which is also decomposed into a set of connected *micro-trees*.

The four representations of ordinal trees can be stored in $2n + o(n)$ bits, supporting a large number of navigation operations. Figure 2.5 shows an example of BPS, DFUDS and LOUDS succinct representations for the same tree.

Sadakane and Navarro [107] proposed a succinct representation of ordinal trees in $2n+o(n)$ bits with a BPS approach and maximum functionality. The proposal is based on a data structure called *range min-max tree* T_{mM}. The method handles a virtual array of excess of opening minus closing parentheses $E[1..2n]$ built over P, where $E[i] = rank_{(}(P, i) - rank_{)}(P, i)$. Later, we partition the BPS P into blocks of fixed size, and for each partition, we create a leaf T_{mM} node that stores summary measures, such as minimum and maximum values of E. We then create a multi-ary hierarchy of internal T_{mM} nodes, with information on

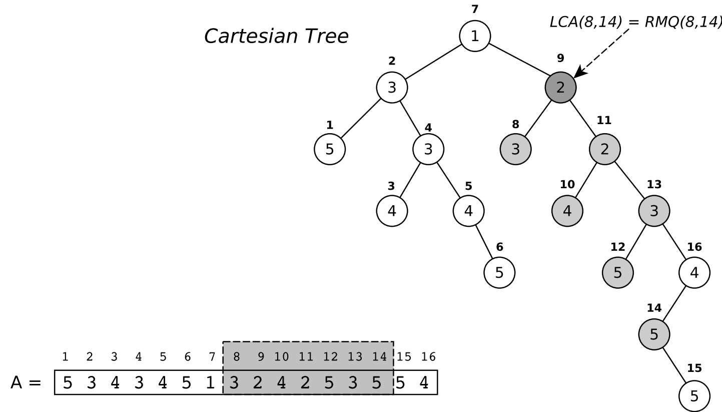

Figure 2.4: The *Cartesian Tree* on the input array $A[1..16]$, with an illustration of the relation between $LCA_{CT}(8,14) = 11$ and $RMQ_{CT}(8,14) = 11$.

the ranges of E, until building the root node, which stores the information of the whole range $E[1..2n]$. They use this information stored in the tree structure to achieve constant time for many tree navigation operations. They also show how with a base set of five operations, we can compute many others. These essential operations on the BPS sequence are: *rank*, *select*, *findopen(i)/findclose(i)* (position of parentheses matching $P[i]$), and *enclose(i)* (position of tightest open parentheses enclosing node i). This approach can be extended to dynamic trees with the same asymptotic space, but the time requirement becomes $O(\log n)$ for most operations.

2.5.3 Cartesian Tree

The Cartesian Tree (CT) [115] is a binary tree data structure built on an array $A[1..n]$ of elements that have a total order. The Cartesian Tree is defined as follows: The root of the Cartesian Tree is $A[i] = min\{A[1], ..., A[n]\}$; its left sub-tree and right sub-tree are Cartesian Trees too, which are computed on $A[1], ..., A[i-1]$ and $A[i+1], ..., A[n]$ respectively (see the example in Figure 2.4). If the array A contains equal elements, then there are different CTs for A. On the other hand, if we always choose the leftmost minimum, the result is called a *Canonical Cartesian Tree*. The Cartesian Tree can be represented succinctly in $2n + o(n)$ bits, such structure supports several navigation operations (see previous section); and it can be built efficiently in $O(n)$ time [39].

2.5.4 Lowest Common Ancestor and Range Minimum Query

The *Lowest Common Ancestor* (LCA) between two nodes v and w in a tree T is the deepest node $l = LCA_T(v, w)$ that is an ancestor of both nodes v and w, assuming that any node is an ancestor of itself. LCA is a commonly required functionality in many problems, including DR.

On the other hand, the *Range Minimum Query* (RMQ) problem is defined as follows. Given an array $A[1,n]$ with elements from a totally ordered set, an RMQ data structure returns the position of the minimum value in any range of A. Formally, it receives a pair of positions $1 \leq i \leq j \leq n$ and returns

$$\mathsf{rmq}_A(i,j) \;=\; \operatorname{argmin}_{i \leq k \leq j} A[k].$$

In many cases one prefers the leftmost position when there are more than one minima in $A[i..j]$.

These two operations are closely related. Gabow et al. [43] showed that LCA in a static tree T can be reduced to RMQ as follows: We store the depths of the nodes in an array H in the same order in which they are visited during an in-order tree traversal of T, and store in $I[j]$ the node from which the depth $H[j]$ comes. Then, letting R be the inverse array of I (i.e., $I[R[j]] = j$), it holds $LCA_T(v,w) = I[RMQ_H(R[v], R[w])]$. On the other hand Bender and Farach [15] compute $rmq_A(i,j) = inorder_{CT}(inorder^{-1}_{CT}(i), inorder^{-1}_{CT}(j))$, where CT is the Cartesian Tree of A.

For RMQ, Fischer and Heun [40] gave an optimal size solution, which requires $2n+o(n)$ bits and computes $\mathrm{RMQ}_A(i,j)$ in constant time, under the word-RAM model, without accessing the original array at query time. This structure is based on a labeled and ordered tree called *2d-Min-Heap*, M_A. The nodes of M_A are $v_0, v_1, ..., v_n$, where v_i corresponds to the cell $A[i]$. The node v_0 is the root and corresponds to a virtual cell $A[0] = -\infty$. The parent node of v_i is v_j if $j < i$, $A[j] < A[i]$, and $A[k] \geq A[i]$ for all $j < k < i$. This rule sets a vertical and horizontal order in the tree, such that the labels between sibling nodes are non-increasing from left to right, and for ancestor nodes their labels are increasing from the root node to each leaf. Then, for a node v with children $v_1, ..., v_k$, it holds $A[v] < A[v_j]$ and $A[v_j] \leq A[v_{j-1}]$ for all $1 < j \leq k$. They store M_A with a succinct tree representation in $2n + o(n)$ bits (see Section 2.5.2) [107], and enable the structure to compute LCA queries. Thereby to obtain $\mathrm{RMQ}_A(i,j)$, we let $l = \mathrm{LCA}_{M_A}(v_i, v_j)$. If $l = i$ then $\mathrm{RMQ}_A(i,j) = i$, and otherwise $\mathrm{RMQ}_A(i,j)$ is given by the child of l on the path from l to j. An example is given in Figure 2.5, where $l = \mathrm{LCA}_{M_A}(i,j) = \mathrm{LCA}_{M_A}(8,14) = 7 \neq i \Rightarrow \mathrm{RMQ}_A(8,14) = 11$.

2.6 Compressed Text Indexes Based on the SA

We have already mentioned how useful is the suffix tree for text indexing. However its great problem is the required space to store its structure, which is around 20 times the size of the indexed text. For that reason, when it is needed to index a big amount of text data it is preferable to replace it by a smaller index. The suffix array (SA) then arises as the ideal candidate to index the text, which still maintains many of the features of the ST. The SA $A[1..n]$ for a text $T_{1..n} = t_1 t_2..t_n$ can be used to compute count and locate, two of the most important operations in text indexing. Given a query pattern p, count refers to the number of times p appears in T. The indices corresponding to every string that starts with the same pattern are in consecutive SA positions. The occurrence interval $I_c = [sp, ep]$ of a pattern $p_{1..m}$ can be found in $O(m \log n)$ time by two binary searches on A (accessing T to compare

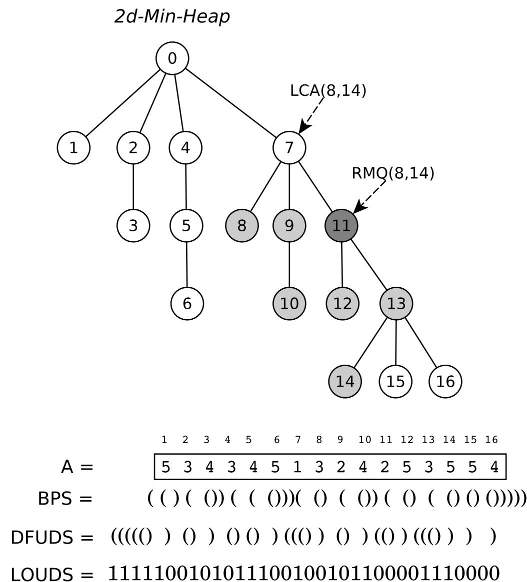

Figure 2.5: The *2d-Min-Heap* data structure on the array A and the BPS, DFUDS and LOUDS succinct representations of the tree. Observe that $RMQ_A(8,14) = 11$ is given by the child of $l = \text{LCA}_{M_A}(8,14) = 7$ on the path from l to $j = 14$.

symbols). After that, count is determined as $ep - sp + 1$. On the other hand, locate obtains the value of $A[i]$ for every suffix of I_c, which can be done in constant time per value with a plain representation of A.

Even though the SA requires only around 20% of the suffix tree size, it still needs too much space for large text collections. The challenge then is to build a smaller representation of the SA. Next we describe some of the most important compressed SA structures.

First we describe the two most important families of text indexes, most of which are self-indexes. These are text indexes that, additionally to computing count and locate, can generate any text substring (extract operation), that is, a self-index replaces the text itself. A self-index structure performs all of these operations in a space close to that of the compressed text, using at most the plain text size plus a sub-linear extra space.

2.6.1 The Compressed Suffix Array

These indexes take advantage of the regularities of the suffix array to represent it in little space. The CSA was introduced by Grossi and Vitter [54] in the year 2000, where it applied only to texts over a binary alphabet. It is not a self-index, because it needs the text to operate. Sadakane turned this CSA into a self-index with some improvements [101, 104]. Grossi and Vitter generalized their initial CSA to general alphabets [55], but it still does not replace the text. The general method of these proposals consists of a hierarchical decomposition of the SA by sampling some cells, and obtaining a smaller sampled SA. The non sampled cell values are obtained from the sampled ones using a so-called *function* ψ. The ψ function, used in

CSA	Required bits	count time locate time	Conditions
[100]	$O(nt(\log n)^{1/t})$	$O(m\,t\log n)$ N/A	$1 \leq t \leq \log\log n$
[104]	$\frac{n}{\varepsilon}H_0 + O(n\log\log\sigma) + \sigma\log\sigma$	$O(m\log n)$ $O(\log^{\varepsilon} n)$	ε *is a constant,* $0 < \varepsilon \leq 1$
[52]	$\frac{n}{\varepsilon}H_k + o\,(n\log\sigma)$	$O(m\log\sigma + \log^{2+\varepsilon} n)$ $O(\log^{1+\varepsilon} n)$	$0 < \varepsilon \leq 1$, $k \leq \alpha\log_{\sigma} n$, $0 < \alpha < 1$; ε,α *constants*
[55]	$(1+\varepsilon^{-1})n\log\sigma + o(n\log\sigma)$	$O(m/\log n + \log^{\varepsilon} n)$ $O(\log^{\varepsilon} n)$	ε *is a constant,* $0 < \varepsilon \leq 1$

Table 2.1: Time-space complexities of most popular CSAs. The time construction for these indexes is $O(n\log\sigma)$.

each level, is a permutation of $[1, n]$. In the i-th position it stores the index of the suffix that is lexicographically the successor to the i-th smallest suffix; that is, $\psi[i] = SA^{-1}[SA[i]+1]$ if $SA[i] < n$, otherwise $\psi[i] = SA^{-1}[1]$. Array ψ can be compressed because it is formed by σ (the alphabet size) increasing sub-sequences. At the last level, the shortest sampled SA can be directly stored.

S. Rao also worked on a similar representation of the suffix array over binary text [100]. His method generalizes of the hierarchical decomposition of the SA above. It chooses the cells that are multiples of a given parameter l, and with these he creates a sampled SA. For the rest of the cells, he uses a generalization of the ψ function, where the cells point to the next position that is a multiple of l and also indicate the distance from the current cell to that multiple, $d - (A[i] \; mod \; l)$.

The summary of the complexities for the main CSAs is shown in Table 2.1.

2.6.2 The FMI Family

FM-Indexes are another approach to compress the SA. In this instance the SA is present implicitly through the *LF Function*. The LF function is the inverse of ψ. It is used to move backwards over SA, that is, $LF[i]$ gives the lexicographic ranking of the suffix with position $SA[i]-1$ in the text. Counting and locating are solved by a method called *Backward Search* [33], using an extension of the LF function, and not by a binary search like the SA or CSA. A backward search first considers the last symbol of the pattern p, computing the interval $[sp_m, \mathrm{e}p_m]$ of all the suffixes that begin with symbol p_m. The next step is to find the subinterval with all suffixes that begin with the string $p_{m-1,m}$. The process is repeated until the symbol p_1 is processed.

The FM-Indexes are based on the *Burrows-Wheeler Transform (BWT)* of T, T^{bwt} [18]. The BWT of T is a permutation of the text symbols. It is formed by orderly traversing the SA and concatenating the symbol preceding each suffix of the text pointed by the SA, where the first suffix is preceded by $T[n]$. T^{bwt} replaces the text and is usually more compressible

Index	Required bits	count time locate time	Conditions
[33]	$5nH_k + o(n \log \sigma)$	$O(m)$ $O(\log^{1+\varepsilon} n)$	$\sigma = O(1)$, $0 < \varepsilon \leq 1$, $k \leq \log_\sigma(n/\log n) - \omega(1)$
[35]	$nH_k + o(n \log \sigma)$	$O(m \log \sigma)$ $O(\log \sigma \log^2 n/ \log \log n)$	$k \leq \alpha \log_\sigma n$, $0 < \alpha < 1$ $\sigma = o(n)$
[36]	$nH_k + o(n \log \sigma)$	$O(m(1 + \log \sigma / \log \log n))$ $O(\log^{1+\varepsilon} n \log \sigma / \log \log n)$	$k \leq \alpha \log_\sigma n$, $0 < \alpha < 1$ $0 < \varepsilon \leq 1$, $\sigma = o(n)$
[7]	$nH_0 + o(n)(H_0 + 1)$	$O(m \log \log \sigma)$ $O(\log n \log \log \log n \log \log \sigma)$	$\sigma = o(n)$
[12]	$nH_k + o(nH_k) + O(n)$	$O(m)$ $O(\log n)$	$k \leq \log_\sigma n - \log_\sigma \log n$ for any σ

Table 2.2: Time-space complexity of main self-indexes of the FM-Index family. The time construction for these indexes is generally $O(n \log \sigma)$.

than the text itself. It is also useful for backward search. Observe that the BWT and the LF array have a close relationship, because $LF[i]$ is the lexicographic ranking of the suffix that begins with the symbol T^{bwt}_{i}. A powerful property of the BWT is that the LF values can be computed from any representation of T^{bwt} that supports $rank_{T^{bwt}_{\mathrm{i}}}(T^{bwt}, i)$ queries. If we maintain additionally a vector $C[1..\sigma]$, which stores in $C[c]$ the number of times that symbols less than c appear in T^{bwt}_{i}, it is not necessary to store the LF array: We can compute the LF values by $LF(i) = C[T^{bwt}_{\mathrm{i}}] + rank_{T^{bwt}_{\mathrm{i}}}(T^{bwt}, i)$. Backward search requires the more general operation $rank_c(T^{bwt}, i)$.

During the year 2000, Ferragina and Manzini [33] designed the first index with this approach, which worked for constant alphabets only. Ferragina et al. [35] improved these previous results with their *Alphabet-Friendly FM-Index*. In this same line of research, they offered later another variation of this index [36]. An improvement was achieved by Barbay et al. [7], who reduced the $o(n \log \sigma)$ bits in the index space and speeded up the time to compute rank. Finally, Belazzougui and Navarro [12] improved all the previous results, locating occurrences in time independent of the alphabet size. They offered a FMI that uses $nH_k(T) + o(nH_k(T)) + O(n)$ bits, computing count and locate in $O(m)$ and $O(\log n)$ time respectively, for any σ value. Table 2.2 summarizes these main results in the FM-index family.

2.6.3 The Locally Compressed Suffix Array

Some of the previous compressed indexes described (CSA or FMI) compute count in $O(m)$ time, for a query pattern of length m. This is an excellent result considering that they work in reduced space. After count, the most frequent task is to compute locate. In this case, the performance of these indexes to retrieve SA values is far from optimal. While by using the plain suffix array it is sufficient to do only one access to memory (i.e., $O(1)$ time), the

Algorithm 1 LCSA Re-Pair on A'

1: Let ab the most frequent pair $(A'[i], A'[i+1])$ in A' with frequency f_{ab}.
2: **while** $f_{ab} > 1$ **do**
3: Create a new integer symbol s, larger than all existing symbols in A'.
4: Add a new rule to the dictionary: $R = R \cup \{s \to ab\}$
5: Replace every occurrence of ab in A' by s.
6: Let ab the most frequent pair $(A'[i], A'[i+1])$ in A' with frequency f_{ab}.
7: **end while**

compressed representations need around $O(\lg^{\varepsilon} n)$ time, where $\varepsilon > 1$ in practice. This is even more relevant when it is necessary to retrieve several consecutive positions, as occurs with locate when retrieving each value of the occurrence interval $A[sp, ep]$. The *Locally Compressed Suffix Array* (LCSA) [51] tries to reduce this gap between the compressed indexes, which take a lot of time to solve locate, and the plain representation that achieves optimal time.

The main advantage of the LCSA is the capability to display the values in a contiguous range of A: it can extract any c consecutive cells $A[i, i+c-1]$ in time $O(c + \lg^{\varepsilon} n \lg\lg n)$. The structure is based in grammar compression and implements a highly-local compression method. This means that an arbitrary segment of the suffix array can be decompressed by accessing mostly contiguous memory areas. The space complexity is given in function of ρ, which is the number of *runs* in A. A run is a maximal segment of length l that differs by one value from another segment, that is $A[i+r] = A[j+r]+1$, $\forall\ 0 \le r < l$. The LCSA requires $O(\rho(1 + \lg \frac{n}{\rho}) \lg n + n \lg^{1-\varepsilon} n)$ bits, for some i, j. Mäkinen and Navarro [74, 75] showed that ρ is smaller as T is more compressible, and if $H_k(T) \le 1$ then $\rho \le nH_k(T) + \sigma^k$. This space can reach, in practice, as little as 25% of the original suffix array size.

The basic structure of the LCSA represents $A[1..n]$ in differential form $A'[1..n]$, where $A'[1] = A[1]$ and $A'[i] = A[i] - A[i-1]$, $\forall\ 1 < i \le n$. The regularity to exploit is that a run $A[i+r] = A[j+r]$ of length l in A produces repetitions in $A'[i, i+l-1]$, $A'[i+r] = A'[j+r]$ for all $1 \le r < l$. Thus they apply Re-Pair, a grammar-based compression technique [71], on A', as described in Algorithm 1. Re-Pair obtains a dictionary of rules R plus a reduced sequence C (the final A'). The alphabet of C is the union of the original alphabet and the new symbols s created.

To decompress $C[i]$, if $C[i] \le n$, it is an original value of A'. Otherwise, we obtain both symbols from $R[C[i]-n]$, and expand them recursively. We can reproduce the corresponding u cells of A' in $O(u)$ time.

The index is completed with some additional structures to retrieve $A[i, i+c-1]$ from $A'[i, i+c-1]$. The total cost to access c contiguous cells is $O(l + d + c)$, where l is the size of a sampling of A and d is a limit for the maximum number of original symbols that can be expanded from any non-terminal symbol.

The second structure applies Re-Pair on on the Ψ function [55], using the observation that runs in A are also runs of 1s in Ψ and vice versa.

Text: tu gusto no gusta del gusto que gusta mi gusto, tu gusto no gusta de mi gusto

LZ77 Parsing:

1 2 3 4 5 6 7 8 9 10 11 12 13 1415 1617 18 19 20 21 22 23 24 25 26 27 28 29
t.u. .g.u.s.t.o. .n.o. gust.a. .d.e.l. gusto .q.u.e. gusta .m.i. gusto.,. .tu gusto no gusta de. mi gusto.

Topology of occurrences:

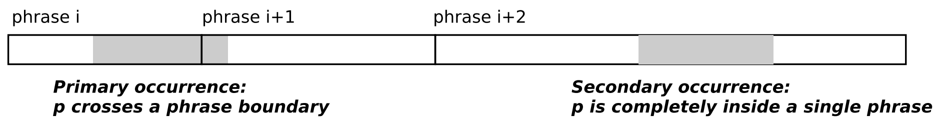

Figure 2.6: At the top, the resulting $z = 29$ phrases after applying the LZ77 parsing on the text $T_{1..77}$: *"tu gusto no gusta del gusto que gusta mi gusto, tu gusto no gusta de mi gusto"*. The dictionary phrases are enumerated on the text and the boundaries are indicated with points. At the bottom, we illustrate the two types of occurrences in the parsed text.

2.7 LZ-Based Compressors

Lempel and Ziv studied different ways to take advantage of the repetitiveness in sequences of symbols. The idea is to represent text segments by pointers to their previous occurrences. They offered the following two important results.

2.7.1 LZ77 Compression

The LZ77 parse [117] is a commonly used compression scheme. That parser obtains compression by replacing strings with pointers to their previous occurrences in the text, achieving one of the best results in terms of compressibility for very similar texts. The parsing scans the text from left to right. In each step of the process it creates a new phrase by searching the preceding text for the longest prefix of the remaining text.

More formally, suppose that we have processed the text $T_{1..i-1}$, and there is at least one match of $T_{i..i+k}$ in $T_{1..i+k-1}$ but not for $T_{i..i+k+1}$ in $T_{1..i+k}$. We then create a new LZ77 phrase for $T_{i..i+k+1}$ pointing to the earliest match found in $T_{1..i+k-1}$. Note that under these conditions, a prefix of a LZ77 phrase can be also the suffix of its own source. The parsing then replaces $T_{i..i+k+1}$ by the pointer to the early occurrence for $T_{i..i+k}$ plus the string length (when $k > 1$) and the mismatch character T_{i+k+1}, as $(start, k, T_{i+k+1})$. The exceptions are the first occurrences of each alphabet's character; in that case we only store the symbol. There are some variants in the way to create the LZ77 phrases, for instance LZ-End [69], which aims at easier decompression of individual phrases.

The LZ77 compression scheme induces a classification of the occurrences of any string in the parsed text [63]. A primary occurrence (or primary match) for a pattern $p_{1..m}$ occurs when the match $T_{i..i+m-1}$ includes a phrase boundary, or when $m = 1$ and it is the first occurrence of that symbol; otherwise it is a secondary occurrence. Figure 2.6 gives an example where

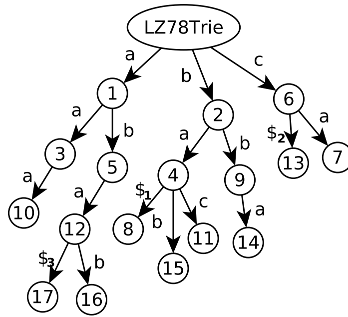

Figure 2.7: The resulting phrases when applying the LZ78 parsing of a collection with 3 texts, $T = abaabaabccaba\$_1bbaaabacabac\$_2bbababababababa\$_3$. The result is a dictionary of $n' = 17$ phrases, which are enumerated and separated by points in the figure. We also show how the phrases are organized in a trie.

the LZ77 parsing outputs $z = 29$ phrases. It also illustrates the two types of occurrences considering the phrase boundaries.

2.7.2 LZ78 Compression

The LZ78 compression algorithm [118] also parses the text $T_{1..n}$ to be compressed into a sequence of *phrases*. Each phrase is formed by appending a new character to the longest possible previous phrase, and is represented with the index of the phrase used and the new character appended. The result is a collection of n' phrases, where $n' \leq n/\log_\sigma n$, and thus the output of the compressor has at most $n'(\lg n' + \lg \sigma) \leq n \lg \sigma + o(n \log \sigma)$ bits if $\sigma = o(n)$. On compressible texts, however, the space decreases. Actually, the number of bits output by the LZ78 compressor can be bounded as $|LZ78| = n'(\lg n + \lg \sigma) \leq n\, H_h(T) + o(n \log \sigma)$ [68] for any $h = o(\log_\sigma n)$.

Figure 2.7 shows an example of LZ78 parsing. The output of the LZ78 compressor are the pairs:
$(0,a)(0,b)(1,a)(2,b)(1,b)(0,c)(6,a)(4,\$_1)(2,b)(3,a)(4,c)\ (5,a)(6,\$_2)(9,a)(4,b)(12,b)(12,\$_3)$.

Phrase number 0 corresponds to the empty string, otherwise phrase number i refers to the ith phrase formed during the parsing. The figure also shows a trie with all the phrases, where the node numbers are the phrase indices. Note that the set is *prefix-closed*, that is, the prefix of a phrase is also a phrase, and thus every trie node corresponds to a distinct phrase.

This trie is used for efficient parsing in $O(n)$ time. It is built as we parse: we traverse the trie with the text to be parsed, and every time we fall off the trie we add a new child with the symbol that was not found among the children, thus creating the new phrase, and return to the root.

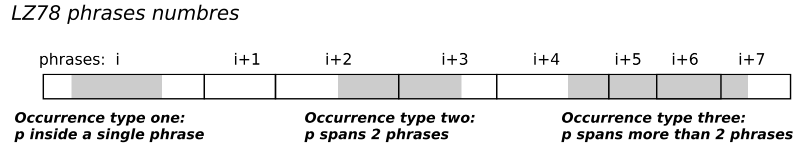

Figure 2.8: The three types of occurrences according to how they span blocks (or phrases).

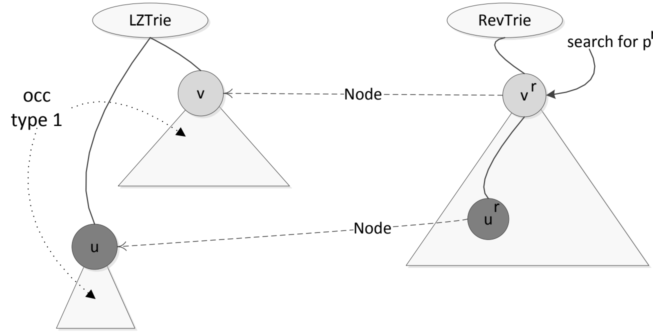

Figure 2.9: The structures to report occurrences of type 1.

2.8 The LZ-Index

This section describes a technique to index text, whose approach is based on LZ78 compression [118] (detailed in Section 2.7.2). Given that two of the results detailed in this thesis (in Chapters 6 and 7) are built on the index of Navarro [85], we now describe the basic members of that structure.

2.8.1 The Basic Structure

The basic LZ-Index [85] builds on the LZ78 parsing of the text $T_{1..n}$ to index. Its first two components are two tries, which store the set of phrases obtained for T using LZ78 (called *LZTrie*, the same shown in Figure 2.7), and the trie of the reversed phrases (called *RevTrie*), that is, the phrases read backwards. Note that LZTrie can be used to find the phrases that start with p, and RevTrie to find those that end with p (by looking for the reverse of p). Note that the set of reversed phrases is not prefix-closed, therefore RevTrie may contain nodes that do not correspond to any phrase.

These two tries are represented in compact form (Section 2.4), so that they support efficient navigation. Apart from basic navigation toward children and parents, we can find in constant time the preorder index of a node v, the node with a given preorder index, and the range of preorder values for the subtree rooted at v. We also store an array that associates the phrase number with each node. The space per trie is $n' \lg n' + O(n' \log \sigma)$ bits [85, 3].

During the search process, it is necessary to travel from a node in RevTrie to the node

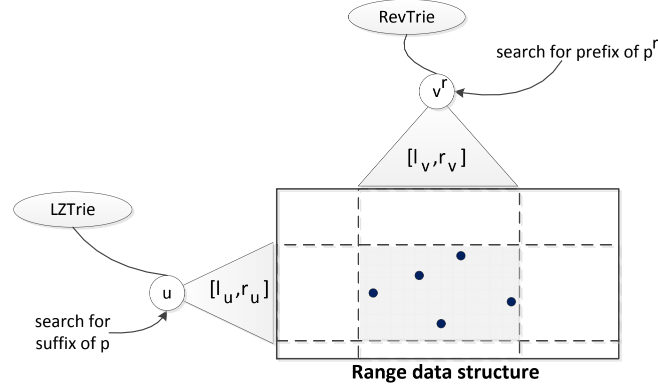

Figure 2.10: The scheme to report the occurrences of type 2.

in LZTrie that represents the same phrase. For this task, the index includes an array called *Node*, which does the mapping between phrase identifiers and preorder values in LZTrie. This array uses other $n' \lg n'$ bits.

The last basic member of the LZ-Index, *Range*, is a data structure used to find the occurrences that begin inside a phrase and end in the next one. This is a two-dimensional $n' \times n'$ grid where we store n' points. If the $(k+1)$th text phrase is represented by the node with preorder i in LZTrie and the kth phrase is represented by the node with preorder j in RevTrie (counting only nodes that represent phrases), then a point at row i and column j is placed in the grid. Note that with the LZTrie preorder value (i.e., the row) we obtain the phrase identifier of a point. The grid is implemented with a wavelet tree (see Section 2.5.1) using $n' \lg n'(1 + o(1))$ bits, so that all the t points in a rectangular query range are retrieved in time $O((t + 1) \log n')$.

With these components, the occurrences of a pattern $p_{1..m}$ in $T_{1..n}$ are found as follows, according to the three possible ways p can occur across the phrases of T (see Figure 2.8).

1. Find the occurrences completely contained in a single phrase (occ_{t1} occurrences of type 1). Search for p^r (the reversed pattern) in RevTrie, arriving at node v^r. Every node u^r in the subtree of v^r corresponds to an occurrence of p at the end of a phrase. Any other phrase formed from that of u^r also contains p, and those form all the occurrences of type 1. Thus, any occurrence of type 1 is at an LZTrie node that descends from u, where u is the LZTrie node that corresponds to u^r. Therefore, for each node u^r, we travel from RevTrie to LZTrie using Node, and report every phrase in the corresponding subtree of LZTrie. The search time for p^r in RevTrie is $O(m)$, and then each occurrence of type 1 is reported in $O(1)$ time, for a total time of $O(m + occ_{t1})$. See Figure 2.9.
2. Find the occurrences that span two consecutive phrases (occ_{t2} occurrences of type 2). The pattern is split in every possible way into $p = p_{start} \cdot p_{\text{end}}$. For each such split, we search for p^r_{start} in RevTrie (finding locus v^r) and for p_{end} in LZTrie (finding locus u). Both searches take $O(m)$ time (for each division of p), and obtain the preorder ranges $[l_v, r_v]$ and $[l_u, r_u]$ of occurrences for all prefixes and suffixes. Now we query Range for $[l_v, r_v] \times [l_u, r_u]$, retrieving all the phrase numbers k that end with p_{start} such that $k+1$

starts with p_{end}. Since this is done for every split, the cost is $O(m^2+m\log n+occ_{t2}\log n)$. See Figure 2.10.

3. Find the occurrences that span more than two consecutive phrases (occ_{t3} occurrences of type 3). Since a phrase must contain a substring of p in each such occurrence, and every phrase is distinct in the LZ78 parsing, there are only $O(m^2)$ possible occurrences of this type. These are found with a more laborious process [85] that takes time $O(m^3)$.

The total space of the LZ-Index is $4n' \lg n' + O(n' \log \sigma)$ bits, which is at most $4nH_h + o(n\log\sigma)$ for any $h = o(\log_\sigma n)$. The time for locating occ occurrences is $O(m^3 + m\log n + occ \log n)$. Later improvements on this structure [3] reduce both time and size: the time to $O(m^2 + m\log n + occ\log n)$ and the size to $(2+\varepsilon)nH_h + o(n\log\sigma)$ bits, for any $\varepsilon > 0$. A practical advantage of the LZ-Index compared to CSAs is that it is faster when many occurrences must be reported [86, 2]. Note that, in this case, the pattern p is usually short and then most of the occurrences are of type 1, which are reported in $O(1)$ time. We exploit this property in our DR indices.

Chapter 3

Document Retrieval Review

This review presents the most important results for document listing and top-k retrieval problems. The first two sections focus in describing data structures on general texts, and the last one on solutions for repetitive texts. Several of our solutions build on these techniques.

3.1 Document Listing

We begin by introducing the method offered by Muthukrisman [84], which is the first optimal time solution for document listing in linear space. He builds his structure on the *Generalized Suffix Tree* (GST) (Section 2.4) of the text $T_{1..n} = \mathrm{d}_1\$_1\mathrm{d}_2\$_2 \ldots \mathrm{d}_D\$_D$. T is the result of concatenating the D documents of the collection, adding a special and unique endmarker symbol $\$_\mathrm{i}$ for each document d_i.

In that proposal he introduces a useful array in the document retrieval field, called *Document Array* $E[1..n]$. $E[i]$ stores the document ID in which the suffix $SA[i]$ begins, that is, $E[i] = j$ *iff* $T_{SA[i]}$ belongs to document d_j. Then, given a suffix rank he uses E to retrieve directly a document ID when it is needed. His structure considers another array $C[1..n]$, where $C[j] = i < j$ *iff* i is the largest index before j such that $E[i] = E[j]$. If i does not exist, then $C[j] = -1$. To complete the framework, he builds an RMQ structure (Section 2.5.4) on the array C.

In order to list the documents that contain a pattern $p_{1..m}$, the first task is to search for p in the GST, reaching the node $v = locus(p)$. Then v represents the lexicographic range of suffixes $I_v = [l, r]$ that start with p (this is equivalent to $SA[l, r]$). After that, Muthukrisman finds each different ID recursively. In each step he determines $i = RMQ_C(l, r)$, and if $C[i] < l$, he reports the document $E[i]$. Next, he continues recursively with the subintervals $[l, i-1]$ and $[i+1, r]$. Each recursion stops when $C[i] \geq l$. Figure 3.1 illustrates this process.

The disadvantage of Muthukrishnan's data structure is the space requirement. His structure considers the text T, the generalized suffix tree, the two arrays C and E, and the structure for RMQ on C. Altogether, it takes $O(n)$ words of space with a high constant factor.

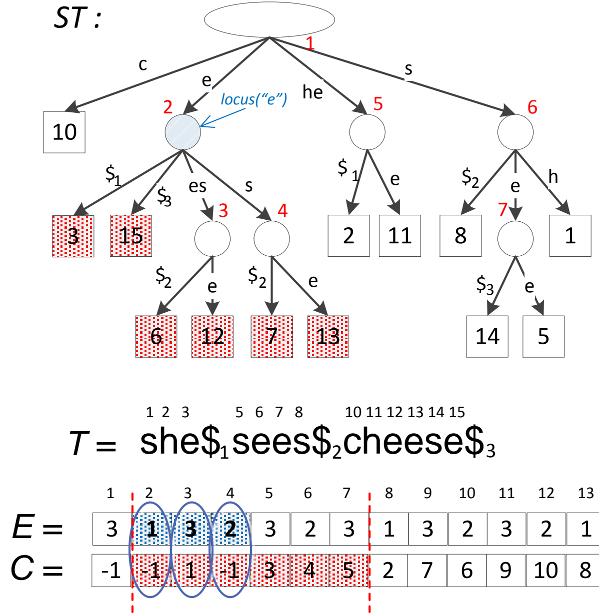

Figure 3.1: The generalized suffix tree of the text T and the arrays E and C that form the structure of Muthukrisman to solve DL queries. We indicate the locus node for the pattern "e", and the three positions, in the arrays, that are reported by the recursive process (i.e., the documents $E[4]$, $E[2]$ and $E[3]$).

Sadakane [102] avoids this drawback by presenting a succinct data structure for that proposal. He reduces these $O(n \log n)$ bits to $|CSA|+4n+o(n)+O(D\log\frac{n}{D})$ bits, where $|CSA|$ denotes the size in bits of some compressed suffix array (see Section 2.6.1). However, the resulting time for document listing is not optimal. The method takes $O(\mathsf{search}(p) + \mathsf{ndoc}\cdot\mathsf{lookup}(n))$ time, where $\mathsf{search}(p)$ is the time to find the interval $[l, r]$ of all suffixes starting with p using the CSA, and $\mathsf{lookup}(n)$ is the time to compute an entry of the SA (or its inverse array) with the CSA. To replace the C array, he constructs another tree τ_C that has $2n$ nodes, adding a unique leaf for each node in the Cartesian tree (see Section 2.5.3). With this he solves RMQ in C by computing LCA over τ_C, storing the tree with a succinct representation of ordinal trees (for instance using [107]). Therefore, his RMQ structure needs $4n + o(n)$ bits without storing C. To replace the E array, he stores a sparse bitvector $B[1..n]$ that marks the first position of each document in the concatenated text, so that $E[j] = rank_1(B, SA[j])$. This bitvector has D 1s out of n and thus is represented in $D\log n/D + O(D) + o(n)$ bits [99].

As we described in Section 2.5.4, Fisher [37] achieves the same optimal time to solve range minimum queries using only $2n + o(n)$ bits. Therefore, we can include this improvement to reduce the size of Sadakane's structure in $2n$ bits. Additionally Sadakane incorporates a bitstring $V_{1..D}$ needed to mark the documents that have already been reported, which replaces the need to compare $C[i]$ with l. This process is shown in Algorithm 2.

In order to obtain the interval of occurrences of the pattern $p_{1..m}$, he uses his CSA, which requires $O(m\log\sigma)$ time. A document listing query is then solved in $O(m\log\sigma + \mathsf{ndoc}\,\cdot$

Algorithm 2 Sadakane's adaptation of Muthukrishnan's algorithm to find in $O(\mathsf{ndoc})$ time all the ndoc different documents corresponding to the suffix interval $SA[l..r]$.

```
 1: procedure CDLP(l, r)
 2:     if l > r return
 3:     x ← RMQ_C(l, r)
 4:     d ← rank_1(B, SA[x])
 5:     if V[d] = 0 then
 6:         output d
 7:         V[d] ← 1
 8:         CDLP(l, x − 1)
 9:         CDLP(x + 1, r)
10:     end if
11: end procedure
```

$\log^{1+\varepsilon} n)$ time with $|CSA| + O(n)$ bits; the time $O(\log^{1+\varepsilon} n)$ corresponds to computing an entry of the SA using his CSA, where $\varepsilon > 0$ is a constant.

The wavelet tree (Section 2.5.1) was first used for DR solutions by Välimäki and Mäkinen [114]. To answer document listing queries, they proposed a structure that uses $|CSA| + 2n + n \log D(1+o(1)))$ bits and reports the ndoc documents in $O(\mathsf{search}(m) + \mathsf{ndoc} \cdot \log D)$ time. In this structure, they construct the wavelet tree on the document array $E[1..n]$ and use it to find any entry of E in $O(\log D)$ time. The wavelet tree computes $rank_{\mathrm{d}}(E, i)$, or $select_{\mathrm{d}}(E, i)$, for any document d and any index i, in $O(\log D)$ time. One advantage of this tree is that it does not need storing the vector C, because any entry $C[i]$ of Muthukrishnan's algorithm is easily determined as $C[i] = select_{E[i]}(E, rank_{E[\mathrm{i}]}(E, i) - 1)$. This proposal also includes a structure for RMQ in $2n + o(n)$ bits to apply the recursive algorithm of Muthukrishnan [84] to solve document listing.

Gagie et al. [46, 45] showed that a wavelet tree can be used for document listing without the need to make range minimum queries, but just a *Depth First Search* (DFS) traversal. They can report the distinct ndoc documents in $E[l, r]$ with their respective frequencies, in $O(\mathsf{search}(m) + \mathsf{ndoc}\ \lg(D/\mathsf{ndoc}))$ time. Navarro et al. [91] achieved nearly 50% compression of the wavelet tree in practice, at the price of nearly doubling the time required. They use this tree to solve DL and top-k retrieval

Hon et al. [58] reduced the space for document listing by modifying the structure of Sadakane. This solution requires $|CSA| + D \log(n/D) + o(n)$ bits, and answers DL queries in $O(\mathsf{search}(m) + \mathsf{ndoc} \log^{1+\varepsilon} n \cdot \mathsf{lookup}(n))$ time, for any $\varepsilon > 0$. They split Muthukrishnan's C vector into segments of size $\log^{\varepsilon} n$, and for each segment they take the smallest value to build a reduced Cartesian tree C_t. To solve queries for a pattern p, they also use the CSA to obtain the interval of all occurrences of p, $I_p = [l, r]$. In order to report the different documents in I_p, they select the nodes in C_t that are smaller than l. Next, they report all different documents found among the segments represented by these selected nodes. They also adopted Sadakane's method for obtaining document identifiers and for not reporting repeated IDs; that is using the CSA for the collection, the bitvector B in $O(D \log(n/D)) + o(n)$ bits and the bitvector $V_{1..D}$, as shown in Algorithm 2.

In order to extend his index to report frequencies, Sadakane [106] adds to his classic DL proposal a single CSA for each document of the collection. Then each time he finds a document d to report, he obtains its $tf(p,d)$ by solving count in the CSA_{d} of that document. First, he determines the interval of all occurrences of the pattern $I_p = [l, r]$ by using the generalized CSA in time $\mathsf{search}(m)$. Next, for each document d to report he obtains the indexes i and j of the leftmost and rightmost occurrences of d in I_p. While i is given by the recursive process of Muthukrishnan, he builds a reciprocal structure to solve *Range Maximum Queries*, and replicates the recursive process, now obtaining the j values. With the bitvector B, Sadakane determines the position z of the first character of d in $T_{1..n}$, $z = select_1(B, d)$. Then, $tf(p,d) = j' - i' + 1$, where $i' = SA_{\mathrm{d}}^{-1}[SA[i] - z + 1]$ and $j' = SA_{\mathrm{d}}^{-1}[SA[j] - z + 1]$ are computed in time $\mathsf{lookup}(n)$ using the CSA and CSA_{d}. Hence, the total time to solve DL with frequencies is $O(\mathsf{search}(m) + \mathsf{ndoc}(\mathsf{lookup}(n) + \log\log \mathsf{ndoc}))$ using $2|CSA| + O(n)$ bits[1]. An important observation is that even though it is an attractive theoretical way to obtain frequencies, in practice this method does not have good performance in terms of space used. Navarro and Valenzuela [94] showed that the extended structure can increase the total size more than three times in practice.

Välimäki and Mäkinen [114] gave the $tf(p,d)$ for each document reported with the same wavelet tree of their index, by simply computing $tf(p,d) = rank_{\mathrm{d}}(E, r) - rank_{\mathrm{d}}(E, l-1)$. This means that they do not add another data structure and report frequencies in the same asymptotic query time.

Another approach was given by Belazzougui et al. [13]. They use *monotone minimum perfect hash functions*[2] *(mmphf)* for counting document frequencies from the interval of occurrences I_p. This additional data structure requires of $O(n \log\log D)$ bits and $O(1)$ time, or $O(n \log\log\log D)$ bits of space and $O(\log\log D)$ time to compute a frequency. So the total time is $O(\mathsf{search}(m) + \mathsf{ndoc} \cdot \mathsf{lookup}(n))$ or $O(\mathsf{search}(m) + \mathsf{ndoc}(\mathsf{lookup}(n) + \log\log D))$, respectively.

3.2 Top-k Retrieval

Several structures to solve top-k problem have been proposed with different trade-offs between query time and space usage. One of the most promising theoretical framework was proposed by Hon et al. [58]. They consider the generalized suffix tree of the collection and say that an internal node v contains an *entry* of document d if and only if at least two children of v contain the document d in their respective subtrees. So, for each internal node v that contains an *entry* of any document d, we store the following values in a structure called *N-structure*: the document d, the pointer to the lowest ancestor that also has an entry for the same document d and its document frequency. The *N-structure* entries are ordered by the preorder value of nodes. Reciprocally, for each *N-structure* entry we store in another structure, called *I-Structure*, an entry with the preorder of the node from which the pointer originates, the same document d and its frequency stored in the *N-structure* entry for this

[1] The $\log\log \mathsf{ndoc}$ time is needed to pair leftmost and rightmost position of each document via sorting.

[2] A minimal perfect hash function maps a set S of n keys into the set $\{0, 1, ..., n-1\}$ bijectively. In a monotone minimum perfect hash function the bijection must preserve the order of the keys [9].

pointer. These entries in *I-structure* are sorted by rank values of their pointers.

To answer a query for the parameters k, p, we first find $v = locus(p)$, which has a preorder range $[l_v, r_v]$. Next, we check each of the $O(m)$ ancestors u of v, searching for the sub-interval $[l_u, r_u]$ within $[l_v, r_v]$ in their *I-structure*. Notice that it is not necessary to check the nodes inside the subtree of v, because the *N-structure* of the subtree of v has a unique pointer corresponding to each different document that points to an ancestor of v for the same document d. They also proved that using an RMQ structure and given a set S of t non-overlapping ranges, we can find the k largest numbers in the union of all the t ranges of S in $O(t + k \log k)$ time. Altogether, the whole structure requires $O(n \log n)$ bits and finds the k documents where p appears most frequently in $O(m + k \log k)$ time.

Hon et al. [58] also gave a structure that uses $2|CSA| + o(n) + D \log \frac{n}{D} + O(D)$ bits and outputs the top-k answer in $O(\mathsf{search}(m) + k \log^{3+\varepsilon} n \cdot \mathsf{lookup}(n))$ time, for an $\varepsilon > 0$. Again the $|CSA|$ corresponds to the size of a compressed suffix array, which is able to compute $SA[i]$ or $SA^{-1}[i]$ in $\mathsf{lookup}(n)$ time, and performs the search for $p_{1..m}$ in $\mathsf{search}(m)$ time. Then, giving a k value, the method splits the tree leaves into segments of fixed size $g = k \log^{2+\varepsilon} n$, taking the leaves in order from left to right. On this partition, they build a reduced tree τ_k incorporating all first and last leaves of each segment. The internal nodes of τ_k are the lowest common ancestors between any consecutive pair of chosen leaves. In an internal node v of τ_k, they store the top-k answer associated with the leaves below v in the GST. They also include the term-frequency for each document in the sorted list, in decreasing order. The complete structure considers a tree τ_k for every k value that is a power of 2, $k \leq D$. It is called a *Sparsified Generalized Suffix Tree* (SGST). As each individual tree takes $O((n/g)k \log n) = O(n/\log^{1+\varepsilon} n)$ bits, the size for all trees is $O(n/\log^{\varepsilon} n) = o(n)$ bits.

This framework enables the following algorithm to find the k most frequent documents for a pattern $p_{1..m}$. We chose the lowest power of two $k' \geq k$, we search the locus node v in $\tau_{k'}$, and retrieve the answer top' stored in it. Note that v represents an interval $I' = [sp', ep']$ that is a subinterval of the leaves interval $[sp, ep]$ of the occurrences of p in GST, that is, $sp \leq sp'$ and $ep' \leq ep$. We then need to consider the document array intervals $E[sp, sp' - 1]$ and $E[ep' + 1, ep]$ to give the final answer to the query. As each of these two segments is shorter than g, this final step is made in time $O(g(\mathsf{lookup}(n) + \log k))$ using a priority queue of length k.

Navarro and Nekrich [89] presented a data structure to solve top-k queries with optimal $O(m + k)$ time and linear space. This work is based on the scheme of Hon et al. [58]. They mark all the nodes that have any *N-structure* entry and use the corresponding term frequency as a weight. This weight is the relevance score of a document d with respect to the string $path(v)$, where $path(v)$ corresponds to the string formed from the root node to the node v (i.e., it is $tf(path(v), \mathrm{d})$). Later, with a preorder traversal of the tree, they assign a unique index value for every entry in the marked nodes. For each marked node v they denote by $[l_v, r_v]$ the integer interval that encloses all indexes assigned to v or its descendants. They store in v the limits l_v, r_v, and encode the unique pointer that points to an ancestor of v with the same document (given the properties of *N-structure* [58]). For each document d that points to some ancestor u of v, they store a point $(l_v + \mathit{offset}, depth(u))$, where $depth(u)$ is

the depth of node u and $l_v+\mathit{offset}$ is the index of this entry. The weight of the points is the term frequency.

In order to solve top-k queries, they find $v = locus(p)$ and find the k heaviest points in the range $[0, depth(v)-1]\times[l_v, r_v]$, where l_v is the preoder value of v and $r_v = l_v+subtreesize(v)-1$. The points can be found in $O(depth(v) + k) = O(m + k)$ time using this grid [64]. The final space can be reduced to $O(n(\log D + \log \sigma))$ bits. Konow and Navarro [67] implemented this index, obtaining a structure that uses 3.3–4.0 bits per character (bpc) and answers top-k queries in k to $4k$ microseconds (μsec). Their time complexity is $O(m+(k+\log\log n)\log\log n)$ with high probability, on statistically typical texts. Gog and Navarro [49] also implemented the compressed solution of Hon et al. [58]. They overcame the results obtained by Konow and Navarro [67], reducing the size from 3.3–4.0 to 2.5–3.0 bpc and maintaining the response time. They achieved this applying two main ideas. First, simplifying the mapping from suffix tree nodes to the grid used in the original and theoretical work. Second, by a smaller representation of the grid. Consequently, this result is the best implementation to date for top-k retrieval on general collections.

Navarro et al. [91] experimented with several reduced-space proposals to solve top-k documents. They worked on the succinct structure of Hon et al., the SGST. Navarro et al. studied various ways to make the rectification on $[sp, sp'-1]\cup[ep'+1, ep]$ more efficient and represent the SGST in less space. Their best experimental result is (generally) to build a unique tree, with LOUDS representation (see Section 2.5.2), combined with a rectification using a variant of Culpepper et al.'s greedy method for wavelet tree structures [25]. Their space reaches 12–24 bpc, depending on the compressibility of the collection, and retrieval times are between 1 to 10 milliseconds, where the time complexity is upper-bounded by $O(m\log\sigma + k\log^{4+\varepsilon} n)$.

As we have seen, Belazzougui et al. [13] offered a proposal for DL with frequencies. Additionally, in this same work, they offered a solution to top-k document retrieval using their *monotone minimal perfect hash functions* (mmphf). Their structure takes a total time of $O(\mathsf{search}(m) + \mathsf{lookup}(n)k\log k\log^{1+\varepsilon} n)$, and the mmphf structures add $O(n\log\log\log D)$ bits to the CSA of the collection. There are several other theoretical proposals [87] that promise to use much less space than current implementations, but that are most likely to be even slower in practice (as already hinted in current studies [91]).

3.3 Document Listing in Repetitive Texts

The next chapter describes a proposal to solve document listing in repetitive texts. Although there are not many document listing structures for repetitive texts, we highlight some attractive results here.

First we introduce the index of Claude and Munro [23], which is based on a grammar-compressed representation of the input text. The proposal starts by compressing the text with a grammar compressor [71] generating a *grammar-compressed sequence*.

First we give some basic definitions.

Our grammar is a tuple $\mathcal{G} = (\mathcal{X} = \{X_1, X_2, ..., X_n\}, \sigma, \Gamma : \mathcal{X} \rightarrow \mathcal{X}^+ \cup \sigma, s)$*, where:*
– $\mathcal{X}$ *represents the set of non-terminal symbols.*
– σ *corresponds to the set of terminal symbols.*
– Γ *is the set of rules that transform a non-terminal into a sequence of non-terminals or just one terminal symbol. It does not allow cycles in the rules, and there is only one rule expanding each non-terminal. That is enough to make sure the grammar generates only one sequence.*
– s *is the identifier of the start symbol* X_s*.*
– *We define* N *as the sum of all the right sides in the grammar, that is.*

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

We define $\mathcal{F}(X_{\mathrm{i}})$ *as the result of recursively replacing the non-terminal,* s*, in* X_{i} *until obtaining a sequence of terminal symbols. In a similar way,* $\mathcal{F}(X_{\mathrm{i}})^R$ *is called reversed sequence, which is obtained by reading* $\mathcal{F}(X_{\mathrm{i}})$ *from right to left. We then say that* $\mathcal{G}$ *compresses* $T = t_1 t_2 ... t_u$*, if* $\mathcal{F}(X_s) = T$*. Finally, we define the height of the grammar as the longest path from the start symbol to a terminal symbol in the parse tree.*

Given a pattern p, we call *primary occurrences* all those non-terminals that contain p because two or more non-terminals generated by their rule, after being concatenated, contain the pattern. On the other hand, *secondary occurrences* are those non-terminals that contain p because they generate a single non-terminal that contains p.

The structure builds a grammar-compressed sequence from the generalized text $T = \$_0 T_1 \$_1 T_2 ... \$_{D-2} T_{D-1} \$_{D-1} T_D \$_D$. We have included $D+1$ symbols $\$_{\mathrm{i}}$ that are not present in the collection. The next step is to compress T with Re-Pair (see Algorithm 1), and with these results to build a grammar index [23]. The rules are:

– X_s generates d non-terminals $X_{t_1}, X_{t_2}, ..., X_{t_D}$.
– $X_{t_{\mathrm{i}}}$ generates the symbols between $\$_{\mathrm{i}-1}$ and $\$_{\mathrm{i}}$, that is $\mathcal{F}(X_{t_{\mathrm{i}}}) = T_{\mathrm{i}}$

The structure is augmented by adding inverted lists recording the documents where each non-terminal symbol is present. As several of such lists can be very similar to others, they again apply grammar-compression on the inverted lists by using Re-Pair. Therefore, in order to solve document listing for a pattern p, first we retrieve the inverted lists associated with each non-terminal where p appears. Then the lists are merged to remove duplicates.

The whole structure for an input text $T[1..u]$ is formed by: (i) the the grammar-index, which requires $2N \log n + N \log u + \varepsilon n \log n + o(N \log n)$ bits, $0 < \varepsilon \leq 1$; and (ii) the inverted lists, which use $t \log D$ bits, where t is the number of document identifiers stored. The size of the inverted lists also can be bounded by $u \log_\sigma u$ bits. The time for a document listing query is $O(t_{\mathsf{tsearch}}(p) + \mathsf{occ}_P \cdot \mathsf{ndoc})$ in the worst case, where ndoc is the output size and occ_P is the number of *primary* occurrences of the pattern p.

An important observation is that this approach does not start from Muthukrisman's struc-

ture, and does not use a suffix array, as all the previous indexes described. This is the first structure that introduces a grammar-index to solve document listing.

We now introduce another structure that also solves document listing on repetitive texts. This index, by Gagie et al. [44], uses a more traditional approach based on suffix arrays, more precisely on the *Run-Length Compressed Suffix Array* (RLCSA) [76]. The idea is to augment the RLCSA structure to answer document listing queries, specifically they store the so-called *Interleaved Longest Common Prefix* array (ILCP).

We define the LCP array LCP_S for a string $S[1..n]$ as follows. LCP_S stores 0 in its first position and for any value, $2 < i \leq n$, $LCP_S[i]$ is the length of the longest common prefix of the lexicographically $(i-1)$th and ith suffixes in S. That is, $S[SA_S[i-1]..n]$ and $S[SA_S[i]..n]$ have a maximum common prefix of length $LCP_S[i]$.

Another vector is introduced in this index, the ILCP array, defined as follows. Given the generalized text $T[1..n] = T_1\$_1 T_2\$_2 ... T_D\$_D$, the document array $E[1..n]$ associated to the SA of T, and the longest common prefix array LCP_T, the interleaved LCP array of T is defined as:

$$ILCP_T[i] = LCP_{T_{E[i]}}[rank_{E[i]}(E, i)], \text{ for any } 1 \leq i \leq n,$$

that is, it interleaves the local LCP array of the documents in the order of the global LCP array.

Let ρ be the number of *runs* (a segment of equal values) in $ILCP_T$. The authors store the structure in $|RLCSA| + \rho \lg(n/\rho) + O(\rho) + D \lg(n/D) + O(D)$ bits, such that DL can be solved in $O(\mathsf{search}(m) + \mathsf{ndoc}(\lg\lg n + \mathsf{lookup}(n)))$ time. The value ρ is low on repetitive collections.

Gagie et al. [44] also experimented with another structure to solve DL on repetitive collections. Given the observation that on highly repetitive documents their associated document array E also is repetitive, they store compressed precomputed answers to document listing queries covering long intervals of suffixes. Given a block size b and a constant $\beta \geq 1$ they build a sparse suffix tree Γ, storing in each node v the DL answer of its SA interval A_v, as follows. The leaves of Γ are the highest nodes $v_1, v_2, .., v_L$ such that these nodes are not ancestors of others and $|A_{v_i}| \leq b$, for $1 \leq i \leq L$. In order to select internal nodes for Γ, they consider the lowest group of leaves $u_1, u_2, ..., u_k$ of Γ, from left to right, so that the total size of sets $D_{u_1}, D_{u_2}, .., D_{u_k}$ is bigger than $\beta \cdot |D_v|$, where $v = LCA(u_1, u_2, .., u_k)$ in the suffix tree; then they include v in Γ and store its DL answer, D_v. They continue until they cover all the leaves of Γ. After that, they process all the internal nodes in the same way, considering now groups of nodes $u_1, u_2, .., u_k$ not ancestors of others, going up until reaching the root and completing Γ. As a final result, they obtain a structure where each node v of Γ satisfies one of the next two conditions:

1. $|A_v| < b$, thus the documents for v, D_v, can be found in time $O(b \cdot \mathsf{lookup}(n))$, where $\mathsf{lookup}(n)$ is the time to obtain a suffix value from the RLCSA index.

2. D_v can be obtained as the union of some sets $D_{u_1}, D_{u_2}, .., D_{u_k}$ of total size at most

$\beta \cdot D_v$, whose nodes $u_1, u_2, .., u_k$ are in Γ.

The precomputed answers of the nodes of Γ are compressed with a grammar-based compressor, which exploits the repetitiveness in the lists. With this, the complete structure includes the RLCSA index, the reduced grammar-compressed lists, the representation of the sparse suffix tree Γ, and some bitvectors. In total, with this structure they solve DL for a pattern $p_{1..m}$ in $O(\mathsf{search}(m) + \log\log n + \beta \cdot h \cdot \mathsf{ndoc})$ time if the answer is stored in Γ, where h is the height of Γ, or $O(\mathsf{search}(m) + \log\log n + b \cdot \mathsf{lookup}(n))$ if it is not. Experimentally they compared this structure with their previous index based on ILCP array, LCP-DL-Index, and a brute force technique, considering only the RLCSA and a bitvector $V[1..n]$. With respect to the two others structures they sowed that the time can be improved up to 2 orders of magnitude at the expense of increasing the size. Sometimes, in the same space required for the LCP-DL-Index, the new DL time is around 10 times faster than their previous solution.

Chapter 4

Contributions in Text Indexing

This chapter describes the first contributions in this thesis, which are focused in the process of text indexing. Even though these results are not properly from the document retrieval field, they deal with fundamental aspects in which most DR indexes build.

The first section describes two implementations of different theoretical proposals to build compressed representations for suffix arrays. These implementations and their results have been included as a part of a previous work published in the Journal of Experimental Algorithmics [51]. The experiments included in this section show that these proposals, in practice, do not perform better than other current and popular implementations in the field, both at query time and required space. Still, demonstrating this is an important contribution to the field, since these promising theoretical proposals had not been tested before.

The second section describes an index, the Hybrid-Index, to solve efficiently the pattern matching problem on highly repetitive texts. This structure introduces a simple technique for reducing the size of conventional indexes when the text contains several segments of repeated strings. We experimentally show its performance in comparison with a conventional index, like the FMI [33, 35, 36], and an index for repetitive text, like the LZ77 of Kreft and Navarro [69], both in query time and required space. This work was published in the Philosophical Transactions of the Royal Society of London [29].

The Hybrid-Index is also the base of Chapter 8, which illustrates how to adapt this basic structure in order to answer document retrieval queries.

4.1 Structures for Compressed Suffix Arrays

This section describes part of the work published in [51], specifically in the section of experiments. We implemented and tested two compressed representations of the suffix array: the structure of Grossi and Vitter [54, 55] (GVCSA) and the proposal of Rao [100] (RaoCSA), which had not been implemented.

A feature in common of these proposals is that their structures need to compress sequences of increasing numbers. To do that, both proposals have used the encoding of non-decreasing integers given by Elias-Fano [26, 27] and Okanohara and Sadakane[95], which do not only guarantee compression, but also to retrieve any number in constant time. We start then describing below that encoding scheme before giving the details of each index.

4.1.1 Elias-Fano Coding

An array of non-decreasing integer numbers $X = \{x_1 \leq x_2 \leq ... \leq x_n\}$, from a universe $U = \{1, ..., m\}$, can be stored explicitly in $n\lceil \lg m \rceil$ bits. However the same set can be stored using less space, and maintaining the constant time to retrieve any number, in $n \lg(m/n) + 2n + o(n)$ bits by using the Elias-Fano code [26, 27, 95].

The method splits the $\lceil \lg m \rceil$ bits for each x_i in two parts: h_i and r_i; such that the $\lceil \lg n \rceil$ most significant bits of x_i are h_i and the remaining $\lceil \lg m \rceil - \lceil \lg n \rceil$ ones are r_i. They encode the sequence $H = \{h_1, h_2 - h_1, h_3 - h_2, ..., h_n - h_{n-1}\}$ by unary coding; that is, a number k is represented with $k + 1$ bits, k copies of **0** followed by **1**. We build a bitvector H of length $n + 2^{\lg n} = 2n$, so this bitvector requires $2n + o(n)$ bits. Note that the i-th **1** in H corresponds to the last bit in the unary code of h_i, and the number of **0**s to that position is h_i itself. So with H we can retrieve any h_i value in constant time computing $h_i = rank_0(H, select_1(H, i)) = select_1(H, i) - i$.

The sequence $R = \{r_1, r_2, ..., r_n\}$, of the least significant bits of X, is stored explicitly. Then R requires $n(\lg m - \lg n) = n \lg(m/n)$ bits of storage[1].

For instance, if we apply Elias-Fano code to $X = \{7, 12, 17, 17, 25, 30\}$ we have $n = 6$, $\lceil \lg n \rceil = 3$, $m = 30$ and $\lceil \lg m \rceil = 5$. The binary codes for X are $X_{(2)} = \{00111, 01100, 10001, 10001, 11001, 11110\}_{(2)}$; so the sequence of header numbers h_i, $\langle h_1, ..., h_6 \rangle$, is $\langle 001, 011, 100, 100, 110, 111 \rangle_{(2)} = \langle 1, 3, 4, 4, 6, 7 \rangle_{(10)}$. Consequently $H = \{h_1, h_2 - h_1, ..., h_6 - h_5\} = \{1, 2, 1, 0, 2, 1\}_{(10)}$, which concatenating each unary code yields $H = \langle 0100101100101 \rangle$. The remaining bits are stored explicitly in $R = \{r_1, ..., r_6\} = \{11, 00, 01, 01, 01, 10\}_{(2)}$. Therefore we can determine x_5 by concatenating the bits of h_5 with r_5; where $h_5 = select_1(H, 5) - 5 = 11 - 5 = 6 = 110_{(2)}$ and $r_5 = 01_{(2)}$. Then $x_5 = 11001_{(2)} = 25_{(10)}$.

4.1.2 The Suffix Array of Grossi and Vitter

The compressed suffix array proposed by Grossi and Vitter [54, 55] represents in a hierarchical structure the suffix array $A[1..n]$ for a text $T[1..n]$. The method applies a recursive decomposition of A in t levels. In each level k, $0 \leq k < t - 1$, it receives an input array $A_k[1..n_k]$ (in the first level $A_0 = A$), and represents it in structures that require less size than its explicit storage. The even numbers of A_k are divided by two and stored in the array $A_{k+1}[1..n_k/2]$, which will be processed in the next level. For the odd numbers they showed that, given the characteristics of the suffix array, it is possible to build an increasing sequence

[1]To simplify the explanation, we write $\lg m - \lg n$ for $\lceil \lg m \rceil - \lceil \lg n \rceil$.

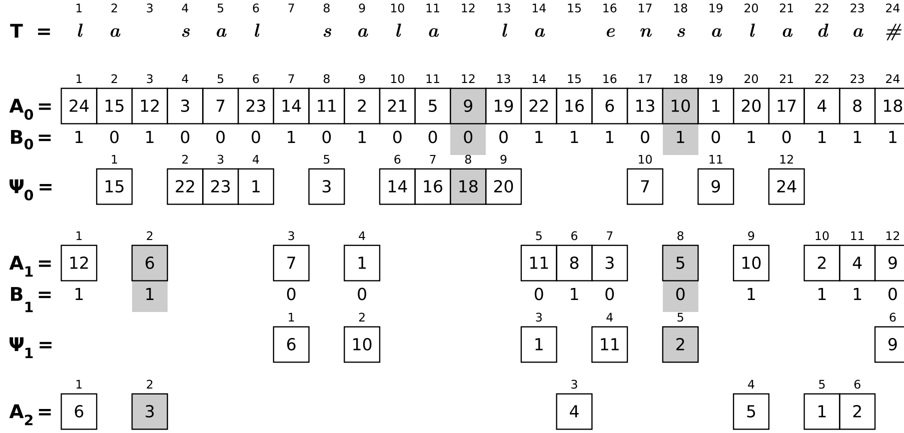

Figure 4.1: GVCSA with $t = 2$ levels of decomposition, where the suffix array $A[1..n]$, $n = 24$, corresponds to the text $T_{1..n}$ = *"la sal sala la ensalada$"*. We shadowed all the positions that we have to access in order to retrieve $A[9]$, according to Algorithm 3.

that represents them. They then compress these sequences in a structure that offers direct access to its values. The scheme finishes by storing explicitly the $n/2^t$ values in the last array A_t, which stores original numbers of A divided by 2^t.

A hierarchical structure. The general idea is summarized as follows. In each level of the decomposition, they divide by two the even values of $A_k[1..n_k]$, $n_k = n/2^k$, and store them in a new array $A_{k+1}[1..n_k/2]$ in the next level. For the other half of A_k, i.e., for each odd value $A_k[i]$, the index stores the position where its next suffix $A_k[i] + 1$ appears in A_k. The representation then starts in the first level with $A_0[1..n_0] = A[1..n]$, marking in a bitvector $B_0[1..n_0]$ all the positions where $A_0[i]$ is even. They divide these numbers by two and store them, forming a new permutation A_1 of $n/2$. If the value $A_0[i]$ is odd then we set $B_0[i] = 0$ and in the array ψ_0 they store the position where $A_0[i] + 1$ occurs. Any A_i is then passed on to the next level $i + 1$. In general, in order to represent $A_k[1..n/2^k]$, $0 \leq k < t - 1$, we let a bitvector $B_k[1..n/2^k]$ identify when $A_k[i]$ is an even value, and it stores $A_k[i]/2$ in $A_{k+1}[1..n/2^{k+1}]$. For odd values, we set $B_k[i] = 0$ and in the array $\psi_k[1..n/2^{k+1}]$ we save the position where the value $A_k[i] + 1$ is. This scheme is repeated for $t = \log\log_\sigma n$ levels, to finish with the last array A_t, saved explicitly. An example of this structure is given in Figure 4.1 for $t = 2$.

A key point of this process is the way followed to store the arrays $\psi_k[1..n_k/2]$. They showed that each ψ_k can be seen as a list that concatenates σ^{2^k} increasing lists. Therefore, they transform ψ_k into a unique increasing list L_k putting, before each $\psi_k[i]$, the bits required to form the value $j \cdot n_k$, where j is the list number where $\psi_k[i]$ appears in this concatenated sequence (i.e., with $2^k \log \sigma + \log n_k$ bits per item). They compress the sequence L_k with Elias-Fano codes [26, 27] —described in Section 4.1.1— offering constant time to retrieve any item.

Algorithm. In order to retrieve $A[i]$ from the representation, we need to call the function GVCSA-LOOKUP$(i, 0)$, detailed in Algorithm 3. As this structure can extract in $O(1)$ time per level, the index retrieves any $A[i]$ in $O(t) = O(\log\log_\sigma n)$ time. An example is given

Algorithm 3 Retrieving $A[i]$ from Grossi and Vitter's data structure.

```
function GVCSA-LOOKUP(i, k)
    if k = t then return A_t[i]
    end if
    if B_k[i] = 1 then return 2·GVCSA-LOOKUP(rank_1(B_k, i), k + 1)
    else return GVCSA-LOOKUP(ψ_k[rank_0(B_k, i)], k)−1
    end if
end function
```

in Figure 4.1, where according to this algorithm, to retrieve $A[12]$ we have to extract the following values:

$$\begin{aligned} A[12] = A_0[12] &= A_0[\psi_0[rank_0(B_0, 12]] - 1 = \\ &= A_0[\psi_0[8]] - 1 \\ &= A_0[18] - 1 \\ &= (2A_1[rank_1(B_0, 18)]) - 1 \\ &= (2A_1[8]) - 1 \\ &= (2(A_1[\psi_1(rank_0(B_1, 8))] - 1)) - 1 \\ &= (2(A_1[\psi_1(5)] - 1)) - 1 \\ &= (2(A_1[2] - 1)) - 1 \\ &= (2((2A_2[rank_1(B_1, 2)]) - 1)) - 1 \\ &= (2((2A_2[2]) - 1)) - 1 \\ &= (2((2 \cdot 3) - 1)) - 1 = 9 \end{aligned}$$

Required space. The explicit array $A_t[1..n_t]$, $n_t = n/2^t$, requires $\frac{n}{2^t}\log\frac{n}{2^t} = \frac{n\log n - nt}{2^t}$ bits. If we set $t = \log\log_\sigma n$, the size becomes $\frac{n\log n - nt}{2^{\log\log_\sigma n}} \leq \frac{n\log n}{\log_\sigma n} = n\log\sigma$ bits.
The length of the t bitvectors B_k is $n + \frac{n}{2} + \frac{n}{4} + .. + \frac{n}{2^{t-1}} \leq 2n$. The size then for these bitvectors, which includes support to rank/select queries in $O(1)$ time, is $2n + o(n)$ bits.
Each sequence ψ_k is encoded with Elias-Fano. As we showed, we create a unique increasing list L_k of length $n_k/2 = n/2^{k+1}$, such that each number in L_k requires $\log(\sigma^{2^k}) + \log n_k = 2^k\log\sigma + \log n_k$ bits. Then each ψ_k is encoded with $2n_k + \frac{n_k}{2}(2^k\log\sigma + \log n_k - \log\frac{n_k}{2}) = \frac{5n_k}{2} + n_k 2^{k-1}\log\sigma$ bits. Therefore all the ψ_k sequences, for the $t = \log\log_\sigma n$ levels, require the following amounts of bits:

$$\begin{aligned} \sum_{i=0}^{t-1}(\tfrac{5}{2}n_k + n_k 2^{k-1}\log\sigma) &= \tfrac{5n}{2}\sum_{i=0}^{t-1}\tfrac{1}{2^k} + t\tfrac{n}{2}\log\sigma \\ &< 5n + \tfrac{1}{2}n\log\log_\sigma n \cdot \log\sigma \end{aligned}$$

Considering all together, the space becomes:

$$n\log\sigma + 2n + o(n) + 5n + \tfrac{1}{2}n\log\log_\sigma n \cdot \log\sigma = (1 + \tfrac{1}{2}\log\log_\sigma n)n\log\sigma + O(n) \text{ bits.}$$

A generalized structure. Grossi and Vitter also generalized the process, not only classifying even and odd numbers. First, they choose the values of A_k that are multiple of a given

parameter l, to be stored (divided by l) in the array A_{k+1} of the level $k+1$. So, they mark these positions in the bitvector (i.e., $B_k[i] = 1$ iff $A_k[i]$ is a multiple of l). The remaining values, in positions where $B_k[i] = 0$, are stored in $\psi_k[1, (1-1/l)n_k]$.

Note that under this scheme we have to access ψ_k up to $l-1$ times before proceeding to the next level. So, the time becomes $O(l\dot{t})$. The space analysis is very similar to the previous structure, except that 2^k is replaced by l^k. The space for the array A_l is $\frac{n}{l^t}\log\frac{n}{l^t} < \frac{n\log\sigma}{l^t}$ bits, the bitvectors B_k also require $O(n)$ bits, and the sub-sequences ψ_k add $n(\log\sigma + \frac{1}{2^{k-1}}) + O(\frac{n}{2^k \log\log n})$ bits.

Then the total space is $(1-\frac{1}{l})n\log\sigma \cdot n\log_l \log_\sigma n + O(n\log\sigma)$. In particular, choosing $l = \log_\sigma^\varepsilon n$ for a constant $0 < \varepsilon < 1$, we have $t = 1/\varepsilon$, the space is $(\frac{1}{\varepsilon} + O(1))n\log\sigma$ bits and the time is $O((1/\varepsilon)\log^\varepsilon n)$.

4.1.3 The Suffix Array of Rao

Similar to the GVCSA, the compressed suffix array of Rao [100] consists of a recursive and hierarchical decomposition of the suffix array. However there are key subtle differences between them. The main idea is that, instead of iterating up to $l-1$ times in a level k before moving to level $k+1$, he stores vectors $d_k[1..n_k]$ with the value $d_k[i] = (l-1) - ((A_k[i]-1) \bmod l)$, that is, at which distance is the value $A_k[i]$ from the next multiple of l. The idea is then that $\psi_k[i]$ stores the position in A_k where the value $A_k[i] + d_k[i]$ is. Therefore we jump directly to the cell answering in one step, not $d_k[i]$ steps as the GVCSA. The final answer is thus $A_k[\psi_k[i]] - d_k[i]$. Unfortunately, this new ψ_k array does not enjoy the monotonicity properties seen before; these hold only within the subsequence associated to a single value of d_k. Thus the array is split into $l-1$ arrays ψ_k^δ, $1 \le \delta < l$, containing the values $\psi_k[i]$ such that $d_k[i] = \delta$. At level k, we also store $l-1$ bitvectors $V_k^\delta[1..n_k]$ where $V_k^\delta[i] = 1$ iff $d_k[i] = \delta$. Then the value $\psi_k[i]$ is found at $\psi_k^\delta[rank_1(V_k^\delta, \mathrm{i})]$ if $d_k[i] = \delta$. Figure 4.2 illustrates the hierarchical decomposition of Rao's CSA.

Now ψ_k^δ is formed by $\sigma^{\delta l^k}$ increasing lists with values in $[1, n_k]$. The renumbering scheme of Section 4.1.2 yields a single list of n_k/l values in $[1, \sigma^{\delta l^k} n_k]$, with $n_k = n/l^k$. Therefore each representation ψ_k^δ with Elias-Fano (Section 4.1.1) yields $(\delta/l)n\log\sigma + O((n_k/l)\log l)$ bits. Summing for $1 \le \delta < l$, at level k we have $O(l \cdot n\log\sigma + n_k\log l)$ bits for the lists. Vector d_k adds $O(n_k \log l)$ bits and vectors V_k^δ add $O(l \cdot n_k)$ bits, for a total of $O(l\cdot n\log\sigma + l\cdot n_k)$ bits at level k. Added over t levels and considering the final explicit array A_t we have $O(t\cdot l\cdot n\log\sigma + l\cdot n + (n/l^t)\log n)$ bits. Choosing $l = \log_\sigma^{1/t} n$ we get space $O(t\cdot\log_\sigma^{1/t} n)n\log\sigma$ bits, and $O(t)$ time to compute any $A[i]$. This gives a number of space/time tradeoffs, consider for example $t = 1/\varepsilon$ or $t = \log\log_\sigma n$.

Algorithm. In order to retrieve $A[i]$ form the representation, we call RaoCSA-LOOKUP$(i, 0)$ in Algorithm 4. The structure of Rao can extract in $O(1)$ time per level, so the index retrieves any $A[i]$ value in $O(t)$ time.

An example is given in the Figure 4.2, where to retrieve $A[18]$, according to this algorithm,

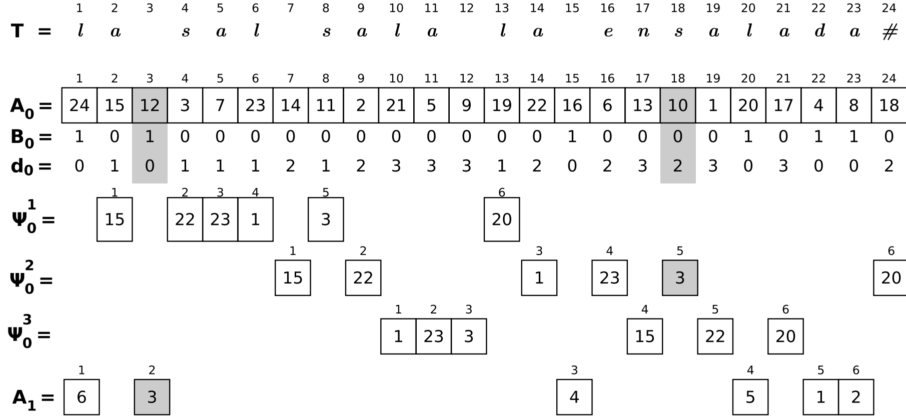

Figure 4.2: RaoCSA with $t = 1$ level of decomposition and for $l = 4$. The suffix array $A_0[1..n]$, $n = 24$, corresponds to the text $T_{1..n} =$ *"la sal sala la ensalada$"*. We shadowed all the positions that, according to Algorithm 4, we have to access in order to retrieve $A[18]$.

Text	$\log \sigma$	H_0	H_4	gzip	bzip2	PPMDi
dna	4.000	1.974	1.910	2.162	2.076	1.943
english	7.814	4.525	2.063	3.011	2.246	1.957
proteins	4.644	4.201	3.826	3.721	3.584	3.276
sources	7.845	5.465	1.852	1.790	1.493	1.016
xml	6.585	5.257	1.045	1.369	1.908	0.745

Table 4.1: Main characteristics for the texts considered in the experiments with the indexes. We show the entropy of order 0 and 4, and also the real compressibility for these texts, using the best-known compressors: gzip (option −9), bzip2 (option −9) and PPMDi (option −1 9).

we have to extract the following values:

$$\begin{aligned} A[18] = A_0[18] &= A_0[\psi_0^{\mathrm{d_0}[18]}[rank_{\mathrm{d_0}[18]}(\mathrm{d_0}, 18]] - \mathrm{d_0}[18] = \\ &= A_0[\psi_0^2[5]] - 2 = \\ &= A_0[3] - 2 \\ &= (4 \cdot A_1[rank_1(B_0, 3)]) - 2 \\ &= (4 \cdot A_1[2]) - 2 \\ &= (4 \cdot 3) - 2 = 10 \end{aligned}$$

4.1.4 Experimental Results

We use text collections obtained from the PizzaChili site[2]. This site offers a collection of texts of various types and sizes. We use the five types (`dna`, `english`, `proteins`, `sources`, and `xml`) for which 100MB files are available. Using larger datasets gives no additional clues on the performance. Table 4.1 summarizes some of their properties.

[2] `http://pizzachili.dcc.uchile.cl/texts.html`

Algorithm 4 Retrieving $A[i]$ from Rao's data structure.

```
function RAOCSA-LOOKUP(i, k)
    if k = t then return A_t[i]
    end if
    if B_k[i] = 1 then return l·RAOCSA-LOOKUP(rank_1(B_k, i), k + 1)
    else return RAOCSA-LOOKUP(ψ_k^{d_k[i]}[rank_{d_k[i]}(d_k, i)], k) − d_k[i]
    end if
end function
```

The experiments were run on an Intel Core2 Duo, running at 3.0 GHz, with 6MB cache and 8GB RAM. The operating system was Linux 64-bit with kernel 2.6.24-31-server, and the compiler was g++ version 4.2.4 with -O3 optimization and -m32 flag (as required by several packages tested). We include the text when we measure the space of the indices.

GVCSA

We implemented a verbatim variant of this data structure, where we used sparse bitvector implementations from the libcds library[3]. Note that one can access any position of any list of ψ_k in constant time by knowing the list number and offset. However, to know the list number we need to know the positions in A where the suffix beginning with any tuple of Σ^k starts. This requires $\sigma^k \lg n$ additional bits.

We implemented a second variant of this structure. Instead of using the σ^k contexts (many of which may actually be empty), we detect maximal runs of increasing numbers in ψ_k and take those as the lists. The beginning of the lists are marked in a sparse bitmap $S_k[1..n_k]$. Then, in order to retrieve $\psi_k[i]$ we compute $j = rank_1(S_k, i)$, to find that i belongs to the j-th list, and use the same numbering scheme as before: since $\psi_k[i]$ is represented as $L[i] = j \cdot n_k + \psi_k[i]$, we compute $L[i]$ from the representation using H and L, and then subtract $j \cdot n_k$. The space for S_k is just $O(n_k)$ bits, and thus the space and time analysis stays the same.

Figure 4.3 compares various (t, l) combinations, t levels marking multiples of l, for both variants of the structure (the basic one and the one using runs), on all the texts; the results are similar It is clear that the best tradeoffs are obtained when using $t = 1$, that is, not using a recursive structure but just one level of sampling, and then storing the samples in plain form. Space can be reduced by using a larger t (i.e., more levels of recursion), but it is always faster to reduce the same space by using a larger l value (i.e., a sparser sampling at the first level). Only on `dna` there are some dominating points using $t = 2$. It is also clear that our variant using runs is much better when using $t > 1$ levels (indeed, the basic variant is almost never affordable for $t > 2$), but there is almost no difference between variants on $t = 1$.

Figure 4.6 (left) compares the two variants, for all the texts, using the dominating points

[3]`https://sourceforge.net/projects/libcds/files/`

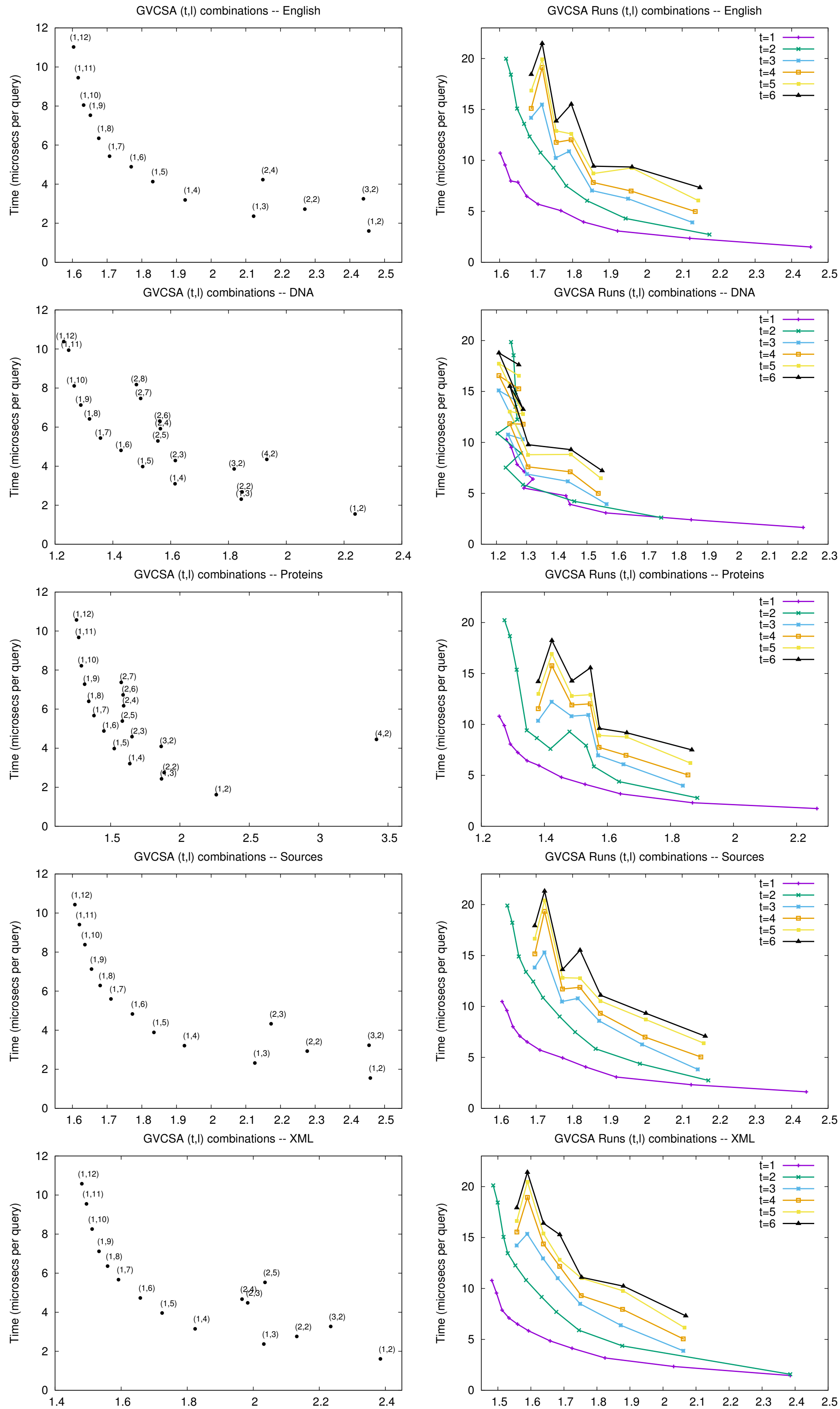

Figure 4.3: Space/time tradeoffs for accessing one cell using various options for (t, l) for GVCSA. On the X-axis we show the value Index size/Text size. On the left we show the basic scheme, and on the right our improvement using runs, showing one curve per t value; the results with l value from 2 onwards are shown right to left in the curve.

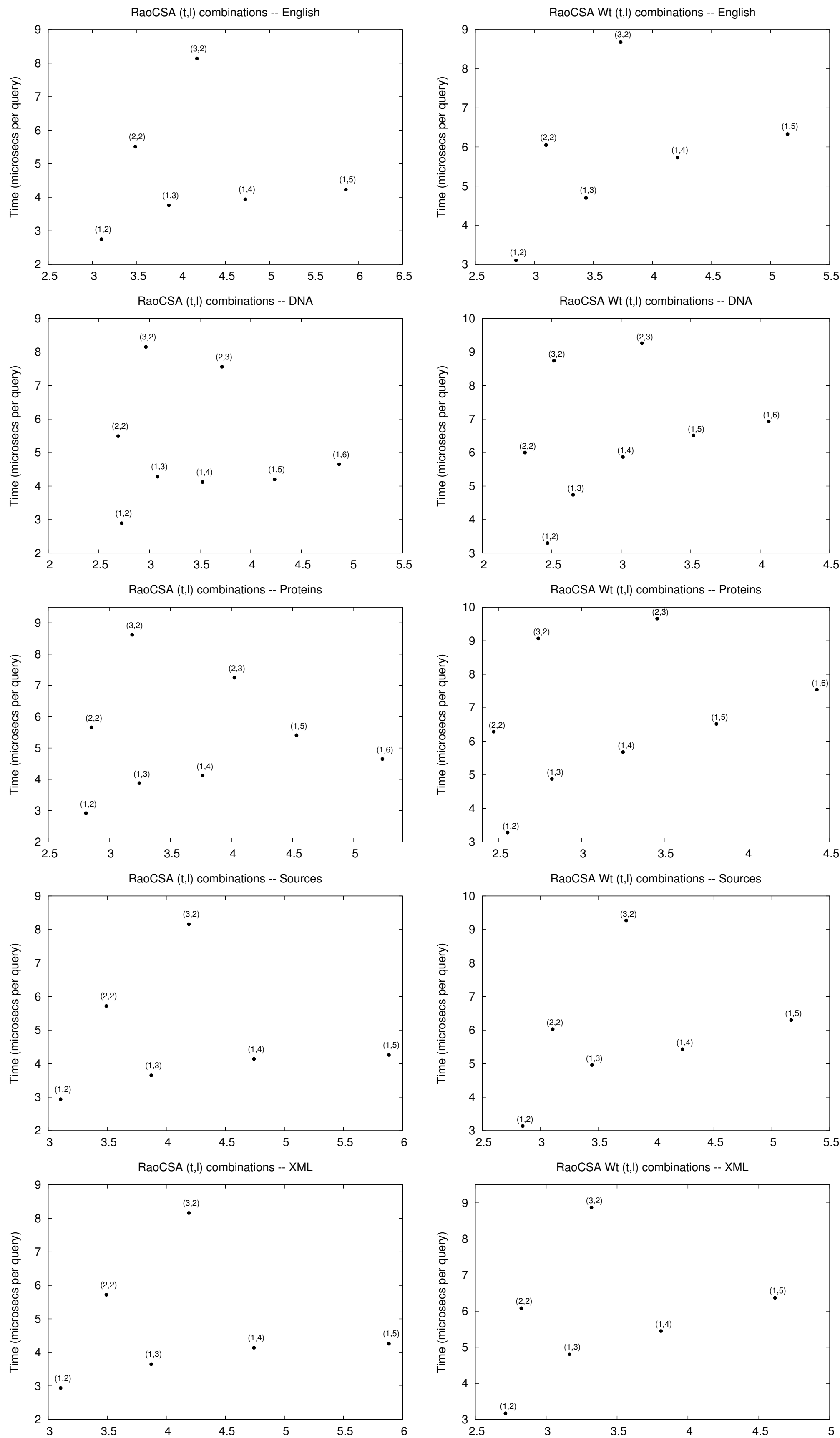

Figure 4.4: Various options for (t, l) for RaoCSA. On the left we show the basic scheme, on the right our improvement using wavelet trees.

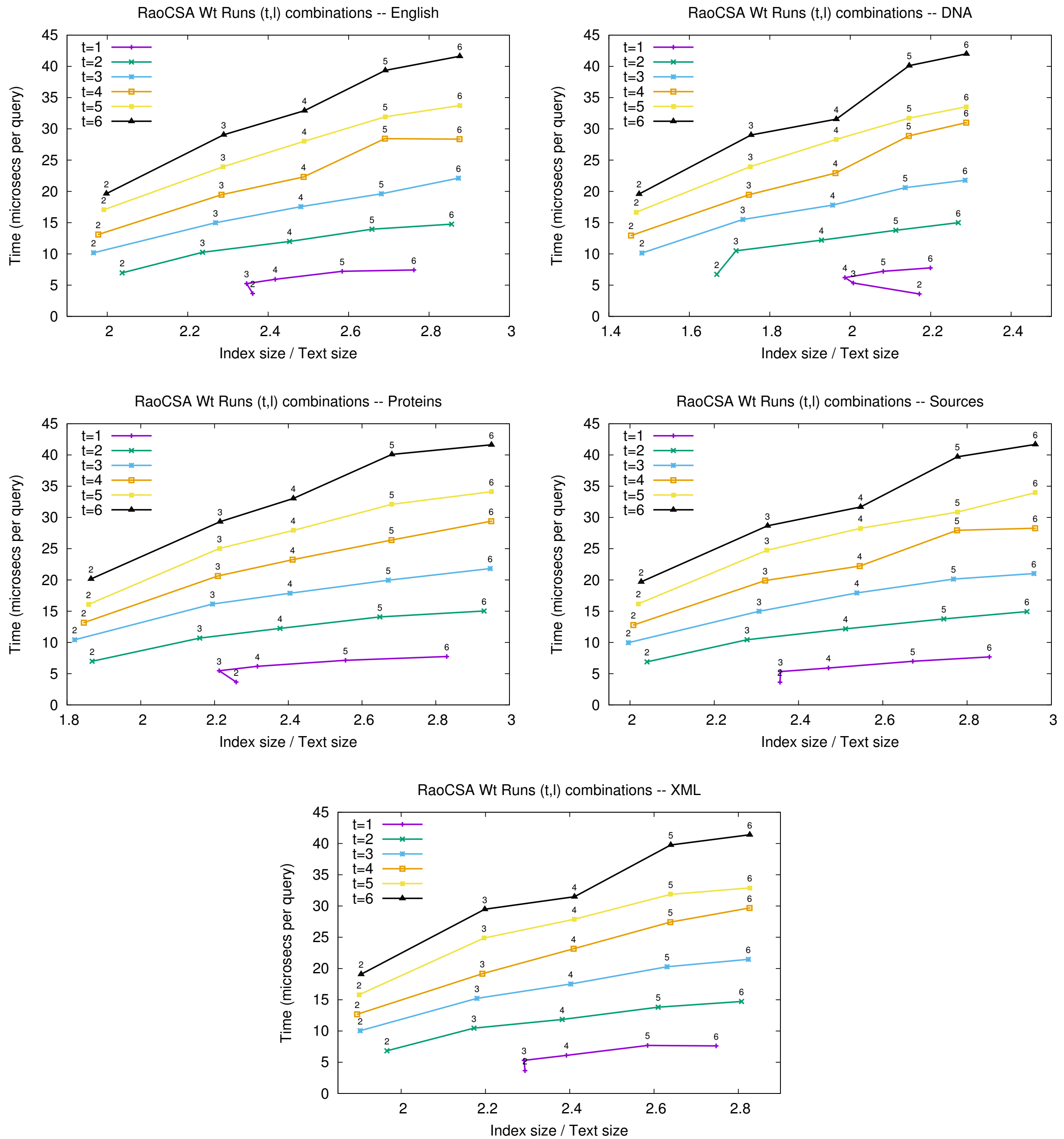

Figure 4.5: Various options for (t, l) for our improvement in RaoCSA using runs and wavelet trees. We show one curve per t value; the l values are marked in the curves.

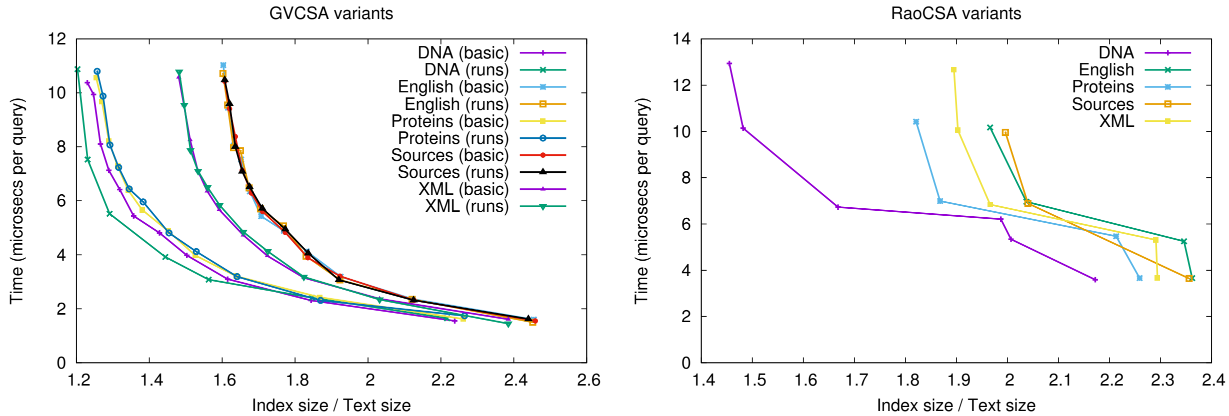

Figure 4.6: Time-space tradeoffs to access one cell. On the left, basic GVCSA versus the version with runs, for all the texts. On the right, the best variants of RaoCSA.

of each scheme (mostly corresponding to $t = 1$, as mentioned). It can be seen that, when using only one level, the differences are minimal. We will use the version with runs as the representative of GVCSA when we compare it with other CSAs in Figure 4.8.

RaoCSA

In practice the V_k^δ bitvectors may occupy considerable space. We implement a second variant where we completely remove them, and instead represent vector d_k as a wavelet tree [52] (see Section 2.5.1). This occupies $O(n_k \log l)$ bits of space instead of $O(ln_k)$, and supports $rank_\delta(\mathrm{d}_k, i) = rank_1(V_k^\delta, i)$ within $O(\log l)$ time instead of $O(1)$. In theory, the asymptotic space does not change and the access time grows to $O(t \log l) = O(\log \log \sigma n)$, which is a mild growth. In practice, this is advantageous, as we see soon.

Figure 4.4 shows the space/time tradeoffs obtained to access a random cell using various (t, l) combinations for this index. On the left we show the basic scheme, where it always holds that the combination $(t = 1, l = 2)$ dominates all the others (note this combination corresponds to the GVCSA). On the right we show the scheme where the bitvectors are replaced by wavelet trees. In this case the combination $(t = 2, l = 2)$ offers better space sometimes. Figure 4.5 shows the variant replacing the strict numbering by runs.

Figure 4.6 (right) shows the results obtained choosing the dominating (t, l) combinations of this variant, for all the texts. This time many more (t, l) combinations are feasible, and various alternatives with $l = 2$ (and even $l = 3$ or $l = 4$) offer relevant space/time tradeoffs. It is also clear that the variant with wavelet trees and runs is always the best. Those will be used to represent RaoCSA in the main experiments with other CSAs in Figure 4.8.

Comparing with Other CSAs

In this section we compare these two CSAs against various alternative suffix array representations. Various of the compressed suffix arrays we wish to compare with [73, 104, 34, 76] already have competitive implementations, which we have used.

LCSA The LCSA from *PizzaChili*[4], considering variants LCSA = RPSP and LCSAΨ = $\mathsf{RP\Psi SP}$, with the parameters set as indicated in the paper [51] ($s = 8$, $\delta = 3/4$ and $\gamma = \log n$), and using samplings $l = 64$ and $d = 256$.

MakCSA The Compact Suffix Array of Mäkinen [73], implemented by himself[5]. The code can only search for patterns and list their positions. In order to extract arbitrary ranges of A we added a bitvector [99] of length n, marking the starting positions of the blocks, so that we could convert positions in A to positions in the compacted array.

SadCSA The Compressed Suffix Array of Sadakane [104], implemented by himself (the implementation is available at *PizzaChili*[6]). This has two parameters: s_Ψ, the sampling step to access the compressed Ψ array, which is left at $s_\Psi = 128$, where it performs best, and s_A, the sampling step to store samples of A, which is used as the space/time tradeoff parameter. We consider values $s_A = \{4, 8, 16, 32, 64, 128, 256\}$. We used routines tailored to extract various consecutive cells, taking advantage of runs of consecutive Ψ values, that were already in Sadakane's code.

RLCSA The Run-Length Compressed Suffix Array [76], implemented by Jouni Sirén (the implementation is available at *PizzaChili*[7]). The RLCSA is a variant of SadCSA specialized on handling repetitive texts. It has the same parameters s_Ψ (which we use at its default value 32) and s_A, which we use as the space/time tradeoff parameter, considering values $s_A = \{4, 8, 16, 32, 64, 128, 256\}$. RLCSA also has routines tailored to extract various consecutive cells.

FMindex The FM-index [34] using a recent and efficient variant [62] implemented by themselves[8]. We used the suffix array sampling parameters $s_A = \{4, 8, 16, 32, 64, 128, 256\}$, and the text sampling parameter set to infinity. We only show the variant using plain bitmaps, as the time/space obtained with compressed bitmaps were almost identical in this scenario.

We remind that SadCSA, RLCSA and FMindex are self-indexes, but we are not interested in this feature for this experiment. We only evaluate their ability to retrieve a cell.

Figure 4.7 compares construction space and time for all the indexes. We have not considered the space and time to build the suffix array, as this is orthogonal to the index construction problem and must be done for all the indexes. It can be seen that all construction spaces are relatively close, except that of LCSA, which in bad cases can require as much as 40 bytes per

[4] At `http://pizzachili.dcc.uchile.cl/indexes/Locally_Compressed_Suffix_Array/`
[5] Downloaded from `http://www.cs.helsinki.fi/u/vmakinen/software/csa.zip`
[6] At `http://pizzachili.dcc.uchile.cl/indexes/Compressed_Suffix_Array`
[7] At `http://pizzachili.dcc.uchile.cl/indexes/RLCSA/`
[8] Thanks to Simon Puglisi for handing us the code.

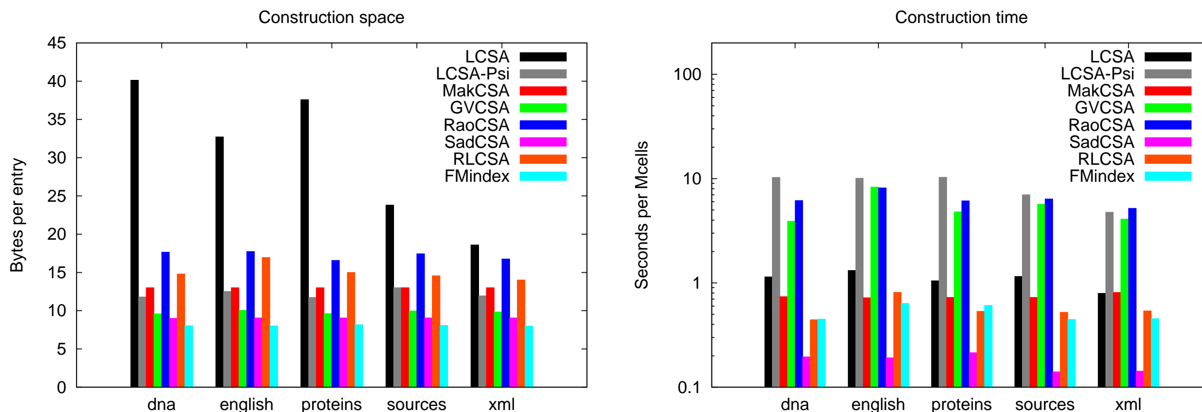

Figure 4.7: Construction time and space for the different indexes on each text.

entry. Those requiring the least space, around 7–9 bytes per entry, are GVCSA, SadCSA, and FMindex. The others are usually within 12 bytes per entry, except RaoCSA and RLCSA, which may require up to 16–17 bytes per entry.

With respect to construction time, SadCSA again excells, requiring less than 0.2 seconds per million cells, whereas the next faster indexes (FMindex, RLCSA, MakCSA, and LCSA, more or less in that order) build at a rate of around 1 second per million cells. The other indexes build ten times slower.

LCSA builds fast (at about 1 second per million cells) but it may require too much extra space (up to 40 times the text size). Variant LCSAΨ, although slower to build (about 10 seconds per million cells, which is still affordable even for large texts), requires reasonable space (near 12 times the text size, not far from the state of the art).

Figure 4.8 shows the space/time tradeoffs, for all the indexes on all the texts, to access a random cell. The space is shown as the index size in bytes divided by n, that is, in bytes per cell.

It can be seen that SadCSA and FMindex are the clear winners in all cases, being faster and smaller than all the others. The size of these indexes is sensitive to the high-order entropy of the texts, whereas GVCSA and RaoCSA are more dependent on the alphabet size. Among the two, GVCSA is always better than RaoCSA. RLCSA, instead, is sensitive to the repetitiveness of the text, performing worst on `dna` and best on `xml`. Finally, in both MakCSA and the variants of LCSA the space depends more on the relation between the high and the zero order entropies of the texts, H_k/H_0. Thus, they perform particularly bad on `dna` and `proteins`, much better on `english` and `sources`, and particularly well on `xml`. Yet, they are still slower than SadCSA and FMindex.

The relation between MakCSA and LCSA variants is mixed. In cases like `dna` and `english`, the former performs better in time and space. On `proteins` and `sources`, LCSA competes in space, but the time is either equal or dominated by that of MakCSA. Finally, on `xml`, where both perform best in space, the LCSA variants use less space than MakCSA, and dominate it in time too.

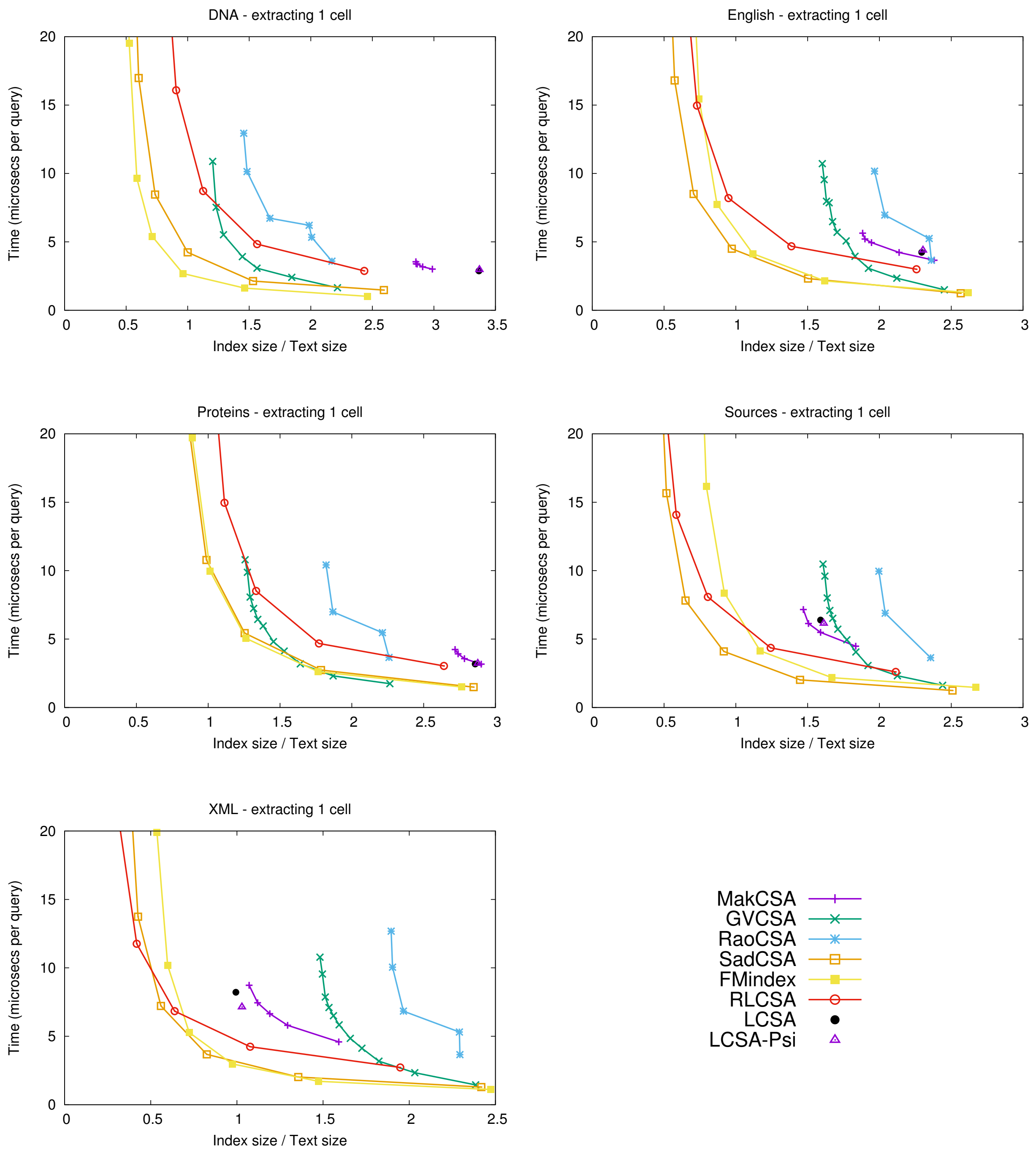

Figure 4.8: Time/space tradeoffs to access one random cell for the different indexes on each text.

4.2 Hybrid Indexing on Repetitive Datasets

Sometimes the text documents that will be indexed share many identical segments. In that case we wish to store them compressed, such that we can later support searches for patterns faster than if we only use a classical text index for general sequences. That technique is called *Text Indexing on Repetitive Text*.

4.2.1 Hybrid Indexing

Given an upper bound on pattern lengths M, we preprocess the text $T[1..n]$ with the LZ77 parser (see Section 2.7.1) to obtain a filtered text $T'[1..n']$, $n' \leq n$, for which we store a conventional index $I_M(T')$. Later, given a query, we find all matches in the filtered text (i.e, in a reduced text) using $I_M(T')$, and then use their positions and the structure of the LZ77 parser to find all matches in the original text. In that process we report secondary matches using the data structure of Kärkkäinen and Ukkonen [63] built on the LZ77-structure (Section 2.7.1). Our structure locates all the *occ* occurrences of a pattern $p[1..m]$ in the text in $O(t_{\mathsf{tsearch}}(m)+\mathsf{occ})$ time, where $t_{\mathsf{tsearch}}(m)$ is the time to search the pattern using the index $I_M(T')$ on T', and occ is the number of times that p appears in T.

Our original work [29] is focused on the *Approximate Pattern Matching Problem*, where the index finds matches between strings that are within a maximal allowed *edit distance* L. However, as in this thesis we are not interested in approximate matching, we simplified the structure and consider only the parameter M.

Finding Primary Matches

Let $T'_{(M)}$ be the text containing the characters of T within distance M to their nearest LZ77-phrase boundaries; characters not adjacent in T are separated in $T'_{(M)}$ by a special character # not in the normal alphabet. We then replace a segment of characters out of a distance M by the symbol #. For example, if T is the text in the Figure 2.6, then $T'_{(3)}$ is

tu gusto no gusta del gu#to que gu#ta mi gusto, tu # de mi#sto

Note how the parsed text $T'_{(3)}$ was reduced from $n = 77$ to $n' = 62$ characters. This is more evident in bigger repetitive collections such as a human genome, for example.

In this first part we only need a structure to find matches that cross at least one phrase boundary. As explained, we locate these primary occurrences with a conventional full text index for the filtered text $I_M(T')$, for instance using an FM-Index [33, 35, 36]. However, $I_M(T')$ alone is not sufficient to check if a string is a primary or a secondary match; we also need to locate the positions of these occurrences in T to validate them as primary. For this, we will use additional structures to: (i) map from any position i' in T' to its respective position i in the original text; and (ii) know if the match in T' spans more than one phrase or it includes the first occurrence of a symbol.

Let L be the sorted list containing the positions of the first character of each phrase in the parse of T, and let $L_{(M)}$ be the sorted lists containing the positions of the corresponding characters in $T'_{(M)}$. We store L and $L_{(M)}$. If $T[i]$ is the first occurrence of a distinct character in T and $T_{(M)}[j]$ is the corresponding character in $T_{(M)}$, then we mark j in $L_{(M)}$.

For our example, L is:

$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 17, 18, 19, 20, 21, 22, 29, 30, 31, 32, 39, 40, 41, 47, 48, 49, 69]$

and $L_{(3)}$ (with asterisks indicating marked numbers) is:

$[1^*, 2^*, 3^*, 4^*, 5, 6^*, 7, 8^*, 9, 10^*, 11, 12, 17^*, 18, 19^*, 20^*, 21^*, 22, 29^*, 30, 31, 32, 39^*, 40^*, 41, 47^*, 48, 49, 56]$

The first part of the algorithm consists in finding all the primary occurrences of the query pattern $p[1..m]$, $m \leq M$, with the following process, which uses the lists L and $L_{(M)}$ and the FM-Index $I_M(T')$.

Given a substring $T'_{(M)}[i'..j']$ that does not contain any symbol #, we can use the lists L and $L_{(M)}$ to map from $T'_{(M)}[i'..j']$ to its correct match $T[i..j]$; and also to check if this is a primary occurrence. To do this, we use binary search to find i's successor $L_{(M)}[s]$. After that, we check:

- if $i' < L_{(M)}[s] \leq j'$ then $T'_{(M)}[i'..j']$ crosses a phrase boundary;
- if $j' < L_{(M)}[s]$ then $T'_{(M)}[i'..j']$ neither contains the first occurrence of a distinct character nor crosses a phrase boundary;
- if $i' = L_{(M)}[s]$ then $T'_{(M)}[i'..j']$ contains the first occurrence of a distinct character or crosses a phrase boundary if and only if $L_{(M)}[s]$ is marked or $L_{(M)}[s+1] \leq j'$.

If the substring $T'_{(M)}[i'..j']$ contains the first occurrence of a distinct character or crosses a phrase boundary, we map its respective string $T[i..j]$ computing $i = L[s] + i' - L_{(M)}[s]$ and $j = i + j' - i' + 1$.

Finding Secondary Matches

We use Kärkkäinen and Ukkonen's method [63] to find secondary matches. They observed that, given the characteristics of the LZ77 parsing, any secondary match is completely contained in a LZ77 phrase. Therefore, any phrase that contains a secondary match has also a match inside its own source substring, and this earlier match can be primary or secondary. That observation leads to the next conclusion: any secondary occurrence always comes from an earlier primary match that was copied one or more times through the text. We then find all the secondary occurrences from the primary ones using the LZ77 structure as follows.

For each primary match $T[\ell..r]$, we find each phrase $T[i..j]$ whose source $T[i'..i' + j - i]$ includes $T[\ell..r]$ — i.e., such that $i' \leq \ell \leq r \leq i' + j - i$. This phrase contains the secondary occurrence $T[\ell'..r'] = T[\ell..r]$, where $\ell' = i + \ell - i'$ and $r' = i - i' + r$ (see Figure 4.9). We

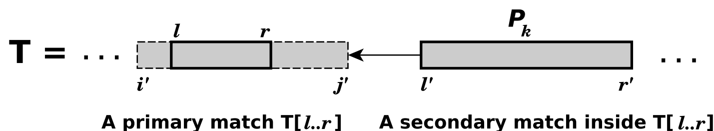

Figure 4.9: The Basic scheme to find secondary matches. A primary occurrence $T[l..r]$ is also found inside $T[l'..r']$.

record $T[\ell'..r']$ as a secondary occurrence and recurse on it to find all the secondary matches that include $T[\ell'..r']$ as part of its source. To do this, we need a representation that enables us to find all the sources that completely cover a specified text segment. That can be see as a grid $G[1..z']$, $z' = z - \sigma$ (σ is the alphabet size), where each of its z' points is a representation of a source $T[x..y]$ that originates the phrase r in the position $L[r]$. Then G is composed of triplets (x, y, r) and we partially store the x and y-coordinates in the arrays $X[1..z']$ and $Y[1..z']$ respectively. For our example the grid $G[1..z']$, $z' = z - \sigma = 29 - 15 = 14$, is:

$$[(2,2,5);(1,1,7);(3,3,9);(8,8,11);(3,7,12);(3,3,14);(3,9,18);(5,5,20);(16,16,21);$$
$$(12,18,22);(3,8,25);(3,3,27);(1,20,28);(38,46,29)]$$

We report secondary occurrences by processing all the primary matches found, searching G for sources that cover each text segment corresponding to any primary match. We must also process each located secondary match in the same way, to look for more secondary matches.

4.2.2 Implementation

The structure to find primary occurrences is composed by a conventional index $I_M(T')$ on the reduced text $T'_{(M)}[1..n']$, the two lists L and $L_{(M)}$, and a list L_σ to find the σ marked positions of the first occurrence of each distinct symbol in T. Then we don not mark these characters with the list L and use a separate list L_σ to do this. We do not restrict the index $I_M(T')$ to a particular family or approach. However, in this implementation we used an FMI [33, 35, 36].

We store L and $L_{(M)}$ using gap coding — i.e., storing the differences between consecutive values — with every gth value stored in plain form, where g is a parameter. We write the differences as $\lceil \log d \rceil$-bit integers, where d is the largest difference in the list, and we write the plain values as $\lceil \log n \rceil$-bit integers. To speed up binary search in $L_{M,K}$, we also sample every bth value in it, where b is another parameter (typically a multiple of g).

Instead of marking values in $L_{(M)}$, we store an array containing the position in $L_{(M)}$ of the first occurrence of each distinct character, in order of appearance. We note, however, that this array is only necessary if there may be matches of length 1.

To find secondary matches, we build a structure to search for points in a given range $[x..y]$ on the grid $G[1..z']$, similar than searches with a two-sided range reporting structure — it was described in Section 4.2.1. We sort the grid by x-coordinate and store separate

Algorithm 5 Report secondaty matches from $T[l..r]$ by RMQs

function SECONDARYREPORT(l, r)

1.- Find, by a binary search, the predecessor $X[k]$ of l in the array X.

2.- Use recursively RMQs to find all the values in $Y[1..k]$ that are at least r, using the Range Maximum structure on the array Y.

3.- For each point (i', j') we find with $i' \leq l \leq r \leq j'$, compute the coordinates (l', r') of the phrase whose source is $T[i'..j']$ as described in Section 4.2.1 (illustrated in Figure 4.9).

4.- Append the pair (l', r') to *List* and recursively call SECONDARYREPORT(l', r').

end function

structures for each coordinate. First, we store the x-coordinates in the array $X[1..z']$ with gap encoding in the same way as the lists L and $L_{(M)}$. Second, we do not store the y-coordinates themselves, but instead build a structure to answer range maximum queries. Fischer and Heun [40] gave an optimal query time structure to find the positions of the smallest value in any array interval $A[l..r]$, where the items of the static array A can be sorted (see Section 2.5.4). We trivially adapt it to find range maximum values on the y-coordinates stored in $Y[1..z]$; after that we discard Y. We report the primary matches found (occ_{pri}) and put all of them in a linked list *List* that will be used in the next step of the process. Then, we report secondary occurrences by applying recursively a process that emulate a 2-sided range reporting by RMQs. We summarize this process in the following steps:

1. Using $I_M(T')$, L, $L_{(M)}$ and L_σ, find the occ_{pri} primary occurrences and store them in the list *List*.
2. For each primary match $T_{l..r}$ in *List*, call the function SecondaryReport(l, r), which is described in Algorithm 5.

When the process finishes, the list *List* contains the endpoints of all primary matches followed by the endpoints of all secondary matches. The described process is very similar to the method followed by Kreft and Navarro [69] to report secondary matches.

4.3 Experiments

In our experiments, we compared a Hybrid-Index based on an FM-Index for the filtered text, to an FM-Index and an LZ77-Index (described in Section 2.7.1). We always used the same implementation for the FM-Index[9]. We set the parameter for the suffix array sampling in 32 and 256, and fixed the sampling for the inverse-SA in 1024 cells. The implementation used to test the LZ77-Index is given by Kreft and Navarro[10] [69], with default parameters. We performed our experiments on an Intel Xeon with with 96 GB RAM and 8 processors at 2.4 GHz with 12 MB cache, running Linux 2.6.32-46-server. We compiled both indexes with `g++` using full optimization.

[9]https://github.com/simongog/sdsl-lite

[10]http://pizzachili.dcc.uchile.cl/indexes/LZ77-index

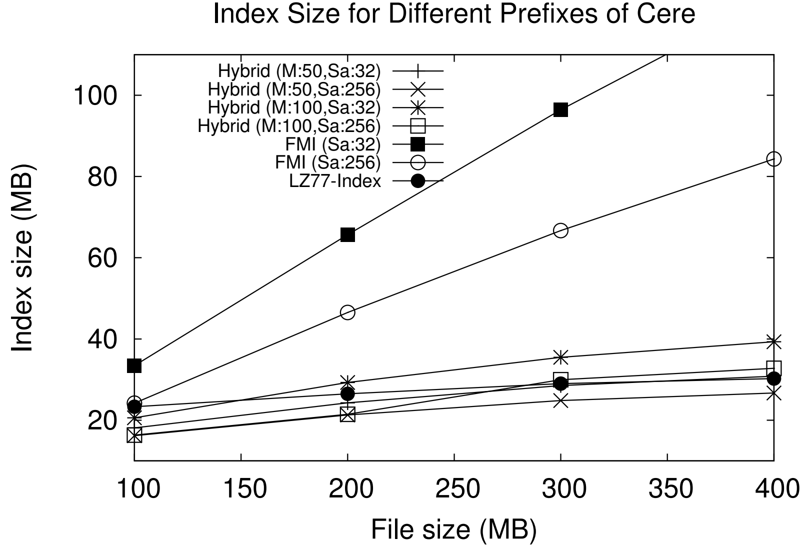

Figure 4.10: Index sizes for prefixes of cere of 100, 200, 300 and 400 MB. We test the LZ77 Index, two variants of the FMI with SA-sampling $Sa = \{32, 256\}$, and four variants of the Hybrid-Index with SA-FMI-sampling $Sa = \{32, 256\}$, and supporting queries for patterns of lengths $M = \{50, 100\}$ characters.

Text	file	7zip	LZ77	FMI(32)	FMI(256)	Hyb(50,32)	Hyb(50,256)	Hyb(100,32)	Hyb(100,256)
cere	440	5.0	31.06	134.32	90.71	32.28	27.85	41.61	34.56
einstein	445	0.3	1.66	73.50	29.30	1.44	1.31	1.71	1.51
fib41	256	0.5	0.04	31.66	7.20	0.01	0.01	0.01	0.01
kernel	246	2.0	15.42	65.53	42.99	11.73	10.79	12.39	11.31

Table 4.2: Sizes in MB of the uncompressed files, the files compressed with 7zip and the three indexes: the LZ77-Index of Kreft and Navarro (with default values), the FM-Indexes with SA-FMI-sampling in 32 and 256, and the hybrid indexes with maximum patterns lengths M in 50 and 100, with SA-FMI-sampling in 32 and 256 in the internal FM-Index for the filtered text. Between parentheses are the parameter values for the FMIs and the hybrid indexes.

We used benchmark datasets from the repetitive corpus of the Pizza&Chili website[11]. Specifically, we used the following files:

- cere — 37 *Saccharomyces cerevisiae* genomes from the Saccharomyces Genome Resequencing Project;
- einstein — versions of the English Wikipedia page on Albert Einstein up to November 10th, 2006;
- fib41 — the 41st Fibonacci word F_{41}, where $F_1 = 0$, $F_1 = 1$, $F_\mathrm{i} = F_{\mathrm{i}-1}F_{\mathrm{i}-2}$;
- kernel — 36 versions of the Linux 1.0.x and 1.1.x kernel.

We set $M = \{50, 100\}$, as that seemed a reasonable value for many applications. Based on preliminary tests, we set the sampling parameters g and b for our Hybrid-Index to 32 and 512, respectively. Notice these parameters have no effect on the other indexes.

[11]http://pizzachili.dcc.uchile.cl/repcorpus.html

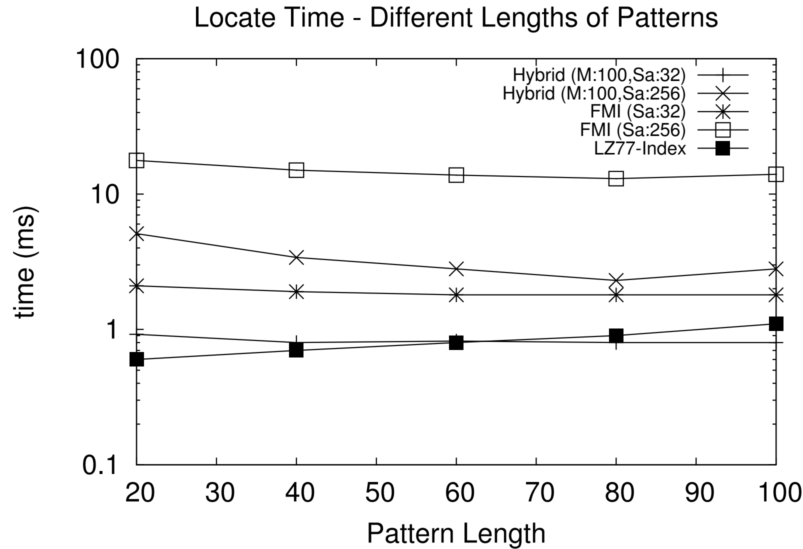

Figure 4.11: Average query times for the different indexes to locate occurrences with patterns of different lengths. The two variants of the FM-Index and the Hybrid-Index have SA-sampling $Sa = \{32, 256\}$. The Hybrid-Index can support queries for maximum length $M = 100$.

Table 4.2 shows the sizes of the uncompressed files, the files compressed with 7zip [12] (which does not support pattern matching), the FM-indexes, and the hybrid indexes.

The first experiment estimates how well the Hybrid and LZ77 indexing take advantage of the repetitive structure, relative to FM-indexing. We truncated cere at 100, 200, 300 and 400 MB, and then built the indexes for those prefixes. Figure 4.10 shows the sizes of those indexes. As expected, the space used by the indexes designed for repetitive text is always less than the space required by the FMI, and does not grow linearly with the file size.

For pattern lengths 20, 40, 60, 80 and 100, we randomly chose 5000 substrings of those lengths from cere and searched for them with the indexes (we validate that the patterns contain at least 2 different characters). Figure 4.11 shows the average query times, using a logarithmic scale. It is not surprising that the Hybrid-Index and the LZ77-Index perform well here: while the FM-index finds all matches with its locate functionality, the Hybrid and LZ77 indexes find secondary matches with a recursive algorithm, which is relatively fast; since cere consists of 37 genomes from individuals of the same species, most matches are secondary.

When indexing repetitive text and searching for small patterns ($M \leq 50$), the Hybrid-Index slightly outperforms the LZ77-Index. Figure 4.12 indicates that the Hybrid-Index, with SA-FMI-sampling $Sa = 32$, is about as fast as the LZ77 Index, while using less space to find patterns of length $M \leq 40$.

[12]http://www.7zip.org

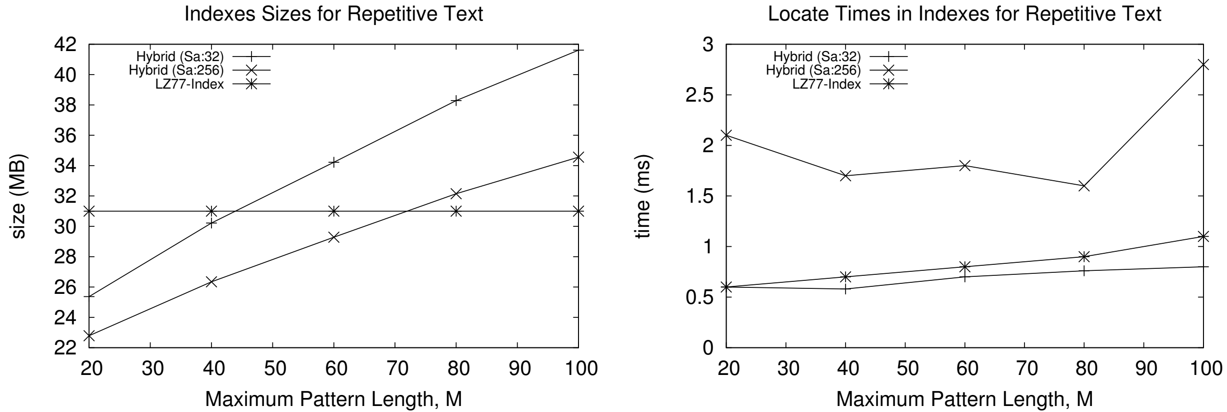

Figure 4.12: Index sizes and locate query time for the Hybrid-Index against the LZ77-Index. We show two variants of the Hybrid-Index, with SA-FMI-sampling $Sa = \{32, 256\}$, that support queries for patterns of maximum length M characters.

4.4 Conclusions

In Section 4.1 we have detailed the implementations and practical optimization of two compressed structures to represent the suffix array. These are the first practical results for those theoretical proposals, which have shown that current implementations, of more popular indexes, indeed perform better.

The best results of the two implementations, GVCSA and RaoCSA, always occur when we use only one level of decomposition in those hierarchical structures (or at most two levels in some cases). This corresponds to the simpler SadaCSA index, which has a better performance. Therefore, our main contribution here is that we confirm that those theoretical proposals do not yield better results that the best currently implemented indices.

Section 4.2 introduces a simple technique, called hybrid indexing, for reducing the size of conventional indexes on highly repetitive texts. In our experiments, this technique is able to match the search speed of the LZ77-Index within less space, provided that maximum pattern length is limited to at most 40 at indexing time.

Chapter 5

Improved Range Minimum Queries

Section 2.5.4 defined the RMQ problem and also described the optimal time solution of Fischer and Heun [40] to answer RMQs without accessing the input array, with a structure that requires $2n+o(n)$ bits. This chapter gives a complete overview of the state of the art with respect to RMQ solutions, and then describes our faster and smaller implementation with a simpler formula to solve RMQs. Our work was published in the 26th Data Compression Conference [32], where it won the Capocelli Prize (best student paper award). An extended version of this work was invited to a special issue of the Journal of Discrete Algorithms.

The RMQ problem is a fundamental one and has a long history, intimately related to another key problem: the LCA (lowest common ancestor) problem on general ordinal trees is, given nodes u and v, return $\mathsf{lca}(u, v)$, the lowest node that is an ancestor of both u and v. Gabow et al. [43] showed that RMQs can be reduced to computing LCAs on a particular tree, called the *Cartesian tree* 2.5.3 of $A[1, n]$. Later, Berkman and Vishkin [17] showed that the LCA problem on any tree can be reduced to an RMQ problem, on an array derived from the tree. In this array, consecutive entries differ by ± 1. Bender and Farach [15] then gave a solution for this so-called ± 1-RMQ problem in constant time and linear space (i.e., $O(n)$ words). Sadakane [105] improved the space of that solution, showing that LCAs on a tree of n nodes can be handled in constant time using $2n + o(n)$ bits (including the tree representation [83]). Finally, Fischer and Heun [40] showed that the Cartesian tree can be represented using $2n + o(n)$ bits so that RMQs on A can be transformed into LCA queries on the succinct tree, and this lead to an RMQ solution that also uses $2n + o(n)$ bits and *does not need to access A at query time.*

Fischer and Heun's solution has become a fundamental building block for many succinct data structures, for example for ordinal trees [105, 61, 93], suffix trees [105, 41], document retrieval [106, 67], two-dimensional grids [90], Lempel-Ziv parsing [21], etc.

Their RMQ computation [40] uses three kinds of operations: several $\mathsf{rank}/\mathsf{select}$s on bitvectors [60, 22], one ± 1-RMQ [15], and one open on parentheses [83]. Although all can be implemented in constant time, in practice the last two operations are significantly slower than $\mathsf{rank}/\mathsf{select}$ [1]. In particular, open is needed just to cover a border case where one node is an ancestor of the other in the Cartesian tree. Grossi and Ottaviano [53] replaced open by

further rank/selects in this case, thus improving the time significantly.

Their formula [40, 53] represents the Cartesian tree using DFUDS [16]. We show that, if we use instead the BP representation for the tree [83], the RMQ formula can be considerably simplified because the border case does not need special treatment. The result is the fastest and most compact RMQ implementation.

5.1 State of the Art

Gabow et al. [43] showed that RMQs can be reduced to computing LCAs on a particular tree, called the *Cartesian tree* 2.5.3 of $A[1, n]$. This is a binary tree whose root is the position p of a minimum in $A[1, n]$ (the leftmost/rightmost one if we want that RMQs return the leftmost/rightmost minimum). Then its left and right children are the Cartesian trees of $A[1, p-1]$ and $A[p+1, n]$, respectively. Any cell $A[p]$ is thus represented by the Cartesian tree node with inorder position p, and it holds

$$\mathsf{rmq}_A(i, j) \;=\; \mathsf{inorder}(\mathsf{lca}(\mathsf{innode}(i), \mathsf{innode}(j))), \tag{5.1}$$

where inorder and innode map from nodes to their inorder values and vice versa. Figure 5.1 shows an example array A and its Cartesian tree, and the translation of a query (ignore the other elements for now).

Later, Berkman and Vishkin [17] showed that the LCA problem on any tree can be reduced to an RMQ problem, on an array $D[1, 2n]$ containing the depths of the nodes traversed along an Eulerian tour on the tree: the LCA corresponds to the minimum in D between a cell of u and a cell of v in the array. Note that consecutive cells in D differ by ± 1. Bender and Farach [15] represented those entries as a bitvector $E[1, 2n]$: $E[i] = 1$ if $D[i] - D[i-1] = +1$ and $E[i] = 0$ if $D[i] - D[i-1] = -1$, with $E[1] = 1$. On top of E, they gave a simple $O(1)$-time solution to this restricted ± 1-RMQ problem using $O(n)$ words of space. Figure 5.1 also shows this arrangement.

Therefore, one can convert an RMQ problem on A into an LCA problem on the Cartesian tree of A, then convert this problem into a ± 1-RMQ problem on the depths of the Eulerian tour of the Cartesian tree, and finally solve this restricted ± 1-RMQ problem in constant time. This solution requires $O(n)$ words of space.

Interestingly, the bitvector $E[1, 2n]$ used to answer LCA queries on a tree of n nodes defines the topology of the tree. If we traverse the tree in DFS order and write an opening parenthesis when we first arrive at a node and a closing one when we leave it, the resulting sequence of parentheses, $P[1, 2n]$, is exactly $E[1, 2n]$ if we interpret the opening parenthesis as a 1 and the closing one as a 0. In particular, consider the following two operations on bitvectors: $\mathsf{rank}_b(E, i)$ is the number of bits equal to b in $E[1, \mathrm{i}]$, and $\mathsf{select}_b(E, j)$ is the position of the jth bit b in E. Both operations can be implemented in $O(1)$ time using just $o(n)$ additional bits on top of E [60, 22]. Then, if we identify a node x with the position of its opening parenthesis in P (which is a 1 in E), then the preorder position of x is $\mathsf{preorder}(x) = \mathsf{rank}_1(E, x)$, the node with preorder i is $\mathsf{prenode}(i) = \mathsf{select}_1(E, i)$, x is a leaf iff $E[x+1] = 0$, and the depth of x is $D[x] = \mathsf{rank}_1(E, x) - \mathsf{rank}_0(E, x) = 2 \cdot \mathsf{rank}_1(E, x) - x$.

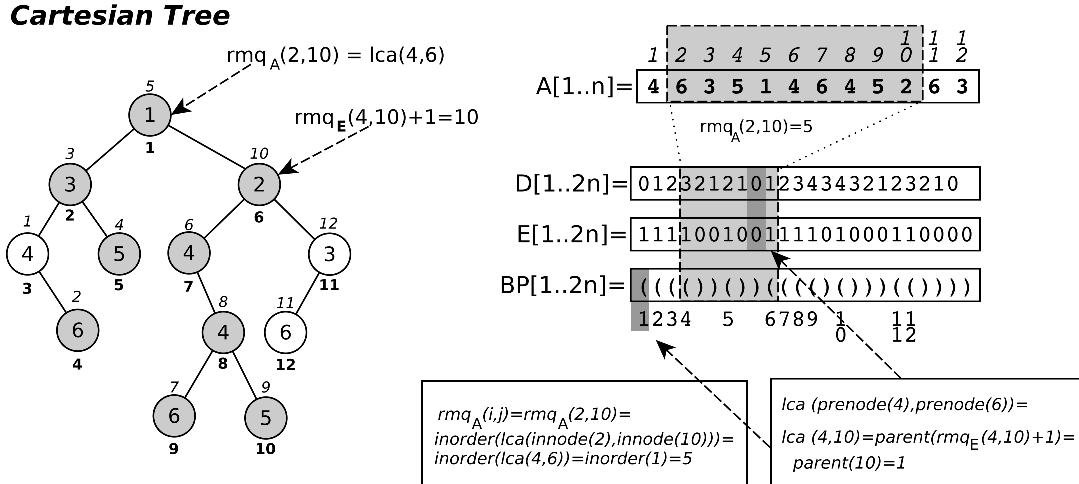

Figure 5.1: An example array $A[1,12]$ (top right) and its Cartesian tree (left). We choose preorder numbers as node identifiers (in bold under the nodes), and also write inorder values on top of the nodes, in slanted font. The left rectangle on the bottom shows how query $\mathsf{rmq}_A(2,10)$ translates into query $\mathsf{lca}(4,6)$ on the Cartesian tree. We also show how this query, in turn, maps into $\mathsf{rmq}_D(4,10)$, on the array D of depths of the tree. Array E tells if consecutive entries of D increase or decrease, and is the same as a BP representation of the tree. The right rectangle on the bottom shows how query $\mathsf{lca}(4,10)$ is solved using $\mathsf{rmq}_E(4,10)$ and parent on the parentheses. This rmq_E query is a simpler ± 1-RMQ problem. Now the nodes 4, 10, and 1 do not refer to preorders but to positions in BP, obtained from preorders with prenode. The corresponding preorder values are written below the BP array.

This parentheses representation (called BP, for Balanced Parentheses) was indeed known, and it was even possible to navigate it in constant time by using just $2n + o(n)$ bits [83, 48]. This navigation was built on top of three primitives on parentheses: $\mathsf{open}(x)/\mathsf{close}(x)$ gave the position of the opening/closing parenthesis matching the closing/opening one at $P[x]$, and $\mathsf{enclose}(x)$ gave the opening parenthesis position y so that $[y, \mathsf{close}(y)]$ contained $P[x]$ most tightly. Many tree traversal operations are built on top of those primitives, for example the parent of x is $\mathsf{parent}(x) = \mathsf{enclose}(x)$, its next sibling is $\mathsf{close}(x) + 1$ (if it exists), its first child is $x + 1$ (if it exists), its subtree size is $(\mathsf{close}(x) - x + 1)/2$, x is an ancestor of y iff $x \leq y \leq \mathsf{close}(x)$, etc.

Now, since E coincides with P, one could add the powerful lca operation to the BP representation! Bender and Farach's solution [15] applied on the bitvector $E[1, 2n]$ actually implements RMQs on the virtual array D. However, their ± 1-RMQ solution used $O(n)$ words. Sadakane [105] improved their solution to use $O(n(\log \log n)^2 / \log n) = o(n)$ bits, and thus obtained a constant-time algorithm for $\mathsf{lca}(x, y)$ on the BP representation (let $x < y$):

$$\text{if } y \leq \mathsf{close}(x) \text{ then return } x$$
$$\text{else return } \mathsf{parent}(\mathsf{rmq}_E(x, y) + 1)$$

where the first line addresses the special case where x is an ancestor of y, and rmq_E refers to the ± 1-RMQ solution on $E[1, 2n]$. The rationale of the second line is that, since x and y descend from two distinct children of $z = \mathsf{lca}(x, y)$, then $D[x, y]$ is minimized at the closing

parenthesis that terminates each child of z, from the one that contains x to the one preceding that containing y. Adding 1 we get to the next sibling of that child, then we return its parent z. See Figure 5.1 once again.

Benoit et al. [16] presented an alternative format to represent a general tree using $2n$ parentheses, called DFUDS. We traverse the tree in DFS order, but this time, upon arriving for the first time to a node with d children, we write d opening parentheses and a closing one (in particular, a leaf is represented with a closing parenthesis). Nodes are identified with that closing parenthesis[1]. It can be shown that the resulting sequence is also balanced if we append an artificial opening parenthesis at the beginning, and many traversal operations can be carried out with the primitives open, close, and $\mathsf{enclose}$. In particular, we can directly arrive at the ith child of x with $\mathsf{next}_0((\mathsf{close}(x-i)+1)$, where $\mathsf{next}_0(t) = \mathsf{select}_0(\mathsf{rank}_0(t-1)+1)$ finds the first 0 from t. The number of children of x can be computed as $\mathrm{d} = x - \mathsf{prev}_0(x)+1$, where $\mathsf{prev}_0(t) = \mathsf{select}_0(\mathsf{rank}_0(t-1))$ finds the last 0 before t. In DFUDS, nodes are also listed in preorder, and there is a closing parenthesis terminating each, thus $\mathsf{preorder}(x) = \mathsf{rank}_0(E, x)$.

Jansson et al. [61] showed that $\mathsf{lca}(x, y)$ can also be computed on the DFUDS representation, as follows (let $x < y$):

$$\text{return } \mathsf{parent}(\mathsf{next}_0(\mathsf{rmq}_E(x, y-1)+1)),$$

where no check for ancestorship is needed[2]. The rationale is similar as before: since in DFUDS D decreases by 1 along each subtree area, $\mathsf{rmq}_E(x, y-1)$ finds the final closing parenthesis of the child of $z = \mathsf{lca}(x, y)$ that precedes the one containing y. Adding 1 and finding the parent gives z. The formula for $\mathsf{parent}(w)$ in DFUDS is $\mathsf{next}_0(\mathsf{open}(\mathsf{prev}_0(w)))$. Figure 5.2 shows our example, now on DFUDS. The formula with DFUDS turns out to be simpler than with BP.

Now we could represent a tree of n nodes in $2n+o(n)$ bits and compute lca on it in constant time, and Eq. (5.1) allowed us to convert rmq_A into an lca operation on its Cartesian tree. It seems that there is a way to build a constant-time rmq_A structure using just the $2n+o(n)$ bits of its Cartesian tree, and without accessing A. However, there was still a problem: how to support the operations $\mathsf{inorder}$ and innode on the Cartesian tree. Sadakane [105] had solved the problem on suffix trees, but in his case the tree had exactly one leaf per entry in A, so one only needed to find the ith leaf, and this could be done by extending $\mathsf{rank}/\mathsf{select}$ operations to find 10s (BP) or 00s (DFUDS) in E. In the general case, one could add artificial leaves to every node, but this would increase the space to $4n + o(n)$ bits.

Fischer and Heun [40] found a solution that used just $2n + o(n)$ bits, which also turned out to be asymptotically optimal. The idea is to use a known isomorphism (see, e.g., [83]) between binary trees of n nodes and general ordinal trees of $n+1$ nodes: We create an extra root for the general tree, and its children are the nodes in the leftmost path of the binary tree. Recursively, the right subtree of each node x in the leftmost path is converted into a general tree, using x as its extra root. A key property of this transformation is that inorders in the binary tree become preorders (plus 1) in the general tree. Fischer and Heun called it tree as *2d-Min-Heap*. As seen, we can easily map between nodes and their preorders in

[1]In some cases, the first opening parenthesis is used, but the closing one is more convenient here.
[2]The check is present in their paper, but it is unnecessary (K. Sadakane, personal communication).

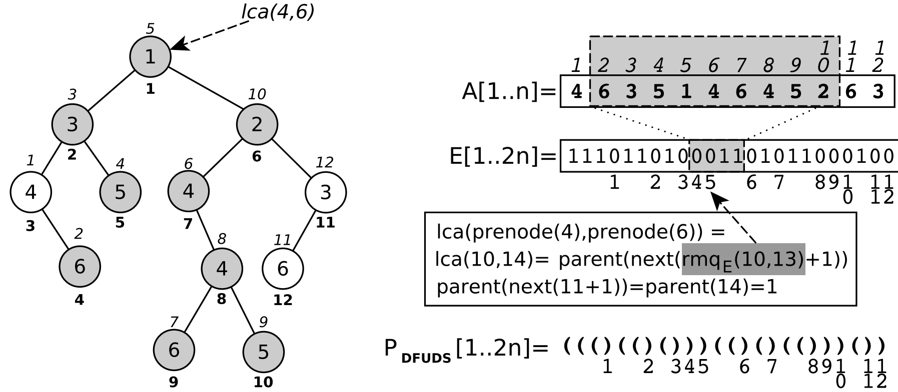

Figure 5.2: The same arrangement of Figure 5.1, now on the DFUDS representation of the Cartesian tree. The query $\mathsf{rmq}_A(2, 10)$ becomes $\mathsf{lca}(4, 6)$, which we translate into $\mathsf{lca}(10, 14)$ when the node identifiers become positions in DFUDS instead of preorders (the translation is shown on the bottom of the sequence P_{DFUDS}).

general trees. Figure 5.3 continues our example.

However, the lca in the Cartesian tree (which is what we want) is not the same lca in the resulting general tree; some adjustments are necessary. Fischer and Heun chose to use DFUDS for their $\mathsf{rmq}_A(i, j)$ solution, where it turns out that the adjustments to use a general tree actually remove the need to compute parent, but add back the need to check for ancestorship:

$$\begin{aligned} & w \leftarrow \mathsf{rmq}_E(\mathsf{select}_0(i+1), \mathsf{select}_0(j)) \\ & \text{if } \mathsf{rank}_0(\mathsf{open}(w)) = i \text{ then return } i \\ & \text{else return } \mathsf{rank}_0(w) \end{aligned} \tag{5.2}$$

The select_0 operations find the nodes with preorder i and $j - 1$ (recall there is an extra root with preorder 1), then w is the position of the closing parenthesis of the result. The next line verifies that x is not an ancestor of y, and the last line returns the corresponding preorder value. For this formula to be correct, it is necessary that rmq_E returns the position of the leftmost minimum. Figure 5.3 (top left) shows a query.

Grossi and Ottaviano [53] replaced the ancestorship test by one that does not use the costly open operation:

$$\begin{aligned} & w \leftarrow \mathsf{rmq}_E(\mathsf{select}_0(i+1), \mathsf{select}_0(j)) \\ & \text{if } D[\mathsf{select}_0(i)+1] \leq D[w-1] \text{ then return } i \\ & \text{else return } \mathsf{rank}_0(w) \end{aligned} \tag{5.3}$$

where as explained we can compute $D[k] = 2 \cdot \mathsf{rank}_1(E, k) - k$.

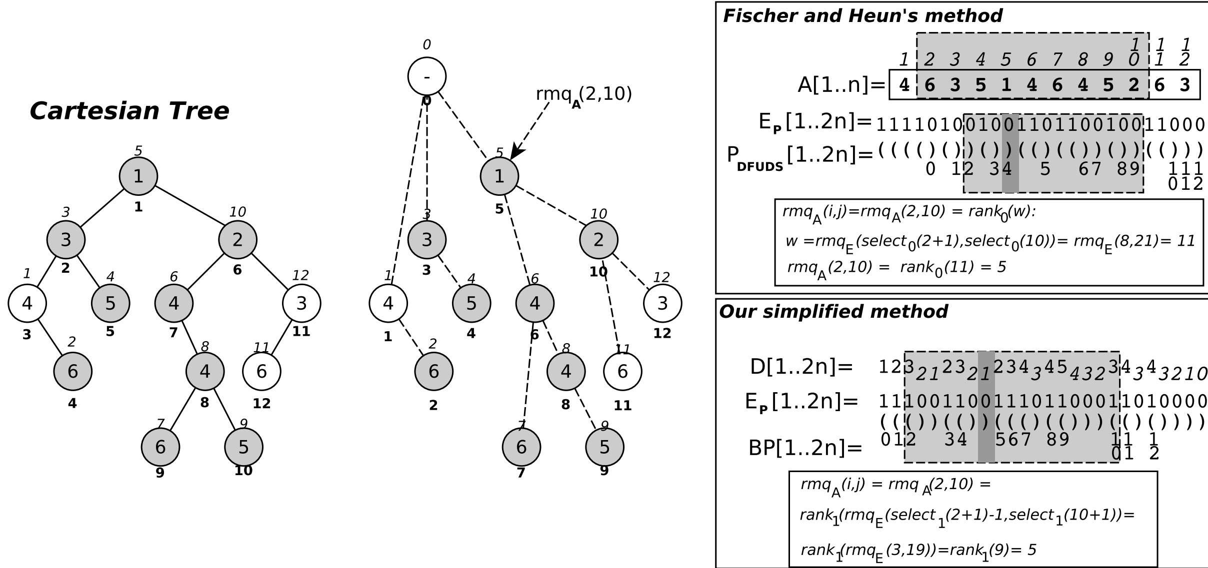

Figure 5.3: The general tree (at middle) derived from the example Cartesian tree. Note how inorder numbers of the binary Cartesian tree became preorder numbers in the general tree (we start preorders from 0 to help see the mapping). On the right, the formulas used by Fischer and Heun based on DFUDS (on the top) and the one proposed in this paper, based on BP (on the bottom). To reuse the same isomorphism of Fischer and Heun, we illustrate the variant of our formula that uses the leftmost path of the tree as the root children.

5.2 A Simplified Implementation

The current implementations of rmq_A build on the DFUDS representation of the general tree derived from the Cartesian tree, and follow either the formula of Fischer and Heun [40] (Eq. (5.2), in *SDSL*), or that of Grossi and Ottaviano [53] (Eq. (5.3), in *Succinct*). We show that, if we use the BP representation instead of DFUDS, we obtain a simpler formula. Let us assume, as before, that rmq_E returns the leftmost minimum. Then, our conversion from the binary Cartesian tree into a general tree must go in the opposite direction: the children of the extra root are the nodes in the *rightmost* path of the binary tree, and so on recursively. With this representation, it turns out that a correct formula is

$$\mathsf{rmq}_A(i,j) \;=\; \mathsf{rank}_0(\mathsf{rmq}_E(\mathsf{select}_0(i), \mathsf{select}_0(j))) \tag{5.4}$$

where no checks for ancestorship are necessary. Now we prove this formula is correct.

Lemma 5.2.1. On a rightmost-path general tree built from the Cartesian tree of A, Eq. (5.4) holds.

Proof. On the rightmost-path representation, the binary tree node with inorder i becomes the general tree node with *postorder* i, which is easily seen by induction. The closing parentheses of nodes x and y, which have postorders i and j, are thus found with $p = \mathsf{select}_0(i)$ and $q = \mathsf{select}_0(j)$. Now let $z = \mathsf{lca}(x, y)$. Then, in the Cartesian tree, x descends from the left child of z, z_l, and y descends from the right child, z_r. In the general tree, z_l is the first

child of z, whereas z_r is its next sibling. Therefore the closing parenthesis of z, at position r, is between p and q. Further, y descends from some sibling z' to the right of z. Between p and q, the minima in D occur at the closing parentheses of z and of its siblings to the right, up to (but not including) z'. Thus the leftmost of those positions is precisely r, where z closes. Finally, $\mathsf{rank}_0(r)$ is the postorder position of z, and the inorder position of the cell in A.

The formula also works if y descends from x in the Cartesian tree. Since $i < j$, the inorder of x is smaller than the inorder of y, and thus y can only descend from the right child of x. Then the first minima in $[p, q]$ is precisely p, the closing parenthesis of x, and thus $z = x$. $\square$

If we want to use the leftmost-path mapping, we need that rmq_E returns the rightmost minimum position in the range. In this case, it holds

$$\mathsf{rmq}_A(i, j) \;=\; \mathsf{rank}_1(\mathsf{rmq}_E(\mathsf{select}_1(i + 1) - 1, \mathsf{select}_1(j + 1))) \tag{5.5}$$

In this case, we must subtract 1 from p (which is now the position where node x opens) to ensure that the rightmost minimum in $[p - 1, q]$ is actually $p - 1$ when y descends from x. Figure 5.3 (bottom right) shows a query.

The distinct operations involved in the solutions for rmq_A, even if constant-time, take widely different times in practice. The original formula of Eq. (5.2) includes 2 operations rank and 2 operations select, one ± 1-RMQ (rmq_E), and one operation open. The last two operations are much costlier than rank and select. For example, in a study of succinct tree representations [1], operations rank required about 50 nanoseconds (ns), select required around 200 ns, open required 200–500 ns, and rmq_E required 400-700 ns. In that paper, they used a succinct tree implementation that used about $2.4n$ bits, based in the so-called Range Min-Max Tree (RMM-tree) [93]. While their operation time is in theory logarithmic, they show that the time growth with n is practically unnoticeable. Other constant-time solutions for open [48] were shown to be significantly slower in practice if using similar space. Our new formula in Eq. (5.4) requires only 2 operations select, one rank, and one rmq_E. In Section 5.4 we show that our formula yields a significant time reduction compared to DFUDS-based ones.

5.2.1 Construction

This representation is easily built in a way similar to the DFUDS-based one [40]. Consider the version using the rightmost-path mapping (the other is similar). We will write the parentheses of $E[1, 2n]$ right-to-left, starting with a 0 (i.e., a closing parenthesis) at its end. We start with an empty stack S, and traverse $A[n]$ to $A[1]$. At the point where we are to process $A[\mathrm{i}]$, the stack S maintains left-to-right minima in $A[\mathrm{i} + 1, n]$. To process $A[\mathrm{i}]$, we pop from S all the elements $\geq A[\mathrm{i}]$, prepending a 1 (i.e., an opening parenthesis) to E each time an element is popped, until S becomes empty or its top is $< A[\mathrm{i}]$. Now we push $A[\mathrm{i}]$ in S and prepend a 0 to E. This is continued until $A[1]$ is processed. Finally, we prepend as many 1s to E as necessary to complete $2n$ bits.

This process requires $O(n)$ time and its extra space for S is proportional to the height of the Cartesian tree of A. While this is usually negligible, the space can become $O(n)$ words

in the worst case. Fischer and Heun [40, Sec. 5.2.2] reduce it to n bits in a way that we can use verbatim in our case.

5.3 Implementing Balanced Parentheses

The most successful implementation of Balanced Parentheses uses Range Min-Max Trees (rmM-trees) [93, 1]. The BP sequence $E[1, 2n]$ is cut into blocks of length b. Each block then becomes a leaf of the rmM-tree, which stores several variables. To describe them, let us introduce the notion of *excess*, which is the number of 1s minus the number of 0s in a bit string up to certain position:

$$\mathsf{excess}(S, \mathrm{i}) \quad = \quad \mathsf{rank}_1(S, \mathrm{i}) - \mathsf{rank}_0(S, \mathrm{i}) \quad = \quad 2 \cdot \mathsf{rank}_1(S, \mathrm{i}) - \mathrm{i},$$

where we note that, if $D[1, 2n]$ is the sequence of depths we have been using and $E[1, 2n]$ is the associated bit sequence, then $D[\mathrm{i}] = \mathsf{excess}(E, \mathrm{i})$.

Then the relevant variables associated with each rmM-tree leaf representing bits $L[1, b]$ are $\mathrm{e} = \mathsf{excess}(L, b)$ (the local excess produced by the leaf), and $m = \min_{1 \leq \mathrm{i} \leq b} \mathsf{excess}(L, \mathrm{i})$ (the minimum left-to-right excess along the leaf). The rmM-tree is a perfect binary tree on those leaves, where the internal nodes store the same fields e and m with respect to the area they cover. That is, let v have left and right children v_l and v_r, respectively, then $v.\mathrm{e} = v_l.\mathrm{e} + v_r.\mathrm{e}$ and $v.m = \min(v_l.m, v_l.\mathrm{e} + v_r.m)$.

We can then compute any operation $\mathsf{rmq}_E(p, q)$ as follows. First, we determine the maximal block-aligned range $[p', q']$ inside $[p, q]$. Then we scan the range $[p, p' - 1]$ sequentially, obtaining the minimum excess min and its excess $exc = \mathsf{excess}(p, p' - 1)$. Then, if $[p', q']$ is not empty, we start at the rmM-tree leaf v started by position p'. We set $min \leftarrow (min, exc + v.m)$ and update $exc \leftarrow exc + v.\mathrm{e}$. Now we start climbing up the path from v. If v is a right child of its parent, we just move to its parent. Otherwise, we see if its right sibling v' is contained in $[p', q']$. If it is, we process it (setting $min \leftarrow \min(min, exc + v'.m)$ and $exc \leftarrow exc + v'.\mathrm{e}$) and then go to the parent of v. If, instead, v' is not contained in $[p', q']$, we switch to $v \leftarrow v'$ and start the descent: Let v_l and v_r be the left and right children of v, respectively. Then, if v_l is is contained in $[p', q']$ we process v_l as before and descend to v_r, otherwise we descend to v_l. At the end, we reach the leaf of position $q' + 1$, which is traversed sequentially up to position q to complete the process.

Once the minimum value is clear, we must find its leftmost occurrence in $E[p, q]$. If it occurred in $[p, p' - 1]$, or occurred only in $[q' + 1, q]$, then we already know its position. Otherwise, its leftmost occurrence is in some rmM-tree node v we know. We then move down from v to find its position: if $v_l.m \leq v_l.\mathrm{e} + v_r.m$, we descend to v_l, otherwise to v_r. We finally reach a leaf and scan it to find the position of the leftmost minimum.

By setting $b = \Theta(\log^2 n)$ and using precomputed tables to process the leaves by chunks of $(\log n)/2$ bits, the total time is $O(\log n)$ and the extra space of the rmM-tree and precomputed tabls is $O(n/\log n) = o(n)$.

Operations rank and select can be solved similarly, the former by computing $exc =$

$\mathsf{excess}(E, \mathrm{i})$ and then using $\mathsf{rank}_1(\mathrm{i}) = (exc + \mathrm{i})/2$ or $\mathsf{rank}_0(\mathrm{i}) = (\mathrm{i} - exc)/2$. For $\mathsf{select}_1(j)$ we move down from the rmM-tree root looking for the position i where $\mathsf{excess}(E, \mathrm{i}) = 2 \cdot j - \mathrm{i}$, and for $\mathsf{select}_0(j)$ we aim to $\mathsf{excess}(E, \mathrm{i}) = \mathrm{i} - 2 \cdot j$.

Our implementation carries out the following optimizations:

1. Instead of the fields e in all the nodes, we store an array $exc[1, 2n/b]$ with $exc[\mathrm{i}] = \mathsf{excess}(E, b \cdot \mathrm{i})$, using as many bits as necessary (in many cases, the maximum excess is not large). Further, when b is even, those sampled excesses are also even, so we save one further bit. To solve rank, we use the table exc to find the rank up to the previous sampled position, and process the last block sequentially.
2. To solve $\mathsf{select}(j)$, we store a table giving the blocks where the answer to every bth value of j fall, using as many bits as necessary. We then compute rank up to that block and sequentially scan from its beginning until reaching the desired rank j. In the conference version [32] we just used binary search on table exc, which saved little space but was considerably slower.
3. To solve rmq_E, we avoid scanning the last area $[q' + 1, q]$ if its block minimum is not smaller than our current minimum value min. Note that its block minimum may be smaller than the minimum in $[q' + 1, q]$, but not larger. In the conference version we stored the position of the minimum to avoid the descent, but this turns out to take too much extra space for a small saving in time.
4. The precomputed tables process bytes of the leaf, so they are very small and usually fit in cache, and we read aligned data.

5.4 Experimental Results

Our first experiment compares our improved implementation with the standard one, which was used in our conference version [32] with block size $b = 256$. We show various block sizes for our new version, so as to choose a good representative. The data are arrays A of sizes from $n = 10^4$ to $n = 10^{10}$, with randomly chosen ranges $[\mathrm{i}, j]$ of fixed length 10,000. Figure 5.4 shows the results, where "rmq-Old" stands for the standard implementation and "rmq-b" for the new ones. The space shown is in addition to the $2n$ bits used by the parentheses.

It can be seen that our new implementation is far more efficient, in space and especially in time. For the rest of the experiments, we will choose $b = 512$ as a compromise value between space and time.

We compare our implementation with those in *SDSL* and *Succinct*, which are based on DFUDS (Eqs. (5.2) and (5.3), respectively). As a control, we also implement ourselves the DFUDS-based solution of Eq. (5.2) using rmM-trees and our rank/select components; this is called *DFUDS* in our charts.

We first compare the four implementations on the same randomly generated arrays A of the previous experiment. Figure 5.5 shows the results (*Succinct* did not build on the largest

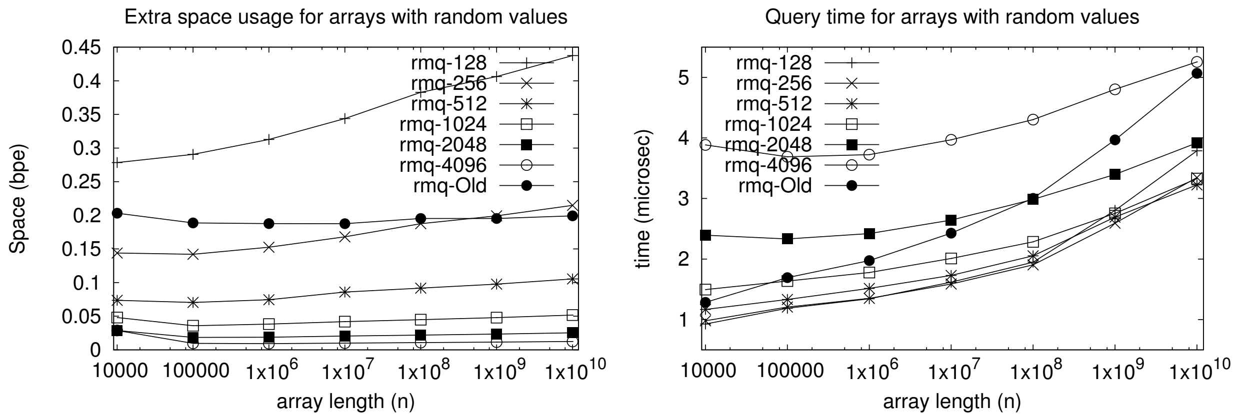

Figure 5.4: Query space and time on random arrays, for ranges of size 10,000, comparing the standard with our new implementations.

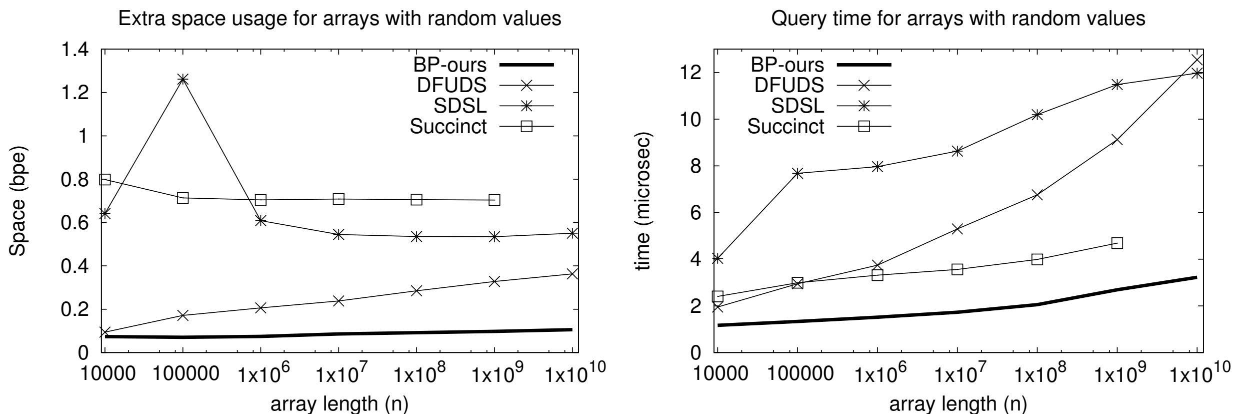

Figure 5.5: Query space and time on random arrays, for ranges of size 10,000.

arrays). Our implementation uses always below 2.1 bits per element (bpe), that is, 0.1 on top of the 2 bpe needed by the BP (or DFUDS) representation. Our DFUDS implementation, instead, increases the space because the average excess grows with n in this format, and thus the rmM-tree counters need more bits. The implementations in *SDSL* and *Succinct* use at least 2.6–2.8 bpe.

Our solution is also the fastest, taking 1–3 microseconds (μsec) per query as n grows. It is followed by *Succinct* and, far away, by *SDSL*. Our *DFUDS* implementation is fast for short arrays, but it becomes slower when n grows. This is probably because operation open matches a farther parenthesis as n grows; the same effect can be seen in *SDSL*. In *Succinct*, instead, operation open is avoided, and thus the growth is much milder. In our BP-based implementation, the growth with n is also mild, owing only to traversing a higher rmM-tree.

Figure 5.6 shows how the times are affected by the size of the query range. As it can be seen, our implementation and *Succinct* show a very slow increase, whereas times grow much faster in *SDSL* and *DFUDS*. This may be due to the open operation, whose time grows in practice with the distance to its parent. Larger intervals return nodes closer to the root, whose former siblings are larger, and so is the distance to the parent in DFUDS.

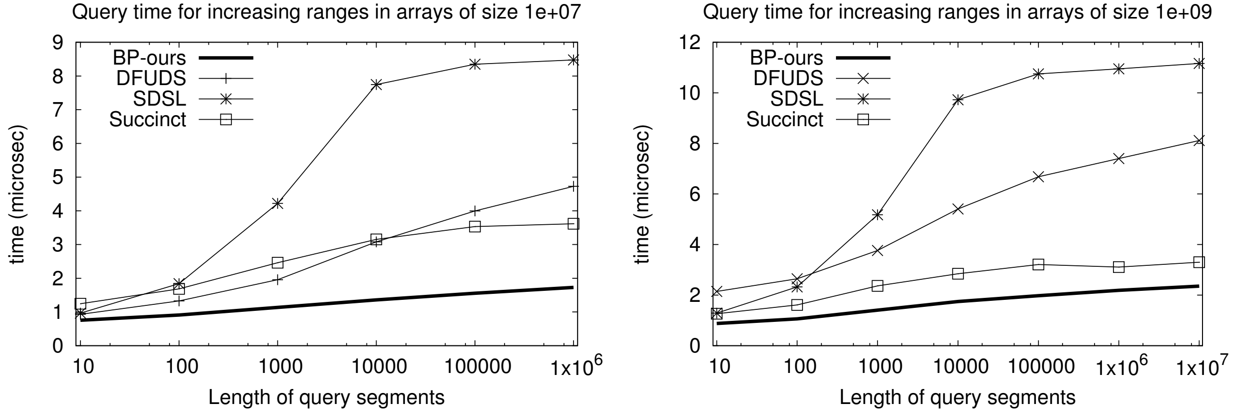

Figure 5.6: Query time on random arrays, for ranges of increasing size and two values of n.

Our final experiment measures the effect of the order in A on the space and time of the structures. Given a parameter Δ, our entry $A[\mathrm{i}]$ is chosen at random in $[\mathrm{i}-\Delta, \mathrm{i}+\Delta]$, or in $[n-\mathrm{i}-\Delta, n-\mathrm{i}+\Delta]$, thus the smaller Δ, the more sorted is A in increasing/decreasing order. Figure 5.7 shows the results.

Our implementation maps the leftmost path of the Cartesian tree to the children of the general tree. As a result, the structure takes slightly more space and time when the array is more sharply increasing, because the general tree is deeper and the rmM-tree stores larger values. Instead, it does not change much when A is decreasing (one could use one mapping or the other as desired, since we know A at construction time, thus never using more than 2.1 bpe). *DFUDS* shows the opposite effect, because the DFUDS excesses are smaller when the tree is deeper. The effect is more pronounced than in our structure, and it also affects the time performance. It is not clear how can one use the rightmost-path mapping in the case of DFUDS, however, as it is not symmetric (we can reverse the array if we do not mind returning the rightmost position of the minimum). The space of *SDSL* and *Succinct* is not affected at all by the lack of randomness, but *SDSL* turns out to be faster on less random arrays, regardless of whether they are increasing or decreasing. *Succinct* performs better when the values tend to be decreasing and worse when they are increasing. Our times are, just like the space, negatively affected by increasing values, but still they are much better than the others and, as said, we can choose to map the rightmost path in this case.

5.5 Conclusions

We have presented an alternative design to Fischer and Heun's RMQ solution that uses $2n + o(n)$ bits and constant time [40]. Our implementation uses $2.1n$ bits and takes 1–3 microseconds per query. This is noticeably smaller and faster than the current implementations in libraries *SDSL* and *Succinct*, which follow Fischer and Heun's design. By using BP instead of DFUDS succinct tree representation, our RMQ formula simplifies considerably, and besides we performed some optimizations to the BP implementation. We have left our implementation publicly available at `https://github.com/hferrada/rmq.git`, and our

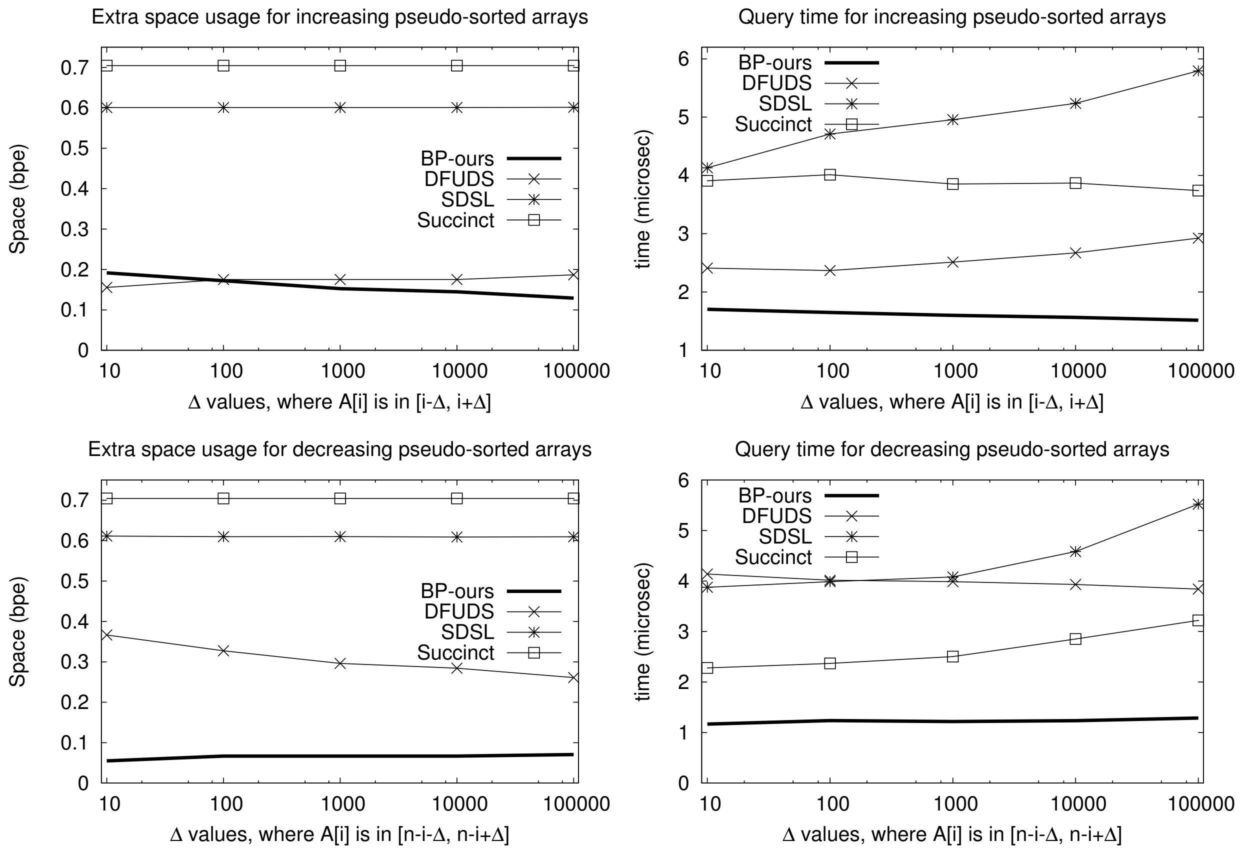

Figure 5.7: Query time on pseudo-sorted arrays, $n = 10^6$ and ranges of size 10,000.

DFUDS-based one at `https://github.com/hferrada/rmqFischerDFUDS.git`.

Any ±1-RMQ implementation can be used together with our new formula. Our current implementation of ±1-RMQs is not formally constant time, as it builds on rmM-trees [93, 1]. Although truly constant-time solutions are not promising in practice [105, 93], and we have shown that the time of rmM-trees grows very slowly with n, it would be interesting to devise a practical and constant-time solution.

Chapter 6

An LZ-based Index for Document Listing

We propose an index to solve the document listing problem, which is based on the classical LZ-Index [85] (detailed in Section 2.8). The resulting index is called LZ-DLIndex, which was published in the 20th International Symposium on String Processing and Information Retrieval [30]. A journal version, together with the results of Chapter 7, was submitted to Information and Computation.

We search the documents by considering the same 3 types of occurrences of pattern matching. The key idea is, instead of collecting each individual occurrence of p, to simulate Sadakane's variant of Muthukrishnan's algorithm (Section 3.1) on ranges of occurrences, even when the information is more fragmented on the LZ-Index than on a suffix array.

For the occurrences of type 1, the pattern matching algorithm finds the locus v^r of p^r in RevTrie and traverses its whole subtree. It maps each node u^r in the subtree of v^r to u in LZTrie, and then traverses the whole subtree of u. Now we want to report all the distinct documents found across this process. We virtually *expand* each node u^r in RevTrie with the subtree of u, recording the document where each node belongs. The result is an array analogous to $E_{1..n}$, where we can use Sadakane's document listing algorithm on the range covered by v^r (see Algorithm 2). We do not store E itself, but just the RMQ structure on the corresponding array C (virtual too). The RMQ structure uses $2n + o(n)$ bits (see the definition in Section 2.5.4) and allows us to find each new document in $O(1)$ time. We present a review of the state of the art on the RMQ problem and our proposal in Chapter 5.

For the occurrences of type 2, we find the $O(\log n)$ nodes that cover the y-interval $[l_u, r_u]$ and project the x-interval $[l_v, r_v]$ to each such node. Each point to report belongs to some document, and we want again to report all the distinct documents. We can do it by brute force (reporting the document of every individual point, avoiding repetitions), or apply Sadakane's algorithm on each of the $O(\log n)$ ranges $[x_m, x_M]$ in the wavelet tree nodes that cover the y-coordinate interval. For this sake, we attach the RMQ structures on the virtual C arrays for the points represented in each wavelet tree node. Therefore, to each bitvector B_v we add an RMQ structure using $2|B_v| + o(|B_v|)$ bits [90].

Finally, occurrences of type 3 are $O(m^2)$ in total and are dealt with one by one. The whole process takes time $O(m^2 \log n + \mathsf{ndoc}\, m \log^2 n)$. However, the ndoc_1 documents found with occurrences of type 1 are listed in time $O(m + \mathsf{ndoc}_1)$. Now we describe the techniques precisely.

We modify the LZ78 parsing so that no phrase crosses a document boundary. The index is composed of the following structures.

6.1 Structure

Tries. We store the topology of RevTrie and LZTrie and also the letters, but only for RevTrie because we perform the searches exclusively with reverse patterns. We then able the index to navigate RevTrie and to search it for patterns in constant time per symbol [3]. These require $2n' \lg \sigma + O(n')$ bits and support constant-time traversals. Note that LZTrie has n' nodes, and thus its topology is represented with $2n' + o(n')$ bits (Section 2.4). However, RevTrie may have up to n nodes, because not every node corresponds to a phrase. From those nodes, some are *unary*, that is, have just one child, and some are *empty*, that is, do not represent any phrase. Since RevTrie has at most n' leaves and exactly n' nonempty nodes, it has at most $2n'$ non-unary nodes. Thus we can represent only the (at most) $3n'$ nodes that are non-unary or nonempty, and collapse the remaining unary paths. Only the symbols that are not in those paths are stored. This leads to a representation that uses $2n' \lg \sigma + O(n')$ bits. The symbols from unary paths are extracted via the connection with the LZTrie [85, 3]. We also store a bitvector of $O(n')$ bits to compute preorder numbers of nonempty nodes. So, we use in this first group the following structures:

P_{lz}: The LZTrie topology represented with parentheses in a preorder traversal, and made navigable in $O(1)$ time, using $2n' + o(n')$ bits (`FF` [1]).

P_{rev}: The tree topology using parentheses and made constant-time navigable, using $2t_{rev} + o(t_{rev})$ bits (`FF` [1]).

E_{rev}: A bitvector marking empty nodes, in preorder, using $t_{rev} + o(t_{rev})$ bits.

U_{rev}: A bitvector marking empty unary nodes (i.e., contracted), from those that are marked empty in E_{rev}, using $t_{rev} - n'$ bits.

L_{rev}: A sequence of the n_{rev} letters that label the non-contracted edges leading to the nodes, in preorder. Used to find the child nodes at searching.

M_{rev}: A sequence of the $t_{rev} - n_{rev}$ letters that label the contracted edges leading to the nodes, in preorder. Used to check that the characters in the contracted edge match the search pattern.

Documents. Instead of storing the phrase identifiers for the n' nodes of LZTrie, we store the identifiers of the document where they occur. We also store the RMQ structure associated with a virtual array of all the documents where each phrase in RevTrie is transitively used.

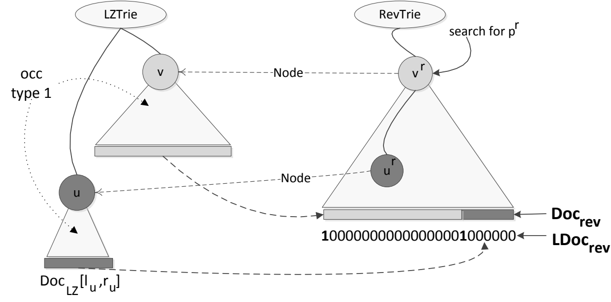

Figure 6.1: The structures to report documents for occurrences of type 1.

In total we store $n' \lg D + 3n + o(n)$ bits, in the following structures (see the arrays on the bottom of Figure 6.1).

Doc_{lz}: The array of n' document identifiers of the LZTrie phrases in preorder order, stored explicitly in $n' \lg D$ bits. This is equivalent to the document array of Muthukrishnan (Section 3.1), but restricted to phrases.

Doc_{rev}: A sequence of n document identifiers built as follows. We traverse RevTrie in preorder, and for each nonempty node v^r, let v be the corresponding LZTrie node. Let $Doc_{lz}[l_v, r_v]$ be the range of all the descendants of v (included). We append $Doc_{lz}[l_v, r_v]$ to Doc_{rev}. The total length of Doc_{rev} is n because n is the internal path length (sum of all node depths) in LZTrie, and each LZTrie node is appended to Doc_{rev} once per ancestor it has in LZTrie.

We do not store Doc_{rev}, but only the $2n+o(n)$-bits RMQ structure on its corresponding C array (Section 3.1). This will be sufficient to run Sadakane's DL algorithm [106] on top of Doc_{rev}. Recall that the RMQ structure does not need to access Doc_{rev} (Section 2.5.4)

$LDoc_{rev}$: A bitvector of n bits that marks the Doc_{rev} positions where the intervals $Doc_{lz}[l_v, r_v]$ start. Since it has only n' bits set, it is represented in compressed form (Section 2.3), so it can use less than n bits.

Node. A mapping from RevTrie to LZTrie. If the node v^r in RevTrie with nonempty preorder i corresponds to the node v in LZTrie with preorder j, then $Node[i] = j$. Array $Node$ uses $n' \lg n'$ bits.

Range. An enhanced binary wavelet tree. Each wavelet tree node implicitly represents a sequence of points (i.e., pairs of phrases $(k, k+1)$). Now consider the array of their corresponding documents (we are not interested in pairs of phrases that span two documents, as no matches occur there). In addition to the bitvector B_v of node v, we store the RMQ structure corresponding to the C array of its (virtual) array of documents (Section 3.1). The total space of Range is then $3n' \lg n' + o(n' \log n')$ bits.

Space. Overall, the LZ-DLIndex requires $4n' \lg n' + n' \lg D + 2n' \lg \sigma + o(n' \log n') + 3n + o(n) \leq 5nH_k(T) + 3n + o(n \log \sigma)$ bits. This is close to the original LZ-Index size [85]. The total LZ-DLindex size includes $2n' \lg n' + o(n' \lg n')$ bits for the RMQ structures in the wavelet tree of Range. A way to reduce this size is to exclude those RMQ structures, paying instead the cost in time to check every secondary occurrence one by one. In that case, the size becomes $\leq 3nH_k(T) + 3n + o(n \log \sigma)$ bits, which is similar to the space achieved by Arroyuelo et al. [3] plus the $n' \lg D + 3n + o(n)$ bits required to store the documents for all the LZTrie nodes and the associated RMQ structure. We test this variant in the experiments of Section 6.3.

Observation. Since phrases are cut at the end of documents, there may appear a few repeated phrases across the collection. Therefore, at construction time, we have to consider the special case when two or more documents end with the same phrase. This is handled by storing a short linked list, both in RevTrie and LZTrie, attached to the nodes representing phrases that appear more than once.

6.2 Queries

As we have introduced, we solve DL incrementally, considering the three types of occurrences.

Occurrences of type 1

We search for p^r in RevTrie, arriving at node v^r. This means that all the occurrences of type 1 are represented by v^r. In particular, as RevTrie is not a typical *trie*, because we contracted paths of empty nodes, we have to be careful during the search of v^r. We need then to validate, using U_{rev}, when an edge contains more than one symbol and retrieve these from M_{rev}. Once that we get the node, let $[i_v, j_v]$ be the range of preorders of nonempty nodes descending from v^r. We find the interval $I = Doc_{rev}[s_v, \mathrm{e}_v]$ of all the documents that contain occurrences of type 1, where $s_v = select_1(LDoc_{rev}, i_v)$ and $\mathrm{e}_v = select_1(LDoc_{rev}, \mathrm{i}_v + 1) - 1$. Next, we report all the distinct documents in I with Sadakane's algorithm using RMQs. For each new position pos of a document $Doc_{rev}[pos]$ reported by an RMQ, we need to report the document identifier. We determine the nonempty preorder $j = rank_1(LDoc_{rev}, pos)$ of the RevTrie node holding that position, and then the preorder of this node in LZTrie, $i = Node[j]$. The difference $d = pos - select_1(LDoc_{rev}, j)$ provides the offset of this position within the leaf interval of the LZTrie node with preorder i. Thus, the document is $Doc_{lz}[i + d]$. The overall time of this step is thus $O(m + \mathsf{ndoc}_1)$.

Occurrences of type 2

We proceed as in the original LZIndex for reporting occurrences from Range, but now we use the RMQ structures in the wavelet tree of Range to report documents. We consider all the $m - 1$ partitions $p = p_{start} \cdot p_{\mathrm{end}}$ and search for these prefixes and suffixes in the

tries. Each such partition then becomes a range search for $[l_v, r_v] \times [l_u, r_u]$ in Range, and is decomposed into $O(\lg n')$ intervals $[x_m, x_M]$ in different wavelet tree nodes v. Each point in those intervals represents a position in a document. The distinct documents in each interval $[x_m, x_M]$ are obtained using Sadakane's algorithm on the RMQs built for the node. To obtain the document identifier for each reported position $pos \in [x_m, x_M]$, we track the position down in the wavelet tree until reaching the leaf, which indicates the row of Range. Since the rows of Range correspond to LZTrie preorders, we simply access Doc_{lz} at the leaf index. This scheme is shown in the Fig. 2.10.

Although unlikely, in the worst case we can output the same document in each of the $O(\lg n')$ intervals for each of the $m-1$ partitions, and each requires $O(\log n')$ time for tracking the point down to the leaves. This gives $O(m^2)$ time for the RevTrie searches plus a (very pessimistic) worst-case bound of $O(\mathsf{ndoc}_2\, m \lg^2 n)$ time for the ndoc_2 occurrences of type 2. The Figure 6.2 illustrates how we use these RMQ'structures, stored in each level of the wavelet tree, to evaluate occurrences type 2 obtained by RMQ.

Occurrences of type 3

The last step is to report documents where the pattern appears as occurrence type 3. We follow the original LZindex search algorithm, yet we have fewer data structures now. First, and following that method, all the searches for all the substrings $p[i,j]$ are carried out in RevTrie, in time $O(m^2)$, and we record the RevTrie and LZTrie preorder values of each. We use *Node* to retrieve the LZTrie preorder giving a RevTrie node. For each i, we store in array A_{i} the information for the substrings of the form $p[i,j]$, sorted by LZTrie preorder value. Note that we have not stored phrase numbers, yet we can still use Range to determine the LZTrie preorder t of the phrase following that of $p[i,j]$, which has RevTrie preorder t^r. If we traverse the wavelet tree of Range starting at position t^r in the root bitmap and track it to the leaves, the final position is precisely t. This operation takes $O(\lg n')$ time. Now we implement a binary search on A_{j+1} for LZTrie preorder t, and if we find it corresponding to a phrase $p[j+1, j']$, we can concatenate $p[i,j]$ to get $p[i,j']$. We can therefore carry out the same process for finding maximal concatenations [85], in total time $O(m^2 \lg n)$. Finally, we have to check if $p[1..i-1]$ precedes the maximal concatenation and if $p[j+1,m]$ follows it. The first question is equivalent to computing whether the preorder interval for $p[1..i-1]^r$ in RevTrie is connected with the LZTrie preorder value t of the first phrase in the maximal concatenation. The second question corresponds to computing the LZTrie preorder interval of $p[j+1,m]$ (which can be done using RevTrie, as before) and then asking if the RevTrie preorder value t^r of the last phrase in the maximal concatenation is connected with some point in the LZTrie interval. We check these connections following the same technique that uses Range. These tests increase the time in $O(m \lg n)$.

Time

The total query time is $O(m^2 \lg n + \mathsf{ndoc}\, m \lg^2 n)$, where we remind that this is a very pessimistic upper bound. We also note that the occurrences of type 1 are reported very early, in

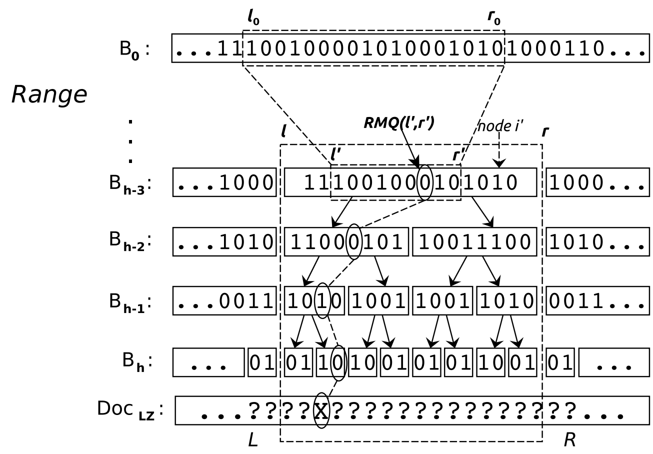

Figure 6.2: The scheme to report the occurrences of type 2 using RMQ structures in each level of the wavelet tree of Range. Suppose that we are looking for points in the rectangle $[l_0..r_0] \times [L..R]$ of Range. Then, we start at the root with the interval $[l_0..r_0]$, in the first bitstring B_0. We continuing go down by the tree updating the new limits $[l', r']$ for each node that we visit in the trip. Then, when we reach at node i', which represents the subinterval $B_{h-3}[l..r]$, we obtain a query interval $I_q = [l'..r']$, where $l \leq l', r' \leq r$, mapping from the original segment $[l_0..r_0]$ of the root to B_{h-3}. We check that $[l, r] \subseteq [L, R]$, then any of the $(r' - l' + 1)$ positions of I_q are occurrences type 2 that we have to evaluate. For this we apply the traditional DL algorithm on I_q using the RMQ structure stored. In this illustration, the first RMQ answer is mapping, go down in the tree, until to obtain the document X in the correct leaf of the tree.

time $O(m + \mathsf{ndoc}_1)$. If the text is generated by an ergodic source, the occurrences of any pattern p appear regularly, every d positions on average (e.g., $\mathrm{d} = \sigma^m$ if the symbols are generated uniformly and independently). On the other hand, since $n' \leq n/\lg_\sigma n$, only $O((n/d)m/\lg_\sigma n)$ of those occurrences hit a phrase boundary on average. This means that a fraction of $1 - O(m/\lg_\sigma n)$ of the occurrences are of type 1, and also $\mathsf{ndoc}_2 = O(\mathsf{ndoc}\, m/\lg_\sigma n) = o(\mathsf{ndoc})$ if $m = o(\lg_\sigma n)$. Thus we report almost all of the occurrences in $O(1)$ time each. If we just lose those $o(\mathsf{ndoc})$ occurrences not of type 1, our time is the optimal $O(m + \mathsf{ndoc})$.

We show in the experiments that, indeed, our index is particularly competitive to show the first occurrences (those of type 1), which are the most for short patterns.

6.3 Implementation

To obtain a practical implementation of the scheme, we make some changes that, although do not preserve the space and time guarantees, perform much better in practice. These refer largely to the implementation of the tries.

The mechanism to avoid storing symbols of unary paths in RevTrie and instead extract them from LZTrie is slow in practice. Instead, we will store them in RevTrie. Moreover, we

will perform all the searches in RevTrie, and do not represent LZTrie at all. RevTrie then has t_{rev} nodes, which can be as large as n, but in practice it is much less.

We represent RevTrie in DFUDS form, using $2t_{rev} + o(t_{rev})$ bits, plus a bitvector that marks the nonempty nodes in DFUDS order, so as to compute the nonempty preorders that are used in searches. We also store a string with the $2t_{rev}$ symbols that label the edges, in the same order they are stored in DFUDS. This allows (1) performing binary searches on the labels toward the children of a node, to find the one to follow efficiently, (2) having in consecutive positions the symbols that label unary paths, so as to compare them efficiently with p. The constant-time method to find the label given in DFUDS [16] is theoretical, and is better replaced with searches on this string.

RevTrie is used directly to find the occurrences of type 1, and also to search for p^r_{start} when looking for occurrences of type 2. To find p_{end}, since we cannot search in LZTrie, we look for p^r_{end} in RevTrie. If it does not exist, or it leads to an empty node, then p_{end} is not a phrase and there are no phrases starting with p_{end} (phrases are built incrementally symbol by symbol in LZ78). If instead we reach a node u^r, with nonempty preorder t, then $i = Node[t]$ is the LZTrie preorder of the corresponding node u, which represents p_{end}. It is also the left end $l_u =$ i of the preorder interval of the descendants of u. To find the right end, r_u, we compute the size ℓ of its interval in Doc_{lz} using $LDoc_{rev}$: $\ell = select_1(LDoc_{\mathrm{rev}}, t+1) - select_1(LDoc_{\mathrm{rev}}, t)$, and then $r_u = l_u + \ell - 1$. Now we have the row interval to search Range.

Finally, we can reduce the space of the RMQs in Range by storing them only for the highest levels of the wavelet tree. The lowest ones have shorter bitvectors, and then traversing them sequentially is not much different from applying Sadakane's algorithm to find the different documents (moreover, as they are closer to the leaves, obtaining their document identifiers is cheaper). This gives a space/time tradeoff.

6.3.1 Experimental Results

We run our experiments on several text collections that were already considered in previous work [92, 67], as well as other larger ones.

- `ClueWiki`: A sample of ClueWeb09. These are Web pages from the English Wikipedia (`boston.lti.cs.cmu.edu/Data/clueweb09/`).
- `Wiki`: A collection of more and shorter documents than `ClueWiki`.
- `KGS`: A collection of sgf-formatted Go game records from year 2009 (`www.u-go.net/gamerecords`).
- `Proteins`: A collection of sequences of human and mouse proteins (`www.ebi.ac.uk/swissprot`).
- `DNA`: A synthetic collection, slightly repetitive with 5% mutations among documents.
- `Influenza`: A repetitive collection of the genomes of influenza viruses. We take the first 70MB.
- `TodoCL`: A collection formed by snapshots of the Chilean Web. This includes real queries, which we use to measure quality. We take the first 100MB for most experiments,

Collection	n (MB)	D	n/n'	compress (bpc)
ClueWiki	131	3,334	17.24	2.78
Wiki	80	40,000	9.58	3.34
KGS	25	18,838	14.97	1.85
Proteins	56	143,244	6.38	4.61
DNA	95	10,000	11.50	2.68
Influenza	70	49,588	21.18	1.89
TodoCL	100	22,850	9.02	3.82
TREC	3500	846,869	19.42	3.74

Table 6.1: Main characteristics of the text collections.

and up to 2.05GB for experiments on collection growth.

- `TREC`: The TREC Corpus FT91 to 94 (`http://trec.nist.gov`). We take the first 3.5GB and use it for experiments on collection growth.

Table 6.1 summarizes the main characteristics of these collections: size n, number of documents D, average LZ78 phrase length n/n' (the larger, the more compressible for our index), and bpc obtained by the LZ78-based Unix `compress` program (another measure of LZ78 compressibility).

The machine used for all experiments is an Intel Xeon with 8 processors of 2.4GHz and 12MB cache, with 96GB RAM. It runs on Linux 2.6.32-46-server, and we use gcc with full optimization.

A 64-bit implementation of our index, *LZ-DLIndex*, is left public [1]. We also use an RMQ implementation of our own[2], which requires around $2.2n$ bits. The bitvector implementations are obtained from the sdsl-library [3].

We also implement the classical DL solution of Sadakane [106], which we also leave public[4]. As the CSA, we use the FM-Index implemented in the sdsl-library, and try different suffix array samplings to obtain space/time tradeoffs.

Space study

Table 6.2 gives the space obtained by our LZ-DLIndex structure on the collections described in Table 6.1. The total bpc for each main component is shown in bold, and between parenthesis its percentage of the total size of the structure. `Influenza`, `ClueWiki` and `KGS` are the most compressible ones, reaching 6.3–8.2 bpc, whereas `DNA`, `Wiki`, `TodoCL` and `Proteins` are the least compressible ones. All are, as roughly expected from the space analysis, 3.7–5.2×|LZ78|, where $|\mathsf{LZ78}| = n'(\lceil \log n' \rceil + \lceil \log \sigma \rceil)/n$. We show how |LZ78| relates to n/n', and how it roughly coincides with the output size of *Compress*, a classical LZW Unix compressor (shown

[1] At `https://github.com/hferrada/LZ-DLIndex.git`.
[2] Available at `https://github.com/hferrada/rmq.git`.
[3] From `https://github.com/simongog/sdsl-lite.git`.
[4] At `https://github.com/hferrada/Sada-DLIndex.git`.

Collection	**RevTrie**	**Doc**	**Node**	**Range**	**Total**	(/|LZ78|)	|LZ78|	(n/n')
ClueWiki	**1.69(23%)**	**3.30(45%)**	**1.33(18%)**	**1.04(14%)**	**7.39**	(4.31×)	1.71	(17.24)
	0.18(topology)	*0.70*(Doc_{lz})						
	1.39(labels)	*2.34*(RMQ_C)						
	0.12(empty)	*0.26*($LDoc_{rev}$)						
Wiki	**1.93(18%)**	**4.38(40%)**	**2.51(23%)**	**2.07(19%)**	**10.89**	(3.68×)	2.96	(9.58)
	0.18(topology)	*1.67*(Doc_{lz})						
	1.39(labels)	*2.34*(RMQ_C)						
	0.12(empty)	*0.37*($LDoc_{rev}$)						
KGS	**2.03(25%)**	**3.65(44%)**	**1.40(17%)**	**1.13(14%)**	**8.21**	(4.56×)	1.80	(14.97)
	0.23(topology)	*1.00*(Doc_{lz})						
	1.61(labels)	*2.36*(RMQ_C)						
	0.19(empty)	*0.29*($LDoc_{rev}$)						
DNA	**1.10(12%)**	**3.87(42%)**	**2.09(23%)**	**2.08(23%)**	**9.14**	(3.89×)	2.35	(11.50)
	0.24(topology)	*1.22*(Doc_{lz})						
	0.80(labels)	*2.32*(RMQ_C)						
	0.06(empty)	*0.33*($LDoc_{rev}$)						
Proteins	**2.06(11%)**	**5.63(37%)**	**3.76(25%)**	**3.76(25%)**	**15.21**	(4.33×)	3.51	(6.38)
	0.44(topology)	*2.82*(Doc_{lz})						
	1.51(labels)	*2.34*(RMQ_C)						
	0.11(empty)	*0.47*($LDoc_{rev}$)						
Influenza	**0.95(15%)**	**3.36(53%)**	**1.04(17%)**	**0.95(15%)**	**6.30**	(5.21×)	1.21	(21.18)
	0.14(topology)	*0.75*(Doc_{lz})						
	0.75(labels)	*2.34*(RMQ_C)						
	0.06(empty)	*0.27*($LDoc_{rev}$)						
TodoCL	**2.05(18%)**	**4.40(39%)**	**2.66(23%)**	**2.21(20%)**	**11.32**	(3.24×)	3.40	(9.02)
	0.35(topology)	*1.66*(Doc_{lz})						
	1.51(labels)	*2.35*(RMQ_C)						
	0.19(empty)	*0.39*($LDoc_{rev}$)						

Table 6.2: Space breakdown of the main components in our LZ-DLIndex structure, with values in bpc. For RevTrie and Doc columns the space is the sum of the components detailed below them (bpc values in italics). The Range columns does not include the RMQ structures to speed up the index. The percentages refer to the total size of the index. The column (/|LZ78|) indicates the ratio of the total size over |LZ78|, and the last column, in turn, gives also (n/n').

in Table 6.1).

The Doc component dominates the space, with at 37%–53% of the total index size. It includes the document identifiers with their boundary values and the RMQ_C data structure on RevTrie. For Range we used the lowest and smallest version of the index, where the wavelet tree of Range does not include any RMQ structure (this corresponds to the highest point of the LZ-DLIndex in Figures 6.3 and 6.4). Range uses 15%–25% of the index size. The distribution varies a bit on the less compressible collections, where the fraction of Node and Range increases, reaching 25%. Note that component Range can be omitted if we only want to list the occurrences of type 1, in which case the index size is reduced by 15%–25%.

Table 6.3 shows the number of documents listed by the queries, averaging over 3,000 patterns randomly extracted from the collections. Many of the listed documents are obtained as type-1 occurrences (70%–96% for $m = 6$, and 50%–92% for $m = 10$ if we exclude DNA). This shows that we could obtain a significant part of the result using just the fastest listing and without representing Range.

Collection	$m = 6$ Type 1	Type 2	Type 3	ndoc	$m = 10$ Type 1	Type 2	Type 3	ndoc	D
ClueWiki	**1860.51**	**69.37**	**0.24**	**1930.12**	**1437.01**	**119.79**	**2.05**	**1558.85**	**3,334**
	96.4%	*3.6%*	*0.0%*	*57.89%*	*92.2%*	*7.7%*	*0.1%*	*46.76%*	
Wiki	**921.66**	**290.04**	**0.16**	**1211.86**	**135.79**	**76.50**	**0.97**	**213.26**	**40,000**
	76.1%	*24.0%*	*0.0%*	*3%*	*63.7%*	*35.9%*	*0.5%*	*0.5%*	
KGS	**4702.26**	**1691.87**	**1.40**	**6395.53**	**2012.27**	**739.57**	**4.66**	**2756.49**	**18,839**
	73.5%	*26.5%*	*0.0%*	*33.9%*	*73.0%*	*26.8%*	*0.2%*	*14.63%*	
DNA	**7527.03**	**1630.21**	**0.01**	**9157.25**	**32.72**	**98.37**	**0.14**	**131.22**	**10,001**
	82.2%	*17.8%*	*0.0%*	*91.56%*	*24.9%*	*75.0%*	*0.1%*	*1.31%*	
Proteins	**52.01**	**21.53**	**0.07**	**73.61**	**25.57**	**16.59**	**3.50**	**45.66**	**143,244**
	70.7%	*29.2%*	*0.0%*	*0.05%*	*56.0%*	*36.3%*	*7.7%*	*0.03%*	
Influenza	**16901.13**	**3302.72**	**0.09**	**20203.94**	**995.46**	**452.63**	**1.18**	**1449.27**	**49,588**
	83.7%	*16.3%*	*0.0%*	*57.89%*	*68.7%*	*31.2%*	*0.1%*	*2.92%*	
TodoCL	**467.09**	**173.88**	**0.16**	**641.13**	**35.85**	**35.34**	**0.93**	**72.12**	**22,850**
	72.9%	*27.1%*	*0.0%*	*2.81%*	*49.7%*	*49.0%*	*1.3%*	*0.03%*	

Table 6.3: Number of occurrences of each type, for pattern lengths $m = 6$ and $m = 10$. Under each number, we give the percentages of the documents output. For the three types of occurrences these refer to ndoc, and for column ndoc this refers to D.

Space/time tradeoffs

Figures 6.3 and 6.4 compare our LZ-DLIndex structures in three modes: (i) the full mode where it returns all the documents for a DL-query (called LZ-Index in the plots); (ii) a mode where it also can return all the documents but we take the time needed to return only those that were found for occurrences of type 1, and use the minimum space for Range (called "up to type 1"); and (iii) a mode where it can only return the documents found by occurrences of type 1 as it does not store Range at all (called "only type 1"). For the full mode, we obtain a space/time tradeoff by representing RMQs only for the highest levels of Range, as explained.

We also compare Sadakane's DL structure [106], showing seven points that use suffix array sampling steps of 4, 8, 16, 32, 64, 128 and 256. We also compare some variants of the proposal that stores a wavelet tree of the document array [92]: (i) the variant using document arrays as plain wavelet trees [114] (WT Plain), (ii) a representation with grammar-compressed wavelet trees (WT RePair), and (iii) an intermediate one called WT Alpha[5]. In order to compute the occurrences interval $SA[l, r]$ in this index we incorporate a CSA with no sampling in order to minimize space (the sampling is not needed here). We use a FM-Index as the CSA in the Sadakane's proposal.

It can be seen that adding RMQs to Range, while theoretically appealing, increases the space without giving a significant speedup in practice. Our LZ-DLIndex is between, on one extreme, Sadakane and WT RePair, which use less space but may be orders of magnitude slower, and on the other extreme, WT Plain, which is orders of magnitude faster but uses much more space. In some collections, like ClueWiki, Wiki, DNA, and TodoCL, WT RePair outperforms the LZ-DLIndex in both time and space, whereas in KGS, Proteins, and Influenza, the LZ-DLIndex is much faster. The LZ-DLIndex is comparable to WT Alpha in various cases, but it is much easier to tune.

[5]We ran the 32-bit code given by the authors [92], which can build the variants (i) and (ii) for any data collection. The "alpha" structure could be built only on the four data collections used in their publication, which include ClueWiki, KGS, and Proteins.

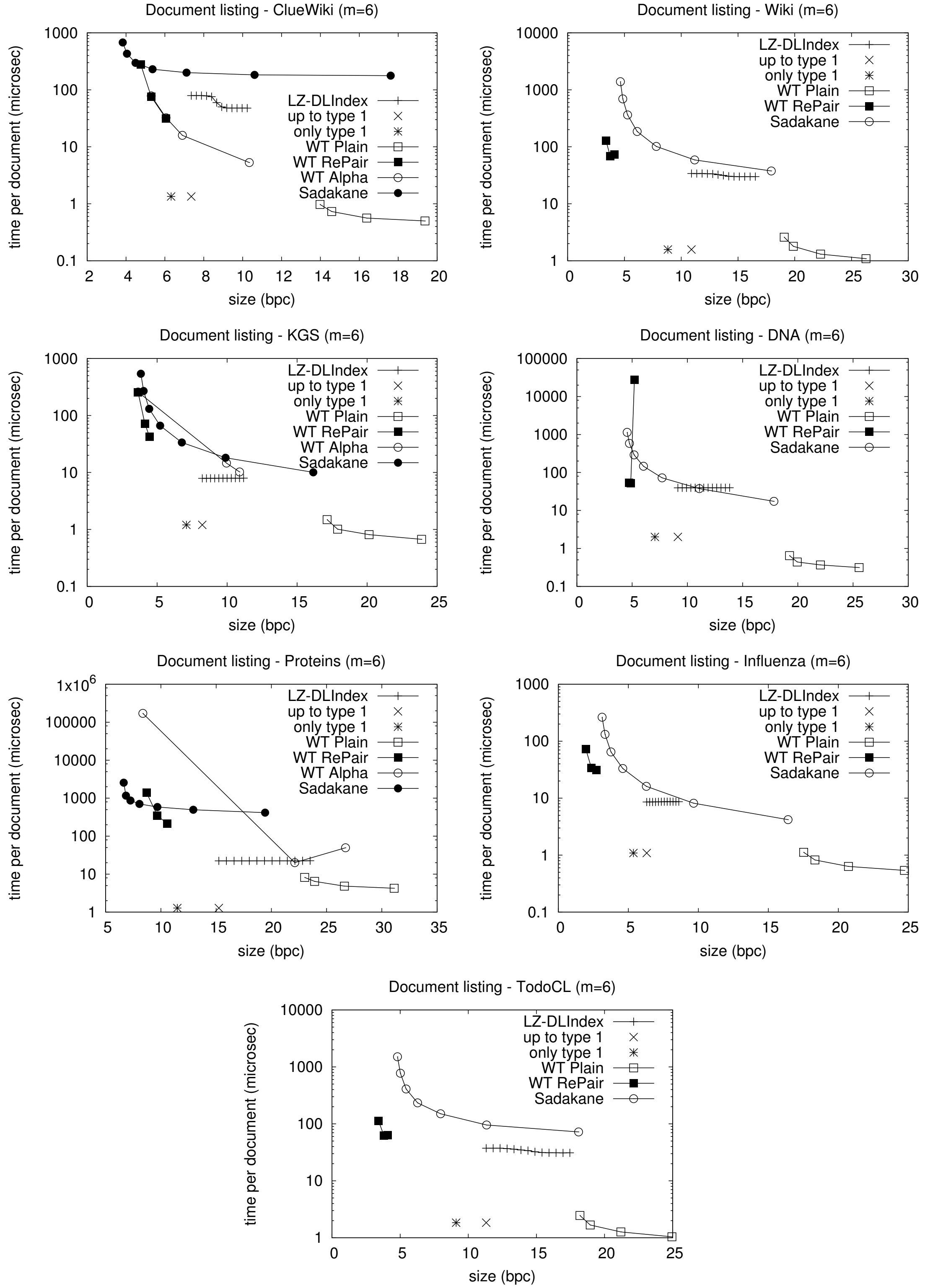

Figure 6.3: Space/time comparison for pattern length $m = 6$.

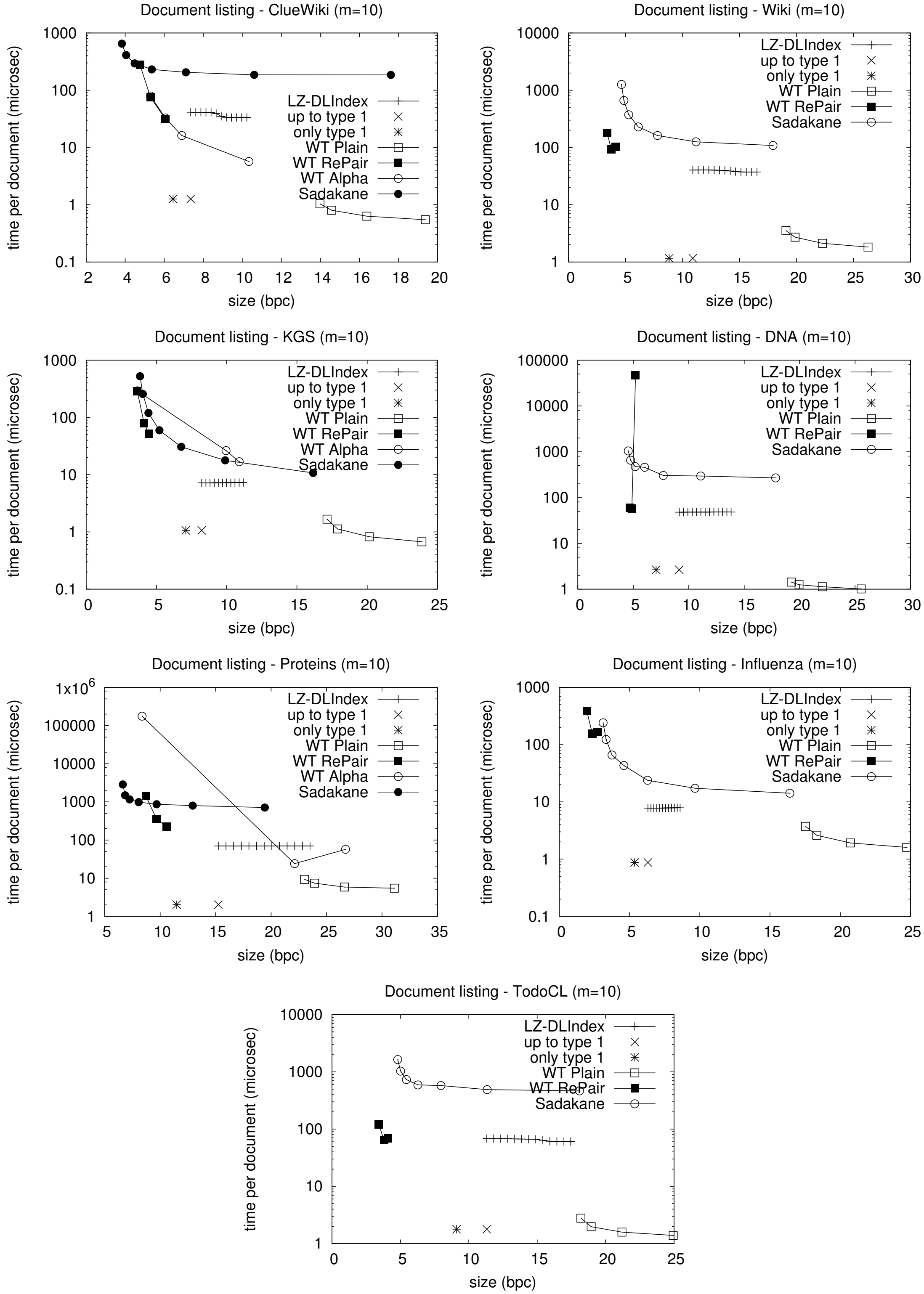

Figure 6.4: Space/time comparison for pattern length $m = 10$.

The non-full variants of the LZ-DLIndex reach much better time, similar and even faster than those of WT Plain. They return about one result per microsecond. Therefore, in scenarios where we return the occurrences progressively, for example to be displayed in an interface, the "up to type 1" structure is very efficient, as it retrieves the first occurrences very fast.

The variant that can only return the occurrences of type 1 is also significantly smaller. Next we study the fraction of the total set that is found with this type of occurrences.

Quality

Now we measure the quality of our small and fast approximation of the LZ-DLIndex. As explained, it returns the documents where p is contained in at least one full phrase. Our analysis showed that, as n grows, the fraction of these documents should asymptotically approach the total answer set.

Figure 6.5 explores the behavior on a large collection, `TodoCL`, with a real query log. We tested one-word and two-word queries. As expected, the ratio of documents returned grows with n and decreases with the query length. When we reach the 2.2GB, the approximation returns about 80% of all the answers. Note that this was already around 75% on just 200MB.

6.4 Conclusions

This chapter has introduced the first document listing data structure based on Lempel-Ziv compression. Apart from offering a competitive space/time tradeoff in general, an interesting feature of the index is its ability to retrieve a large number of documents very fast. The index outputs the most of the documents only in a few microseconds and always using less than 7 bpc (excluding `Proteins` which is less compressible). The structure is also able to output the complete answer at cost of extra size and time. However it continues to offer competitive trade-off between time response and space usage. This makes it an ideal choice in interactive scenarios, where one must show some answers immediately and others can be calculated in the background, and in cases where only some answers are sufficient.

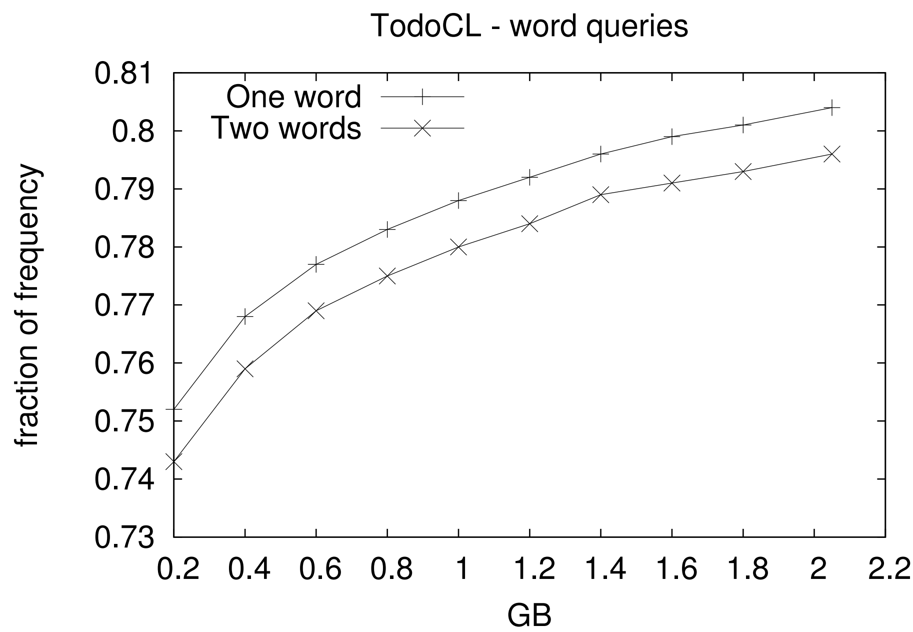

Figure 6.5: Fraction of the real answer of our LZ-DLIndex for real queries, as a function of the prefix size of `TodoCL`in GB, for words and phrases of two words.

Chapter 7

An LZ-based Index for Top-k Retrieval

This chapter introduces the extended version of the index described in Chapter 6, which computes approximate answers for top-k queries. We detail both how to retrieve the complete output for a top-k query, and an approximate answer. This work was published in the 21th International Symposium on String Processing and Information Retrieval [31]. A journal version, together with the results of Chapter 6, was submitted to Information and Computation.

The LZ-DLIndex of Chapter 6 is fast to retrieve a large portion of the documents where p appears, and estimating the top-k from this list might also yield a good approximation to the top-k set where p occurs most often. Our approximate top-k index, LZ-TopkApp, builds on this idea to be small and fast, close to the LZ-DLIndex variant "only type 1". Compared to that index, it also stores the top-κ documents for some RevTrie nodes, for a κ that depends on the number of occurrences of the node. A top-k query on that node where $k \leq \kappa$ can simply return the first k precomputed answers. Otherwise, we solve the query by brute force, extracting all the occurrences of type 1.

This idea is has some resemblances with the succinct approach of Hon et al. [58, 92], which stores this information for some chosen suffix tree nodes. A parameter g determines the RevTrie nodes that will store their top-κ answer. The (empty or nonempty) RevTrie nodes representing a string with at least g occurrences of type 1 are then marked in a bitvector B_{top}. Yet, we never mark empty unary nodes because their set of occurrences is the same as for their child. Each marked node, however, stores a different number κ of documents where it appears most often: if the node has o occurrences of type 1, then it stores $\kappa = \lfloor o/g \rfloor$ precomputed answers. This guarantees that, if $k > \kappa$ and we need to find all the o occurrences by brute force, it is because p has less than gk occurrences of type 1, and thus the effort to collect them individually is no more than $O(g)$ per result returned.

The exact top-k index, LZ-TopkIndex, also stores the structures to collect the occurrences of type 2 and 3 by brute force.

7.1 Structure

The structure of LZ-TopkIndex includes the LZTrie, RevTrie, and Node components of the LZ-DLIndex. We also include Doc_{lz}, but not Doc_{rev} nor $LDoc_{rev}$. In addition we include Range, to find the occurrences of type 2, but not the RMQ structures the LZ-DLIndex associates with it. In exchange, we include a new structure, *Top*, where the precomputed top-κ answers are maintained. Apart from the bitvector B_{top} that tells which nodes are marked, we store the top-κ answers for each marked node in an array K_{top}, and mark the beginning of each answer set in a bitvector LK_{top}. The detail is as follows:

B_{top}**:** A bitvector marking which RevTrie nodes have top-κ answers precomputed, in preorder.

K_{top}**:** The sequences of κ most frequent documents where each node marked in B_{top} appears, concatenated in the same order of B_{top}. The identifiers are stored using $\lg D$ bits, in decreasing frequency order.

LK_{top}**:** A bitvector marking the starting positions of the sequences in K_{top}.

A_{top}**:** Since there may be less than κ distinct documents where the marked node appears, this bitvector indicates whether a node marked in B_{top} already lists all of the possible documents.

Space. The larger g, the fewer RevTrie nodes store their top-κ documents. Consider a RevTrie node. If it has o occurrences, then it stores $\kappa \leq o/g$ precomputed answers (including zero, being not marked, if $o < g$). Adding over all the RevTrie nodes representing strings of the same legnth, no more than n/g precomputed results are stored, since the occurrences must be disjoint and can only add up to n. Therefore, if h is the maximum length of a phrase (or, equivalent, the height of LZTrie), we can have n/g results per length, adding up to $h(n/g)\lg D$ bits in total. If we assume that the text is generated by a memoryless source, then the LZTrie can be thought of as the trie induced by n' infinite and statistically independent strings. Under a wide set of probabilistic models, the height of such a trie is $O(\log n')$ [112]. The result still holds if T is generated by a finite-memory source, where each symbol depends on $O(1)$ previous symbols.

Under these assumptions, the space of Top structures is $O((n/g)\log n' \log D)$ bits. By choosing $g = \Theta(\log n' \log D)$, this space becomes $O(n)$ bits. The bitvectors B_{top}, LK_{top} and A_{top} add just other $O(n')$ bits. Thus the overall space is $n'(\lg n' + \lg D + 2\lg\sigma) + O(n) \leq 2nH_k(T) + O(n) + o(n\lg\sigma)$ bits.

7.2 Queries

At query time, we perform in RevTrie an optimal time search for the locus v^r of p. This means that there are at least one occ_{t1} for p which are represented by v^r. We have the same precautions than the searches with the LZ-DLIndex, with edges that represent more than

one symbol in RevTrie, because we also contracted paths for empty nodes. And we need to validate using U_{rev} these nodes and to retrieve the extra letters from M_{rev} in order to compare them.

Once that we have retrieved v^r, the next step is to check B_{top} to see if v^r is a marked node (i.e., the index contains stored the top-k answers for some k^* in K_{top}). If that happens, by $rank/select$ queries on LK_{top} we obtain the range of document identifiers stored k^*. So, if this range indicates that it stores $k' \geq k$ top documents, we return the top-min(k, κ) documents stored for v^r in K_{top} and finish. Otherwise, we check in A_{top} if v^r stores all the documents for p, and return the complete range of identifiers stored in K_{top}.

Otherwise, the stored answers are not sufficient (or do not exist or when v^r is not found) and we have to proceed by brute force. Then, just as for the basic index, we collect all the occurrences of type 1, by construction, this takes place only if v^r has $k' < k$ answers stored (including the case $k' = 0$). This indicates that if $k^* < k$ is the power of 2 closest to k, then v^r does not store its top-k^* answer (perhaps there is a list stored for a lower k^*). We map every descendant u^r of v^r to node u in LZTrie using Node, and traverse the range of Doc_{lz} covered by u. In order to obtain each identifier document for v^r sequentially, we use P_{rev} to compute its preorder i_v and its subtree size s_v. Thus all the subtree of v^r has the preorder interval $[i_v, i_v + s_v - 1]$. We then use $rank$ on E_{rev} to map it to the interval $[i_1, i_2]$ of nonempty preorder values. For each i in this interval, we compute $i_u = Node[i]$, which is the preorder of the corresponding node in LZTrie, and then use P_{lz} to obtain the corresponding node u in LZTrie. Then, we similarly compute the size s_u of u from P_{lz} and obtain the interval $[i_u, i_u + s_u - 1]$ of all the descendants of u in LZTrie. We process all the document identifiers in $D_{lz}[i_u, \mathrm{i}_u + s_u - 1]$, for all the nodes u in LZTrie that correspond to all the RevTrie descendants u^r of v^r. Along this process, we accumulate the frequencies of the documents found in an initializable array [81, Sec. III.8.1], and at the end collect the k documents with the highest frequencies. In case of the LZ-TopkIndex, we also collect the occurrences of type 2 and 3, to ensure that the answer is completely correct.

Figure 7.1 illustrates the main components of our index and how we retrieve top-k answers in both cases: when the locus of the pattern contains the answer precomputed, or when the output is computed by brute force.

Time. The LZ-TopkIndex guarantees to spend $O(g \lg n')$ time per occurrence returned when p has occurrences of type 1. Otherwise, there is no guarantee. However, let us follow the analysis of Section 6.2. On texts generated by ergodic sources, the probability that p, appearing o times in T, has no occurrences of type 1, is $(1 - \Theta(m/\log_\sigma n))^o$. Taking the worst value $m = 2$ and multiplying by the cost $O(o \log n')$ to find all such occurrences, this is upper bounded by $\mathrm{e}^{-\Theta(o/\log_\sigma n)} o \log n$, which is maximized for $o = \Theta(\log_\sigma n)$. Thus we absorb this case, on average, by adding $O(\log^2 n)$ time. Considering the time for searching the tries and handling the occurrences of type 3, we obtain $O(m \log^2 n + kg \log n)$ time. The LZ-TopkApp structure, instead, reports nothing when p has no occurrences of type 1, and otherwise spends $O(g)$ time per occurrence returned. Thus its total time is always $O(m + kg)$.

If we assume that the text is generated by a memoryless source, then the LZTrie can be thought of as the trie induced by n' infinite and statistically independent strings. Under a

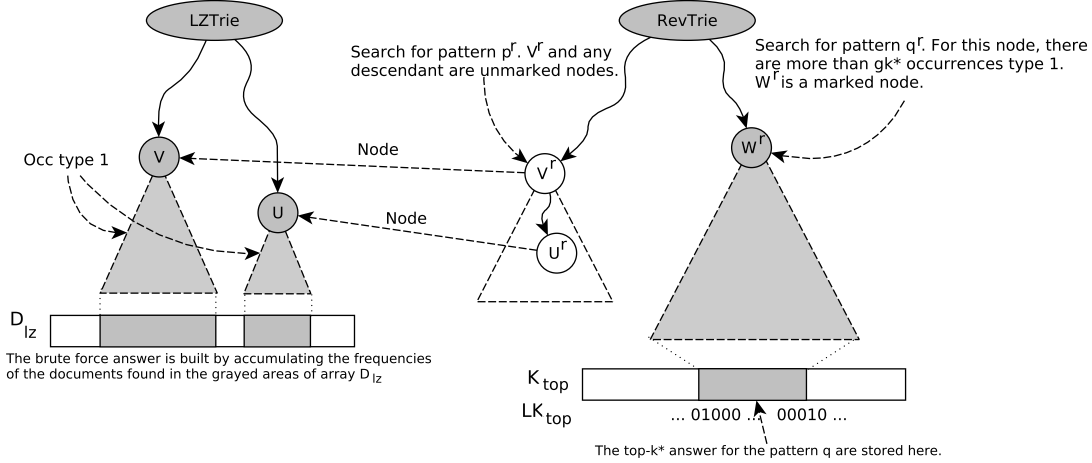

Figure 7.1: The main data structures of our approximate top-k index. The search for the pattern q^r reaches node w^r in RevTrie, which is marked in B_{top}. The marks in LK_{top} also indicate that there are $\kappa \geq k$ document identifiers stored. Therefore, the answer is retrieved from K_{top} using the marks in LK_{top}. The search for p^r, instead, reaches node v^r in RevTrie. Since this node is not marked, the answer is computed by accumulating frequencies from the document array of phases, D_{lz}. We use $k*$ for κ in the drawing.

wide set of probabilistic models, the height of such a trie is $h = O(\log n')$ [112]. The result still holds if T is generated by a finite-memory source, where each symbol depends on $O(1)$ previous symbols. Therefore, we have that $g = h \lg D = O(\log^2 n)$. Our previous calculations then yield time $O(m \log^2 n + k \log^3 n)$ for LZ-TopkIndex and $O(m + k \log^2 n)$ for LZ-TopkApp.

7.3 Improving the Quality

A simple way to improve the quality of the approximation is to note that, when top-κ answers are precomputed, we can perfectly do the precomputation for the *actual* top-κ answers, instead of using just the occurrences of type 1. Note that the length of these stored sequences is the same, then the size for K_{top} will be maintained. We just need to locate the occurrences for some RevTrie's paths and accumulate the document frequencies to determine the top-ranked documents. This process can be implemented by using, at construction, a temporal self-index of the collection. Later, to sort the the document frequencies we can use a maximum priority queue of length k. The process needs extra time to build the self-index and compute each top-k answer for marked nodes. Then, in practical terms, the cost to have a better answer will imply that the construction time will increase. However, the structure will retrieve the right answer when it finds it precomputed. It will only give an approximation when it has to scan all the occurrences of p one by one.

7.4 Experimental Results

We leave public a 64-bit implementation of our index, called LZ-TopKApp[1]. We also included the version that gives the full answer for Top-k, called LZ-TopkIndex[2]. We compare our index with previous work [67, 92] in terms of query time and space usage. We use the same document collections described in Table 6.1.

7.4.1 Space study

Figure 7.2 gives the space breakdown for our index, for various values of g. We group the data structures in four components: (1) LZTrie contains the tree topology and the document identifiers Doc_{lz}; (2) RevTrie considers the tree topology, the symbols of the edges, and the other bitvectors to perform pattern searches; (3) Node is the array mapping RevTrie to LZrie; and (4) Top counts the storage of the best documents for marked nodes and the bitvectors to extract them. Only the size of Top varies with g. It can be seen that reasonable values of g, depending on the collection, start at 32–256. The impact of g is slightly smaller on DL-TopkIndex.

7.4.2 Space/time tradeoffs

We compare our top-k indexes with the best previous solutions. We denote IDX-KN the implementation [67] of a fast and large structure [89] (there is an even more recent implementation [49], but it is not very different from the one we show in the range of interest of this paper). We also include a choice of relevant space/time tradeoffs from the small and slow structure based on Hon et al.'s sampling [58] combined with wavelet trees [92], which we call HON-WT.

We consider search patterns of lengths $m = 6$ and for $m = 10$ in Figures 7.3 and 7.4. We take strings from random positions in the collection, checking that they appear in at least k documents[3]. We test $k = 10$ and $k = 100$. For LZ-TopKApp, we try the values $g = 256, 128, 64 \ldots$ until the size of component Top exceeds 24 bpc. For LZ-TopkApp we also include the case $g = +\infty$ (i.e., not precomputing any answer) to see if storing answers is worth the space.

Since n/n' is the average node depth in LZTrie, we set $g = (n/n') \lg D$ as a natural value. According to Table 6.1, this yields values in the range 110–330 for g. In most texts LZ-TopkApp uses 4–7 bpc with those values of g (except `Proteins`, where it uses 10 bpc), and solves top-k queries in around k–$5k$ μs. LZ-TopkIndex uses 5–8 bpc (12 bpc on `Proteins`) and solves queries in $10k$–$100k$ μs. Using a smaller g improves performance significantly in

[1] At `https://github.com/hferrada/LZ-AppTopK.git`.

[2] At `https://github.com/hferrada/LZ-TopK.git`.

[3] It does not make much sense to compute the top-k documents for patterns whose DL-list is shorter than k.

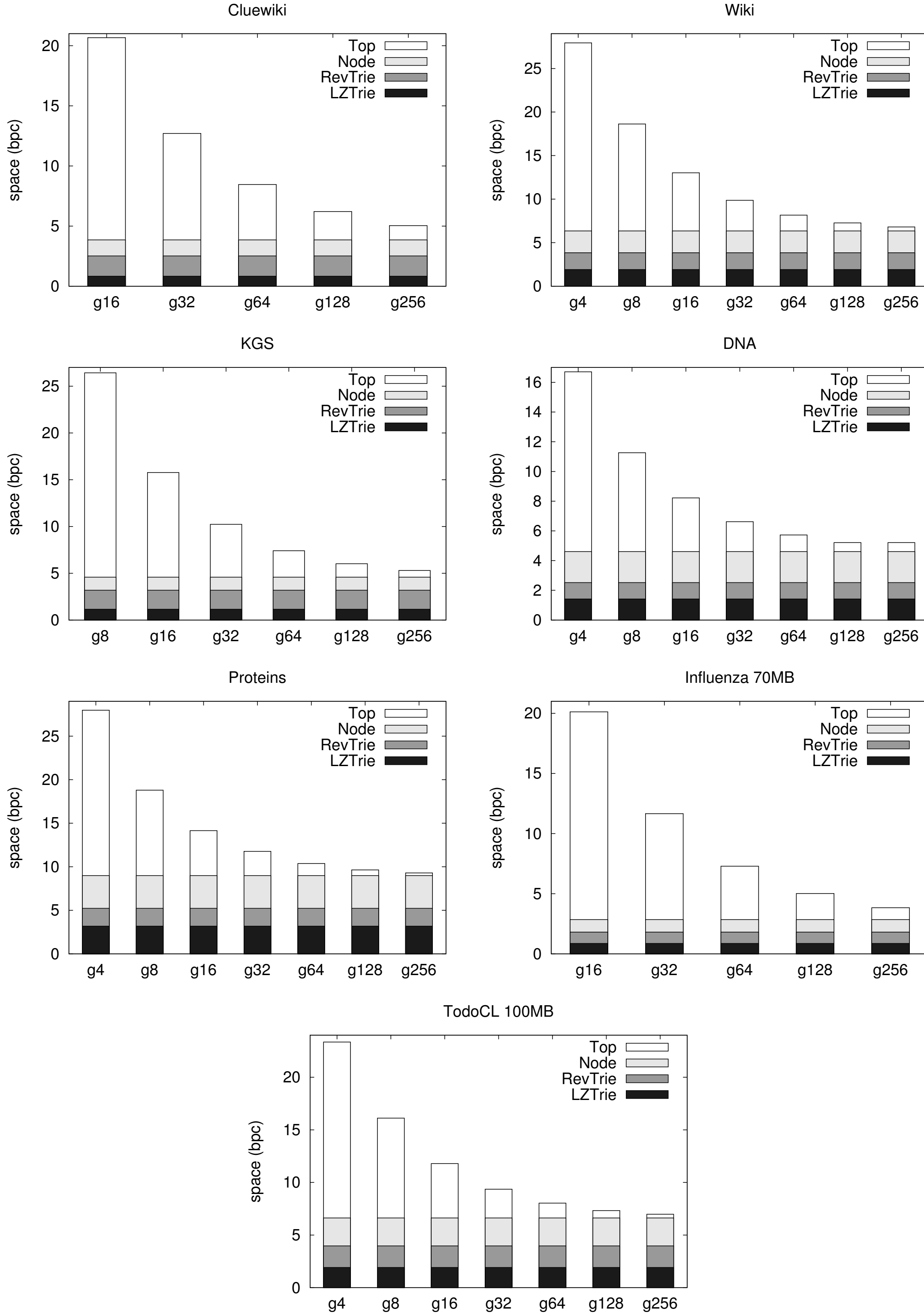

Figure 7.2: Space breakdown of our structures for different g values (g is the x-axis).

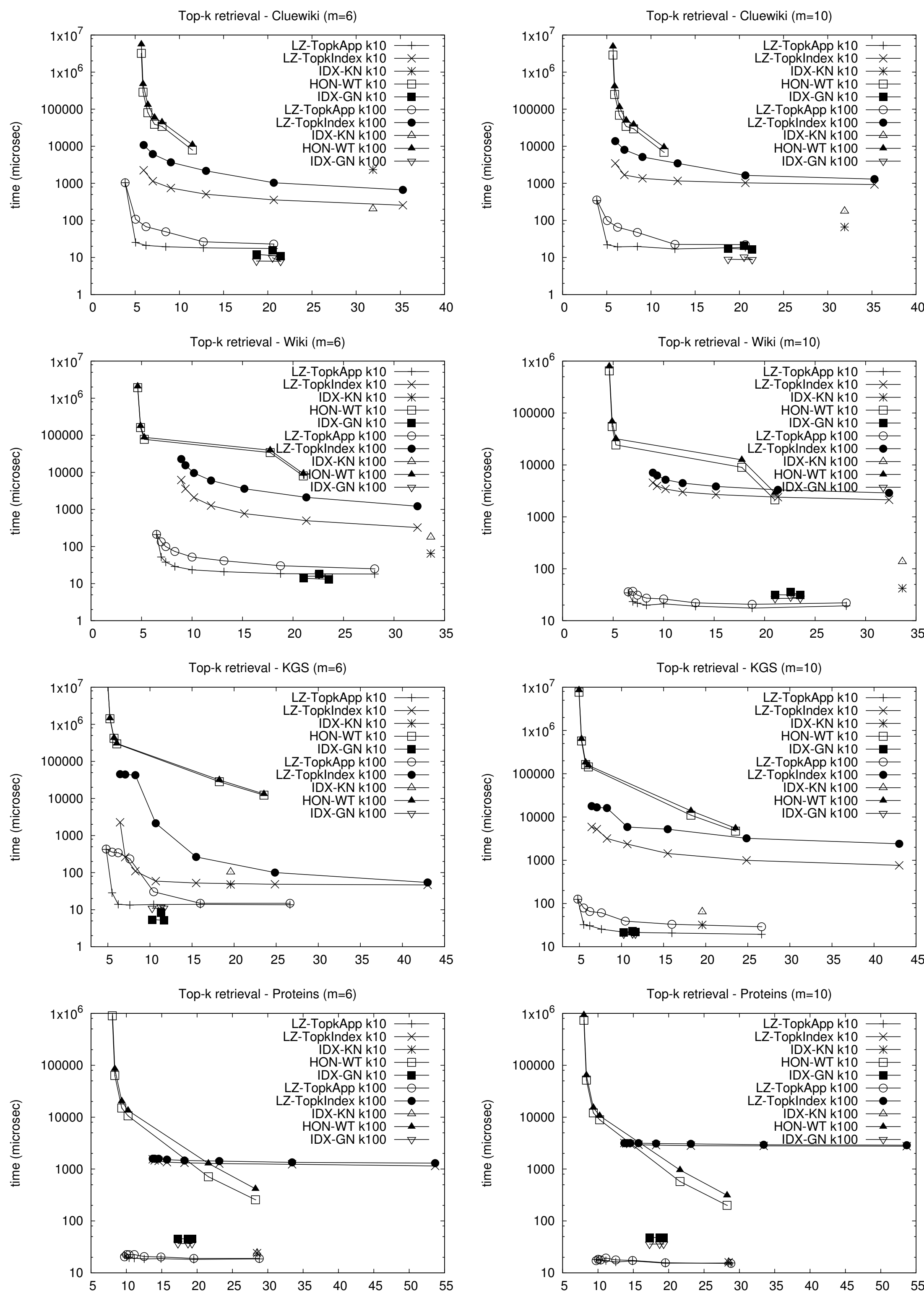

Figure 7.3: Space/time comparison for pattern length $m = 6$ (left) and $m = 10$ (right). Space (bpc) is the x-axis.

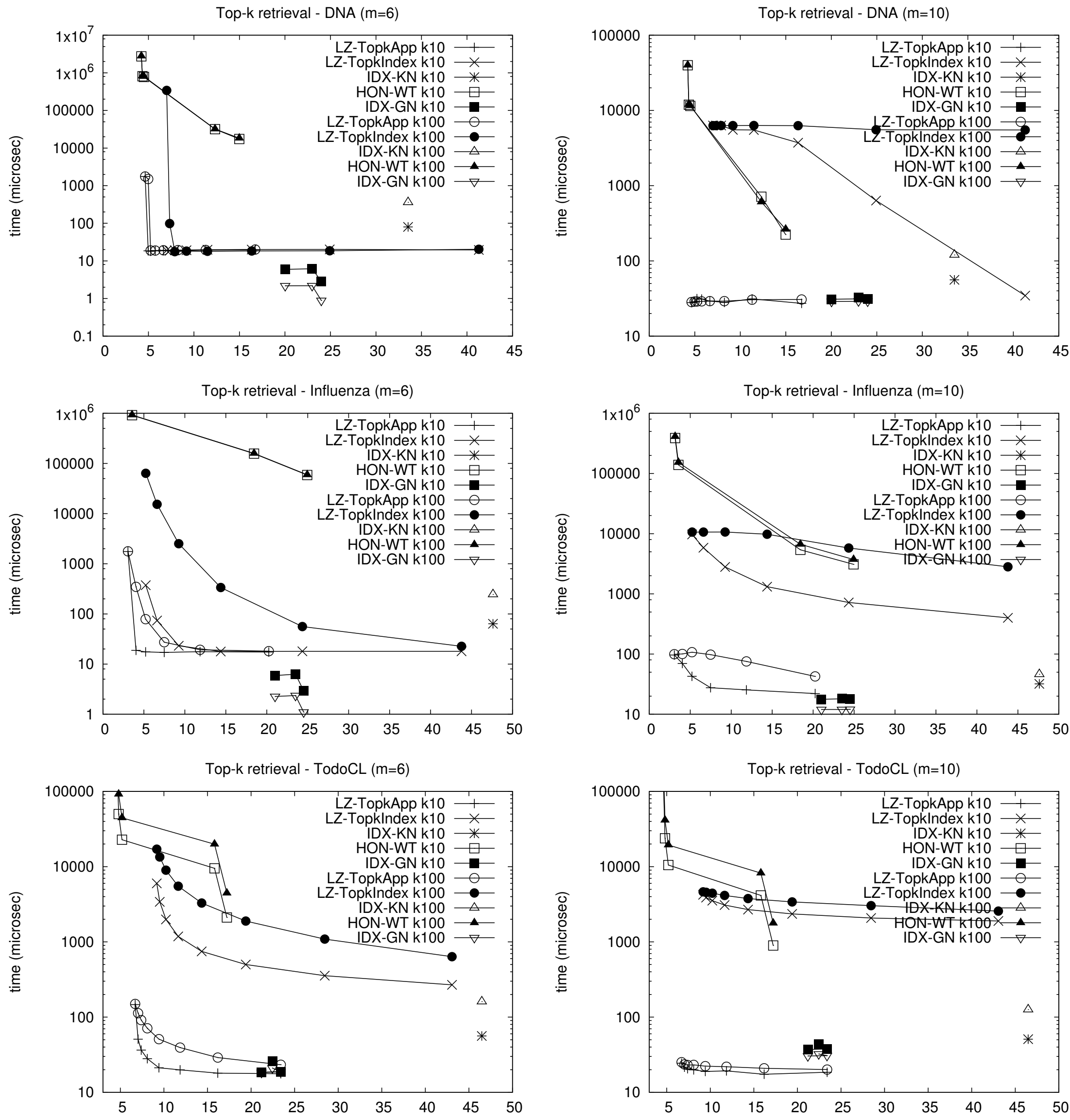

Figure 7.4: Space/time comparison for pattern length $m = 6$ (left) and $m = 10$ (right). Space (bpc) is the x-axis.

some cases, while increasing the space still within competitive bounds. Instead, not using top-κ answers at all significantly increases the times.

The structure HON-WT can use similar or less space than LZ-TopKApp, but at the cost of being 3–6 orders of magnitude slower. Even if using much more space, HON-WT is at least 2 orders of magnitude slower than LZ-TopKApp. On the side of the large and fast structures, IDX-GN obtains time similar to LZ-TopkApp, but it uses 2–4 times more space.

7.4.3 Quality

Our LZ-TopkApp index offers an excellent space/time tradeoff. However, it does not always ensure that the answer is completely accurate. In order to estimate how good the approximation is, we computed two measures of quality for the top-k approximation. The first one is the traditional *recall*, measured in the following way: for each value $k' \in [1, k]$, we measure how many of the (correct) top-k' documents are reported within the (approximate) top-k results. This is shown in Figure 7.5. In this experiment we have selected the largest g value for each collection, which ensures that the total size of the index is around 12–16 bpc.

The point at $k' = 1$ (i.e., 0.1 in the x-axis for $k = 10$ and 0.01 for $k = 100$) indicates how many times the most relevant document is contained in the top-k approximate answer. The point at $k' = k$ (i.e., 1.0) gives traditional recall: how many of the correct top-k documents are actually returned.

This indicator is useful for applications where the top-k answer is postprocessed with a more sophisticated relevance function in order to deliver a final answer of $k' \ll k$ results. For example, except for $m = 10$ on `Proteins` (where few occurrences of type 1 are found), we obtain a recall of 70%–100% if we use this top-k approximation to later extract the best 30% of the results (0.3 in the plots).

In most collections the recall is 60%–100% even for $k' = k$ (except on `Proteins` and `DNA`, which do not compress well). There are no large differences between $k = 10$ and $k = 100$. When there are, the quality is much better for $k = 100$.

If our index fails to return a top-k document, but returns another one with the same frequency, we take it as a hit, as both are equally good. In this sense, recall is too strict of a measure of relevance: if the system returns a document with only slightly fewer occurrences than the correct one, it counts as zero. As the frequency is only a rough measure of relevance, a fairer measure of quality is the sum of the frequencies of the documents in the approximate top-k answer as a fraction of the sum in the correct top-k answer. This is the second indicator we compute. We omit the figures because the improvement is not that large compared to recall: now we obtain 70%–100% of quality for $k' = k$ (except for `Proteins` and `DNA`, which do not improve much), and 80%–100% for $k' = 30\%$ of k (except for `Proteins`).

On the other hand, the fact that better quality is obtained for shorter patterns coincides with our probabilistic analysis. Figure 7.6 illustrates this effect more closely, for increasing pattern lengths (using our second measure of quality from now on). For the moderate col-

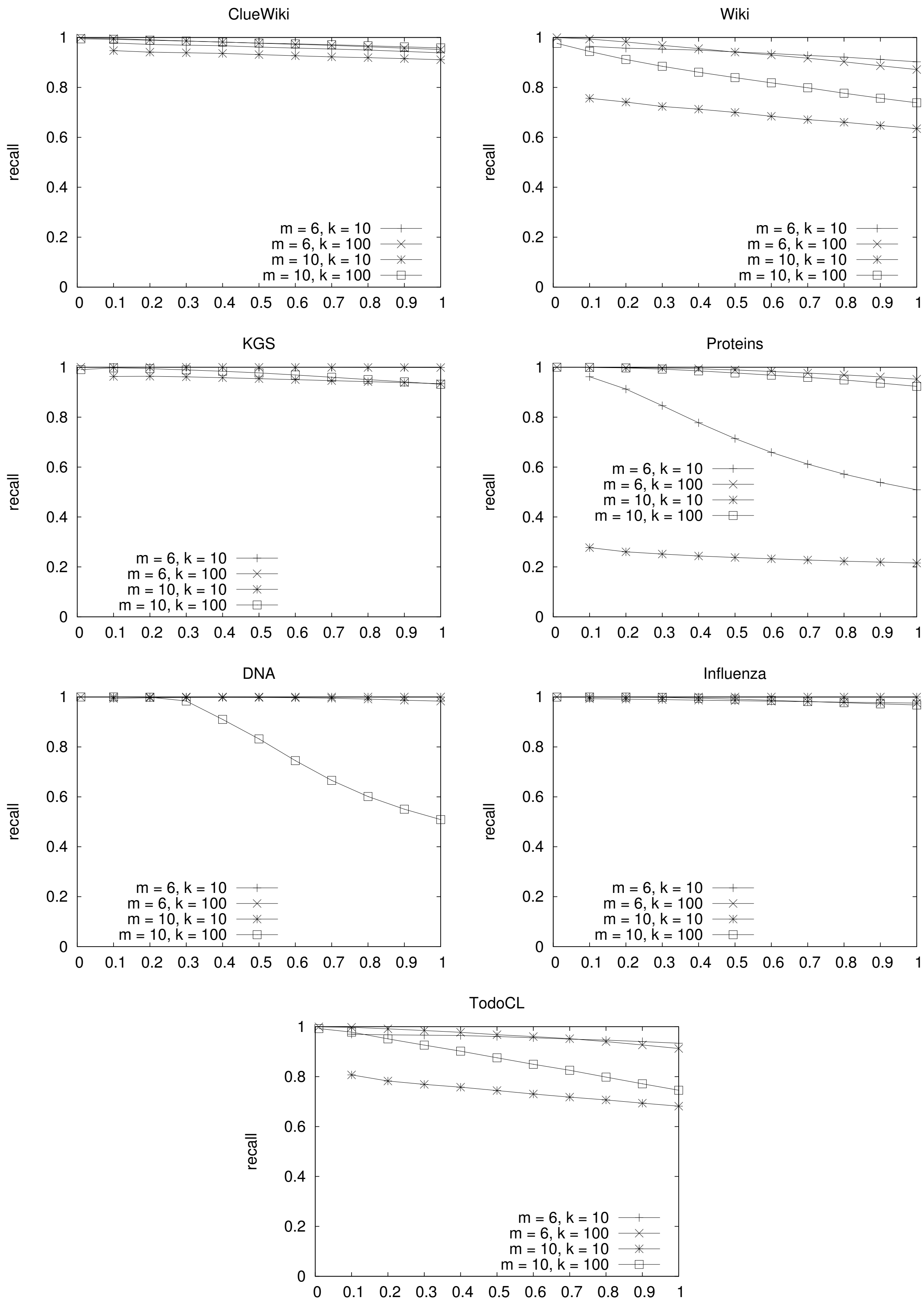

Figure 7.5: Recall of our approximate top-k solution, as a function of the fraction of the answer (x-axis).

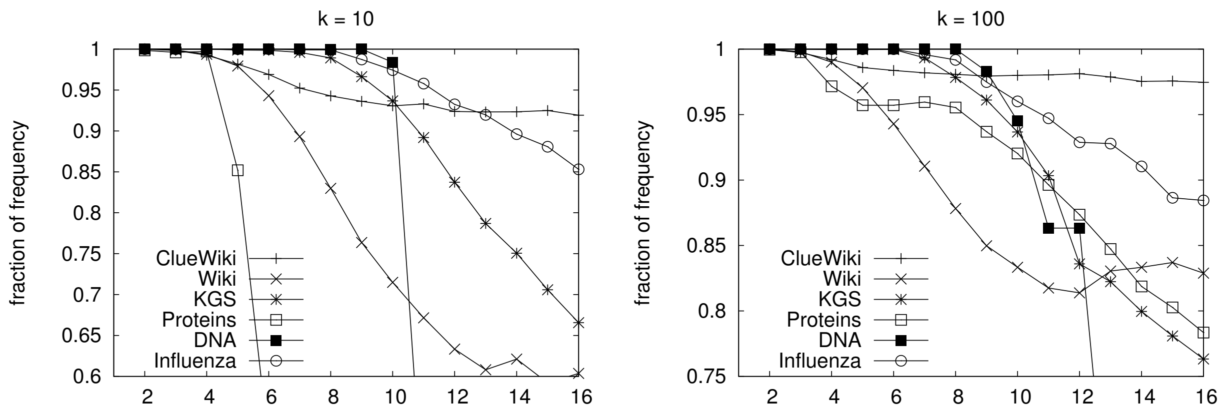

Figure 7.6: Quality of our approximate top-k solution, as a function of the pattern length, for top-10 (left) and top-100 (right). Each pattern appears at least in k documents.

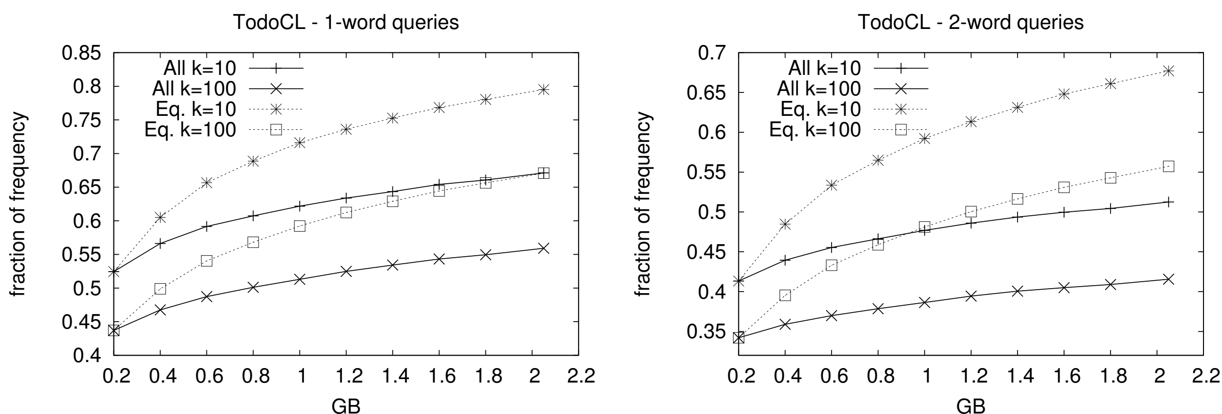

Figure 7.7: Fraction of the real answer found by LZ-AppTopK for real queries, as a function of the prefix size of `TodoCL` for words (left) and phrases of two words (right). Solid lines include new sets of query-patters for each prefix (labels "All k"). Dashed lines consider always the same set of query-patterns from the first 200MB of the collection (labels "Eq. k").

lection sizes of 25–130 MB we considered, we obtain quality well above 80% for $m = 2$–8 in top-10 (`Proteins`, again, is the exception). In most of the collections, the quality is over 90% for $m \leq 10$. For top-100, we obtain quality well above 80% for $m \leq 14$ (except for `DNA`, where the results are good only up to $m \leq 12$).

Our analysis also predicts that the quality improves as n grows. In the next experiment we build the structure for increasing prefixes of `TodoCL`. Figure 7.7 (solid lines) shows the quality obtained for real query words (of length > 3 to exclude most stopwords), with average length 7.2, and 2-word phrases, with average length 8.0. We convert `TodoCL` to lowercase (as the distinction is generally not made in natural language queries). As predicted, the quality improves with n, from 44%–52% on 200MB ($n/n' = 10.1$) up to 56%–67% on 2.05GB ($n/n' = 12.7$) for words; and for 2-word phrases from 34%–42% on 200MB up to 42%–52% on 2.05GB.

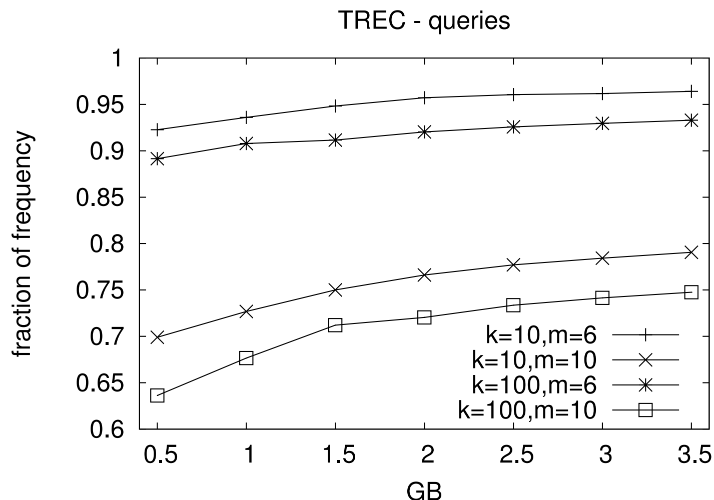

Figure 7.8: Fraction of the real answer found by LZ-TopkApp as a function of the prefix size of `TREC`, for arbitrary patterns of lengths 6 and 10, in top-10 and top-100.

The percentages are much lower than before, because many queries may appear just a few times in the collection. In those cases, a brute-force pattern matching is a better approach. Our LZ-TopKApp index performs better then the words appear many times, and thus a top-k query is more relevant. Figure 7.7 (dashed lines) repeats the experiment, but now we only use patterns that appear in the first 200MB, to query the structure for all the prefixes. The results are much better because the queries appear more often.

In the last experiment, we measure the improvement with n without the problem of real queries that may appear infrequently, and with another large text collection, `TREC`. We extract patterns of lengths $m = 6$ and $m = 10$ from random text positions, and that appear at least in k documents, for $k = 10$ and $k = 100$. The resulting quality is shown in Figure 7.8. Once again, our index gives an answer of high quality on large text collections.

7.5 Conclusions

We have introduced a top-k retrieval index for general string collections, based on Lempel-Ziv compression. Our implementations achieve competitive space/time tradeoffs compared to existing solutions, dominating a significant part of the space/time tradeoff map. The approximate variant of our index (LZ-TopkApp) is orders of magnitude faster, and uses much less space, than previous work. It typically uses 4–7 bpc and returns each result in about 1–5 microseconds. As its results are approximate, we have shown that its quality improves asymptotically with the size of the collection, reaching over 90% of the accumulated term frequency of the real answer already for patters of length ≤ 8 on rather small collection, and improving for larger ones. This makes our index an ideal low-cost structure to obtain a quick and rough approximate top-k answer, which might then be postprocessed (as in many real applications).

We have strived for speed in our index. A variant using less space, yet possibly somewhat slower, can be obtained by avoiding the representation of the explicit arrays of documents, D_{exp}. Instead, we can represent the documents associated to LZTrie nodes, which are only n', and at query time traverse all the RevTrie descendants u^r of v^r, map them to u in LZTrie, and traverse the documents of the subtree of u. That is, avoid the concatenation done to build D_{exp}, and instead do it on the fly at query time. This process has less locality of reference than the current one, and it requires mapping a fair number of nodes from RevTrie to LZTrie, but given the large performance gap, it is likely to still be many times faster than competing schemes. In exchange for the sharp reduction in the size of D_{exp}, we need to represent the LZTrie topology and Node mapping of the LZIndex, but the reduction of space should still be significant.

In natural language, retrieving approximate top-k answers to improve efficiency is a common practice. This avenue has not been explored much for general string collections. Our work shows that this idea is promising, as large space and time reductions are possible while still returning answers of good quality.

Chapter 8

An LZ77 Based Index for Document Listing

Collections of highly repetitive texts are usually bigger than sets of traditional files, and unfortunately classical text indexing is not designed to take advantage of the repetitiveness among the files. Nowadays, these types of databases have become very common in many fields. A good example is the encyclopedia Wikipedia, which is composed of millions of versioned documents, where any person can create new articles or suggest corrections to existing items. This process generates multiple versions of documents.

In view of the above, repetitive documents need special treatment in their storage and query if we want an efficient document retrieval system on them. The challenge then is to build a representation that takes advantage of the similarity between documents in order to index the full text, answering queries quickly and using smaller size compared to the approaches designed for conventional text collections.

This chapter presents the first attempts to use the LZ77 parsing in order to solve document listing on collections of highly repetitive texts.

8.1 A Document Listing Approach Based on the Hybrid-Index

Section 4.2 describes the Hybrid Index, which uses the LZ77 parsing to solve the pattern matching problem restricted to a predefined maximum pattern length M. The general idea to solve queries is first to find primary matches using a conventional index on a reduced text, in which consecutive phrase boundaries are located at maximum distance $(2M + 1)$ of one another. In a second step, beginning with each primary occurrence found, it locates secondary matches using the structure of Kärkkäinen and Ukkonen [63] built on the LZ77-structure.

The Hybrid Index locates all the *occ* occurrences of a pattern $p_{1..m}$ in $O(\mathsf{locate}_{pri}(m) + \mathsf{occ})$

time, where $\mathsf{locate}_{pri}(m)$ is the time to locate all the primary occurrences of the pattern using the index $I_M(T')$ on a reduced text T', and occ is the number of times that p appears in T.

The New Idea. This proposal adapts the Hybrid-Index in order to report documents instead of occurrences, in the following way. The original structure to find primary matches and its algorithm are maintained. Additionally we augment the structure so as to determine the identifiers of documents associated with each primary occurrence. We include a bitvector of length z, which marks the first phrase of each document. The document can be easily identified using the LZ77 structure and this bitvector. With respect to secondary matches, we design a scheme that takes a primary occurrence $T_{l..r}$ and can retrieve the list of documents containing phrases that have copied $T_{l..r}$ from this primary match. We adapt the basic algorithm in order to find only one secondary occurrence per document in each one of the t primary matches found. Our method finds documents by querying separate structures, depending on whether the source of a secondary occurrence is shared with another source.

As a result, our proposal solves any document listing query in $O(\mathsf{locate}_{pri}(m) + t \cdot \mathsf{ndoc})$ time, where ndoc is the output size.

8.1.1 The Structure for Primary Matches

The first task, like with the hybrid indexing, is to parse the input text $T_{1..n}$ with the LZ77 parser, which obtains a dictionary of z phrases and the list $L[1..z]$ that contains the positions where each phrase starts in T. We also build a bitvector $B_{1..z}$ whose D positions associated with the first phrase in each document are set to 1. B requires $z + o(z)$ bits. So, in order to find the document d_j that contains the phrase r, we just compute $j = rank_1(B, r)$.

We handle the special case when the length of the queried pattern is 1, storing in an additional list L_σ the σ phrases corresponding to the first occurrence of each one of the σ different characters of the alphabet. This list uses $\sigma\lceil\log z\rceil$ bits.

The next step is to filter the input text $T_{1..n}$ to create the reduced text T' (see Section 4.2.1) plus the list $L_{(M)}$ to map positions from T' to T. The list $L_{(M)}$ stores the positions where each phrase starts in the filtered text T'. We complete the structure building a conventional full text index, $I(T')$, for the filtered text.

8.1.2 The Structure for Secondary Matches

Given the characteristics of the original method to find secondary matches, we see that in the recursive algorithm, for each primary match the secondary occurrences are found in increasing order of positions in T. We start from a primary match $T_{l..r}$ and find all the secondary matches $s_1, s_2, ..., s_k$ whose sources include $T_{l..r}$. These occurrences can be in the same document of $T_{l..r}$ or in later ones. Next, we repeat the same procedure with each of the occurrences s_i. We exploit this order with a method to obtain increasing sequences of

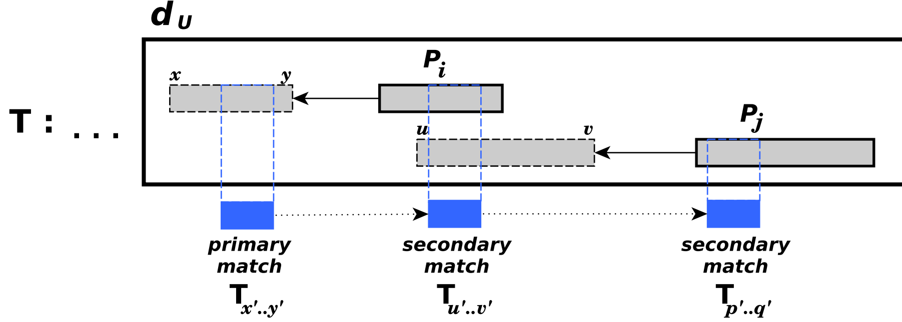

Figure 8.1: The basic scheme with non-overlapped phrase sources. Two phrases P_{i} and P_j in the document d_U, whose sources $T_{x..y}$ and $T_{u..v}$ (framed with segmented gray boxes) are also in the same document, and $T_{u..v}$ overlaps P_{i}.

documents that contain these secondary matches s_{i}. It is clear that if we simply retrieve the document for each secondary occurrence found, we obtain an inefficient solution for DL with $O(t_{\mathsf{tsearch}}(m) + \mathsf{occ})$ time, where occ is the number of times that the pattern $p_{1..m}$ appears in the whole collection. In order to avoid exploring the complete set of secondary occurrences generated by each primary occurrence, we have to overcome at least two problems. One is to modify the original grid (described in Section 4.2.1), so that now, for each primary occurrence we will retrieve only one phrase per different document. The second and main problem happens when a substring is shared by more than one source of subsequent phrases. These succeeding phrases can belong to the same document, so we can report the same identifier several times.

We propose to store inverted lists for all segments that are used by two or more sources, For those that have a single match we adapt the structure to retrieve a different document in each step of the recursive algorithm. We build a structure that can retrieve the k_{i} different documents obtained from the ith primary occurrence in $O(k_{\text{i}})$ time.

Non-Overlapped Phrase Sources

Let us consider the most basic situation, which is illustrated in Figure 8.1. It shows two phrases P_{i} and P_j in a document d_U, with their sources $T_{x..y}$ and $T_{u..v}$ in the same document and the source of P_j overlapping P_{i}. Consequently, the original grid G includes the points $\{(x, y, P_{\text{i}}), (u, v, P_j)\}$. Under that scheme, if we find a secondary match inside the phrase P_{i}, it would be possible to include it in the source $T_{u..v}$ of the phrase P_j. Therefore we will report the same document 3 times, one for the primary match of the pattern in $T_{x..y}$ and the other two secondary matches in the phrases P_{i} and P_j. Therefore we modify the grid G to avoid the described redundancy.

Step 1.- Deleting unnecessary points. Note that none of the two points that represent the phrases illustrated in Figure 8.1 are necessary. We then delete these from the grid G. In general, we delete from G every point whose phrase and source are in the same document

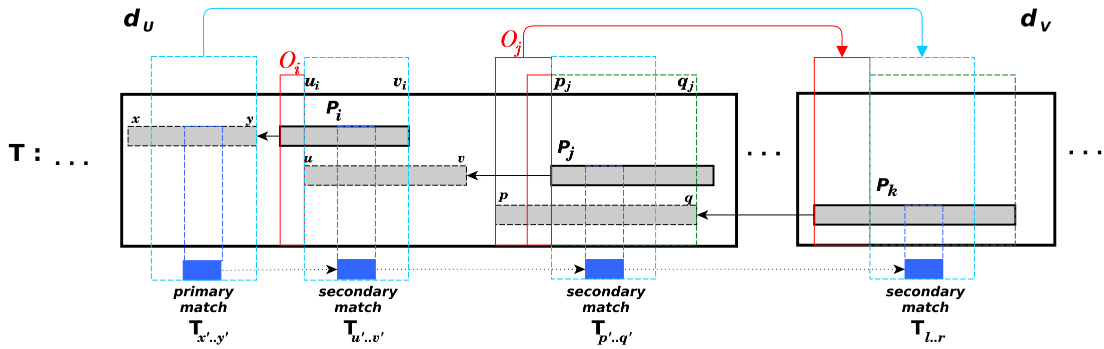

Figure 8.2: An example with non-overlapped phrase sources distributed in two documents d_U and d_V. There are three phrases $P_{\rm i}$, P_j and P_k, whose sources $T_{x..y}$, $T_{u..v}$ and $T_{p..q}$, respectively are pointed with solid arrows and framed with gray boxes in dashed line. The red boxes indicate offsets from the beginning of the phrases with respect to the beginning of other phrase sources. There are four occurrences of a pattern: the primary one $T_{x'..y'}$ and the three subsequent secondary matches that have been copied from $T_{x'..y'}$.

and the phrase does not overlap with any other source. This is the case of phrase P_j in Figure 8.1, and after it is deleted, the same situation occurs with phrase $P_{\rm i}$. Consequently both points are deleted. This rule ensures that the algorithm reports once the document d_U (i.e., only for its primary occurrence inside the source $T_{x..y}$). It is possible, however, that d_U is reported again by other primary matches. Besides, note that under these conditions, we also delete every point whose source possibly overlaps with other source.

Consider now Figure 8.2, which shows two documents d_U and d_V. Inside d_U there are two phrases $P_{\rm i}$ and P_j whose sources $T_{x..y}$ and $T_{u..v}$ are also in the same document. In d_V the phrase P_k has its sources $T_{p..q}$ in the previous document d_U. The original grid G represents the described scenario with the points $\{(x, y, P_{\rm i}), (u, v, P_j), (p, q, P_k)\}$. Suppose now that we found a primary match $T_{x'..y'}$ inside the source $T_{x..y}$, as the figure shows, $x \leq x' < y' \leq y$. Then given the characteristics of the LZ77 parser, the string $T_{x'..y'}$ is also in the phrases $P_{\rm i}$, P_j and P_k in this example. Therefore, the original algorithm locates the primary occurrences and the three subsequent secondary ones inside those phrases, meaning that a document is reported several times under the original scheme. We now describe a solution for that situation.

Step 2.- Building a new grid G Let focus only in the phrases P_j and P_k of Figure 8.2 and forget $P_{\rm i}$ for the moment. Observe that the phrase source $T_{p..q}$ of P_k includes the substring $T_{p_j..q_j}$ of the phrase P_j. Therefore, if we replace the point (u, v, P_j) by $(u, u + (q_j - p_j), P_k)$ we ensure that if, and only if, we find a secondary match inside $T_{u..u+(q_j-p_j)}$ in the source of the phrase P_j, we will obtain a secondary match inside P_k too, reporting a different document as we desire. We will also have to search for other sources that cover the string $T_{u..u+(q_j-p_j)}$, but now inside the phrase P_k. For that reason another component is necessary in the structure that specifies where the string $T_{u..u+(q_j-p_j)}$ starts inside the phrase P_k, so that we can continue accurately with the recursive algorithm. Figure 8.2 indicates the offset O_j from the beginning of P_k that we need to consider in order to know where the substring

Original Grid	New Grid
(p, q, P_k)	$(p, q, P_k, 0)$
(u, v, P_j)	$(u, u + q_j - p_j, P_k, O_j)$
(x, y, P_i)	$(x + O_\mathrm{i}, y, P_k, O_j)$

Table 8.1: New points in the grid G for the three phrases of our example at Figure 8.2.

$T_{u..u+(q_j-p_j)}$ begins in the phrase and to continue with the recursion. Similarly, in the same figure if we found an occurrence inside the source the phrase P_i, we then need the offset O_i to continue the search of phrases whose sources include the substring $T_{x+O_\mathrm{i}..y}$. Therefore, for each source non-overlapped with any other source but overlapped with another phrase in another document, we add an offset in the fourth element of its point as follows —remember that the list $L[1..z]$ stores the starting positions for each phrase in the text.

1. If the phrase P_k and its source $T_{p..q}$ are in different documents, then we change (p, q, P_k) by $(p, q, P_k, 0)$ (i.e., we include the offset $O_k = 0$).
2. If the phrase P_j and its source $T_{u..v}$ are in the same document d_U, and also there is a phrase P_k in a different document d_V whose source includes the substring $T_{p_j..q_j}$ of P_j in $T_{(L[k]+O_j)..(L[k]+O_j+q_j-p_j)}$, then we change (u, v, P_j) by $(u', u'+(q_j-p_j), P_k, O_j)$, where u', $u \leq u' \leq v$ is the point where the string $T_{p_j..q_j}$ starts inside $T_{u..v}$. This new point added may trigger further opportunities to apply this rule. In our example, once we change (u, v, P_j), we can modify (x, y, P_i).

Note that points that have not been deleted in Step 1 can satisfy only one of these two previous conditions. As an example, Table 8.1 shows the new values for the points corresponding to the scheme illustrated in Figure 8.2. In the figure, we see that the offset O_i is included in the final part of the offset O_j and $O_\mathrm{i} = (y - x) - (v_\mathrm{i} - u_\mathrm{i})$.

Overlapped Phrase Sources

Considering Figure 8.3, where there are four phrases P_i, P_j, P_k and P_l, whose sources are $T_{c..f}$, $T_{a..g}$, $T_{\mathrm{d}..h}$ and $T_{b..\mathrm{e}}$, respectively. These phrases can be in the same document d_U or any other subsequent document in the text. Observe that the sources share substrings in common with each other, which implies that the search of secondary matches in this point will extend in many different ways. We handle this case by storing inverted lists.

The idea is that for each primary occurrence $p_{pr\mathrm{i}}$, we find the documents recursively in our new grid (created in Step 2) until either we find an inverted list with all the remaining documents for $p_{pr\mathrm{i}}$ or there is no other secondary match in G in this search branch generated by $p_{pr\mathrm{i}}$. When that happens, we move on to the next primary match and again search for secondary matches.

Step 3.- Creating inverted lists. In a new structure I_L, for all the segments in sources that overlap with at least one other source, we store their inverted lists with all the subsequent

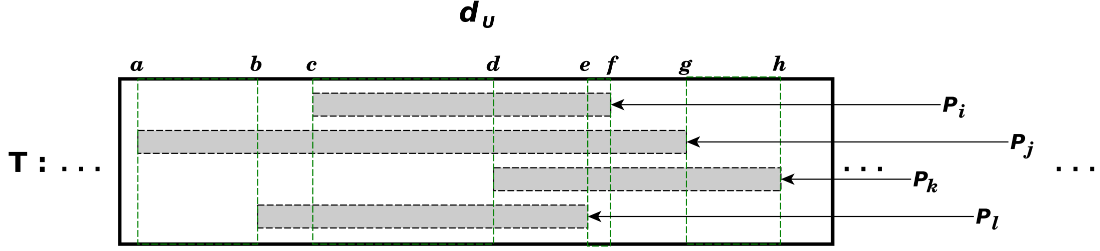

Figure 8.3: An example with several overlapped phrase sources in a document d_U.

documents that contain a substring copied from here. In other words, if a segment $T_{x..y}$ is shared by more than one source in a document d_U, then we save the increasing list with all the subsequent documents (to the right of U) that contain phrases that have copied the segment $T_{x..y}$ from d_U. As an example, the following inverted lists correspond to the scheme of Figure 8.3:

$T_{b..e} \rightarrow L_{b..e}$, for the overlapped segment between the sources P_j and P_l.

$T_{c..f} \rightarrow L_{c..f}$, for the overlapped segment between the sources P_i and P_j.

$T_{d..g} \rightarrow L_{d..g}$, for the overlapped segment between the sources P_j and P_k.

$T_{c..d} \rightarrow L_{c..d}$, for the overlapped segment between the sources P_i, P_j and P_l.

$T_{d..f} \rightarrow L_{d..f}$, for the overlapped segment between the sources P_i, P_j and P_k.

$T_{d..e} \rightarrow L_{d..e}$, for the overlapped segment between all the sources.

Observe that with this structure a substring s would be included in more than one inverted list. In that case, our algorithm selects the bigger list (i.e., the inverted list for the smallest segment that covers s). Note that, for example, we do not include $T_{b..c}$ because its list $L_{b..c}$ contains the same elements as the longer interval $T_{b..e}$.

To represent all the n_I overlapped segments $T_{x..y}$ and their inverted lists we use separate structures. We store the x-coordinates in an array $X_I[1..n_I]$, the length of each segment in the array $Len[1..n_I]$ and the N_I documents of all the inverted lists in $I_L[1..N_I]$. Then for any string $T_{x_i..y_i}$ with its associated inverted list with i' documents we have an entry in this new structure with three components: $(X_L[i] = x_i, Len[i] = y_i - x_i + 1, I_L[k..k+i'-1])$, for some k. In order to obtain the position k where a list starts we use a bitvector $V_I[1..N_I]$ to mark these positions. We sort the entries of these structures by their x-coordinates and store X_I with gap encoding as done with the array X in the grid G. We save the array Len explicitly in $n_L\lceil \log max_m \rceil$, where max_m is the maximum length among all the overlapped segments. Given that we create increasing inverted lists, we can store these with gap encoding too, concatenating all these lists in the array I_L and marking in V_I the first document of each list. In Section 8.1.4 we will describe how to store the inverted lists in order to save more space and we also give a method to search the lists efficiently.

After we have created the inverted lists we cannot simply delete all these points from G, because there are non-overlapped segments that we need to cover, such as $T_{a..b}$ and $T_{g..h}$ in our example. Additionally, we need to consider larger substrings $T_{a'..b'}$, $a \leq a' < b < b' \leq g$, that have not been covered by the lists. Consequently, we only delete the points (x, y, P_i)

Algorithm 6 Find documents with secondary matches from the string $T_{l..r}$

```
1: function SecondaryMatchesDL(l, r)
2:     Find the biggest inverted list L_i[1..i'] that covers T_{l..r}.
3:     if (L_i is not NULL) then
4:         Merge the list L_i with occ, updating V and the counter for ndoc
5:     else
6:         Find, by a binary search, the predecessor X[k] of l in the array X.
7:         Use recursively RMQs to find all the maximal values in Y[1..k] that are at least
   r, using the Range Maximum structure on the array Y.
8:         for each point (i', j'), we find (l', r') of the phrase T_{l'..r'} whose source is T_{i'..j'} do:
9:             doc = rank_1(B, l')
10:            if (V[doc] = 0) then
11:                occ[ndoc] = doc
12:                ndoc = ndoc + 1
13:                V[doc] = 1
14:            end if
15:            SecondaryMatchesDL(l', r')
16:    end if
17: end function
```

from G whose source $T_{x..y}$ is completely covered by an overlapped string (i.e., there is a document list associated to this string). Otherwise, we conserve the point to be processed as we detailed in Step 2.

8.1.3 The Document Listing Algorithm

The following process returns the list occ[1..ndoc] with the ndoc documents that contain the queried pattern $p_{1..m}$ as a substring. Similar to Sadakane's method to check if a document has already been reported or not, we use the same bit-string $V[1..D]$ to mark documents. We follow the original method of the hybrid indexing to locate all the primary matches occ_{pri}, by using the conventional index for the filtered text, $I(T')$, plus the lists L and $L_{(M)}$ to map from $T'[i'..j']$ to its correct match $T[i..j]$. Next, for each primary match $T_{i..j}$ found, we determine its document by computing $doc = rank_1(B, i)$. We use V to check if doc is not in occ, before including it in the output list and increasing the counter for ndoc. After that we call the function SecondaryMatchesDL(i, j) defined in Algorithm 6, which finds all the documents with secondary occurrences that have copied the pattern from this primary occurrence.

The line 2 of Algorithm 6 checks if a string $T_{x..y}$ is covered by an inverted list. We then perform a search for the predecessor k of T_x in the sorted array X_I and validate that $(y - X_I[k]) > Len[k]$. We do this with the same method implemented for predecessor search in the array X of the grid G. In this point, it is possible that we find various segments for the same starting position $X_I[k] = X_I[k+1] = ... = X_I[k+k']$ (i.e., various inverted list too); if this is the case, we need to obtain the greatest inverted list, which is associated with the segment of minimum length that is bigger than $y - X_I[k]$. There are some structures to store Len and perform this search efficiently, for instance with Elias-Fano codes (see Section 4.1.1).

However, in Section 8.1.4 we will detail another method to perform this search efficiently and at the same time saving space in the inverted lists. Once we obtain the correct list $L_{\rm i}[1..{\rm i}']$, we merge it with the output list occ. In the line 8, we compute the coordinates (l', r') of the phase whose source is $T_{{\rm i}'..j'}$ as described in Section 4.2.1 (scheme illustrated in Figure 4.9). Each time a new document $L_{\rm i}[t]$ is included in occ (as $V[L_{\rm i}[t]] = 0$), we set $V[L_{\rm i}[t]] = 1$ and increment the counter for ndoc.

8.1.4 Reducing the Size of the Inverted Lists

Two improvements are detailed below in relation with the space usage of the inverted lists.

In Section 8.1.2, we defined how to build the inverted lists. For a pair of overlapped phrase sources in a document d_U, we detailed how to generate the increasing inverted list $L_{\rm i}[1..k_{\rm i}]$ with all the $k_{\rm i}$ documents that contain the overlapped string copied from d_U. Each list $L_{\rm i}[1..k_{\rm i}]$ is stored with gap encoding using $k_{\rm i}\log(max\{L_{\rm i}[0], {\rm d_i}\})$ bits, where $d_{\rm i}$ is the maximum difference between any consecutive cells of the list $L_{\rm i}$, $d_{\rm i} = max_{0 \leq j < k_{\rm i}-1}(L_{\rm i}[j+1] - L_{\rm i}[j])$. Because $L_{\rm i}[0]$ can be a bigger value ($L_{\rm i}[0] \leq D$), the required space for the list $L_{\rm i}$ also can be proportional to this initial document $L_{\rm i}[0]$. However, we see that these documents are to the right of d_U, and therefore we know that $L_{\rm i}[0] > {\rm d}_U$. Hence, we can rewrite the header of the list $L_{\rm i}$ corresponding to the first document $L_{\rm i}[0]$ as $(L_{\rm i}[0] - {\rm d}_U)$, because at query time we easily know the value of the initial document d_U. As a result, we now require $k_{\rm i}\log(max\{L_{\rm i}[0] - {\rm d}_U, {\rm d_i}\})$ bits, which helps reduce the size of the list. This first improvement looks minimal, but we believe that it can be significant in reducing the bits for each cell when d_U is close to d. Applying that rule we obtain a complete sequence of gap coding values (including the first position $L_{\rm i}[0]$) for the array I_L.

The second and more significant improvement to reduce the size of the lists takes into account the containment relationships between sets. Returning to the example of Figure 8.3, let us focus on the overlapped strings contained in other overlapped zones. For instance, $T_{c..{\rm d}}$ is covered by $T_{c..f}$, which implies that the inverted list $L_{c..f}$ corresponding to the longer string is a subset of the inverted list $L_{c..{\rm d}}$ associated with the shorter string. Another dependency between sets is given by the segments $T_{\rm d..e}$, $T_{{\rm d}..f}$ and $T_{{\rm d}..g}$, where for their lists we have $L_{{\rm d}..g} \subseteq L_{{\rm d}..f} \subseteq L_{\rm d..e}$. We will exploit these containments between inverted lists in order to save space. Observe that in Section 8.3 we forced all overlapped strings to either start at the same position of the text or end in a common position in T. Therefore, these relationships can be determined by the y-coordinates of the segment $T_{x..y}$, as it happens between $T_{c..f}$ and $T_{{\rm d}..f}$ or by the x-coordinates of these segments, which we store in the array X_I. Observe the situation illustrated in Figure 8.3, where the overlapped strings $T_{c..{\rm d}}$ and $T_{c..f}$ start in a same point c of the text, and the overlapped strings $T_{\rm d..e}$, $T_{{\rm d}..f}$ and $T_{{\rm d}..g}$ start in $T_{\rm d}$, sharing their initial point. We take advantage of this order as follows.

Suppose that we have identified some containment relationships between sets given by k lists, whose the respective overlapped strings are $T_{l..r_1}, T_{l..r_2}, ..., T_{l..r_k}$, such that $r_1 < r_2 < ... < r_k$. Then, for their lists we have $L_{l..r_k} \subseteq L_{l..r_{k-1}} \subseteq ... \subseteq L_{l..r_1}$; we call these lists *complementary lists*. Formally, at construction time, for each one of these complementary lists whose overlapped strings start in T_l, we do the following:

1. We store the lists from the smallest one (at the left) to the biggest one (to the right), that is $L_{l..r_k}, L_{l..r_{k-1}}, ..., L_{l..r_1}$ from left to right.
2. The first list is stored without changes, and for the subsequent lists only the documents that have not been included in the early ones are stored. Then we assure that each document can be present once in the complete representation for each complementary list. Then the final sets to store are: $\{L_{l..r_k}\}, \{L_{l..r_{k-1}} - L_{l..r_k}\}, \{L_{l..r_{k-2}} - (L_{l..r_{k-1}} \cup L_{l..r_k})\}, ..., \{L_{l..r_1} - (L_{l..r_2} \cup ... \cup L_{l..r_{k-1}} \cup L_{l..r_k})\}$.
3. We order the bitvector $V_I[1..N_I]$ to indicate where each list starts.

We now describe how to find the inverted list we need in line 2 of Algorithm 6. Once we have obtained the predecessor $X_I[k]$ for the string queried $T_{l..r}$ as we detailed previously, we add to the output all the complementary lists whose segment length is at least $Len[k]$. Therefore, we first validate that $(r - X_I[k]) > Len[k]$; if it is true we add to occ the kth inverted list from the position $sp = select_1(V_I, k)$ to $ep = select_1(V_I, k+1)$ in I_L. We continue validating if the segment of the next list, $k+1$, also starts in $X_I[k]$, checking if $X_I[k] = X_I[k+1]$. If it is true then we validate the length $(r - X_I[k]) > Len[k+1]$ before merging the list $k+1$. We continue with this process until a list $k+k'$ does not start in the same point, that is, $X_I[k] < X_I[k+k']$, or the length of the segment does not cover the string, that is, $(r - X_I[k+k']) \leq Len[k]$. This method ensures optimal $O(k_{\mathrm{i}})$ time to retrieve the k_{i} documents from the inverted lists that include the segment $T_{l..r}$. Note also that, for the complementary lists we can store their segment lengths of the array Len with gap encoding too, because these values are in decreasing order.

Additionally, we have to add the first document retrieved from any inverted list the value of the document d_U where the overlapped string occurs and the list was generated, which is obtained at query time. Then we include in the procedure $SecondaryMatchesDL$ a third parameter d_U. For the first call d_U will be the document where the primary occurrence was found.

8.2 Including Frequencies

We follow the most natural way to retrieve the frequencies of the documents reported. We extend the presented structure to include new arrays that store the frequencies for the inverted lists and the points of the grid G. For the case of the lists, we will store all the frequencies corresponding to the concatenated array of inverted lists $I_L[1..N_I]$ in a new array $F_I[1..N_I]$. Then, during the construction, we compute these values by accumulating the number of secondary matches for these documents respect to each overlapped segment that we have associated with an inverted list. We just store explicitly each of these frequencies using $k \log\lceil M_{\mathrm{i}} \rceil$ bits, where M_{i} is the maximum frequency among all the documents included in I_L.

With respect to the new grid (detailed in the second step of Section 8.1.2), we consider a new array F_G to store the frequencies of each point, but only when it is greater than one. We then include another bitvector B_F to mark the points that have an associated frequency

in F_G.

In order to include the frequencies associated with the document reported in our DL algorithm, we use a temporary array $F_V[1..D]$ to accumulate the frequencies for the documents found at query time. The accumulated frequencies of F_V are associated with the marked positions in V, updating their frequencies each time we find a document.

8.3 Conclusions

We have introduced a novel structure based in the LZ77 parser to index repetitive texts and solve document listing queries. Since we have reduced the number of points to store in the grid G for secondary occurrences, and we efficiently store a reduced number of inverted lists; we can anticipate that our structure (without frequencies) will use space similar to that of the original Hybrid-Index (described in Section 4.2), now finding documents instead of pattern occurrences.

Chapter 9

Conclusions and Further Research

This PhD thesis offers novel contributions in the Document Retrieval (DR) field. As a result of this research we propose new approaches to building compressed data structures that help solve the most important DR problems on general string collections. Our overview of the state of the art showed that most of the proposals are built on indexes based on the Suffix Array or the Suffix Tree, which are used to find the set of all the suffixes that contain the search pattern as a prefix. On these suffixes, different techniques are used to identify the documents to output. In particular, for document listing (DL) all the solutions follow the pioneer optimal time solution of Muthukrisman [84]. Various improvements in space requirements result in compact DL solutions whose answer time is far from optimal. On the other hand, the most useful structure for Top-k was offered by Hon et al. [58], which stores precomputed answers for some suffix tree nodes and its answer time is close to the optimal. Again, most of the subsequent structures improve the space but affect the answer time adversely.

The central contribution of this thesis is a new approach to DR based on the LZ78 parser [118] instead of on suffix arrays and trees. We started from the basic LZ-Index of Navarro [85] and designed two novel indexes for solving DL (the LZ-DLIndex [30]) and Top-k retrieval (the LZ-TopkIndex [31]). The experiments show that our structures are very competitive both in query time and space requirements compared to the best previous approaches. Our novel indices are also capable of retrieving approximate answers for both problems, using much less space and/or time. The LZ-DLIndex outputs most of the documents in only a few microseconds and generally uses less than 7 bpc (bits per character). It gives the complete answer at the cost of some extra space and time. On the other hand, the LZ-TopkApp index offers an excellent approximation to the LZ-TopkIndex. It uses 4–7 bpc for most collections and offers a query time around k μs per query. We have shown that the quality of this approximation improves asymptotically with the size of the collection, reaching over 90% of the accumulated term frequency of the real answer already for patterns of length ≤ 8 on rather small collections, and improving for larger ones. The tradeoff between time-space consumption and the quality of its answer makes our structures very relevant in scenarios where approximate answers to these problems are sufficient.

Our second main contribution are the first steps in developing DR solutions for highly

repetitive text collections. We build on the LZ77 parser [117], which is the best for such collections. We first develop an efficient pattern-matching index. Given an upper bound on pattern lengths M, our Hybrid-Index [29] offers better search time than the LZ77-Index [69] and better space and occurrence location time than the FM-Index [33, 35, 36], given a sufficiently low bound M. The experiments indicate that the Hybrid-Index is the best option to locate patterns of length less than 40 characters, a boundary that is sufficient in many applications.

On the Hybrid-Index we design a new solution for DL. Our new DL index is then useful on highly repetitive texts, and it is capable of retrieving the frequency for each reported document. While not yet implemented, this is undoubtedly a promising result for the DR field, and the first one building on LZ77 compression to solve DR problems.

We also contributed to basic data structures that are used, in particular, for DR. Our main result is an alternative design to Fischer and Heun's Range Minimum Query (RMQ) solution [40], which was the best proposal to date in terms of time and space. It uses $2n+o(n)$ bits and answers RMQs in constant time. We simplified their formula using a Balanced Parenthesis tree representation [83] instead of a DFUDS [16] one. Our implementation uses $2.2n$ bits and takes 1–4 microseconds per query, for any input array of length n. This is noticeably smaller and faster than the current implementations in libraries *SDSL* and *Succinct*, which follow Fischer and Heun's design.

We also implemented and tested the Compressed Suffix Arrays (CSAs) of Grossi and Vitter [54, 55] and Rao [100], since all classical compact solutions use a CSA. However, we did not obtain good results in terms of space usage and query time with these indexes. These negative outcomes are also of interest.

Further Research

Our thesis has opened important research avenues. The first is the use of Ziv-Lempel based indexing to solve document retrieval problems. We have shown that this path is promising and that competitive space/time tradeoffs can be obtained.

The second is the research in indexes that provide incomplete or approximate solutions to these problems. While this is a very natural direction on DR in natural language, because in most cases inaccurate answers are acceptable, it had not been explored before on general strings collections. We have shown that much better space/time tradeoffs can be obtained, while retaining good quality in the results.

The third is the search on DR on highly repetitive text collections, which are becoming a central actor in the sharp growth of the available digital data. There are almost no previous solutions of this kind.

We expect that these three future research lines will flourish in the next years, and also to participate in them. In particular, we plan to implement our DL proposal for highly repetitive text collections. We also plan to further improve our RMQ solution and to do further research in the redundancy of the document array, which we believe admits compression beyond the

current results.

Bibliography

[1] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. 12th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 84–97. SIAM Press, 2010.

[2] D. Arroyuelo and G. Navarro. Practical approaches to reduce the space requirement of lempel-ziv-based compressed text indices. *ACM Journal of Experimental Algorithmics (JEA)*, 15(1.5), 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*. ACM Press / Addison-Wesley, 1999.

[5] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval - the concepts and technology behind search, Second edition*. Pearson Education Ltd., 2011.

[6] J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, 69(1):232–268, 2014.

[7] 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.

[8] A. Bartsch, B. Bunk, I. Haddad, J. Klein, R. Münch, T. Johl, U. Kärst, L. Jänsch, D. Jahn, and I. Retter. Genereporter - sequence-based document retrieval and annotation. *Bioinformatics*, 27(7):1034–1035, 2011.

[9] 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.

[10] D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. In *Proc. 19th Annual European Symposium on Algorithms (ESA)*, LNCS 6942, pages 748–759, 2011.

[11] 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.

[12] D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms (TALG)*, 10(4):article 23, 2014.

[13] D. Belazzougui, G. Navarro, and D. Valenzuela. Improved compressed indexes for full-text document retrieval. *J. of Discrete Algorithms*, 18:3–13, January 2013.

[14] T. Bell, I. Witten, and J. Cleary. *Text Compression*. Prentice Hall, Englewood Cliffs, N.J., USA, 1990.

[15] M. Bender and M. Farach-Colton. The lca problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, LATIN '00, pages 88–94, London, UK, UK, 2000. Springer-Verlag.

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

[17] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

[18] M. Burrows and D. J. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[19] S. Bttcher, C. Clarke, and G. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.

[20] G. J. Chaitin. *The Limits of mathematics: A course on information theory and limits of formal reasoning*. Springer Series in Discrete Mathematics and Theoretical Computer Science. Springer, 1998.

[21] G. Chen, S. J. Puglisi, and W. F. Smyth. Lempel-Ziv factorization using less time & space. *Mathematics in Computer Science*, 1:605–623, 2008.

[22] D. Clark. *Compact Pat Trees*. PhD thesis, 1998. University of Waterloo, Canada.

[23] F. Claude and J. I. Munro. Document listing on versioned documents. In *Proceedings of the 20th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 8214, pages 72–83, 2013.

[24] F. Claude and G. Navarro. Improved grammar-based compressed indexes. In *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 7608, pages 180–192, 2012.

[25] J. S. Culpepper, G. Navarro, S. J. Puglisi, and A. Turpin. Top-k Ranked Document Search in General Text Databases. In *Proc. 18th Annual European Symposium on Algorithms (ESA)*, LNCS 6347, pages 194–205 (part II), 2010.

[26] P. Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*,

21(2):246–260, 1974.

[27] R. M. Fano. *On the number of bits required to implement an associative memory.* Memorandum 61, Computer Structures Group, MIT Project MAC Computer Structures Group, 1971.

[28] A. Farzan, R. Raman, and S. Rao. Universal succinct representations of trees? In *Proc. 36th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 451–462 Part I, 2009.

[29] H. Ferrada, T. Gagie, T. Hirvola, and S. J. Puglisi. Hybrid indexes for repetitive datasets. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 372(2016), 2014.

[30] H. Ferrada and G. Navarro. A Lempel-Ziv compressed structure for document listing. In *Proc. 20th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 8214, pages 116–128, 2013.

[31] H. Ferrada and G. Navarro. Efficient compressed indexing for approximate top-k string retrieval. In *Proc. 21st International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 8799, pages 18–30, 2014.

[32] H. Ferrada and G. Navarro. Improved range minimum queries. In *Proc. 26th Data Compression Conference (DCC)*, pages 516–525, 2016.

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

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

[35] 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.

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

[37] J. Fischer. Optimal succinctness for range minimum queries. In *9th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 158–169, 2010.

[38] J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE). LNCS*, pages 459–470, 2007.

[39] J. Fischer and V. Heun. Range median of minima queries, super-cartesian trees, and text indexing. In *Proc. 19th International Workshop on Combinatorial Algorithms (IWOCA)*, pages 239–252, 2008.

[40] 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.

[41] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.

[42] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[43] H. Gabow, J. Bentley, and R. Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th Annual ACM Symposium on Theory of Computing (STOC)*, pages 135–143, 1984.

[44] T. Gagie, K. Karhu, G. Navarro, S.J. Puglisi, and J. Sirén. Document listing on repetitive collections. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 7922, pages 107–119, 2013.

[45] T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426427:25–41, 2012.

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

[47] R. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms*, 2(4):510–534, 2006.

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

[49] S. Gog and G. Navarro. Improved single-term top-k document retrieval. In *Proc. 17th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 24–32. SIAM Press, 2015.

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

[51] R. González, G. Navarro, and H. Ferrada. Locally compressed suffix arrays. *ACM Journal of Experimental Algorithmics*, 19(1):article 1, 2014.

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

[53] R. Grossi and G. Ottaviano. Design of practical succinct data structures for large data collections. In *Proc. 12th International Symposium on Experimental Algorithms (SEA)*, LNCS 7933, pages 5–17, 2013.

[54] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications

to text indexing and string matching (extended abstract). In *Proc. 32nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 397–406, 2000.

[55] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[56] W. Hon, M. Patil, R. Shah, and S. Wu. Efficient index for retrieving top-k most frequent documents. *Journal of Discrete Algorithms*, 8(4):402–417, 2010.

[57] W. 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)*, pages 173–184, 2012.

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

[59] W. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. *Foundations of Computer Science, IEEE Annual Symposium on*, pages 713–722, 2009.

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

[61] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees with applications. *Journal of Computer and System Sciences*, 78(2):619–631, 2012.

[62] J. Kärkkäinen and S. J. Puglisi. Fixed block compression boosting in FM-indexes. In *Proc. 18th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 174–184, 2011.

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

[64] M. Karpinski and Y. Nekrich. Top-k color queries for document retrieval. In *Proc. 22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 401–411, 2011.

[65] D. E. Knuth. *The Art of Computer Programming. Vol.3: Sorting and Searching*. 1973.

[66] A. N. Kolmogorov. On tables of random numbers. *Theoretical Computer Science*, 207(2):387–395, November 1998.

[67] R. Konow and G. Navarro. Faster compact top-k document retrieval. In *Proc. 23rd Data Compression Conference (DCC)*, pages 351–360, 2013.

[68] S. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel–Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 2000.

[69] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[70] S. Kurtz. Reducing the Space Requirement of Suffix Trees. *Software Practice and Experience*, 29:1149–1171, 1999.

[71] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Proc. Data Compression Conference (DCC)*, pages 296–305, 1999.

[72] A. Lempel and J. Ziv. On the complexity of finite sequences. *Information Theory, IEEE Transactions on*, 22(1):75 – 81, 1976.

[73] V. Mäkinen. Compact suffix array — a space-efficient full-text index. *Fundamenta Informaticae*, 56(1–2):191–210, 2003.

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

[75] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.

[76] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

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

[78] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

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

[80] Y. Matias, S. Muthukrishnan, S. Sahinalp, and J. Ziv. Augmenting suffix trees, with applications. In *Proc. 6th Annual European Symposium on Algorithms (ESA)*, pages 67–78, 1998.

[81] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1984.

[82] J. I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42, 1996.

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

[84] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. 13th*

Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 657–666, 2002.

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

[86] G. Navarro. Implementing the LZ-index: Theory versus practice. *ACM Journal of Experimental Algorithmics*, 13(article 2), 2009.

[87] G. Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys*, 46(4):article 52, 2014.

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

[89] 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.

[90] G. Navarro, Y. Nekrich, and L. M. S. Russo. Space-efficient data-analysis queries on grids. *Theoretical Computer Science*, 482:60–72, 2013.

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

[92] G. Navarro, S. J. Puglisi, and D. Valenzuela. General document retrieval in compact space. *ACM Journal of Experimental Algorithmics*, 19(2):article 3, 2014.

[93] G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):article 16, 2014.

[94] G. Navarro and D. Valenzuela. Space-efficient top-k document retrieval. In *Proc. 11th International Symposium on Experimental Algorithms (SEA)*, LNCS 7276, pages 307–319, 2012.

[95] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 60–70, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[96] M. Pătraşcu. Succincter. In *Proc. 49th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 305–313, 2008.

[97] S. J. Puglisi, W. Smyth, and A. Turpin. Inverted files versus suffix arrays for locating patterns in primary memory. In *Proc. 13th international conference on String Processing and Information Retrieval (SPIRE)*, pages 122–133, 2006.

[98] S. J. Puglisi, W. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2), 2007.

[99] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[100] S. Rao. Time-space trade-offs for compressed suffix arrays. *Information Processing Letters*, 82(6):307–311, 2002.

[101] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. 11th International Conference on Algorithms and Computation (ISAAC)*, pages 410–421, 2000.

[102] K. Sadakane. Space-efficient data structures for flexible text retrieval systems. In *Proc. 13th International Conference on Algorithms and Computation (ISAAC)*, pages 14–24, 2002.

[103] 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.

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

[105] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, pages 589–607, 2007.

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

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

[108] D. Salomon. *Data Compression: The Complete Reference*. Springer-Verlag New York, Inc., 2006.

[109] G. Salton, A. Wong, and C. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[110] E. Selfridge-Field. *Beyond Midi: The Handbook of Musical Codes*. Mit Press, 1997.

[111] C. E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[112] W. Szpankowski. On the height of digital trees and related problems. *Algorithmica*, 6(2):256–277, 1991.

[113] W. Szpankowski. A generalized suffix tree and its (un)expected asymptotic behaviors. *SIAM Journal on Computing*, 22(6):1176–1198, 1993.

[114] N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *Proc.*

18th Annual Symposium on Combinatorial Pattern Matching (CPM), pages 205–215, 2007.

[115] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

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

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

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