UNIVERSIDADE TÉCNICA DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Enhanced Full-Text Self-Indexes based on Lempel-Ziv Compression

Luís Manuel Silveira Russo
(Licenciado)

Dissertação para obtenção do Grau de Doutor em
Engenharia Informática e de Computadores

Orientador: Doutor Arlindo Manuel Limede de Oliveira

Júri

Presidente: Reitor da Universidade Técnica de Lisboa

Vogais: Doutor Arlindo Manuel Limede de Oliveira

Doutor Gonzalo Navarro

Doutor Mário Alexandre Teles de Figueiredo

Doutor José Carlos Alves Pereira Monteiro

Doutora Margarida Paula Neves Mamede

Doutor Paulo Alexandre Carreira Mateus

Setembro 2007

Abstract

Full-text indexes provide an efficient method to search for any sub-string of a text. However, these indexes require a large amount of space. Compressed indexes, that explore the compressibility of the text, have recently been proposed to address this problem. Self-indexes, that in addition are able to reproduce any sub-string of the text without storing it explicitly, represent a further step towards saving space.

This thesis studies self-indexes based on the Lempel-Ziv data compression technique. It starts by analyzing the search algorithm of these indexes, pointing out the causes of a quadratic dependency on the pattern size. It then proposes a new search procedure that solves this problem and confirms empirically that this modification has significant practical results. The thesis also proposes a method to extend the functionality of these indexes, so that it becomes possible to find a longest common sub-string and to compute approximate matches. These results are verified empirically, demonstrating that these indexes are very efficient.

Keywords: information storage, data compression, text indexing, pattern matching, approximate pattern matching, longest common sub-string

Acknowledgments

The past 4 years of my life have been a maturing period, in many ways. During this time I was faced with new and intriguing challenges. Transiting from being an undergraduate student to a PhD student was not always easy and straightforward, but, in the end, I am profoundly satisfied that I was able to study string processing. In fact, string processing had already fascinated me as an undergraduate. Research is the most consuming activity I have ever engaged in. It has been a disorienting process, full of setbacks and breakthroughs. I learned important lessons such as recognizing the value of simple ideas. It is probably natural that I feel strongly about such an important period of my life. I think Charles Dickens expressed it best: "It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair ...". Still it was well worth it.

Arlindo Oliveira, my thesis advisor, deserves a special acknowledgment, for actively supporting my research, introducing me to new problems and new persons. He has systematically given me good advises, even though I sometimes learned this the hard way by my own fault. He has also helped me with several practical issues, including correcting a huge amount of manuscripts filled with typos and bugs. He always finds time when I need it, despite his incredibly busy schedule. Finally for believing in me and pushing me forward when I felt most lost.

I am also deeply grateful to Gonzalo Navarro for supporting this work in several ways. He has strongly influenced my ideas about compressed indexes and about my own work. His work in Lempel-Ziv indexes motivated me to start investigating this area. I will always recall his insightful words of incentive in the Workshop on Compression, Text, and Algorithms at DCC in November of 2005, along with a series of stimulating discussions. I am also grateful for having worked with him on approximate string matching. His suggestions and corrections on the SPIRE 2006 paper, as well as Arroyuelo's and those of several anonymous reviewers were greatly appreciated.

I would also like to say thanks to Amílcar Sernadas for helping me to start this PhD and also for continuously showing interest in my work.

This thesis was also greatly improved by my thesis committee, namely Gonzalo Navarro, Margarida Mamede and José Monteiro, that made important comments and corrections on a preliminary version of it.

I am particularly grateful to everyone that aided my research by providing prototypes: Heikki Hyyrö, Paolo Ferragina, Rodrigo González, Veli Mäkinen, Pedro Morales, Gene Myers, Gonzalo Navarro, Sadakane and Rossano Venturini.

I am grateful to all the members of the ALGOS group for making this period of time more stimulating and enjoyable. In particular I would like to thank Luís Coelho, Miguel Bugalho, Alexandre Francisco and Alexandra Carvalho for countless stimulating discussions about all sort of algorithms and other more mundane issues.

In the course of this work I interacted with several people form abroad of whom I have pleasant memories: Diego Arroyuelo, Ricardo Baeza-Yates, Benjamin Bustos, Edgar Chávez, Shane Culpepper, George Dupret, Heikki Hyyrö, Rodrigo González, Veli Mäkinen, Pedro Morales, Rodrigo Paredes.

I am grateful to several institutions for supporting this research, Fundação para a Ciência e a Tecnologia (grant SFRH/BD/12101/2003 and BIOGRID project POSI/SRI/47778/2002), CYTED VII.19 (project RIBIDI), Orient Foundation (grant IS/DCAS/L.12.6.5/05).

To my wife, for her daily support in pursuing this goal, even by spelling if necessary, as "Stanley and Iris". Also for improving my ability to deal with multiple aspects of life, from social skills to Sisyphus tasks. Above all I am deeply grateful to her for being considerate, not only with me but also with others.

To my family and friends, for the unconditional support they have always given me. In particular to my parents for being particularly mindful of my education, always stimulating my curiosity and my understanding of the world, ultimately allowing me to cast "alohomora" throughout this thesis.

Contents

List of Figures

List of Tables

List of Algorithms

1

Introduction

The exact matching problem consists in finding all the occurrences, if any, of a short sequence of characters (pattern) P in a longer sequence of characters (text) T. The size of P will be denoted by m and the size of T by u. P and T are generated from alphabet Σ of size σ.

We agree with Gusfield's [52] opinion that " The practical importance of the exact matching problem should be obvious to anyone who uses a computer". In fact exact matching is a building block for a considerable amount of applications. Therefore, developments in exact matching tend to have a wide impact. The applications of this problem are so numerous that listing them would prove to be a hard task. Due to the central role that this problem plays, it is also sometimes referred to as "string matching", "pattern matching" or "text searching", among others. Depending on the nature of the application at hand, we must choose one of the two following approaches to solve the problem.

The sequential approach consists in searching for P directly on the plain representation of T, i.e. without using any auxiliary data structure. The indexed approach consists in first building an auxiliary data structure, known as an index, that is used to improve the time it takes to solve the problem.

The sequential approach should be applied when the text is small enough or too dynamic. For example, this is a viable approach for text editors, where the text is very dynamic and not excessively large. Naive and linear algorithms for this approach can be found in undergraduate computer science textbooks [28]. In fact this approach has been extensively researched and a considerable number of linear time algorithms for it are now available. For a book on the subject consult Navarro and Raffinot [96].

1.1 Text Indexes

It should be clear that the sequential approach must consult a considerable amount of the text. If the text is very large, this may require too much time. In this case the indexed approach can provide a faster alternative to solve the problem. In this kind of approach we assume that the text is known and preprocessed a priori, i.e., that the index is built before the pattern is given. Therefore, the time to build the index is not taken into account when searching for a pattern. Building the index structure is likely to require a considerable amount of space and time. Hence indexes can only be used when these resources are available. In order to amortize this cost, a considerable amount of searches should be performed. The text should also be relatively static, since a change in the text may imply a costly update to the index.

Indexes play an important role in search engines, such as Google or Yahoo. *Inverted indexes* are the standard type of index used in search engines for *natural language*. However, inverted indexes assume that there is a logical partition of the text into words. Hence, inverted indexes are not efficient for all patterns, as they are restricted to word or phrase queries. This is in fact a significant limitation, as pointed out by Navarro and Makinen [95]:

> "First, the keyword "natural language" excludes several very important Human languages. It refers only to languages where words can be syntactically separated and follow some statistical laws such as Heaps' (which governs the number of distinct words) and Zipf-Mandelbrot (which governs their frequency distribution) [Baeza-Yates and Ribeiro 1999]. This includes English and several other European languages, but it excludes, for example, Chinese and Korean, where words are hard to

separate without understanding the meaning of the text; as well as agglutinating languages, where particles are glued to form long "words", yet one wishes to search for the particles (such as Finnish and German). Second, natural language excludes many symbol sequences of interest in many applications, such as DNA, gene or protein sequences in computational biology; MIDI, audio, and other multimedia signals; source and binary program code; numeric sequences of diverse kinds; etc."

Despite these limitations, there are several reasons why inverted indexes are used. They have the merit of being simple to understand, and in fact, they correspond to the notion of index usually found in books. They are also very efficient in terms of time and space.

In this thesis however we are concerned with *full-text indexes*, i.e. we make no assumption on the structure of the text and expect to be able to search for any given pattern. The main problem with full-text indexes is their space requirements: from 4 to 20 times the size of the text [67, 79, 81]. Recall that indexes pay off when searching the text is time consuming because the text is large. The space requirements of full-text indexes severely compromise their applicability. In fact most full-text indexes are not designed to work in secondary memory (a notable exception was given by Ferragina et al. [34]). If a full-text index is forced to use secondary memory, its performance usually deteriorates considerably.

A common argument in favor of using inverted indexes instead of full-text indexes is precisely the huge space requirements of full-text indexes. Several attempts to reduce the redundancy present in full-text indexes originated new, more compact, full-text indexes [4, 16, 27, 30, 44, 61, 67, 124]. These indexes provided only a constant-factor reduction of space and, as such, were important, but not spectacular. In recent years, however, this situation has changed dramatically. A new kind of full-text index was exposed by a series of researchers: Kärkkäinen and Ukkonen [58, 59], Grossi and Vitter [51], Sadakane [110, 111], Ferragina and Manzini [35, 36]. These indexes became known as *compressed full-text indexes* and obtained their impressive results by combining *text compression* techniques with *succinct data structures* and full-text index theory.

A text compression technique is a way to encode the text in a format that requires less space than that of the original raw sequence and that still represents the original text. Since we are interested in reducing the space requirements of full-text indexes, a simple idea is to consider using text compression. It quickly becomes obvious this idea is not that simple. The common use of data compression focus only on reducing the text size. Once compressed, texts cannot be manipulated in any way. Compressed indexes however are radically changing this scenario, and with it our perception of data compression.

The text compression technique cannot be used as black box. In fact compressed indexes are heavily dependent on the technique that is used. From a high-level point of view, this idea makes perfect sense. We know that text compression provides a trade-off between the size necessary to store the text and the time it takes to consult a part of the text. This is an advantageous trade for full-text indexes, since we can afford that they become slower in exchange for requiring less space, which in turn makes the overall performance better, either because they do not need secondary memory or because they need less secondary memory. This trade off also makes sense from cache to main memory. Of course for this to be possible, the cost of consulting a part of the text must not be excessive. The naive approach of decompressing the whole text in order to read a specific character is not acceptable. Most compressed indexes actually solve this problem in a much more efficient way. Compressed indexes that are able to reproduce any portion of the original text efficiently are know as compressed self-indexes.

1.2 Succinct Data Structures

A succinct data structure is a compact representation of a data structure. Trees are a recurrent data structure in computer science and, in particular, play a central role in full-text indexing theory. It is therefore natural to consider succinct representations of trees. Clearly the less space we need to represent a tree, the less space our indexes will require.

Jacobson [57] was the first to study succinct data structures, such as trees and bitmaps (strings of 1's and 0's). Trees are commonly implemented with pointers which may not be the most space efficient way to store them. A tree can, for example, be represented as a string of left and right parentheses. This representation does not support by itself common operations efficiently, such as moving to a father node or to a child node, but it does represent the tree. Therefore a tree with n nodes can be represented with $2n$ bits. The work presented by Jacobson showed how to simulate tree traversals efficiently using only $o(n)$ extra bits. Clearly this kind of results is relevant for producing smaller full-text indexes. The fundamental tools supporting these kinds of data structures are the Rank and Select operations over bitmaps. The Rank operation counts the number of 1's up to a given position in the bitmap. The Select operation returns the location of the i-th 1 in the bitmap. Jacobson showed how to compute Rank in constant time, with only $o(n)$ extra bits. Later on, Munro [84] and Clark obtained constant-time solutions for Select, with $o(n)$ extra bits. The set of operations provided by succinct trees has been successively enlarged and improved by several researchers: Munro et al. [86], Benoit et al. [14] and Geary et al. [43]. The Rank and Select operations also proved to be useful for representing permutations [85]. Trees and permutations play a central role in full-text indexing theory, and therefore this kind of results account for a significant part of the success of compressed indexes.

1.3 Thesis contribution

The combination of text compression and succinct data structures was so impressive that a second wave of compressed indexes quickly followed, with contributions from Grossi et al. [49], Navarro [98] and Mäkinen et al. [76]. The interest on compressed indexes has grown since then. Several researchers have systematically improved the practical an theoretical performance of compressed indexes. Their functionality has also been greatly extended.

For a comprehensive and up to date survey see Navarro and Mäkinen [95]. Ferragina and Navarro gave an interesting description of the situation [128, Prologue]:

> "The new millennium has seen the birth of a new class of full-text indexes which are structurally similar to Suffix Trees and Suffix Arrays, in that they support the powerful substring search operation, but are succinct in space, in that it is close to the empirical entropy of the indexed data. They are therefore called compressed Suffix Trees and compressed Suffix Arrays, or in general compressed indexes.
>
> ...
>
> This interest is motivated by the large availability of textual data in electronic form, by the ever increasing gap in performance among the memory levels of current PCs, and by the "non negligible" space occupancy of classic data structures like Suffix Trees and Suffix Arrays which are pervading the BioInformatics and the Text Mining fields."

An interesting phenomenon, that motivated this thesis, was occurring in this "revolution". Essentially, compressed indexes based on the Lempel-Ziv compression algorithm [126, 127] seemed to be less efficient than other alternatives. This seemed bizarre since the Lempel-Ziv compression method is extremely popular (the gzip application is based on this type of compression). In fact a file compressed with these methods can be stored in $uH_k + o(u \log \sigma)$ bits [66], where H_k is a lower bound on the compression ratio of the best practical compressors. Our main contribution is to analyze this situation. We present a theoretical improvement of compressed indexes based on Lempel-Ziv compression method, which is significant in practice.

The first and main contribution of this thesis is to propose a compressed self-index based on the Lempel-Ziv, the inverted-LZ-index (ILZI), that requires $O(uH_k) + o(u \log \sigma)$ bits and whose dependency on m for finding patterns is linear, more precisely the time to report occurrences is $O((m+occ) \log u)$. We present a detailed study of this compressed self-index, both from a theoretical point of view and from a practical point of view. From a theoretical point of view, we present a new insight into the data structures used in Lempel-Ziv based compressed indexes and point out the main obstacle to linear time indexes based on the Lempel-Ziv compression. From a practical point of view, we explain the most important decisions that we took to implement this index in practice, achieving a fast and small index.

The second contribution of this thesis extends the functionality of the index we presented. In fact some full-text indexes, such as suffix trees, are useful for solving other problems related to strings. One such problem, that we consider to be fundamental, is the longest common substring problem: given two strings, a longest common substring is a largest string that occurs simultaneously in both strings. It is trivial to solve this problem in linear time using suffix trees. We consider that one string is the pattern P and that the other is the text T. This extra functionality of suffix trees is heavily used in bioinformatic applications.

We propose a solution for the longest common substring problem using the ILZI. We introduce the theoretical tools necessary to solve this problem efficiently. We also explain how to solve this problem from a more practical point of view.

We also explain how to adapt the ILZI for approximate string matching (ASM). Approximate string matching is an important subject in computer science, with applications in text searching, pattern recognition, signal processing and computational biology. The problem consists in locating all occurrences of a given pattern string in a larger text string, assuming that the pattern can be distorted by errors. To solve this problem we abstract away some of the structure of the ILZI and consider it as a variable-length q-samples index. By using this approach we present some new insights that are of independent interest. We compare our approach with other indexed alternatives for ASM, including other algorithms based on compressed indexes.

The remainder of this thesis is organized in the following way: Chapter 2 introduces fundamental notions of classical full-text indexes, namely suffix trees, suffix arrays, inverted indexes and a brief introduction to approximate string matching. Chapter 3 gives a brief introduction to compressed indexes, with a special focus on Lempel-Ziv compressed indexes. The chapter starts by explaining some data compression techniques and succinct data structures. In chapter 4 we describe our first contribution, a Lempel-Ziv based compressed index (ILZI), including demonstrations of theoretical properties and experimental results. In chapter 5 we explain how the ILZI can be used to determine a longest common substring between an indexed text and a given pattern. Chapter 6 considers approximate string matching over Lempel-Ziv compressed indexes, abstracted as irregular q-samples indexes. Chapters 5 and 6 also contain experimental results. In chapter 7 we present some conclusions and future work. Appendix A presents more exhaustive experimental results related to chapters 4, 5 and 6.

2

Basic Full-Text Indexes

In this chapter we describe the basic concepts related with the most popular full-text indexes, since they form the underlying theory of compressed indexes. We are interested in indexes that work for any kind of text, not just English. We are not particularly interested in variations of these indexes, designed to reduce the space requirements by a constant factor with no relation to text compression.

2.1 Basic Concepts and Notation

2.1.1 Model of Computation

In this thesis logarithms are assumed to be base 2 unless explicitly stated otherwise.

The model of computation we use is the *word* RAM [53]. This model considers a machine with $2^{O(w)}$ registers, each of which can store w bits. The model is random access, which means that consulting or updating a register requires $O(1)$ time. Note that we can address any of the $2^{O(w)}$ registers.

Moreover, in the word RAM model we assume that we can perform the operations usually available on modern computers (flow control, comparisons, basic arithmetic operations, bitwise shift and Boolean operations, etc.) in $O(1)$ time. The main difference between the word RAM model and weaker RAM models is that these operations are not always considered constant and therefore may require linear ($O(w)$) time to compute. This means

Table 2.1. Main symbols used in this thesis.

Symbol	Pages	Meaning
T	11	Text string
u	11	Original length of text string in characters, i.e. $\|T\|$
P	11	Pattern string
m	11	Length of pattern string in characters, i.e. $\|P\|$
Σ	10	Alphabet for P and T
σ	10	Alphabet size, $\sigma = \|\Sigma\|$
k		Number of errors
occ, occ_1, $occ_{>1}$	11,77	Number of occurrences of the pattern in the text, inside a block and spanning more than one block respectively
occ'	51	occurrences determined by an orthogonal range query
H_0,H_k	42,45	0-th/k-th order entropy of a text character
i, j	20	counters in the Descend and Suffix Walk algorithm or generic indexes
Z_i	48	Lempel-Ziv block
n	48	Number of LZ78 blocks of the text
ϵ	11	either the empty string or a small positive real number
$\mathcal{T}$, $\mathcal{T}_{78}$	73,82	dictionary and Lempel-Ziv suffix tree
d, t	72,73	number of nodes/points in the Lempel-Ziv suffix tree,
t	20	the tree depth in the FOR variant
$\mathcal{ST}$, $\mathcal{ST}_{78}$	74,83	Lempel-Ziv sparse suffix tree
d'	76	number of nodes in the Lempel-Ziv sparse suffix tree
$\mathcal{T}_{78}(T)$	73	$\mathcal{T}_{78}$-maximal parsing
f	73	size of the $\mathcal{T}_{78}$-maximal parsing
R	24	reverse mapping between trees
V	19,85	Descend and Suffix walk variant and block bitmap

that on a weaker model our results hold multiplied by a factor of w, which is $O(\log u)$ for a model with u registers. In general this is not a severe slow down.

2.1.2 Strings

A **string** S is a finite sequence of symbols taken from a finite **alphabet** Σ of size σ. The **size** of a string S, i.e. the number of symbols in S, is denoted by $|S|$. By $S[i]$ we denote the symbol at position ($i \bmod |S|$) of S, where $S[0]$ denotes the first symbol. The **empty string** is denoted by ϵ. The **concatenation** of two strings S, S', denoted by $S.S'$, is the string that results from appending S' to S.

The set of strings of Σ is denoted by Σ^*. The **reverse string** S^R of a string S is the string such that $S^R[i] = S[-i-1]$. A **string point** $S\langle i \rangle$ in a string S is the space between letters $S[i-1]$ and $S[i]$. A **prefix** $S[..i-1]$, **substring** $S[i..j]$ and a **suffix** $S[j+1..]$ of a string S are (possibly empty) strings such that $S = S[..i-1].S[i..j].S[j+1..]$.

Definition 2.1. *The* ***exact matching problem*** *consists in finding all occurrences of a (shorter) pattern string P in a (longer) text string T, i.e. $O = \{i \mid T[i..i+|P|-1] = P\}$. We denote $|P|$ by m, $|T|$ by u and $|O|$ by occ.*

As a running example we shall consider string $T = cbdbddcbababa$. We have that $u = 13$, $\Sigma = \{a, b, c, d\}$, $\sigma = 4$, $T[0] = c$, $T[-1] = a$ and $T^R = abababcddbdbc$. String points $T\langle 7 \rangle$ and $T\langle 10 \rangle$ are shown in Figure 2.1. The strings $cbdbddc = T[..6]$, $bab = T[7..9]$ and $aba = T[10..]$ are respectively a prefix, a substring and a suffix of T. Figure 2.1 shows the solution of the exact matching problem for patterns $cbdbddc$ and b. Usually we

```
                1
   0123456 789 012
T: cbdbddc.bab.aba           cbdbddcbababa
P: cbdbddc                    b b   b b b
```

Fig. 2.1. Schematic representation of the solution of the exact matching problem for patterns *cbdbddc* (left) and *b* (right) and text *cbdbddcbababa*.

consider 3 sub problems related to the exact matching problem: counting, which consists in determining occ; reporting, which consists in enumerating O; outputting, which consists in determining $T[i..i+\ell]$ for every i in O and some fixed ℓ.

It is usual to assume that T is terminated, i.e. that there is a terminator character \$ appended to T. This character does not match any other character in Σ and is lexicographically smaller than all the other characters.

2.1.3 Permutations

Permutations are a fundamental tool in computer science. They emerge naturally in the context of sorting algorithms and consequently in several areas of computer science. For a detailed explanation the reader should refer to the reference work by Knuth [64].

Definition 2.2. *A* ***permutation*** π *is a one to one mapping from a set to itself.*

Given a string we can associate to it a permutation, known as suffix array. We explain this process in Section 2.3. We will therefore start by studying permutations. The concepts presented will be useful to explain some concepts of strings in a simple and abstract way.

The number of permutations of u elements is $u! = 1 \times \ldots \times u$. By $\pi[i]$ we denote the value that π attributes to element $(i \bmod u)$. Index i starts at position 0. As an example consider the following permutation

$$\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 10 & 7 & 11 & 8 & 13 & 12 & 9 & 6 & 3 & 5 & 2 & 4 & 1 & 0 \end{pmatrix}.$$

So for example $\pi[3] = 8$. A permutation is an invertible function. The inverse of a permutation is also a permutation and will be denoted by π^{-1}. The inverse of the previous permutation is

$$\pi^{-1} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 13 & 12 & 10 & 8 & 11 & 9 & 7 & 1 & 3 & 6 & 0 & 2 & 5 & 4 \end{pmatrix}.$$

Orbits and Shift Permutations

We can apply a function iteratively to a given element e, i.e. $\pi^0[e] = e$ and $\pi^{i+1}[e] = \pi[\pi^i[e]]$. We refer to the set $\{\pi^i[e] \mid i \geq 0\}$ as the orbit of e. Observe that since the co-domain of permutations is finite its orbits are finite. This allows us to represent the orbit of e as $(e\ \pi^1[e]\ \pi^2[e]\ \ldots\ \pi^o[e])$ where $o+1$ is the size of the orbit. Permutations can therefore be represented as the set of its orbits. This is called cyclic notation. For simplicity we omit the set brackets in cyclic notation. In our example we have that $\pi = (0\ 10\ 2\ 11\ 4\ 13)(1\ 7\ 6\ 9\ 5\ 12)(3\ 8)$.

Definition 2.3. *A* ***cycle*** *is a permutation that has only one orbit.*

Cycles will play an important role in string processing. In fact, there is a general way to compute a cycle from a permutation. The idea is that given a permutation π we can build the following cycle: $(\pi[0]\ \pi[1]\ \pi[2]\ \ldots\ \pi[u])$.

Definition 2.4. *The* ***shift permutation*** $shift_\pi$ *of a permutation* π *is the cycle permutation such that its orbit is the permutation* π*, i.e.* $\pi[i] = shift_\pi^i[\pi[0]]$.

In our example $shift_\pi = (10\ 7\ 11\ 8\ 13\ 12\ 9\ 6\ 3\ 5\ 2\ 4\ 1\ 0)$, and therefore

$$shift_\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 10 & 0 & 4 & 5 & 1 & 2 & 3 & 11 & 13 & 6 & 7 & 8 & 9 & 12 \end{pmatrix}.$$

Observe that by computing the orbit of a shift permutation one can recover the original permutation π, provided we know $\pi[0]$. In fact it should be clear that u distinct permutations share the same shift permutation. Recovering π from its shift permutation plays a central role in compressed indexes and data compression. The shift permutation of π is given analytically as $shift_\pi[i] = \pi[\pi^{-1}[i]+1]$. This can be verified by iterating the shift permutation, i.e. $shift_\pi^2[i] = shift_\pi[shift_\pi[i]] = \pi[\pi^{-1}[\pi[\pi^{-1}[i]+1]]+1] = \pi[\pi^{-1}[i]+1+1] = \pi[\pi^{-1}[i]+2]$. In general $shift_\pi^j[i] = \pi[\pi^{-1}[i]+j]$, therefore $shift_\pi^i[\pi[0]] = \pi[\pi^{-1}[\pi[0]]+i] = \pi[0+i] = \pi[i]$. Likewise it is easy to see that $shift_\pi^{-1}[i]$ can be computed by $\pi[\pi^{-1}[i]-1]$. In fact one should observe that for cycles we have that $shift_\pi^u[i] = i$ and therefore $shift_\pi^{u-1}[i] = shift_\pi^{-1}[i]$. Therefore we can conclude that $\pi[\pi^{-1}[i]-1] = \pi[\pi^{-1}[i]+u-1] = shift_\pi^{u-1}[i] = shift_\pi^{-1}[i]$.

Runs

Runs allows us to explore the structure of permutations.

Definition 2.5. *A* ***run*** *of a permutation $shift_\pi$ is a maximal interval $[a, b]$ such that $shift_\pi[i+1] > shift_\pi[i]$ for all $i \in [a, b]$.*

As an example we show the runs of $shift_\pi$ separated by bars.

$$shift_\pi = \left(\begin{array}{c|ccc|ccccc|ccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 10 & 0 & 4 & 5 & 1 & 2 & 3 & 11 & 13 & 6 & 7 & 8 & 9 & 12 \end{array} \right).$$

Runs end in descents, i.e. a descent is an index i such that $shift_\pi[i] > shift_\pi[i+1]$. Therefore a permutation with k descents has $k+1$ runs. In our example $shift_\pi$ has 3 descents and 4 runs. We will use the notation $\left\langle {u \atop k} \right\rangle$ to denote the number of permutations of $\{1, 2, \ldots, u\}$ that have exactly k descents. These numbers are usually called *Eulerian numbers*. Given a permutation of $\{1, 2, \ldots, u-1\}$ we can form u new permutations by inserting u in all possible positions. If the original permutation has k descents then $k+1$ of the new permutations have k descents. The remaining ones will have $k+1$ descents. The idea is that by inserting u at the end of a run it remains a run, but inserting it in the middle splits it into two. Therefore we have the following recurrence:

$$\left\langle {u \atop k} \right\rangle = (k+1)\left\langle {u-1 \atop k} \right\rangle + (u-k)\left\langle {u-1 \atop k-1} \right\rangle, \quad 0 < k < u.$$

Finally we convention that the empty permutation has no descents, i.e.,

$$\left\langle {0 \atop 0} \right\rangle = 1.$$

The cases not contemplated by the recurrence or the above convention are assumed to be 0.

The length of the upwards runs in permutations can be used to test the randomness of a sequence [64]. What happened in the area of compressed indexes was that it became clear that suffix arrays (definition 2.14) are not random. This can be seen because they have very few runs, which are generally long runs. Since suffix arrays are not entirely random we can explore their regularity to reduce their space requirements.

The reverse permutation of a permutation $shift_\pi$ is the permutation that results from reading $shift_\pi$ left to right, i.e. $shift_\pi[-i-1]$. The average number of runs of a permutation is $(u+1)/2$. In fact the number of descents of a permutation plus the number of descents of the reverse permutation must add up to $(u+1)$. Observe that $shift_\pi$ has 4 runs but a random permutation of size 14 should have around 7.5 runs. The probability that a randomly chosen permutation has 4 runs or less is $\sum_{k=0}^{3} \left\langle {14 \atop k} \right\rangle / 14! \approx 0,002328$, which is fairly unlikely.

A fundamental observation used in compressed indexes is that the runs of the shift permutations expose regularities in π^{-1}. In fact the runs in $shift_\pi$ may be even more regular.

Definition 2.6. *A* ***natural run*** *of a permutation $shift_\pi$ is a maximal interval $[a, b]$ such that $shift_\pi[i+1] - shift_\pi[i] = 1$ for all $i \in [a, b]$.*

We show the natural runs of $shift_\pi$ that are longer than 1 in square brackets,

$$shift_\pi = \left(\begin{array}{c|ccc|ccccc|ccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 10 & 0 & [4 & 5] & [1 & 2 & 3] & 11 & 13 & [6 & 8 & 7 & 9] & 12 \end{array}\right).$$

Mäkinen [75] proposed the *Compact Suffix array*, a compressed index based on the exploration of natural runs. Suppose we would like to represent π^{-1} in a compressed form and that we find a natural run in $shift_\pi$. From the definition we get that for some interval $\pi[\pi^{-1}[i] + 1] = i + q$ which gets rewritten to $\pi^{-1}[i+q] - \pi^{-1}[i] = 1$. Let us look at an example. The interval $[4, 6]$ is a natural run of $shift_\pi$ of the form $shift_\pi[i] = i - 3$ therefore $\pi^{-1}[i-3] - \pi^{-1}[i] = 1$ for $i \in [4, 6]$, i.e.

- $\pi^{-1}[1] - \pi^{-1}[4] = 12 - 11 = 1$,
- $\pi^{-1}[2] - \pi^{-1}[5] = 10 - 9 = 1$,
- $\pi^{-1}[3] - \pi^{-1}[6] = 8 - 7 = 1$,

This property of π^{-1} is known as a self-repetition and can be explored to reduce the space requirements to store π^{-1}. The idea is to avoid storing the values of $\pi^{-1}[4], \pi^{-1}[5], \pi^{-1}[6]$ and compute their values from the values of $\pi^{-1}[1], \pi^{-1}[2], \pi^{-1}[3]$. Observe that this conclusion can just as easily been drawn from $shift_\pi^{-1}$ since a natural run in $shift_\pi$ implies a natural run in $shift_\pi^{-1}$. This can be seen by rewriting the above equation as $\pi[\pi^{-1}[i+q]-1] = i$.

$$\pi^{-1} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 13 & [12 & 10] & 8 & [11 & 9 & 7] & 1 & 3 & [6 & 0 & 2 & 5] & 4 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 13 & [12 & 10 & 8] & [11 & 9] & [7 & 1 & 3 & 6] & 0 & 2 & 5 & 4 \end{pmatrix}.$$

This approach may seem a bit far fetched. In fact there seems to be no clear reason to expect that this approach should work. Also it may seem that just because we are interested in permutations with long runs that does not necessarily entail that they should have a significant amount of natural runs. However, Mäkinen and Navarro [76] showed that this approach results in a compressed index, the compact suffix array.

2.2 Suffix Trees

We will now present a brief description of suffix trees, since they play a central role in full text-indexing theory and in the inverted-LZ-index. None of the concepts presented in this Section is new, but the way to expose them is. For a more detailed explanation, the reader is referred to one of the several references on the subject, e.g. the book by Gusfield [52].

Suppose we wish to locate all occurrences of pattern P in T. We start by considering simultaneously all suffixes of T. Then we discard all the suffixes that do not start by $P[0]$ and from those we discard the ones whose second letter isn't $P[1]$ and so on until we determine all the suffixes that start by P or we run out of suffixes. Clearly a naive way to use this approach would scan T in $O(um)$ time. A better way to do it is to arrange all the suffixes of T in a tree so that discarding the suffixes can be performed simply by descending in the tree.

First we need some concepts about trees. A **compact tree** is a tree that has no unary nodes, except, possibly, the ROOT. A **labeled tree** is a tree that has a nonempty string label for every edge. In a **deterministic tree**, the common prefix of any two different edges out of a node is ϵ. In a deterministic tree the first letters of every edge are referred to as **branching letters**. A **point** p in a labeled tree is either a node or a string point in some edge-label. The **path-label** of a point p in a labeled tree is the concatenation of the edge-labels from the root down to p. For deterministic trees we refer indifferently to points and to their path-labels, also denoted by p. The **string-depth** of a point p in a labeled tree is $|p|$, denoted by SDEP(p). FATHER(v) is the father node of node v. LETTER(v, i) equals $v[i]$, i.e. the i-th letter of the path-label of node v. DESCEND?(p, c) is true iff it is possible to descend from point p with c and DESCEND(p, c) returns the resulting point. By DFS(v) we refer to the depth-first time-stamp [28] of a node v in a tree and by DFS'(p) to the depth-first time-stamp of a point p in a labeled tree.

Definition 2.7. *The* ***generalized suffix tree*** $\mathcal{T}_{S_1,\ldots,S_k}$ *of a set of strings* $\{S_1, \ldots, S_k\}$ *is the deterministic compact labeled tree such that the path-labels of the leaves are the suffixes of the* $S_1\$, \ldots, S_k\$$ *strings, where* $\$$ *is a terminator symbol that does not belong to* Σ.

Usually the definition of generalized suffix trees uses different terminators for each string, but this is not necessary for this work. We will refer to generalized suffix trees just as suffix trees. Whenever it is convenient, we will omit the terminator symbol. Moreover, by leaves, we also refer to the internal nodes that are created by the terminator symbol.

The suffix tree of T is shown in Figure 2.2 and the suffix tree $\mathcal{T}$ of strings $a, b, ba, bd, cba, cbd, d$ is shown in Figure 2.3 (top-right). Solving the exact matching problem with a suffix tree consists in reporting all the leaves below point P. This takes $O(m + occ)$ time. Suffix trees can be stored in $O(u \log u)$ bits and built in $O(u)$ time, with a number of well known algorithms [81, 120, 121]. In this classical context edge-labels are stored as pointers to T. Therefore storing a suffix tree requires $O(u \log u)$ bits. In a good implementation the factor hidden by the O notation is around 10 [67].

2.2.1 Suffix-links and Descend and Suffix Walks

An element that is responsible for the flexibility of suffix trees is the suffix link. The **suffix-link** of a node v of a suffix tree is a pointer to node $v[1..]$, denoted by SuffixLink(v). Figure 2.2 shows the suffix links of nodes *ba* and *baba*. We define in an artificial way Suffix_Link(Root) as a node that descends to the root by every letter including terminator symbols. Several suffix tree algorithms use suffix links. One such algorithm is a greedy traversal of the tree. The traversal is greedy in the sense that the algorithm traverses the tree trying to maximize the string depth at all times. Suppose we are given pattern P and a suffix tree $\mathcal{T}$. A greedy traversal of P in $\mathcal{T}$ consists in trying to read a string P by starting from the root and descending as much as possible. When it is impossible to descend any further, we follow suffix-links until descending becomes possible again.

Definition 2.8. *The **descend and suffix walk** of a string P over a suffix tree $\mathcal{T}$ is the sequence $p_0 \ldots p_{2m}$ of points of $\mathcal{T}$ computed by Algorithm 1.*

It is important to notice that Algorithm 1 starts by appending to P a new terminator character $\$'$ that fails to match with any other character. The following lemma explains why the point at the beginning of the for cycle corresponds to the largest suffix of $P[..i-1]$ that is a point in $\mathcal{T}$.

Lemma 2.9 (For Invariant). *Before executing line 5 of Algorithm 1, it is always true that if $j' < j$ then $P[j'..i-1]$ is not a point in $\mathcal{T}$.*

Proof. First it should be obvious that, except in line 10, $point = P[j..i-1]$, since the SuffixLink (resp. Descend) and $j{+}{+}$ (resp. $i{+}{+}$) instructions are consecutive.

The lemma is proved by induction on i. The base is trivial. We assume that before line 7 is executed, if $j' < j$, then $P[j'..i]$ is not a point in $\mathcal{T}$. Our result follows immediately from this property by observing that the *point* and i are updated before reaching line 5 again.

The previous property can be proved by induction on the number of times the while loop ran on an iteration of the for loop. The base follows from the induction hypotheses of the lemma, by observing that, since $\mathcal{T}$ is suffix closed, if point $P[j'..i-1]$ is not in $\mathcal{T}$, neither is point $P[j'..i]$. Finally assume that the while's guard is true, i.e. NOT DESCEND?($P[j..i-1]$, $P[i]$). Therefore $P[j..i]$ is not a point in $\mathcal{T}$. Hence if $j' < j+1$ then $P[j'..i]$ is also not a point in $\mathcal{T}$. □

This lemma shows that the value of the point in line 6 is left maximal, i.e. cannot be extended to the left. Likewise the points in line 8 are right maximal, since the while's guard has just evaluated true. This gives a way to classify the points that were reached by the descend and suffix walk.

Definition 2.10. *The* ***left, right traces*** *of a string P over a suffix tree $\mathcal{T}$ are the subsequences of the descend and suffix walk given respectively by lines 6 and 8 of Algorithm 1.*

By **father_right**[i] (resp. **father_left**[i]), we refer to the lowest ancestor of $trace\_right[i]$ (resp. $trace\_left[i]$) that is a node of $\mathcal{T}$ and by **child_right**[i] (resp. **child_left**[i]), to the highest descendant of $trace\_right[i]$ (resp. $trace\_left[i]$) that is a node of $\mathcal{T}$. Table 2.3 (top) shows the descend and suffix walk of *cbdbddc* in $\mathcal{T}$.

Finally we will briefly explain why Algorithm 1 runs in $O(m)$ time. First it should be clear that Algorithm 1 does terminate.

Theorem 2.11. *Expression $V(i) = 3m - i - 2j - t$ is a variant of the for loop, where t is the tree depth of the point. Therefore Algorithm 1 terminates.*

Proof. Suppose that $V(i) \leq 0$. Since $t \leq i - j$, then $3m - 2i - j \leq 3m - i - 2j - t$. Since $j \leq i$, then $3m - 3i \leq 3m - 2i - j$. Therefore $3m - 3i \leq 0$, hence $m \leq i$ and the for cycle terminates.

Except for instruction 10, it should be evident that $\varDelta V = V(i+1) - V(i) < 0$ for any i, since j is non-decreasing and i is strictly increasing for each iteration of the for loop. The problem with the Suffix_Link operation is that it may cause t to decrease. However t can decrease at most by 1. The factor 2 associated with j compensates this effect. Therefore in every iteration of the while cycle $\varDelta V < 0$. □

Algorithm 1 Descend and Suffix Walk

1: **procedure** $Descend\_and\_Suffix(P)$
2: $P \leftarrow P.\$'$
3: $j \leftarrow 0$
4: point $\leftarrow$ Root
5: **for** $i \leftarrow 0, i < |P|$ **do**
6: trace-left$[i]$ $\leftarrow$ point
7: **while** NOT Descend?(point, $P[i]$) **do**
8: trace-right$[j]$ $\leftarrow$ point
9: j++
10: point $\leftarrow$ Suffix_Link(point)
11: **end while**
12: point $\leftarrow$ Descend(point, $P[i]$)
13: **end for**
14: **end procedure**

i	0	1	2	3	4	5	6	7
P[i]	c	b	d	b	d	d	c	$'
trace_left[i]	ϵ	c	cb	cbd	b	bd	d	c
DFS'(father_left[i])	0	0	6	8	2	4	9	0
DFS'(trace_left[i])	0	5	6	8	2	4	9	5
DFS'(child_left[i])	0	6	6	8	2	4	9	6
trace_right[i]	cbd	bd	d	bd	d	d	c	ϵ
DFS'(father_right[i])	8	4	9	4	9	9	0	0
DFS'(trace_right[i])	8	4	9	4	9	9	5	0
I(trace_right[i])	[8,8]	[4,4]	[9,9]	[4,4]	[9,9]	[9,9]	[5,8]	[0,9]
DFS'(child_right[i])	8	4	9	4	9	9	6	0

Table 2.3. Descend and suffix walk of *cbdbddc* in $\mathcal{T}$

For now we assume that the operations DESCEND and DESCEND? are computed in constant time and later give a more realistic analysis. The problem of analyzing the time of Algorithm 1 is that operation SUFFIXLINK is computed for points, not just nodes, and therefore doesn't necessarily run in constant time.

Lemma 2.12 (Skip/count trick). *The* SUFFIX_LINK *function runs in* $O(\Delta t + 2)$ *time, where* Δt *is the variation of tree depth.*

Proof. Computing the SUFFIX_LINK for the nodes of $\mathcal{T}$ can be done in $O(1)$ by storing the suffix-links in $\mathcal{T}$. For a point, the idea is to first use the suffix link of its father node and then descend until the string depth is equal to the string depth of the original point less 1. In order to descend, it is not necessary to read the whole edge labels. The reason is that $P[j+1..i-1]$ must be a point in $\mathcal{T}$ since $P[j..i-1]$ is. Therefore we only need to check the branching letters of the nodes we find along the way. Hence we conclude that this procedure can be computed in $O(\Delta t + 2)$ time. $\square$

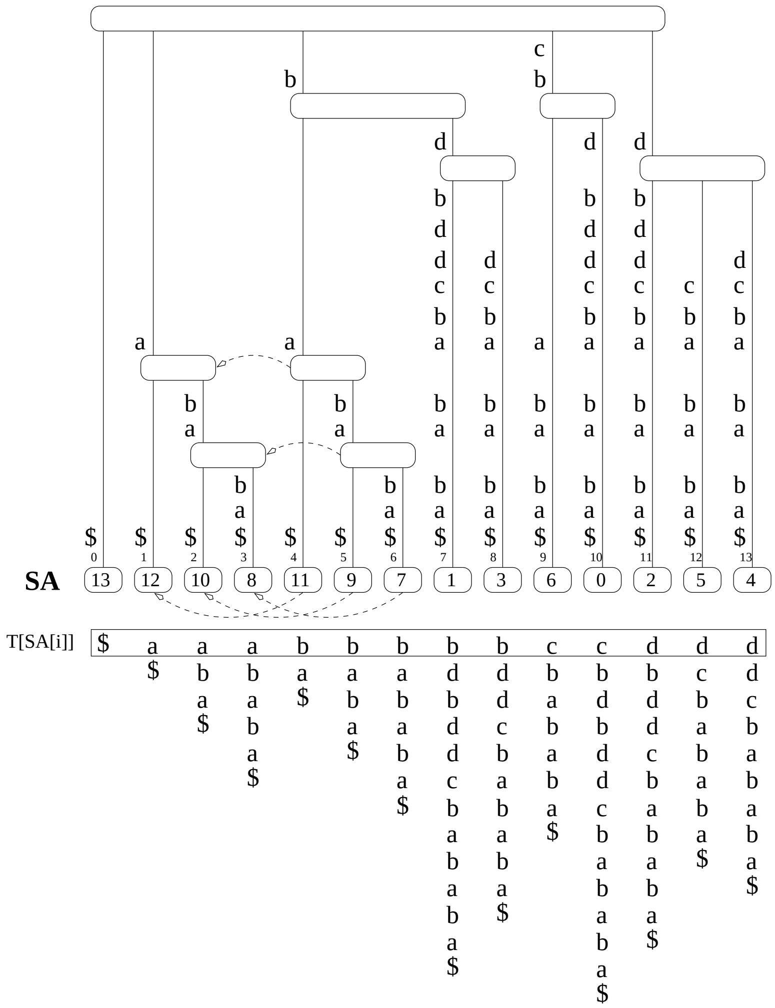


Fig. 2.2. (top) Suffix tree for T. Some suffix links are shown (dashed arrows). (bottom) Suffix array of T.

Observe that $-\Delta V$ counts all the operations executed in an iteration of the for loop, including the time to compute Suffix_Link. Therefore Algorithm 1 runs in $O(V(0)) = O(m)$ time.

The algorithm presented by McCreight and Ukkonen to build a suffix tree consists in computing the descend and suffix walk of T on an evolving suffix tree. The suffix tree begins with the Root and Suffix_Link(Root) nodes. The algorithm is a descend and suffix walk, where before line 9 is executed we add a new leaf to the tree. The new leaf

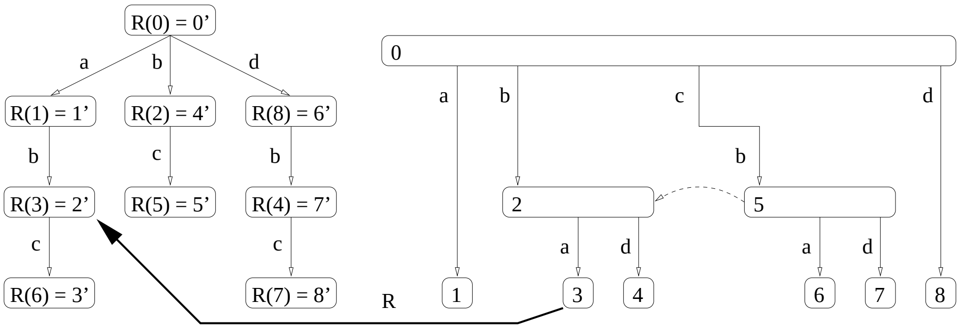


Fig. 2.3. (right) Suffix tree for strings $\{a, b, ba, bd, cba, cbd, d\}$. Suffix link from *cb* to *b* shown by a dashed arrow. Nodes show their DFS value in $\mathcal{T}$. (left) Reverse tree of the suffix tree on the right. Nodes show their DFS value in $\mathcal{T}^R$. The R mapping is shown and $R(3)$ is indicated by a bold arrow.

is linked to an internal node, possibly new, located in the current point position and edge labeled by $T[i..]$. Since it is a descend and suffix walk, this algorithm takes $O(u)$ time.

2.2.2 The Suffix tree/Suffix-link Duality and Backward Searching

Weiner presented the first algorithm for building suffix trees [121]. Interestingly enough this algorithm was based on backward search, a concept that was recently introduced by Ferragina and Manzini [35] in compressed indexes.

The idea of backward search consists in using the suffix tree backwards. To determine the point P in $\mathcal{T}$ we start by descending by $P[-1]$, instead of $P[1]$. For example suppose we wish to locate pattern *cbd* in $\mathcal{T}$ of Figure 2.3. We start by descending by $d = P[-1]$. We now have a serious problem, since it is not clear how to move to point *bd*. In this particular case since d corresponds to a node, the one with DFS value 8, we can use another data structure to solve this problem.

Definition 2.13. *The* ***reverse tree*** $\mathcal{T}^R$ *of a suffix tree* $\mathcal{T}$ *is the minimal labeled tree that, for every node* v *of* $\mathcal{T}$*, contains a node* v^R*, where* v^R *denotes the reverse string of* v.

The tree $\mathcal{T}^R$ is shown in Figure 2.3 (top-left). Observe for example that, since *cbd* is a node of $\mathcal{T}$, there is a node $cbd^R = dbc$ in $\mathcal{T}^R$. We define a canonical mapping R that, for every node v in $\mathcal{T}$, maps $\textsc{Dfs}(v)$ to $\textsc{Dfs}(v^R)$ (see Figure 2.3). We will use $R(v)$ to denote $R(\textsc{Dfs}(v))$. Note that since the nodes of $\mathcal{T}$ form a suffix closed set, the nodes of $\mathcal{T}^R$ form a prefix closed set.

It is now very easy to solve our problem by descending in $\mathcal{T}^R$. We map the node d to $\mathcal{T}^R$, by R, and obtain a node with $\textsc{Dfs}_{\mathcal{T}^R}$ value $6'$. If we descend by b and then by c we reach a node with a $\textsc{Dfs}_{\mathcal{T}^R}$ value of $8'$. This node can be mapped back to $\mathcal{T}$ with R^{-1} to the node *cbd* with the $\textsc{Dfs}$ value 7. This way we are able to search for P backwards in $\mathcal{T}$.

Of course this may look like cheating since we are building another data structure to be able to do this search. The interesting fact is that we are not. In fact the structure of $\mathcal{T}^R$ is already present in $\mathcal{T}$ in the form of suffix-links. Observe for example that the suffix-link from *cb* to *b* can be computed by: mapping *cb* to $\mathcal{T}^R$ (node $5'$); moving up to the father of that node (node $4'$); mapping back to $\mathcal{T}$ (node 2). This means that the structure of $\mathcal{T}^R$ is actually stored in the suffix-links. This information however only moves upwards in $\mathcal{T}^R$. Weiner stored the suffix-links reversed, i.e. as pointers from $v[1..]$ to v for every node v of $\mathcal{T}$. Note that we also need to store letter $v[0]$.

The $\mathcal{T}^R$ tree is, by itself, not enough to do backward search for every point p of $\mathcal{T}$, specially when p is not a node. Consider for example how to move from point *bab* to point *abab* in the suffix tree of Figure 2.2. The best way is to move up to node *ba*, then follow a Weiner-link to node *aba* and finally descend to point *abab*. The reader might get the impression that it is enough to follow the Weiner-link from the father node of the point, this however is not the case. Consider for example how to move from point *bab* to point *cbab*. In this case we must move up to node *b*, from that node we follow a Weiner-link to node *cb* and then descend to point *cbab*. Note that the time of reading a pattern P backwards on $\mathcal{T}$ amortizes to $O(m)$ and that the time to move from one point to the next is equal to the variation in tree depth of the points when using the skip/count trick to move in the tree.

In order to be able to use the skip/count trick coherently we need to store some more information. Note for example that with the previous procedure it might seem that it is possible to move from point *bdd* to point *cbdd* while, in fact, we would end up in point *cbdb*. In fact it is not enough to store $\mathcal{T}^R$. We need information about the suffix tree of T^R. In particular for the nodes v we traverse upwards, we need to know if there is a point p such that $p[1..] = v$, even if p is not a node. Only if there is such a point p for which $p[0]$ is the letter we are prepending is this procedure valid. Weiner stored this information as bits in the nodes of $\mathcal{T}$.

We show the duality between the suffix trie (suffix tree not in compact form) of $T = aaabbb$ and T^R in Figure 2.4 This duality between suffix trees was first explored by Weiner [121], that presented the first algorithm for building suffix trees. It was later pointed out by Gusfield [52]. Recently it was explored by Stoye [115] and Maaß [72]. Ferragina and Manzini [35] found an efficient process to do backward search over suffix arrays that exploits the compressibility of T.

2.3 Suffix Arrays

Suffix arrays resulted from an attempt to reduce the space requirements of index structures [46, 79]. Like suffix trees, suffix arrays give valuable insight into the structure of strings.

Definition 2.14. *The* ***suffix array*** SA *of a text* T *is the permutation that gives the lexicographical ordering of the suffixes of* T*, i.e.* $T[SA[i-1]..] < T[SA[i]..]$ *for all* $0 < i < u$.

Therefore $SA[i]$ responds to the question: "which suffix is in the i-th position in the lexicographical ordering ?". Suffix arrays can be obtained from suffix trees by traversing its leaves in lexicographical order (see Figure 2.2). In practice, however, it is better to build them directly. Several algorithms have been proposed to build suffix arrays, from the original $O(u \log u)$ time [79] to the newest $O(u)$ time algorithms [17, 63, 65]. Storing a suffix

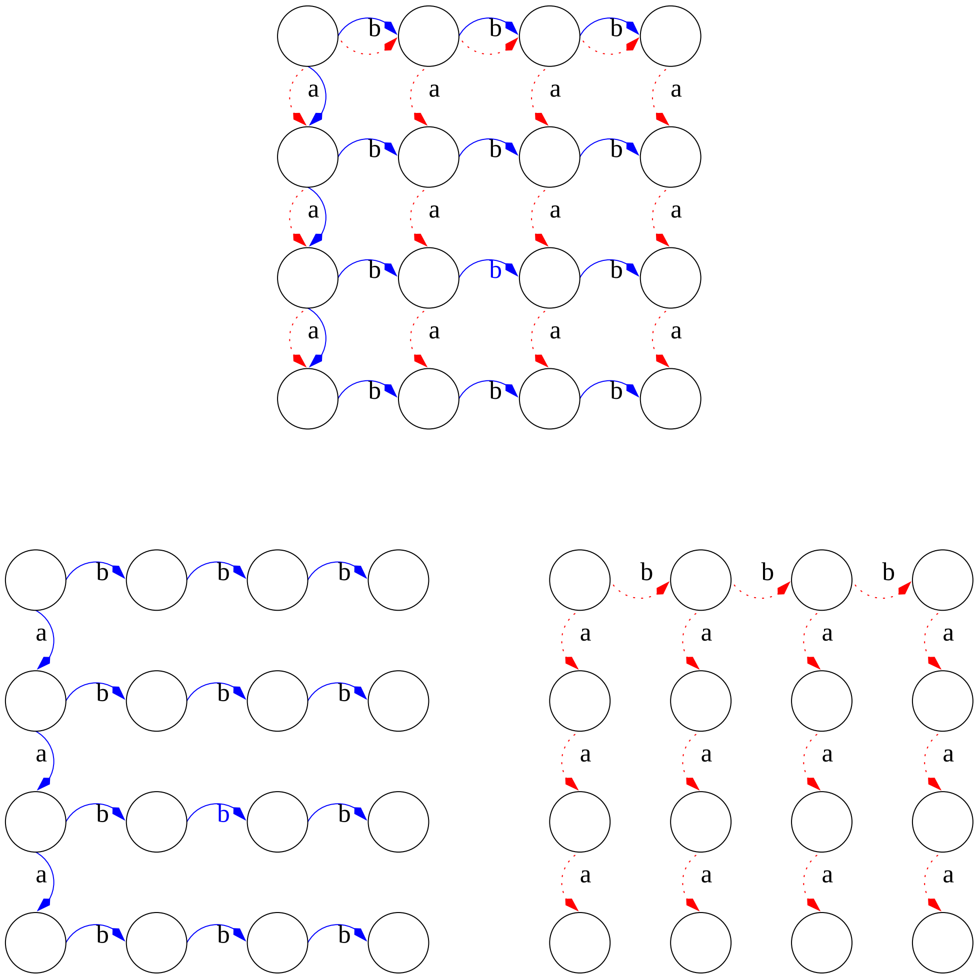


Fig. 2.4. Suffix trie for string *aaabbb* (left) and string *bbbaaa*(right). Overlap of both suffix tries (top)

array requires $u(\log u + \log \sigma)$ bits, since we must also store T. Figure 2.2 shows the suffix array of T.

To every point p, we associate the range of leaves that are descendants of p. For example string ba is associated with the interval $[4, 6]$ that corresponds to suffixes $\{11, 9, 7\}$. The exact matching problem can be solved using suffix arrays by computing the range associated to P. Using binary searches, this can be achieved in $O(m \log u + occ)$ time. This can be further improved to $O(m + \log u + occ)$ by storing information about the longest common prefix for consecutive suffixes [2, 79]. Observe that, after computing the range associated to P, we can compute *occ* in constant time, i.e. count the number of occurrences. These values are obtained from the edges of the range, in our example it would be computed as $6-4+1=3$. This property of suffix arrays led to a division of the exact matching problem in two versions. The first, referred to as counting, consists in computing *occ*; the second, referred to as reporting, consists in reporting the locations of each occurrence of P in T.

2.3.1 Suffix-Links over Suffix Arrays

Since suffix arrays are permutations their inverse are also permutations. The inverse of the suffix array SA is the **rank array**, i.e. $RA = SA^{-1}$. Therefore $RA[i]$ responds to the question: "what is the ranking of suffix i in the lexicographical order ?". The rank array for the suffix array of Figure 2.2 is the following :

$$RA = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 10 & 7 & 11 & 8 & 13 & 12 & 9 & 6 & 3 & 5 & 2 & 4 & 1 & 0 \end{pmatrix}.$$

The rank array indicates where in the suffix array is stored a given suffix. For example consider suffix aba of T. Since $aba = T[10..]$ and $RA[10] = 2$ we should have that $aba = T[SA[2]..]$, i.e. $SA[2] = 10$. Observe that RA of T is the same permutation that was presented in Subsection 2.1.3 as π.

We introduce the ψ function. The ψ function is a permutation that plays the role of suffix-links in suffix arrays. In the context of compressed indexes it is also used to explore the regularities of SA [51, 110]. Our exposition of the ψ function is motivated by the work of Crochemore et al. [29].

Definition 2.15. *The ψ function of a text T is the shift permutation of the rank array of T. The inverse of the ψ function is called the Last-to-First mapping.*

Figure 2.5 shows the ψ function and the LF mapping of T. According to Subsection 2.1.3 ψ can be computed by the expression $RA[RA^{-1}[i]+1] = SA^{-1}[SA[i]+1] = RA[SA[i]+1]$. The last expression gives the intuition behind the ψ function. The expression reads "where in the

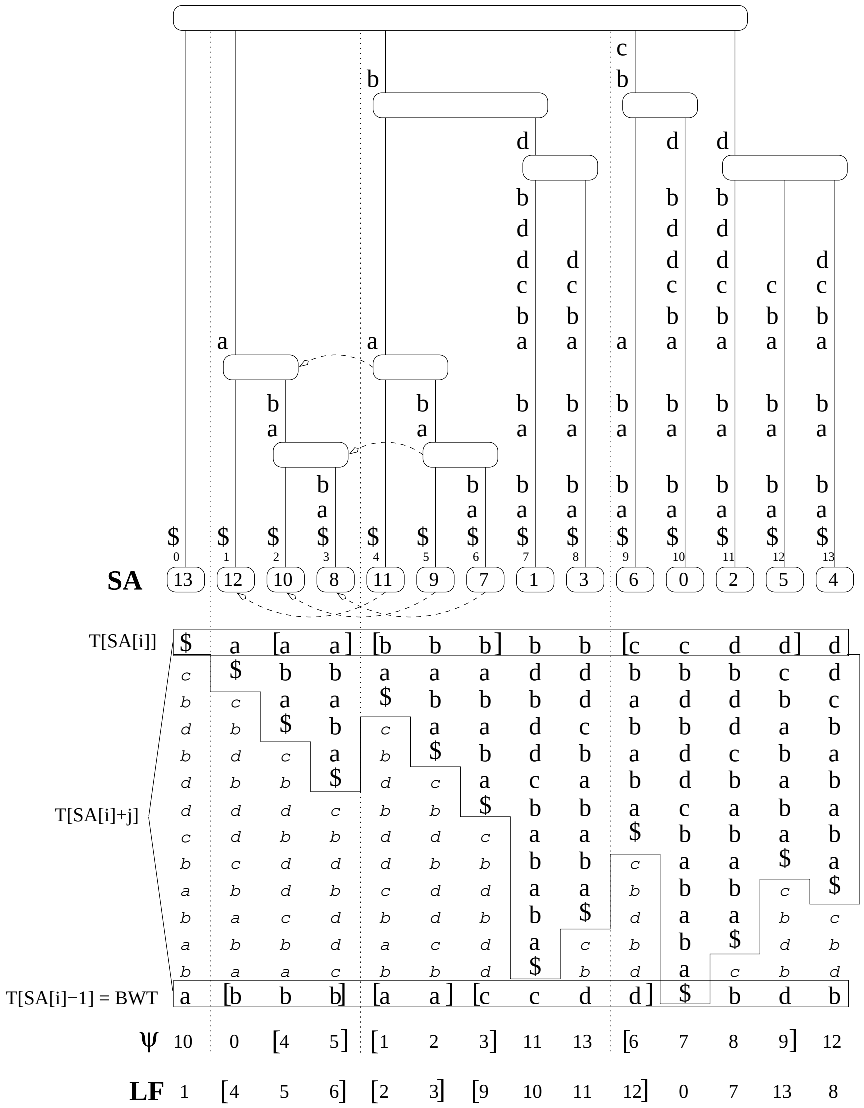


Fig. 2.5. (top) Suffix tree for T. Some suffix links are shown (dashed arrows). (bottom) Suffix array of T, its Burrows-Wheeler transform, the ψ and LF permutations.

suffix array is the suffix $SA[i]+1$?", i.e. the suffix after $SA[i]$. The expression in the middle shows this statement in another way since $T[SA[\psi[i]]..] = T[SA[SA^{-1}[SA[i]+1]]..] = T[SA[i]+1..]$. For example $aba = T[10..] = T[SA[2]..]$ and $T[SA[\psi[2]]..] = T[SA[4]..] = T[11..] = ba$. Obviously the LF-mapping does exactly the inverse of the ψ function, i.e. it indicates where is the previous suffix.

The LF-mapping can be used to perform backward search in a way similar to using Weiner-links (see Subsection 2.2.2). The LF-mapping itself is much like the reverse tree of a suffix tree. In order to be able to perform backward searching we need to extend it to the ELF-mapping, that for a given leaf with index i, corresponding to suffix $T[SA[i]..]$, and a letter ℓ returns another leaf with index $ELF[i, \ell]$, corresponding to suffix $T[SA[ELF[i, \ell]]..]$, such that any prefix of $\ell.T[SA[i]..]$ that is a node of the suffix tree of T is also a prefix of $T[SA[ELF[i, \ell]]..]$. This definition however is ambiguous, since there can be different values that satisfy this property. For our purposes any value that satisfies the previous property is acceptable. In Subsection 3.1.2 we give a closed formula for ELF.

For example consider $i = 8$, note that $SA[8] = 3$ and that $T[SA[8]..] = bddcbababa\$$, suppose we consider $\ell = c$. There is no string $c.bddcbababa\$$ in T, the largest prefix of this string that is a substring of T is *cbd*. Therefore the only node we need to worry about is node *cb* because of node *b*. In this case both leaves at indexes 9 and 10 satisfy the property we want, in particular we choose 10. Therefore since $SA[10] = 0$ we have that $ELF[8, c] = 0$.

Observe that the ELF-mapping does extend the LF-mapping. In fact $ELF[i, T[SA[i] - 1]] = RA[SA[i] - 1] = LF[i]$. The last equality is essentially the definition of LF mapping. The first equality is the observation that if we consider the suffix $T[SA[i]..]$ and prepend to it the letter $T[SA[i]-1]$, we end up with suffix $T[SA[i]-1..]$ which is a leaf and therefore a node of the suffix tree of T. In this case there is no ambiguity for ELF which must respect the first equality.

In order to perform backward searching effectively on suffix arrays we need to compute the ELF mapping efficiently. However, before we can do that, we need further insight into suffix arrays.

2.3.2 Self-Indexing and Counting Suffix Arrays

This subsection is based on the work of Sadakane [110] and of Schürmann and Stoye [114] and aims to explain some properties of suffix arrays.

Consider the application that assigns to every string T the permutation SA. A curious fact about this application is that it is not injective, i.e. distinct strings may have the same suffix array. Studying this phenomena is important since it will allow us to construct self-indexes. A self-index is an index from which we can recover the original string. If two distinct strings can share the same suffix array then suffix arrays are not by themselves self-indexes. This means that in order to determine which string ST originated a given suffix array SA we need to store more information. The important question is: how much more information ?

As a simple example, observe that strings *aaaa* and *dcba* share the same suffix array. Understanding the relation between the strings that share the same suffix array will allow us to add information to suffix arrays in order to turn them into self indexes. One obvious way to achieve this goal is to add T itself. We are however interested in using less than $O(u \log \sigma)$ bits.

The following characterization of suffix arrays is essential to describe the strings that share the same suffix array. This characterization was first given by Burkhardt and Kärkkainen [17] and equivalent propositions were given by Duval and Lefebvre [32].

Theorem 2.16. *A permutation SA is the suffix array of string T iff the following conditions hold:*

- $T[SA[i]] \leq T[SA[i]+1]$
- $\psi[i] > \psi[i+1] \implies T[SA[i]] < T[SA[i+1]]$

Proof. Suppose that SA is the suffix array of T. Since SA is the lexicographical order of the suffixes, the first condition must hold (see Figure 2.2). The second condition states that if the relative order of two entries of SA changes when we remove the first character of both then they must disagree in that character, i.e. $T[SA[i]] < T[SA[i+1]]$.

Proving that a permutation with the above properties is the suffix array of T consists in using induction with the two above properties. The reader is referred to the work of Schürmann and Stoye [114]. $\square$

Since Theorem 2.16 is a complete characterization of suffix arrays it should now be clear that what is essential to tell apart the strings that share the same suffix arrays is the string $T[SA[i]]$. In fact, given a permutation SA and a compatible string $T[SA[i]]$ we can recover T by inverting SA, since $T[SA[SA^{-1}[i]]] = T[i]$. A self-index must therefore find a way to store string $T[SA[i]]$. This string however is extremely regular and there is no point in storing u letters (see Figure 2.2).

Definition 2.17. *The* ***count*** *function* $C : \Sigma \rightarrow [0, u-1]$ *of a string* T *is defined as follows:*

- $C(l) = i$ *if* $T[SA[i]] < l$ *and* $T[SA[i+1]] = l$.

The count function of our running example is the following:

$$C = \begin{pmatrix} a\ b\ c\ d \\ 0\ 3\ 8\ 10 \end{pmatrix}.$$

Simply storing the C function in an array allows for C to be computed in $O(1)$ time and uses up only $O(\sigma \log u)$ bits. In this representation we can compute $T[SA[i]]$ in $O(\log \sigma)$ time by doing a binary search. This can be further improved with a bitmap or a compressed bitmap. We will discuss these approaches in Chapther 4. However in practice this was the way we implemented self-indexing in the ILZI, the compressed index that we will discuss in Chapther 4. Note also that it is easy to extract a substring $T[SA[i]..SA[i]+\ell]$ by using the ψ function and outputting $T[SA[i]]$, $T[\psi[SA[i]]]$, $T[\psi^2[SA[i]]]$ and so on.

The second condition of Theorem 2.16 is illustrated in Figure 2.5 by vertical dotted lines. This condition relates the smallest alphabet that can generate a suffix array and the descents of ψ. Let k be the number of descents of ψ ignoring position 0. In fact since position 0 corresponds to the terminator character we must always have that $\$ = T[SA[0]] \neq T[SA[1]]$. According to this condition we must have that $k < \sigma$. In our example $\sigma = 4$ and $k = 2$. Counting the number of strings that share the same suffix array is now a simple matter of counting the number of C functions that are compatible with ψ. The number of such compatible C functions is $\binom{u+\sigma-1-k}{\sigma-1-k}$.

For every suffix array there is a base string for which $\sigma = k - 1$. Bannai et al. [11] presented an algorithm to compute this string. It consists essentially in considering $T[SA[i]]$ as the string that has always the same letter in the runs of ψ. In our example this is not the case because of the last run of ψ (see Figure 2.5).

This analysis not only explains how one can use suffix arrays as self indexes, but it also gives some insight into why compressed indexes have troubles with large alphabets. The fewer runs a permutation has the more regular it is and therefore the less space we will need to represent it. In this Section we have shown that σ is an upper bound on the number of runs of ψ and, therefore σ influences the complexity of storing ψ. This, of course, is also a measure of the complexity of storing RA and SA (see Subsection 2.1.3).

This relation between σ and the complexity of storing an index of T allows us to give another overview of how compressed indexes work. Observe that if σ was close to u there probably would be very few redundancy both in T and in its suffix array. This means that, in this case, the suffix array would already be compressed [41]. Therefore a good way to reduce the space requirements of SA is to transform T into another (shorter) string with a larger alphabet. An index for that string would already be compressed. The problem is how to use such an index as an index for the original string. In fact this is how most compressed indexes work. The mapping usually consists in grouping letters into blocks. In the Lempel-Ziv based indexes the way to group letters is determined by the Lempel-Ziv parsing. The compressed suffix arrays group the letters into blocks of size $2^{\lceil \log \log u \rceil}$. However, for FM-indexes this description is not very natural.

Table 2.4. 2-grams index for $T = cbdbddcbababa$.

2-grams	\$	a\$	ab	ba	bd	cb	db	cc	dd
suffixes	13	12	8, 10	7, 9, 11	1, 3	0, 6	2	5	4
gaps	13	12	8, 4	7, 2, 2	1, 2	0, 6	2	5	4

Table 2.5. 2-samples index for $T = cbdbddcbababa$, with $h = 2$.

2-grams	a\$	ab	cb	db	dd
suffixes	12	8, 10	0, 6	2	4

2.4 Inverted Indexes and q-grams/q-samples Indexes

For some applications we do not need to search for patterns larger than a given q, i.e. $m \leq q$. If this is the case we can cut the suffix tree of T at string-depth q, i.e. we do not need to organize the suffixes beyond string-depth q. This means that the sub-trees below string-depth q are stored as arrays that are not lexicographically sorted. In general they may be sorted by the position in T, as shown in Table 2.4. This kind of index is known as q-grams index.

This kind of index still requires a lot of space since we must store $O(u)$ numbers. This can be stored in a compressed format by storing the gaps between consecutive suffixes (see Table 2.4) with Elias-δ coding [33, 122] (see Subsection 3.1.1). With Elias-δ coding, representing an integer x requires $\log x + 2 \log\log x + O(1)$ bits, but it is possible to recover the numbers from a binary string of consecutive numbers. Representing the array of suffixes in this way requires $u(q \log \sigma + 2 \log(q \log \sigma) + O(1))$ bits in the worst case, i.e. when the gaps between the samples are of size σ^q. Therefore, for a small q, this requires less space than suffix arrays.

An index that requires even less space than the q-grams index is the q-samples index. The idea of the q-samples index is that the text is sampled every h position, i.e. we are only going to index suffixes $T[0..]$, $T[h..]$, $T[2h..]$ and so on. Usually, we assume that the samples do not overlap, i.e. that $q \leq h$. This means that a q-samples file only needs to store

Table 2.6. Inverted index for $T = to\_be\_or\_not\_to\_be$.

dictionary	be	not	or	to
suffixes	3, 16	9	6	0, 13

u/h numbers. Moreover each number requires even less space since it can be stored divided by h. Table 2.5 shows an example of a q-samples index with $h = q = 2$. The problem with the q-samples index is that it cannot find every occurrence of a pattern of size q efficiently, since the pattern may appear spread across q-samples.

A variation of the q-samples index for which this problem is not very relevant is the inverted file. Inverted files are used for natural language texts, like English. Instead of considering all q-samples we are given a dictionary containing the words of T. We store the suffixes only for the words in the dictionary. See Table 2.6.

The reader may wonder if it is possible to use the same reasoning that transformed q-grams into q-samples to reduce the space requirements of suffix trees. This was explored by Kärkkäinen et. al. [61] that introduced the notion of sparse suffix tree. This idea is also used in compressed suffix arrays [110].

2.5 Approximate String Matching

We finish this chapter with a brief overview of the basic concepts related with approximate string matching. In Chapther 6 we will present an algorithm to perform approximate string matching with the inverted Lempel-Ziv index, that uses some of these concepts.

Approximate string matching is an important subject in computer science, with applications in text searching, pattern recognition, signal processing and computational biology. The problem consists in locating all occurrences of a given pattern string in a larger text string, assuming that the pattern can be distorted by errors. If the text string is long, it may be infeasible to search it on-line, and we must resort to an index structure. This approach has been extensively investigated in recent years [8, 9, 10, 24, 56, 88, 100, 119, 123].

State of the art algorithms are hybrid, and divide their time into a *neighborhood generation* phase and a *filtration* phase [88, 99].

During the neighborhood generation phase we compute all the strings that can be obtained from the pattern by a small amount of distortions. This can be computed with dynamic programming over a suffix tree [8, 24, 119].

Filtration consists in determining text areas, that do not contain matches, using techniques less expensive than dynamic programming. This approach has the obvious drawback that it cannot exclude all such areas, the remaining points have to be inspected with other methods. In the indexed version of the problem, filtration can be used to reduce the size of neighborhoods, hence speeding up the algorithm. The most common filtration technique splits the pattern and later on tries to expand it around potential matches. We will give a more detailed explanation of filtration in Chapther 6. The way the pattern is split determines the balancing point for the hybrid algorithm. Myers [88] and Baeza-Yates and Navarro [99] presented a detailed treatment of this subject. They also describe the limitations of the method, including the fact that above a given error level the complexity of the method becomes linear on the size of T.

2.5.1 Basic Concepts

Definition 2.18. *The* edit *or* Levenshtein *distance,* $\mathrm{ed}(S, S')$*, between two strings is the smallest number of edit operations that transform S into S'. We consider as operations insertions (I), deletions (D) and substitutions (S).*

For example:

```
D S I
abcd
 bedf
```

$ed(abcd, bedf) = 3$

The edit distance between strings S and S' can be computed by filling up a dynamic programming table $D[i, j] = ed(S[0..i-1], S'[0..j-1])$, constructed as follows:

Table 2.7. Table $D[i,j]$ for *abbaa* and *ababaac*.

	col	0	1	2	3	4	5	6	7
row			*a*	*b*	*a*	*b*	*a*	*a*	*c*
0		0	1	2	3	4	5	6	7
1	*a*	1	0	1	2	3	4	5	6
2	*b*	2	1	0	1	2	3	4	5
3	*b*	3	2	1	1	1	2	3	4
4	*a*	4	3	2	1	2	1	2	3
5	*a*	5	4	3	2	2	2	1	$\boxed{2}$

$$D[i,0] = i, \qquad D[0,j] = j$$

$$D[i+1,j+1] = D[i,j], \text{ if } S[i] = S'[j]$$

$$1 + \min\{D[i+1,j], D[i,j+1], D[i,j]\}, \text{otherwise.}$$

Table 2.7 shows an example of the dynamic programming table D.

A different, yet related, approach for the computation of the edit distance is to use a non-deterministic automaton (NFA). We can use a NFA, denoted N_P^k, to recognize all the words that are within *edit* distance k from another string P. Figure 2.6 shows an automaton that recognizes words that are at distance at most one from *abbaa*, where Σ represents any symbol. It should be clear that the word *ababaa* is recognized by the automaton since $ed(abbaa, ababaa) = 1$. A comprehensive survey about these algorithms is available [97], and should be consulted for a more complete description.

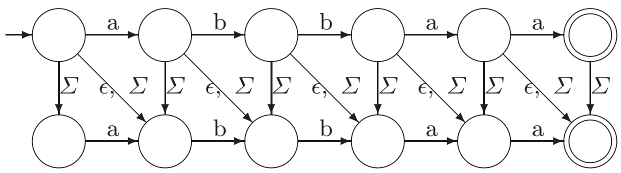


Fig. 2.6. Automaton N_P^k for *abbaa* with at most one error.

Figure 2.7 shows the computation performed by automaton N_P^k when the input string is *ababaac*.

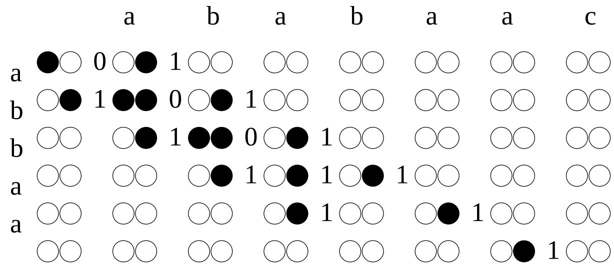


Fig. 2.7. Computation of *ababaac* using automaton N_P^k for *abbaa* with at most one error (flipped horizontally and rotated 90 degrees clockwise, with active states marked in black).

A useful variation of table D is table $D'[i,j] = \min_{0 \leq l \leq j}\{ed(P[1..i], T[l\ ..\ j])\}$, computed as table D but setting $D[0,j] = 0$ (see Table 2.8).

Table 2.8. Table $D'[i,j]$ for *abbaa* and *ababaac*.

col / *row*	0	1 *a*	2 *b*	3 *a*	4 *b*	5 *a*	6 *a*	7 *c*
0	0	0	0	0	0	0	0	0
a	⋮	⋱	⋮	⋱	⋮	⋱	⋱	⋮
1	1	0	1	0	1	0	0	1
b	⋮	⋮	⋱	⋮	⋱	⋮	⋮	⋱
2	2	1	0	1	0	1	1	1
b	⋮	⋮	⋮	⋱	⋮	⋱	⋮	⋮
3	3	2	1	1	1	1	2	2
a	⋮	⋮	⋮	⋱	⋮	⋱	⋱	
4	4	3	2	1	2	1	1	⋯ 2
a	⋮	⋮	⋮	⋮	⋱	⋮	⋱1 ⋮	⋱1 ⋮
5	$\boxed{5}$	$\boxed{4}$	$\boxed{3}$	$\boxed{2}$	$\boxed{2}$	$\boxed{2}$	$\boxed{1}$	$\boxed{2}$

According to the definition of D', the last line, $D'[m,j]$, stores the smallest edit distance between S and a sub-string of S' starting at some position l and ending at position j. Suppose we want to find all occurrences of *abbaa* in *ababaac* with at most one error. By looking at row $D'[5,j]$ we find out that such occurrences can end only in position 6. In particular, there are two such occurrences, *ababaa* and *abaa*. To compute the same result we can use an automata N'^k_P, built by adding a loop labeled with all the characters in Σ to the initial state (see Figure 2.8).

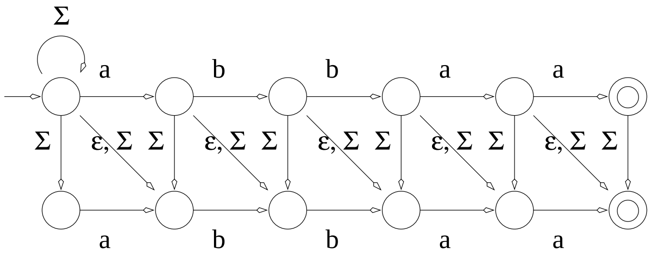


Fig. 2.8. Automaton N'^k_P for *abbaa* with at most one error.

2.5.2 Indexed Approximate Pattern Matching

If we wish to find the approximate occurrences of P in T in sub-linear time (with $O(n^\alpha)$ complexity, for $\alpha = m/k < 1$) we need to use an index structure for T. Suffix arrays [99] and q-grams have been proposed in the literature [56, 88]. An important class of algorithms for this problem are hybrid in the sense that they find a trade-off between neighborhood generation and filtration techniques.

A first and simple-minded approach to the problem consists in generating all the words at distance at most k from P and looking them up in the index T. The set of generated words is the *k-neighborhood* of P.

Definition 2.19. *The* k-neighborhood *of P is $U_k(P) = \{P' \in \Sigma^* : ed(P, P') \leq k\}$*

Let us denote the language recognized by the automaton N^k_P as $L(N^k_P)$. It should be clear that $U_k(P) = L(N^k_P)$. Hence, computing $U_k(P)$ is achieved by computing $L(N^k_P)$. This can be done by performing a DFS search in Σ^* that halts whenever all the states of N^K_P became inactive.

The *k-neighborhood*, turns out to be quite large. In fact $|U_k(P)| = O(m^k \sigma^k)$ [118]. Instead we use *condensed neighborhoods* (CU_k) [88, 99] and *super condensed neighbor-*

hoods (SCU_k) [104, 105] that contain $|SCU_k(P)| = |CU_k(P)| = O(n^{pow(m/k)})$ strings, where

$$pow(\alpha) = \log_{|\Sigma|} \frac{(\alpha^{-1} + \sqrt{1+\alpha^{-2}}) + 1}{(\alpha^{-1} + \sqrt{1+\alpha^{-2}}) - 1} + \alpha \log_{|\Sigma|}(\alpha^{-1} + \sqrt{1+\alpha^{-2}}) + \alpha$$

.

Figure 2.9 shows an example of the *1-neighborhood*, the *1-condensed neighborhood* and the *1-super condensed neighborhood* of *abbaa*. Observe that $SCU_k(P) \subseteq CU_k(P) \subseteq U_k(P)$.

Filtration can also be used to reduce the size of the *k-neighborhoods*. The most common filtration method divides the pattern into pieces of equal size and distributes the errors by these pieces, thus generating neighborhoods with smaller k's and m's. The disadvantage of this approach is that it will forces us to verify the text around the occurrences of these pieces and sometimes there are no approximate occurrences of P in those positions. A detailed description is given in Chapther 6.

SCU_1: abaa, abba, abbba, bbaa

CU_1: aabaa, ababaa, babbaa, bbbaa, aabbaa

U_1: abaaa, abbaa, abbaaa, abbaab, abbab, abbaba, abbbaa

Fig. 2.9. Figure representing the one-neighborhoods of *abbaa*.

3

Related Work

In this chapter we explain data compression methods, succinct data structures and how they can be used to design compressed indexes. We focus mainly on Lempel-Ziv compressed indexes.

3.1 Data Compression

A *text compression* technique is a way to encode the text in a format that requires less space than that of the original raw sequence and that still represents the original text. By representation we mean that we can consult any part of the original text, even if this implies that first we must decompress the whole string. It should be clear that we wish to recover exactly the original text, i.e. we are interested only in "lossless" data compression methods. Text compression usually provides a trade-off between the size necessary to store the text and the time it takes to consult a part of the text. This trade-off might be advantageous for storing a text or for transmitting it, such as over the Internet or from secondary memory to main memory. Storing compressed files saves storage space. Transmitting compressed files saves time when the overall time to encode, transmit and decode the file is smaller than the time to transmit the original text. Therefore applications such as gzip or bzip2 became popular for compressing and decompressing texts.

3.1.1 The 0-th Order Empirical Entropy

Text compression cannot compress a string indefinitely. In fact a simple argument proves that even if we had enough computational power available, it is not possible to compress every text by 1 bit. As a lower bound on the compressibility of a string we use the *empirical entropy* given by Manzini [80]. Which limits how much a string, considered as a finite object, can be compressed in practice by a class of known of "good" compressors. This concept is the best limit for our purposes. In Shannon's theory [12] different strings are grouped together into ergodic sources. However this tells us very little about a concrete string since it can be generated by different sources. Hence we are not interested in how the text was generated only in how much it can be compressed by a given algorithm. Of course this immediately raises the problem that for any given text, individually, it is possible to find a program that outputs it. Hence we must also measure the size of the program that outputs the string. In other words we could consider Kolmogorov Complexity [69]. However this complexity is not computable. Hence it is better to use the empirical entropy, which is computable and can be achieved by a class of known "good" compressors.

We start by defining the 0-th order empirical entropy.

Definition 3.1. *Let T, of size u, be a text over alphabet Σ. The zero-order empirical entropy of T is defined as*

$$H_0 = H_0(T) = \sum_{c \in \Sigma} (u_c/u) \log(u/u_c),$$

where u_c is the number of occurrences of character c in T.[1]

There are several methods to compress T such that the compressed version requires only uH_0 bits, such as Huffman coding, arithmetic coding [12]. In this thesis we briefly explain the Elias delta coding [33]. This code allows us to represent an integer x in $\log x + 2 \log\log x + O(1)$ bits in a way that it is possible to recover a sequence of numbers

[1] We assume that $0 \log \infty = 0$.

from a string of concatenated bits. For example, the binary representation of 2 and 3 are respectively 10 and 11. However it is not possible to retrieve this numbers from the string 1011 in an unambiguous way, i.e. this string may be representing eleven instead. A simple way to solve this problem is a unary code. In this code an integer $x \geq 1$ is encoded as $x-1$ bits followed by a zero bit, e.g. the unary code for 3 is 110 (see Table 3.1).

Table 3.1. Unary, gamma and delta coding

x	Unary	γ	δ
1	0	0	0
2	10	10 0	100 0
3	110	10 1	100 1
4	1110	110 00	101 00
5	11110	110 01	101 01
6	111110	110 10	101 10
7	1111110	110 11	101 11
8	11111110	1110 000	11000 000
9	111111110	1110 001	11000 001
10	1111111110	1110 010	11000 010

Unary coding becomes very ineffective for representing large numbers. The γ-codes correct this problem by representing x as $1+\lfloor \log x \rfloor$ bits followed by $\lfloor \log x \rfloor$ bits. The first part represents $1 + \lfloor \log(x) \rfloor$ with the unary coding, the second part represents $x - 2^{\lfloor \log x \rfloor}$ in binary, e.g. 2 is represented as 10, which is the unary representation of $1 + \lfloor \log 2 \rfloor = 2$, followed by $0 = 2 - 2^{\lfloor \log 2 \rfloor}$. The δ coding takes this process one step further and represents an integer by encoding $1 + \lfloor \log x \rfloor$ using γ-codes and $x - 2^{\lfloor \log x \rfloor}$ in binary. This requires $\log x + 2 \log \log x + O(1)$ bits.

A common application of δ-coding is to represent the gaps of inverted indexes or q-grams/q-samples indexes (see Subsection 2.4). Observe for example that a 1-gram index using gaps and δ-coding can be represented in $O(u) + u \sum_{c \in \Sigma} (u_c/u)(\log(u/u_c) +$

$2 \log\log(u/u_c)) = uH_0 + O(\log\log\sigma)$ bits, since the maximum space it can occupy occurs when the gaps are equal, i.e. of size u/u_c. This bound was pointed out by Navarro et al. [95]. Therefore 1-gram indexes can be considered compressed indexes. This reasoning can also be applied to 1-samples indexes and to inverted files. However, in that case, T must be considered a sequence of q-samples or words instead of letters. This decreases the value of u, to u/q, but may increase the value of H_0, also a table that maps from the q-samples to the original bits will contain σ^q entries.

The fact that by grouping letters we increase the value of H_0 gives us a way to design data compression methods that achieve a higher compression ratio. Note that H_0 reflects the redundancy of T. The smaller H_0 the more redundant T is. A text that has no redundancy whatsoever is already compressed. Therefore some compression methods focus in grouping the letters of T in order to obtain higher compression ratios.

3.1.2 The Burrows-Wheeler Transform and the k-th Order Empirical Entropy

The Burrows-Wheeler Transform is an invertible application that maps a string into another string of the same size, $BWT : \Sigma^* \to \Sigma^*$ that is easier to compress. In 1994 Burrows and Wheeler used it as a way to reorder a string such that information about groups of consecutive letters is stored in the same location, instead of spread across T [18].

Definition 3.2. *The* ***Burrows-Wheeler*** *transform of a string T is the string T^{bwt}, where $T^{bwt}[i] = T[SA[i]-1]$ assuming T ends with a \$, where SA is the suffix array of T* [2].

Figure 2.5 illustrates this concept. The Burrows-Wheeler transform of $T = cbdbddcbababa\$$ is $T^{bwt} = abbbaaccdd\$bdb$. Note that the property we mentioned that T^{bwt} groups the information of consecutive groups of strings has to do with the fact that T^{bwt} is based on the suffix array of T. More formally by T^{bwt}_s of a string s of size k we refer to the substring $T^{bwt}[i..j]$ such that s is a prefix of all the strings between $T[SA[i]..]$ and $T[SA[j]..]$. For example $T^{bwt}_{ba} = aac$.

[2] See definition 2.14.

The insight behind the Burrows-Wheeler transform is that T^{bwt} should be easier to compress since each T^{bwt}_s should have a smaller H_0 than the overall string. Note for example that since the "th" is very common in English the T^{bwt}_h string of an English text should have a disproportionated number of t's and therefore a smaller H_0 than the overall string. Therefore by using the entropy of this substrings for contexts s of size k we obtain the notion of *k-th order empirical entropy* $H_k(T)$ given by Manzini [80]. The *k-th order empirical entropy* gives a lower bound on the best compression ratio that T can achieve, if when compressing a character of T, we consider only the context of the k characters that follow it in T. Obviously the larger the context we consider, the better the compression should be, i.e. $0 \leq H_k(T) \leq \ldots \leq H_0(T) \leq \log \sigma$. Therefore the size of the compressed text will range from $uH_k(T)$ to $uH_0(T)$ depending on the compressor we use.

Definition 3.3. *Let T, of size u, be a text over alphabet Σ. The k-th order empirical entropy of T is defined as*

$$H_k = H_k(T) = \sum_{s \in \Sigma^k} (|T^{bwt}_s|/u) H_0(T^{bwt}_s).$$

Note that our definition is not the one usually given in text compression literature. In fact our definition is the k-th order empirical entropy of T^R. This makes little difference. In theory Ferragina et al. [37] showed that $uH_k(T) - O(\log u) \leq uH_k(T^R) \leq uH_k(T) + O(\log u)$. In practice, $H_k(T)$ and $H_k(T^R)$ can also be shown to be similar.

A series of techniques are used to encode T^{bwt} in uH_k bits, such as move-to-front transform and run-length encoding [18] but we will not discuss them here.

The surprising fact about the Burrows-Wheeler transform is that it is reversible, i.e. T^{bwt} contains enough information to allow us to recover T. First observe that by sorting the letters in T^{bwt} we obtain $T[SA[i]]$. Now the key to recovering T comes from observing that $T[SA[RA[i]]] = T[SA[SA^{-1}[i]]] = T[i]$. All that we need to do is compute RA. The T^{bwt} string contains enough information to compute the LF-mapping and in fact the ELF-mapping. Observe that according to the definitions of LF-mapping (definition 2.15) and of shift permutation (definition 2.4) we have that $RA[i] = \psi^i[RA[0]]] = LF^{-i}[RA[0]]]$. Observe also that because of the terminator character $LF[RA[0]] = 0$ and therefore $RA[i] = LF^{-i-1}[0]$. Hence we conclude that $T[i] = T[SA[LF^{-i-1}[0]]]$, i.e. $T[-i] = T[SA[LF^{i-1}[0]]]$. This equation shows that we can recover T from right to left by iterating LF from 0. All that is left to show is how to compute the LF mapping from T^{bwt}.

Recall that $LF[i] = RA[SA[i] - 1]$, i.e. the LF mapping corresponds to finding where in the suffix array is the suffix that precedes $T[SA[i]..]$. Hence the LF mapping can be obtained by the following process. Start with a suffix array. Prepend T^{bwt} to every string $T[SA[i] - 1..]$. Now sort the resulting array back into back into lexicographical order. Figure 3.1 shows an example of this process. Clearly since what we are sorting is already sorted, except for the first character, the sorting procedure is stable. This means that the relative order of suffixes $T[SA[i] - 1..]$ that start by the same letter is unaltered by the sorting process. We exemplify this for letter b. Observe that the arrows in Figure 3.1 do not cross. The LF mapping is called the last to first mapping since $T^{bwt} = T[SA[i] - 1]$ is the last column of matrix $T[SA[i] + j]$ and $T[SA[i]]$ is the first, see Figures 2.5 and 3.1.

Lemma 3.4. *The LF mapping of a string T can be computed as*

$$LF[i] = C(T^{bwt}[i]) + 1 + Occ(T^{bwt}[i], i),$$

where $Occ(\ell, i)$ is the number of occurrences of ℓ in $T^{bwt}[..i-1]$ and C is the count function of Subsection 2.3.2.

For example $LF[13] = C(b)+1+Occ(b, 13) = 3+1+4 = 8$ (see the last suffix of Figure 3.1, Figure 2.5 and Subsection 2.3.2).

In general the ELF-mapping can be computed as $ELF[i, \ell] = C(\ell) + 1 + Occ(\ell, i)$. Therefore the computation time of ELF depends essentially in computing $Occ(\ell, i)$, since C can be computed in $O(1)$ time by storing $O(\sigma \log u)$ bits. Computing Occ efficiently and in compressed space, $O(uH_k)$ bits, is possible by using succinct data structures.

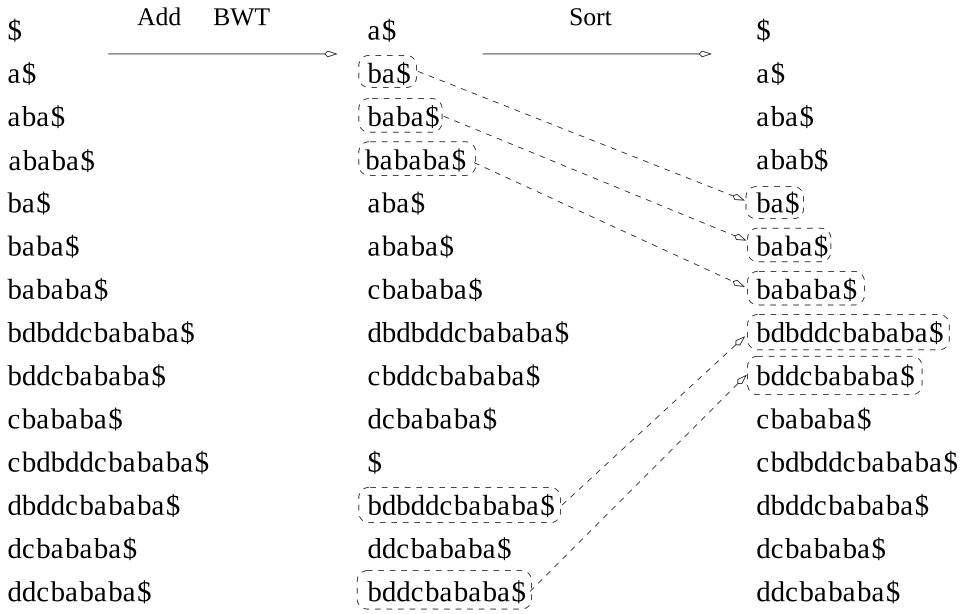


Fig. 3.1. Computation of the LF mapping of T.

Just to be able to recover T from T^{bwt} we do not need to support the general Occ operation. We can instead compute the values of $Occ(T^{bwt}[i], i)$ and store them in an array. This can be computed with a scan over T^{bwt}, which, however, requires $u \log u$ bits. For this reason popular implementations of this algorithm start by dividing T into blocks and using the transformation in each of the blocks separately [129].

3.1.3 Lempel-Ziv Data Compression

A common approach to data compression are dictionary-based compression methods. The idea is to first analyze T to infer a suitable dictionary D and then represent T as a concatenation of words in D. Several questions on how to infer and represent the dictionary are immediately raised. How many strings should the dictionary have ? How big should the strings in the dictionary be ? Which strings should be part of the dictionary ? How should the dictionary be represented ?

The total space that is necessary to store the string in compressed form includes the dictionary. Therefore a partial answer to the above questions is that the dictionary must be such that its size plus the size of the compressed text is as small as possible. Computing this optimal dictionary is surprisingly difficult.

In the 1970s, Ziv and Lempel [126, 127] proposed a greedy way of inferring a dictionary that was very simple. Moreover their approach had the added bonus of encoding the dictionary implicitly in the compressed text. Their idea was to replace each substring of the text with a pointer to a place where it had occurred before.

Definition 3.5. *The **LZ77 parsing** of a string T is the sequence $Z_1, \ldots, Z_n$ of strings such that $T = Z_1 \ldots Z_n$ and, for every i, $Z_i = S_i c$ where c is a letter and S_i is the largest prefix of $Z_i \ldots Z_n$ that is a substring of $Z_1 \ldots Z_{i-1}$.*

The LZ77 parsing of T can be computed in $O(u)$ time and space by using the algorithm of McCreight or Ukkonen (see Subsection 2.2.1).

The compressed indexes proposed by Kärkkäinen and Ukkonen [58, 59] were based on a variation of the LZ77 parsing. The dictionary underlying the LZ77 parsing is relatively complex and so Ziv and Lempel proposed a variation of the above parsing that yields a simpler dictionary.

Definition 3.6. *The **LZ78 parsing** of a string T is the sequence $Z_1, \ldots, Z_n$ of strings such that $T = Z_1 \ldots Z_n$ and, for every i, $Z_i = Z_j c$ where c is a letter and Z_j is the largest prefix of $Z_i \ldots Z_n$ belonging to $\{Z_1, \ldots, Z_{i-1}\}$.*

The dictionary associated with the LZ78 parsing of string T can be represented as a **trie**, i.e. a labeled tree where every label has only one letter. The Z_i strings are referred to as **LZ78-blocks**. The **LZ78 trie** of a string T is the trie of its LZ-blocks.

Definition 3.7. *The **LZ78 trie** of a string T is the trie of strings $\{Z_1, \ldots, Z_n\}$.*

The LZ78 parsing of T can be easily computed in $O(u)$ time by incrementally computing the LZ78 trie of T. For reasons that will become clear later, we show the LZ78 parsing of T^R. In our example T^R is parsed into $a, b, ab, abc, d, db, dbc$. Figure 3.2 shows the LZ78

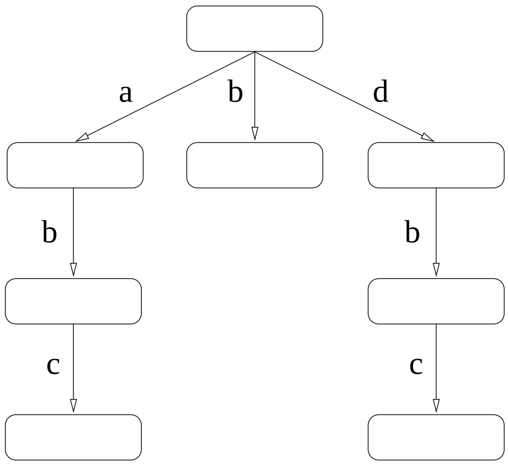


Fig. 3.2. LZ78 trie of T^R.

trie of T^R. We assume that n represents the number of blocks of the LZ78 parsing of T. The relation between n and H_k was established by Kosaraju et al. [66] who showed that $n \log u = uH_k + o(u \log \sigma)$ for $k = o(\log_\sigma u)$. Therefore, representing T in compressed form consists in storing the LZ78 trie. This is done by storing the nodes in the same order that they appear in the text, i.e. storing $Z_1, \ldots, Z_n$. Decompressing consists in decompressing each block. Since the blocks are stored in the same order that they appear in the text, when we need to decompress a block we only need to add a letter to a block that is already decompressed.

3.2 Succinct Data Structures

The only missing piece we need to describe compressed indexes is succinct data structures. A *succinct data structure* is a compact representation of a data structure. The Rank and Select operations, over bitmaps, are a crucial element to obtaining these data structures.

3.2.1 Basic Rank and Select

By bitmap B we refer to a string over $\{0, 1\}$. The operation $\textsc{Rank}(B, i)$ counts the number of 1's in $B[..i-1]$ and $\textsc{Select}(B, i)$ returns the smallest j such that $\textsc{Rank}(B, j+1) = i$,

```
   1
   012345678901234567
B: 110110100110100100
```

Fig. 3.3. Sample bitmap.

i.e. the position of i-th 1. For the bitmap in Figure 3.3, we have that $\textsc{Rank}(B, 3) = 2$ and $\textsc{Select}(B, 2) = 1$. Munro [84] and Clark showed how to support these operations in $O(1)$ time and $|B| + o(|B|)$ bits. Succinct data structures can also be combined with data compression techniques, i.e. when B is compressible, solutions that require $|B|H_0(B) + o(|B|)$ bits may be more adequate. This line of work was initiated by Pagh [102] and extended by Raman et al. [103], whose approach can store a bitmap in $|B|H_0 + o(|B| \log\log|B|/\log|B|)$ bits supporting $\textsc{Select}_1$ in $O(1)$ time.

We will briefly explain the basic $u + o(u)$ bits solutions. To avoid having to deal with floors and ceilings we assume, in this description, that u is a power of 4.

Let us start by $\textsc{Rank}$. A simple solution would be to store an array with the precomputed solution for every i. This however would require $u \log u$ bits. Instead we divide B into blocks of size $(\log u)/2$, $\log u$ and $\log^2 u$ referred to as small blocks, blocks and super blocks. The super blocks store the precomputed solutions for positions that are multiples of $\log^2 u$, each such value requires $\log u$ bits. The blocks store the precomputed solution for positions $\log u$ but only relative to the super block they are in. Each such value requires only $\log\log^2 u$ bits, since each block can contain at most $\log^2 u$ ones. Finally the small blocks are handled by the four-Russians technique, which consists in precomputing and storing all solutions for each of the $2^{(\log u)/2} = \sqrt{u}$ possible small blocks. Each value requires $\log((\log u)/2)$ bits. Therefore storing a solution for every position of a small block, for every type of small block, requires $2^{(\log u)/2}((\log u)/2)\log((\log u)/2)$ bits. Overall storing these tables and B requires only $u + o(u)$ bits.

Consider for example the bitmap $(01)^{32}$, i.e. the string 01 repeated 32 times. In our example $(\log u)/2 = 3$, $\log u = 6$ and $\log^2 u = 36$. Table 3.2 shows the values of blocks and super blocks of our example and table 3.3 shows the preccomputed solutions for small blocks. In order to compute $\textsc{Rank}(B, i)$ we just add up the values relative to the corresponding super block, block and, possibly, two small blocks. Consider for example $\textsc{Rank}(B, 47) = \textsc{Rank}(B, 1 \times 36 + 1 \times 6 + 3 + 2) = \mathit{super}[1] + \mathit{block}[1] + \mathit{small}[010, 3] + \mathit{small}[101, 2] = 18 + 3 + 1 + 1 = 23$.

Table 3.2. Table exemplifying RANK for blocks and super blocks.

B	$(01)^3$	$(01)^3$	$(01)^3$	$(01)^3$	$(01)^3$	$(01)^3$	$(01)^3$	$(01)^3$	$(01)^3$	$(01)^3$	$(01)^2$
super block	0						18				
block	0	3	6	9	12	15	0	3	6	9	12

Table 3.3. Small blocks table.

small block	0	1	0		1	0	1	
i mod 4	0	1	2	3	0	1	2	3
	0	0	1	1	0	1	1	2

The SELECT operation is a bit more complicated but works in a similar way. The space of possible arguments $[0, u]$ is broken into blocks, but this time there is no bound on the number of bits that the partial solutions for blocks require. For a detailed explanation of how to solve this problem consult Munro [84] or Navarro et al. [95]. In practice, SELECT can be computed with a binary search in $O(\log u)$ time [47].

3.2.2 Wavelet Trees

Another kind of succinct structures we require are those to compute orthogonal range queries. The idea is to preprocess a grid of $f \times f$ points to be able to determine the points inside a two-dimensional range (see Figure 4.1 (bottom-right)). For this purpose, we use a structure by Chazelle [22], that uses $f \log f(1 + o(1))$ bits and $O(f \log f)$ time to be built. It reports points in $O((1 + occ') \log f)$ time.

Wavelet trees are a recurrent succinct data structure. They were proposed by Grossi et al. [49] as a structure for supporting Rank and Select for sequences over an alphabet larger than 2. They were also proposed by Chazelle [22] for performing orthogonal range queries. Obviously the algorithms over the structure are different. However, both use Rank and Select over bitmaps. This description of the structure given by Chazelle was pointed out by Navarro et al. [95].

Consider for example the sequence $0, 3, 3, 7, 9, 4$. The wavelet tree of this sequence is shown in Figure 3.4. The wavelet tree is a perfect binary tree of height $\lceil \log \sigma \rceil$. Each node stores a sub-sequence of the original sequence. The root stores the whole sequence. Starting from the most significant bit, the left node stores the sub-sequence for which this bit is 0, the right node stores the sub-sequence for which this bit is 1. In our example the left sub-sequence is $0, 3, 3, 7, 4$ and the right sub-sequence is 9. This process continues until all bits have been used. Moving from one node in the tree into a child node consists of a Rank operation. For the left node, we use Rank_0 and, for the right node, Rank_1, i.e. we count the number of zeros or count the number of ones. In our example, we can track the element 4, that is in position $i = 5$, by computing $\text{Rank}_0(000010, 5) = 4$ at the root node. Note that element 4 is in position 4 of the left child of the root. Obviously moving upwards uses the inverse procedure, i.e. the Select_0 operation. In this case we compute $\text{Select}_0(000010, 4 + 1) = 5$. Every leaf of the wavelet tree represents a type of element in the sequence. Moving from the root to a leaf allows us to compute Rank for the element type of that leaf. Conversely, moving from a leaf to the root allows us to compute Select for a given element type. For example we can compute $\text{Rank}_3(033794, 4)$ by using the following sequence of rank operations $\text{Rank}_0(000010, 4) = 3$, $\text{Rank}_0(00011, 3) = 2$, $\text{Rank}_1(0111, 2) = 1$, $\text{Rank}_1(11, 1) = 1$. Since $033794[4] = 9 \neq 3$ we must compensate for that fact and add 1 to obtain $\text{Rank}_3(033794, 4) = 1 + 1 = 2$. Note that the sequence of indexes in the Rank operation is 0011 which corresponds to 3 in binary. Also note that by using Select in the reverse order we can obtain $\text{Select}_3(033794, 2) = 2$. Hence the wavelet tree can be used to compute the *Occ* operation of the ELF-mapping.

The tree structure is only conceptual. In fact the only information that is stored are the bitmaps highlighted in Figure 3.4. Further Rank and Select operations can be used to delimit the bits that correspond to a given node of the tree.

The wavelet tree can also be used to compute orthogonal range queries. Consider a grid $[1, f] \times [1, f]$ with f points inside. An orthogonal range query consists in determining the points inside a rectangle (see Figure 4.1). Provided that the points are all distinct in the first coordinate they can be stored in a wavelet tree, by building a list of the second coordinate ordered by the first coordinate. In the example of Figure 4.1 the resulting sequence is $0, 3, 3, 7, 9, 4$. This requires $f \log f(1 + o(1))$ bits. In fact, it is easy to extend the space of the second coordinate, i.e. extend the space to $[1, f] \times [1, f']$. This will require $f \log f'(1+o(1))$ bits instead. To compute a range query $[i, i'] \times [j, j']$ we start by locating the range $[i, i']$ at the root of the wavelet tree. When we descend, we track the elements i and i'. The idea is to track every path that is contained in the $[j, j']$ range. Obviously we can avoid descending by nodes for which the corresponding range $[i, i']$ is empty. Therefore, whenever a leaf is reached, an occurrence is found, i.e. it takes $O((1 + occ') \log f')$ time to report occ' occurrences. A simpler procedure can be used to count the number of occurrences in range $[i, i'] \times [j, j']$. The procedure consists in descending by j and j', the total of occurrences associated with the non-shared part of these paths gives the number of occurrences. This takes $O(\log f')$ time.

There exist other range data structures, proposed by Alstrup et al. [3], that require $O(f \log^{1+\gamma} f)$ bits of space, for any constant $\gamma > 0$. A counting query requires $O(\log \log f)$ time and each occurrence can be reporting in constant time. They also propose another structure that takes $O(f \log f \log \log f)$ bits of space and requires $O((\log \log f)^2)$ time for a query and reports each occurrence in $O(\log \log f)$ time.

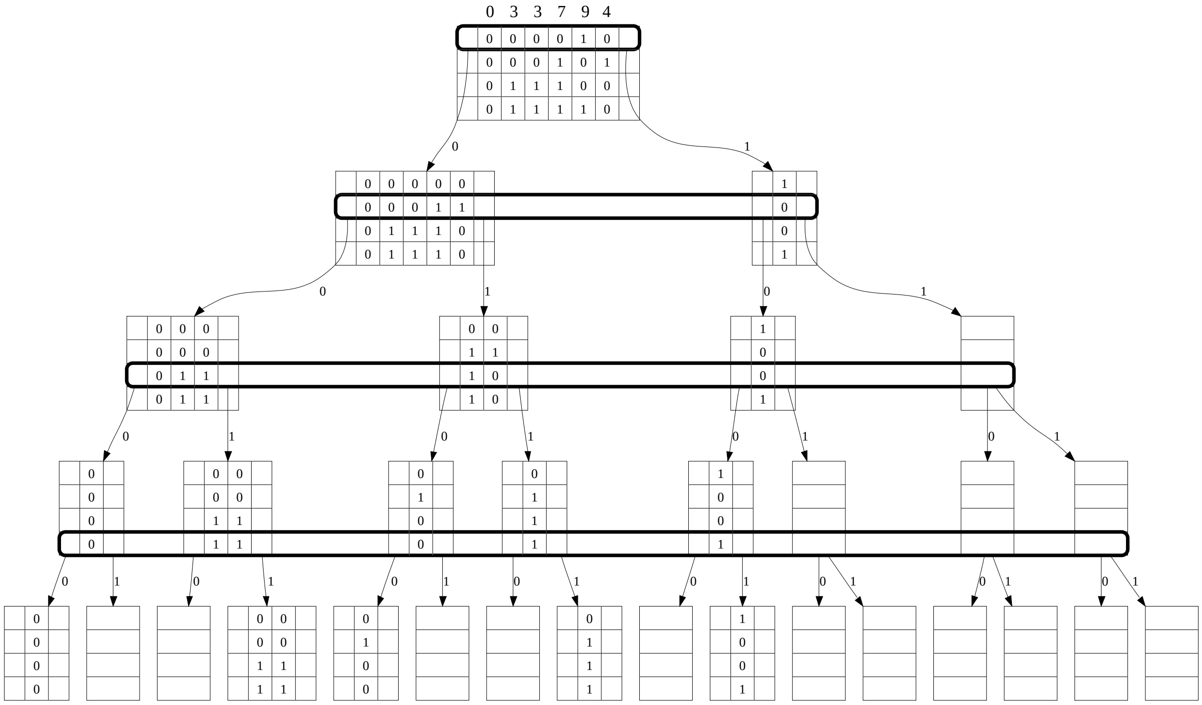


Fig. 3.4. Wavelet tree for sequence $0, 3, 3, 7, 9, 4$.

3.2.3 Permutations and Trees

Trees are a recurrent data structure in computer science and, in particular, play a central role in *full-text indexing* theory. It is therefore natural to consider succinct representations of trees. Clearly, the less space we need to represent a tree, the less space our indexes will require. Jacobson [57] was the first to study succinct data structures, such as trees and bitmaps.

The set of operations provided by succinct trees has been successively enlarged and improved by several researchers [14, 23, 39, 43, 86, 108].

The representation of Geary et al. [43] requires $2n + o(n)$ bits, for a tree with n nodes, supporting, among others, the following operations in constant time:

- Anc(v, j) returns the j-th ancestor of node v (for example Anc$(v, 1)$ is Father(v));
- LeftRank(v) returns Dfs(v);
- RightRank(v) returns the largest Dfs value among the descendants of v;

- SELECT(j) returns the node with DFS time j;
- CHILD(v, j) returns the j-th child of node v;
- DEG(v) returns the number of children of node v;
- DEPTH(v) returns the tree depth of node v.

The underlying techniques of these representations are similar in spirit to the ones behind RANK and SELECT, i.e. packing the nodes into blocks of different sizes.

The RANK and SELECT operations also proved useful for representing permutations. Munro et al. [85] showed how to represent a permutation of d elements and its inverse in $(1+\epsilon)d \log d + O(d)$ bits, where ϵ is constant and $0 < \epsilon \leq 1$. An element of the permutation can be computed in $O(1)$ and an element of the inverse in $O(1/\epsilon)$.

Note that a simple way to invert permutation π is to use the SELECT operation, i.e. $\pi^{-1}[x] = \text{SELECT}_x(\pi, 1)$. Therefore the wavelet tree can be used to invert a permutation in $O(\log d)$ time. Moreover reading an element $\pi(x)$ also requires $O(\log d)$ time. The solution given by Munro et al. consisted in using the orbits of the permutation. Note that if $\pi^{k+1}[x] = x$ then $\pi^k[x] = \pi^{-1}[x]$. However scanning an entire orbit may require $O(d)$ time, i.e. $k+1 = d$. Therefore we sample π^-1 every $1/\epsilon$ position. Assume for simplicity that $1/\epsilon$ is an integer. In this way, we need to scan at most $1/\epsilon$ positions. A bitmap indicating the positions that store the π^-1 pointers can be used with RANK to locate the positions of these pointers in an array.

3.3 Compressed Indexes

In this section we describe other compressed indexes. We start by briefly explaining FM-Indexes and compressed suffix arrays and then describe Lempel-Ziv compressed indexes.

3.3.1 Full-text Index in Minute Space

The FM-index algorithm searches for a pattern P by reading it form right to left, using the backward search, see Subsection 2.2.2. The idea is to update the range $[sp, ep]$ for every suffix of P. For example for $P = cbdbddc$ we would consider the following sequence of suffixes c, dc, ddc, $bddc$, $dbddc$, $bdbddc$, $cbdbddc$ which correspond to the following sequence of ranges $[0, 13]$, $[9, 10]$, $[12, 12]$, $[13, 13]$, $[8, 8]$, $[11, 11]$, $[7, 7]$, $[10, 10]$ (see Figure 2.5). The fundamental tool to achieve this goal is the ELF mapping, see Subsection 3.1.2. The procedure is shown in algorithm 2.

Algorithm 2 Backward Search

$sp \leftarrow 0$
$ep \leftarrow u - 1$
for $i \leftarrow m - 1, 0$ **do**
    $sp \leftarrow ELF[sp, P[i]]$
    $ep \leftarrow ELF[ep + 1, P[i]] - 1$
    if $sp > ep$ **then**
        return $\emptyset$
    end if
    $i \leftarrow i - 1$
end for

As we pointed out in Subsection 3.1.2, the key to computing the ELF mapping is to compute the Occ operation. We also explained in Subsection 3.2.2 that this can be computed with the wavelet tree in $O(\log \sigma)$ time, requiring $u \log \sigma(1 + o(1))$ bits. Hence, this representation is not compressed. However, by using compressed bitmaps, we can represent a wavelet tree in $uH_0 + o(u \log \sigma)$ bits. For more elaborated implementations of Occ that can achieve $uH_k + o(u \log \sigma)$ bits of space, see the survey by Navarro et al. [74, 95].

3.3.2 Compressed Suffix arrays

Compressed suffix arrays are an abstract optimization of the suffix array data structure. The search algorithm is the usual algorithm over suffix arrays, i.e. two binary searches over SA (see algorithm 3). However, instead of storing the suffix arrays explicitly, its values are computed by some process. This means that the time of this procedure will depend on the time it takes to compute the entries of the suffix array.

Algorithm 3 Suffix Array Search

```
sp ← 0
st ← u − 1
while sp < st do
    s ← ⟨(sp + st)/2⟩
    if P > T[SA[s]..] then
        sp ← s + 1
    else
        st ← s
    end if
end while
ep ← sp − 1
et ← u − 1
while ep < et do
    e ← ⟨(ep + et)/2⟩
    if P < T[SA[e]..] then
        et ← e − 1
    else
        ep ← e
    end if
end while
```

There are two main ways to obtain this abstract data structure, the *Compact Suffix Array* of Mäkinen [75] and the *Compressed Suffix Array* of Grossi and Vitter [51]. Both of these ideas appeared, simultaneously and independently, during the year 2000.

The compact suffix array explores the self-repetitions of the suffix arrays. In Subsection 2.1.3, we explained how self-repetitions can be detected, if we consider π^{-1} as SA we have that $shift_\pi = \psi$ and, therefore, self-repetitions are discovered by using the natural runs of ψ.

The compressed suffix array of Grossi and Vitter is based on the idea of hierarchical decomposition of SA. The idea is to reveal the relations between the sparse suffix arrays of the text, i.e. the suffix arrays associated with the sparse suffix trees of T (see Subsection 2.4). Consider, for example, the suffix array of T, denoted as SA_0, and the suffix array of the text T where the letters are grouped in 2-samples, denoted as SA_1. The relation between SA_0 and SA_1 is that SA_1 only contains the even entries of SA_0. We define a bitmap B_0 such that $B_0[i] = 1$ iff $SA_0[i]$ is even. We can compute a value of SA_0, from SA_1, B_0 and ψ_0. If $B_0[i]$ is 1 then $SA_0[i] = SA_1[\textsc{Rank}(B_0, i)]$. Otherwise $SA_0[i] = SA_0[\psi_0[i]] - 1$, as shown in Table 3.4, note that in this case $B_0[\psi_0[i]]$ is 1. It may seem that storing B_0, ψ_0 and SA_1 is not much better than storing only SA_0. However if we store SA_1 divided by 2 and we use δ coding for the gaps of ψ_0 as for inverted indexes (see Subsection 3.1.1), it may indeed require less space. We continue this process for successive powers of 2 up to $2^{\lceil \log \log u \rceil}$, repeating the process $\lceil \log \log u \rceil$ times. Sadakane [112] modified the suffix array into a self-index so that it was possible to compare P with $T[SA[i]..]$ without having to compute $SA[i]$.

Table 3.4. The first level of the hierarchical decomposition of the suffix array of Grossi and Vitter.

SA_0	13	12	10	8	11	9	7	1	3	6	0	2	5	4
B_0	0	1	1	1	0	0	0	0	0	1	1	1	0	1
ψ_0	10	0	4	5	1	2	3	11	13	6	7	8	9	12
SA_1		12	10	8						6	0	2		4

3.4 Lempel-Ziv Compressed Indexes

We will now explain Lempel-Ziv compressed indexes in detail. The first Lempel-Ziv compressed indexes represented the initial work on compressed indexes. They were proposed by Kärkkäinen and Ukkonen [59].

3.4.1 The LZ-Index of Kärkkäinen and Ukkonen

The LZ-Index of Kärkkäinen and Ukkonen uses a variation of the LZ77 parsing. In fact the parsing used by Kärkkäinen and Ukkonen is the one given in definition 3.5. In the original definition by Ziv and Lempel [126, 127] the condition "a substring of $Z_1 \dots Z_{i-1}$" is replaced by "a substring of T that starts in $Z_1 \dots Z_{i-1}$".

The occurrences of P in T are classified as *primary* when they span more than one LZ77-block or as *secondary* when they occur completely inside an LZ77-block.

The approach presented by Kärkkäinen and Ukkonen consisted in finding primary occurrences first and then finding the secondary occurrences. Note that the secondary occurrences are repetitions of other primary or secondary occurrences. The fundamental observation behind their approach is that the first occurrence of P in T must be a primary occurrence, unless $m = 1$. Since this exception is easy to handle we disregard it here.

In order to find primary occurrences we need to consider every possible partition of P. When P occurs spread across LZ77-blocks a prefix $P[..i]$ of P is a suffix of the first block that contains the occurrence of P, while $P[i+1..]$ occurs as the concatenation of LZ77-blocks. The search consists in associating a range to $P[..i]$ and another range to $P[i+1..]$.

One of the most interesting characteristics of the approach of Kärkkäinen and Ukkonen is that it reduced the pattern matching problem to a two-dimensional range query problem. The idea is to store the information relative to consecutive blocks such that we can determine in which locations in T it occurs simultaneously an LZ77-block that contain as a suffix $P[..i]$ and a sequence of blocks that contain string $P[i+1..]$.

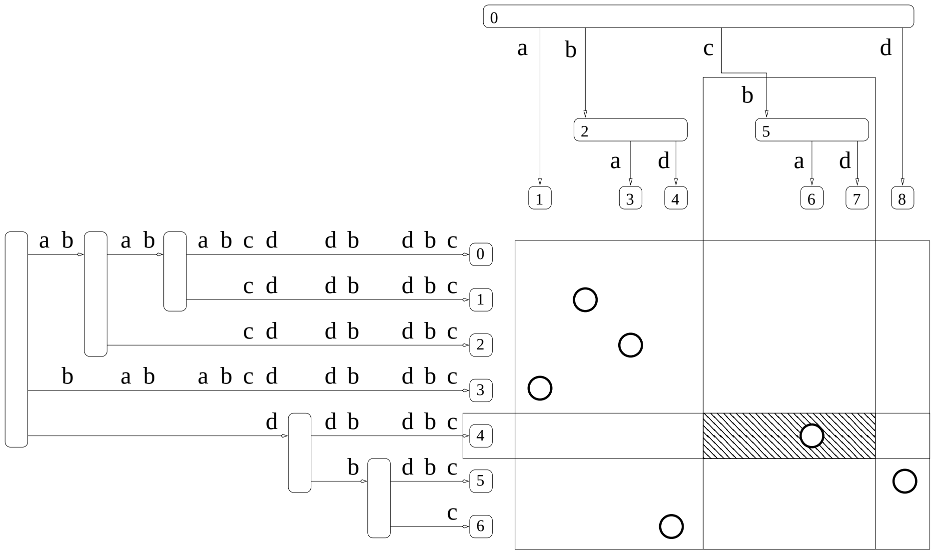


Fig. 3.5. (top-right) Tree for strings $\{a, b, ba, bd, cba, cbd, d\}$, nodes show their DFS values. (bottom-left) Sparse suffix tree, leaves, show their indexes, (bottom-right) linking points over spaces supported by DFS and suffix array indexes. Orthogonal range query [4,4]:[5,7].

We search for $P[i+1..]$ in a sparse suffix tree that only indexes suffixes of T that start at blocks. Suppose that we use the LZ78 parsing on $T^R = abababcddbdbc$ (we will explain this in further detail in chapter 4). For simplicity, we will use the LZ78 parsing, since the properties of the LZ77 parsing we need are also present in the LZ78 parsing. In this example the LZ78 parsing is $a.b.ab.abc.d.db.dbc$. The sparse suffix tree contains the following strings: $a.b.ab.abc.d.db.abc$, $b.ab.abc.d.db.abc$, $ab.abc.d.db.abc$, $abc.d.db.abc$, $d.db.abc$, $db.abc$ and abc. This tree is shown in Figure 3.5(left). The range we are interested in is the range corresponding to $P[i+1..]$ in the suffix array associated to the sparse suffix tree.

Consider for example, that we are searching for $P = cddbdbc$ and that we are trying the case $c = P[..i]$ and that $ddbdbc = P[i+1..]$. In that case, we would descend by $ddbdbc$ in the sparse suffix tree, and would only have to consider the leaf indexed by 4, i.e. the range $[4, 4]$.

In order to obtain a range for $P[..i]$ we build a labeled tree with the LZ77-blocks reversed. In this case we need a labeled tree with the strings $\{a, b, ba, bd, cba, cbd, d\}$. This tree is denominated *RevTrie* in the work of Navarro [98] and by $\mathcal{T}_{78}$ suffix tree in our work. Note that using the LZ77 parsing does not guarantee that this structure is a suffix tree. Since this tree stores the blocks reversed, finding the blocks that finish by a given suffix can be achieved simply by descending in this tree. In this example, we descend by c in the tree on the top-right of Figure 3.5. The corresponding range is $[5, 7]$.

For the orthogonal range search to locate the occurrences we need to store points with the necessary information. We store points representing the relation between strings Z_i and $Z_{i+1} \dots Z_n$. In our example this corresponds to the following pairs of strings $\langle a, b.ab.abc.d.db.dbc\rangle$, $\langle b, ab.abc.d.db.dbc\rangle$, $\langle ab, abc.d.db.dbc\rangle$, $\langle abc, d.db.dbc\rangle$, $\langle d, db.dbc\rangle$ and $\langle db, dbc\rangle$. This corresponds to the following sequence of points $\langle 3, 1\rangle$, $\langle 1, 2\rangle$, $\langle 2, 3\rangle$, $\langle 4, 6\rangle$, $\langle 5, 8\rangle$ and $\langle 6, 4\rangle$. In this way, when we execute the $[4, 4] : [5, 7]$ query, the only point in this range is $\langle 4, 6\rangle$ which corresponds to the occurrence $ab\underline{c.d.db.dbc}$.

It is important to notice that even though this process is very ingenious it requires $O(m^2)$ operations since we must descend in the RevTrie by $(P[..i])^R$ for every $0 \leq i < m$ by $P[i + 1..]$ in the sparse suffix tree.

As mentioned before, secondary occurrences are those for which P is a substring of some Z_i. According to definition 3.5, if P is a substring of some Z_i then either that substring contains the respective c or it does not. If the occurrence of P contains c then this can be found in RevTrie by descending by P^R. If the substring does not contain c, then P is a substring of the corresponding S_i. To find these occurrences we store an array S with these strings. However, instead of storing S_i, it stores the interval of T that corresponds to S_i, i.e., S stores the pair of integers $\langle x_i, y_i\rangle$ for every $S_i = T[x_i..y_i]$. This definition, however, is ambiguous, since there might be more than one pair of integers for which $S_i = T[x_i..y_i]$. This is not a relevant problem since any pair of integers will do, provided that, overall, no interval is contained in another interval, i.e. there are no distinct i and i' such that $x_i \leq x_{i'}$ and $y_{i'} \leq y_i$. Kärkkäinen and Ukkonen show how to ensure this property. The entries in the S array are sorted by x_i. We also store a bitmap B that is set to 1 for the x_i's and to

0 for the other positions. To obtain a secondary occurrence from another predetermined occurrence of $P = T[j..j+m-1]$, we need to determine which of the intervals of S contain the interval $[j, j+m-1]$. Since the elements of S are sorted by x_i and are not contained inside each other, this can be done by checking $S[\textsc{Rank}(B, j)]$ and scanning the previous elements of S until the first one that does not contain the interval $[j, j+m-1]$.

The space requirements of this index are $O(uH_k) + o(u \log \sigma)$ bits. This can be proved by using the bound by Kosaraju et al. [66], who showed that $n \log u = uH_k + o(u \log \sigma)$ for $k = o(\log_\sigma u)$. The trees and the S array requires $O(n \log u)$ bits of space. If we use a wavelet tree as the range data structure, it requires another $n \log n(1+o(1))$ bits. Therefore the overall space requirement of this index is $O(uH_k) + o(u \log \sigma) + u \log \sigma$ since we need to store T because this index is not a self-index.

The time it takes to count occurrences is $O(m^2 + (m + occ) \log u)$. Note that, as all Lempel-Ziv based indexes, it is not possible to compute counting queries without an *occ* time dependency. This difficulty is mainly because of the occurrences of P inside a block, i.e. secondary occurrences. The $O(m^2)$ parcel is the time it takes to descend in RevTrie and in the sparse suffix tree. The $O((m+occ) \log u)$ part is the time to compute the range queries.

3.4.2 The LZ-Index of Navarro

The LZ-Index of Navarro uses the LZ78 parsing instead of the LZ77 parsing since it has very nice characteristics for string matching, in particular for secondary occurrences.

Navarro's approach discards the sparse suffix tree and uses the LZ78 trie. What happens is that primary occurrences have to be further divided into occurrences that span exactly two blocks (occ_2) and occurrences that span more than two blocks (occ_3), since this last type of occurrences cannot be found using only the LZ78 trie. To find occurrences that span exactly two blocks Navarro uses the same procedure as for primary occurrences. The LZ78 trie plays the role of the sparse suffix tree in this procedure. In practice Navarro observed that using a range data structure might not be the best alternative. In fact, his prototype used a naive procedure, which scanned all the points of the smallest interval to determine which ones are also in the other interval.

To find occurrences that span 3 or more blocks the algorithm searches around the LZ78 blocks that are substrings of P, i.e. the $Z_i = P[j..j']$. Note that, in an occurrence that spans at least tree blocks, one of the blocks must be a substring of P. However since the Z_i blocks are distinct among themselves, there can be at most $O(m^2)$ such blocks. Therefore searching around them does not require more than $O(m^2)$ time. Note that this checking is done block-wise and not character-wise. Hence removing the sparse suffix tree will not increase the asymptotic complexity of the search.

Since this index is based on the LZ78 parsing finding the occurrences that are completely contained inside one block can be done in a simpler way. We use this procedure in our prototype and hence it is described in Subsection 4.2.3.

Like the LZ-index of Kärkkäinen and Ukkonen, the LZ-index of Navarro requires $O(m^2 + (m + occ) \log u)$ time to report the occurrences of P in T. However, since Navarro used self-indexing, the structure only requires $O(uH_k) + o(u \log \sigma)$ bits of space. Moreover Navarro proposed a succinct representation of this index that was based on representations of trees using bitmaps, as described in Subsection 4.2. In particular the representation proposed was based on the work of Munro et al. [86]. The resulting index required $4uH_k + o(u \log \sigma)$ bits and the queries can be answered in $O(m^3 \log \sigma + (m + occ) \log u)$ time [98]. This representation has been recently improved by Arroyuelo et al. [6] to require $(2 + \epsilon)uH_k + o(u \log \sigma)$ bits, for any $\epsilon > 0$, and the search complexity is $O(m^2 \log m + (m + occ) \log u)$. They can display ℓ letters in $O(\ell / \log_\sigma u)$ time.

3.4.3 The LZ-Index of Ferragina and Manzini

The LZ-Index of Ferragina and Manzini was the only existing LZ-index that was able to count occurrences with a linear time dependency on m, i.e. $O(m)$ time prior to our work. They proposed an implementation of the sparse suffix tree and of RevTrie based on the FM-Index. This means that they are able to search for $(P[..i])^R$ and $P[i+1..]$ by doing backward search, as described in Subsections 2.2.2 and 3.3.1. The time required for backward search is linear on m.

They propose to implement RevTrie as the FM-Index of the text $(Z_1)^R\#\ldots\#(Z_n)^R\#$, where the #'s are special characters that do not appear in T. In our example, since we used the LZ78 parsing for $T^R = a.b.ab.abc.d.db.dbc$, this would yield the text $a\#b\#ba\#cba\#d\#bd\#cbd\#$. They show that storing the RevTrie in this form requires $O(uH_k)$ bits. In Lemma 4.13 we give a simple proof of this fact.

Instead of using a sparse suffix tree they use an FM-Index for T. This means that, in fact, they have another compressed index inside their LZ-index. However, with the proper implementation, it is possible to guarantee that both these structures occupy asymptotically the same space (Alphabet friendly FM-Index [95]), $uH_k + o(u \log \sigma)$ bits. Nonetheless using another compressed index means they are not really addressing the problems underlying Lempel-Ziv compressed indexes.

In fact the LZ-Index of Ferragina and Manzini is much more of an FM-index with improved reporting performance than an LZ-index. As we have explained, by using the wavelet tree, reporting requires $O(\log u)$ time per occurrence of P. However the FM-index is not so efficient at reporting. In fact, the version by Ferragina and Manzini required $O(\sigma \log^{1+\epsilon} u)$ time and more recent versions need $O((\log^{1+\epsilon} u)(\log \sigma)/\log\log u)$ time, where $\epsilon > 0$ is an arbitrary constant. Ferragina and Manzini were in fact able to obtain $O(1)$ reporting time per occurrence. To achieve this result they used the more space demanding range data structure by Alstrup et al. [3] that requires $O(n \log^{1+\gamma} n)$ bits of space and can report occurrences in $O(1)$ time, for any constant $\gamma > 0$. In Subsection 4.4.2 we confirm experimentally that Lempel-Ziv based indexes are fast at reporting occurrences. However this good performance is not due to the range data structure in question, since the Lempel-Ziv prototypes use the scanning proposed by Navarro.

Their approach for finding secondary occurrences is similar to Navarro's, which we will describe in Subsection 4.2.3. Their approach for finding primary occurrences is similar to the approach of Kärkkäinen and Ukkonen.

Assuming the underlying FM-indexes used are the alphabet friendly FM-Index, the Lempel-Ziv index of Ferragina and Manzini requires $O(uH_k \log^\gamma u) + o(u \log \sigma \log^\gamma)$ bits, counts occurrences in $O(m(1 + \log \sigma / \log \log u))$ time, requires $O(1)$ time to report an occurrence and displays ℓ characters in $O(\ell + \log^\epsilon u)$ time, for any constants $\gamma > 0$ and $0 < \epsilon < 1$, $\sigma = o(u^{1/\log \log u})$, $k = o(\log_\sigma u)$. Note that since their index contains the FM-index of T, one can remove the *occ* variable from the counting time.

4

The Inverted Lempel-Ziv Index

In this Chapter we present a new compressed self-index, the inverted Lempel-Ziv index (ILZI). The ILZI is based on the Lempel-Ziv data compression technique, just like the indexes described in Chapter 3. We begin by giving some observations that motivated us to develop this new index. Next we give a theoretical description of the ILZI based on generic dictionaries. Finally, we explain the practical decisions made in the implementation of ILZI and present some empirical results obtained with our prototype.

4.1 Observations and Motivation

There were several reasons for developing a new compressed index based on the Lempel-Ziv data compression technique. Some remain valid today and others have become less important as the theory of compressed indexes evolved.

The main reason to develop a new LZ-index was the quadratic nature of its time requirements. Let us ignore the index proposed by Ferragina and Manzini [37] for now. A quick look at the time performance of LZ-indexes (see Section 3.4) shows that the dependency on m of these indexes is at least $O(m^2)$. This is an intriguing phenomenon, specially since compressed indexes based on other types of data compression have a dependency of only $O(m)$. Naturally, it is interesting to investigate why this phenomenon occurs and whether the performance can be improved. Moreover, we are not only interested in this problem because it is an intriguing theoretical question, we are also very concerned about its consequences in practice.

The reason why the dependency on m is $O(m^2)$ is that any sub-string of P can be a Lempel-Ziv block and all such blocks must somehow be considered in the search algorithm. Obviously the sub-strings that are not LZ blocks do not need to be considered. This means that in fact, on average, this dependency may not be $O(m^2)$. Since the average size of the LZ-blocks of T is asymptotically $(\log u)/H_k$ the average dependency should be closer to $O(m \min(m, (\log u)/H_k))$. This means that in practice we are proposing to improve previous LZ-indexes by a factor of $(\log u)/H_k$. This factor varies and some patterns will have a bigger speedup than others [117]. Speeding up LZ-indexes by a factor of $(\log u)/H_k$ is a considerable improvement in practice. This means that addressing this problem is not merely a theoretical exercise.

Now that we established our cause let us address the index proposed by Ferragina and Manzini [37]. The time performance of this index has a linear dependency on m. Moreover it is able to report each occurrence in $O(1)$ time, which is an extraordinary result. Therefore it seems the problem we are trying to solve already has a solution. In part this is true, but there are some shortcomings to this solution. The first obvious problem with this index is that it requires too much space, $O(uH_k \log^\gamma u) + o(u \log \sigma \log^\gamma u)$ bits for any $\gamma > 0$, or in a more compact version $O(uH_k \log\log u) + o(u \log \sigma \log\log u)$ bits. This argument however is not enough to discard this approach right away. In fact the culprit for this exaggerate space requirements is the data structure used for range queries, since all the remaining components of this index have more "moderate" space requirements. If they use the structure given by Chazelle [22], the time performance would still be linear on m, although now affected by a $\log u$ factor, and the space requirements would be significantly reduced. Note that in this scenario the reporting time is no longer $O(1)$ per occurrence but $O(\log u)$ instead. It is therefore crucial to understand how Ferragina and Manzini solve the problem of the dependency on m. The solution they presented was intriguing since they used another compressed index. One component of their LZ-index is an FM-index, in fact more than one. Their solution is interesting because it suggests that the original LZ-parsing is probably not very appropriate and that it should be replaced. However they replaced it by using another data compression technique, namely the Burrows-Wheeler transform. This solution has some shortcomings since the FM-Index has some problems of its own, namely its dependency on the alphabet size. The original FM-index presented by Ferragina and Manzini [35] contained a large constant $\sigma \log \sigma$ in the sub-linear part, which does not decrease with the entropy and an enormous additive constant larger than σ^σ. These problems however were systematically addressed [38, 48, 73] and are nowadays much less preeminent. Therefore in theory it is possible to achieve the same result we present using the approach proposed by Ferragina and Manzini with a state of the art FM-Index.

However, such an approach would be very complex. Moreover its real performance would, most likely, not be significantly improved. In practice this index can be implemented either with the range data structure or without it. If we use the range data structure the result is an FM-Index with improved reporting performance. However this improvement will most likely be null in practice since Navarro pointed out that using the range data structure is less effective, both in terms of space and time, than a direct scan (see Subsection 3.4.2). If we do not use the range data structure we can either use only the FM-Index we have been mentioning or do direct scans. In the first case the resulting index is nothing more than an FM-Index. The second case contains all the same components of the index we use in practice, however we do not need to implement them as FM-Indexes.

4.2 Theoretical Description and Results

In this Section we explain the first major contribution of this thesis. In Subsection 3.1.3 we presented the Lempel-Ziv data compression. We pointed out that this technique works by inferring a dictionary that is appropriate for T. Therefore we present our algorithm by assuming that we are given a dictionary. This way we can explain our algorithm by exploring the similarities that it has with an inverted file. We use this similarity to provide insights into the algorithm. As we mentioned in Subsection 2.4, inverted indexes work by dividing the text into words and for every word storing the suffixes by which it starts. In general however the dictionary is predefined. In Lempel-Ziv indexes we use the Lempel-Ziv data compression to infer the dictionary instead.

The ranges we are going to use are obtained from other structures in our index, in particular from suffix trees.

Definition 4.1. *The* ***range*** $I(p)$ *of a point* p *of a suffix tree* $\mathcal{T}$ *is the interval of the* DFS' *values of the points that are descendants of* p.

For the example in Figure 4.1 we have that $I(c) = [5, 8]$.

4.2.1 Succinct Suffix Trees

Since our approach is based on suffix trees, we begin our description by presenting a succinct representation of suffix trees that is adequate for our index. We have already mentioned that trees can be represented as a sequence of parentheses, i.e. they can be represented as a bitmap. For example, the bitmap in Figure 4.2 represents the suffix tree $\mathcal{T}$ in Figure 4.1 (top-right). The node with DFS value 2 is represented by the parentheses at positions 3 and 8 of B. The DFS value can be obtained from B as $\textsc{Rank}(B, 3) + B[3] - 1 = 2 + 1 - 1 = 2$. This is the definition of the LEFTRANK operation, i.e. LEFTRANK corresponds to DFS. The $\textsc{RightRank}(v)$ corresponds to the largest DFS value among the descendants of v. This operation can be computed as $\textsc{Rank}(B, 8) + B[8] - 1 = 5 + 0 - 1 = 4$. This is consistent with Figure 4.1, where the node with DFS value 4 is the last descendant of the node with DFS value 2.

We assume that the tree structure of $\mathcal{T}$ and $\mathcal{T}^R$, the reverse tree of $\mathcal{T}$ (see Subsection 2.2.2), are stored using the representation of Geary et al. [43] (see Subsection 3.2.3).

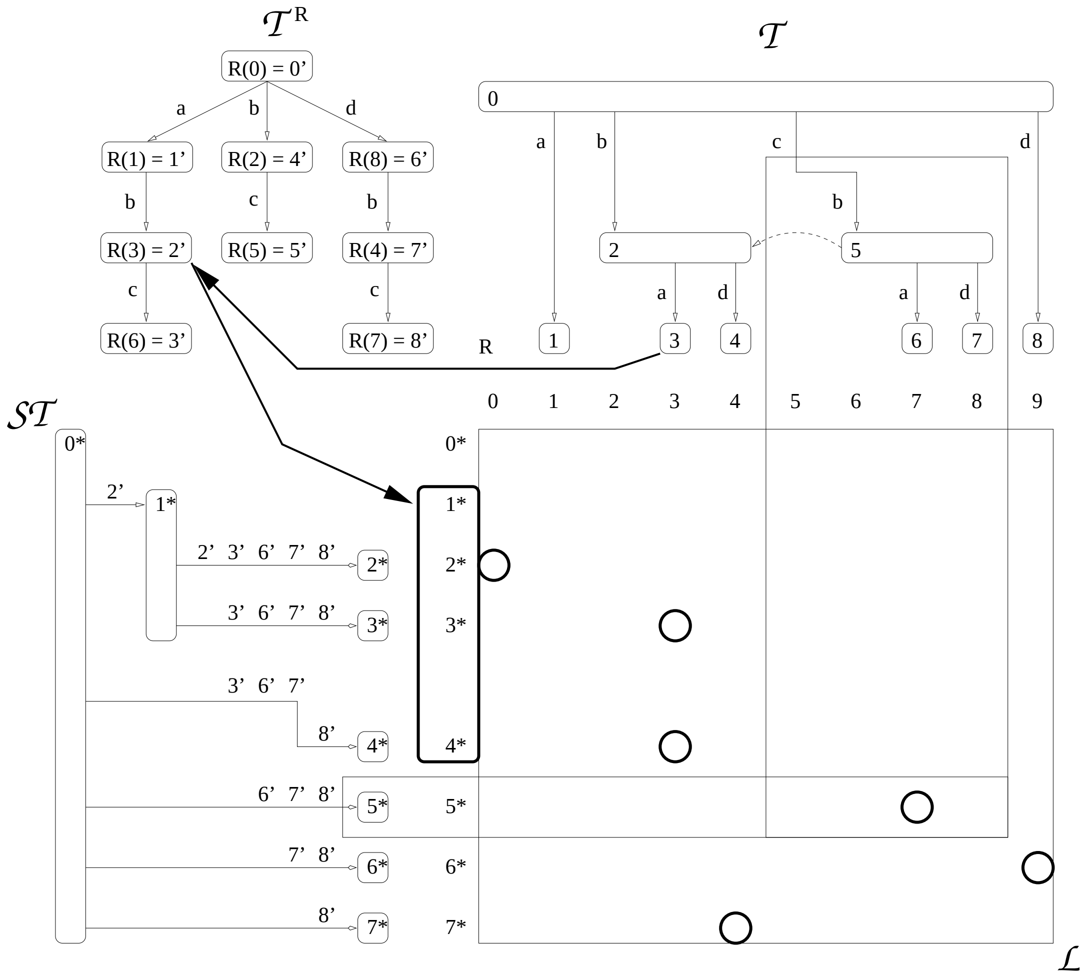


Fig. 4.1. (top-right) Suffix tree for strings $\{a, b, ba, bd, cba, cbd, d\}$. Suffix link from cb to b shown by a dashed arrow. Nodes show their DFS value in $\mathcal{T}$. (top-left) Reverse tree of the suffix tree on the right. Nodes show their DFS value in $\mathcal{T}^R$. The R mapping is shown and $R(3)$ is indicated by a bold arrow. (bottom-left) Sparse suffix tree of T, nodes show their DFS$_{\mathcal{ST}}$ values. Weak descent $W(\text{ROOT}_{\mathcal{ST}}, 2')$ shown in bold rectangle. (bottom-right) Linking points over spaces supported by DFS' and DFS$_{\mathcal{ST}}$ values. Orthogonal range query [5*,5*]:[5,8].

```
             1
   012345678901234567
B: 110110100110100100
   (()(()())(()())())
```

Fig. 4.2. Sample bitmap that represents the suffix tree $\mathcal{T}$ in Figure 4.1 (top-right)

Arroyuelo et al. [6] proposed a way to represent the R mapping. Since R is a permutation, R and R^{-1} can be stored using the representation of Munro et al. [85] in $(1+\epsilon)d\log d+o(d)$ bits, where ϵ is fixed and $0<\epsilon\leq 1$. This way R and R^{-1} can be computed in $O(1)$ and $O(1/\epsilon)$ time respectively (see Subsection 3.2.3).

Lemma 4.2. *A suffix tree $\mathcal{T}$ with d nodes can be stored in $(1+\epsilon)d\log d+5d+o(d)$ bits. Let p be a point, c a letter and v a node of $\mathcal{T}$. This representation provides the operations given by Geary et al. in $O(1)$ time. Moreover it provides* $\textsc{Sdep}(v)$ *in $O(1)$ time,* $\textsc{Suffix\_Link}(v)$ *and* $\textsc{Letter}(v,i)$ *in $O(1/\epsilon)$ time and* $\textsc{Descend?}(p,c)$*,* $\textsc{Descend}(p,c)$ *in $O((\log\sigma)/\epsilon)$ time.*

Proof. We compute $\textsc{Sdep}(v)$ as $\textsc{Depth}_{\mathcal{T}^R}(R(v))$. The operation $\textsc{Suffix\_Link}(v)$ is computed as $R^{-1}(\textsc{Father}_{\mathcal{T}^R}(R(v)))$. Observe that $v[0]$ represents the letter just below the root. For example $cbd[0]=c$. We define a bitmap D to compute $v[0]$, in a way similar to Sadakane [112]. We have that $D[0]=1$ and, for $i>0$, $D[i]=0$ iff $\textsc{Dfs}(v)=i$, $\textsc{Dfs}(v')=i+1$ and $v[0]=v'[0]$. In our example $D=11001001$. We can compute $v[0]$, when v is not the $\textsc{Root}$, in $O(1)$ as the letter in position $\textsc{Rank}_1(D,\textsc{Dfs}(v))$ of Σ. This requires $d+o(d)$ bits. The operation $\textsc{Letter}(v,i)$ can be computed from $R^{-1}(\textsc{Anc}_{\mathcal{T}^R}(R(v),i))$. This expression represents the node obtained after following i suffix-links, hence $\textsc{Letter}(v,i)$ is the first character of its path-label, i.e. $\textsc{Letter}(v,i)=R^{-1}(\textsc{Anc}_{\mathcal{T}^R}(R(v),i)[0]$. When p is not a node, $\textsc{Descend?}(p,c)$ can be computed in $O(1/\epsilon)$ time by consulting $\textsc{Letter}$ for the point below p. If p is a node, we do a binary search among the children of p. If we find a child that starts with c, we return true. Procedure $\textsc{Descend}(p,c)$ updates the value of p. When p is a point, this is done in $O(1)$ time. When p is a node, we first proceed as $\textsc{Descend?}$. $\square$

Finally observe that with this representation we cannot compute $\textsc{Dfs}'(v)$. The $\textsc{Dfs}'$ values are essential to our algorithm because they serve as a supporting space for range queries. This result can be obtained with a compressed bitmap of size t.

Lemma 4.3. *For a suffix tree $\mathcal{T}$ with t points, operations* $\textsc{Dfs}'(p)$ *and* $I(p)$ *can be computed in* $O(1)$ *time using* $tH_0 + o(n) + O(\log\log t)$ *extra bits, where* H_0 *is the empirical entropy of a bitmap with* $(t - 2n)$ *ones and* $2n$ *zeros.*

Proof. Consider the bitmap that for every point of $\mathcal{T}$ stores 1 if the corresponding point is a node and 0 if it is not a node. The bitmap is sorted in DFS' order. In our running example the points of $\mathcal{T}$ that correspond to nodes are the ones with DFS' values $0, 1, 2, 3, 4, 6, 7, 8, 9$. Therefore the resulting bitmap is 1111101111. Using the compressed representation of Raman et al. [103] this bitmap can be stored in $tH_0 + o(n) + O(\log\log t)$ bits supporting $\textsc{Select}_1$ in $O(1)$ time. Observe that for a node v we have that $\textsc{Dfs}'(v) = \textsc{Select}_1(\textsc{Dfs}(v))$.

For a point p, $\textsc{Dfs}'(p)$ is computed as $\textsc{Dfs}'(v) - \textsc{Sdep}(v) + \textsc{Sdep}(p)$, where v is the highest node that is a descendant of p. Also $I(p) = [\textsc{Dfs}'(p), \textsc{Dfs}'(\textsc{Select}(\textsc{RightRank}(v)))]$. $\square$

4.2.2 Generic Inverted Index

Throughout Section 4.2 we assume that we are given an arbitrary suffix tree $\mathcal{T}$ with d nodes, that we will use as a dictionary. We consider as dictionary *words* the path-labels of the nodes of $\mathcal{T}$. The first thing we should do is to organize T according to our dictionary $\mathcal{T}$, much like what is done in inverted files when given a lexicon.

Definition 4.4. *The* $\mathcal{T}$***-maximal parsing*** *of string* T *is the sequence of nodes* $v_1, \ldots, v_f$ *such that* $T = v_1 \ldots v_f$ *and, for every* j, v_j *is the largest prefix of* $v_j \ldots v_f$ *that is a node of* $\mathcal{T}$.

We assume that $\mathcal{T}$ is appropriate for T, i.e. that it is possible to parse T in a maximal way. In our example, the $\mathcal{T}$-maximal parsing of a string T is the sequence cbd, bd, d, cba, ba, ba. We refer to the elements of the $\mathcal{T}$-maximal parsing of T as *blocks*. This notion, although simple, is absolutely crucial to our algorithm. We intend to replace the LZ78 parsing of T

with the $\mathcal{T}$-maximal parsing. The reason for this is that usually LZ-indexes consider all the sub-strings of P that are LZ-blocks, and there can be $m(m+1)/2$ such sub-strings, i.e. $O(m^2)$. By replacing the LZ78 parsing with the $\mathcal{T}$-maximal we are able to reduce the sub-strings we consider to the sub-strings of P that are nodes in $\mathcal{T}$ and maximal. A sub-string of P is maximal in this context if it is not the prefix of another sub-string of P that is also a node of $\mathcal{T}$. For example for $P = cbdbddc$ the sub-string cb is not maximal since the sub-string cbd is also a node of $\mathcal{T}$, cbd however is indeed a maximal sub-string of P. This is in fact the first step towards obtaining a linear dependency on m, since P cannot have more than $O(m)$ maximal sub-strings.

We will store the $\mathcal{T}$-maximal parsing of T in compact form as a string of numbered blocks.

Definition 4.5. *The* ***translation*** $V(v_1 \ldots v_f)$ *of a sequence* $v_1 \ldots v_f$ *of nodes is a string such that* $V(v_1 \ldots v_f)[i] = \textsc{Dfs}(v_i)$.

We denote by $\mathcal{T}(T)$ the translation of the $\mathcal{T}$-maximal parsing of T. Since the $\mathcal{T}$-maximal parsing of T is the sequence cbd, bd, d, cba, ba, ba, its translation is the string $\mathcal{T}(T) = 748633$. Note that word ba is associated with two blocks, v_5 and v_6.

Inverted files usually store a list of occurrences for every word of the dictionary. To play this role we will use a stronger indexing structure, a sparse suffix tree. This sparse suffix tree indexes the reverse of $\mathcal{T}(T)$. This "reverser" is obtained by extending the canonical mapping (see Subsection 2.2.2) R to sequences in the following way: $R(v_1 \ldots v_f) = R(v_f) \ldots R(v_1)$. In our example $R(\mathcal{T}(T)) = R(748633) = R(3)R(3)R(6)R(8)R(4)R(7) = 2'2'3'6'7'8'$. This corresponds to the notion of reverse string, because the concatenation of the path-labels of $R(\mathcal{T}(T))$ in $\mathcal{T}^R$ is $ab.ab.abc.d.db.dbc = T^R$.

Definition 4.6. *The* ***sparse suffix tree***[1] $\mathcal{ST}$ *of a string* T *and a suffix tree* $\mathcal{T}$ *is the suffix tree of* $R(\mathcal{T}(T))$.

[1] Similar to a concept defined by Kärkkäinen et al. [61]

The sparse suffix tree of our example is shown in Figure 4.1 (bottom-left). We can descend in the sparse suffix tree in the usual way with $\textsc{Descend}_{\mathcal{ST}}$. In LZ-indexes it is necessary to deal with the blocks that contain the extremes of P. In particular to deal with the block that contains the left edge of P we will need to specify in some way all the blocks that terminate by a given suffix. This corresponds to a range of nodes in $\mathcal{ST}$. The notion of weak descend gives a precise definition of this range. Since $\mathcal{T}^{\mathcal{R}}$ provides the alphabet for $\mathcal{ST}$, we can also take that into consideration when descending.

Definition 4.7. *The* ***weak descent*** $W(p, v^R)$ *for a point* p *in* $\mathcal{ST}$ *and a node* v^R *in* $\mathcal{T}^R$ *is the interval of* $\textsc{Dfs}_{\mathcal{ST}}$ *values of the nodes below the following points:*

$\{p.\textsc{Dfs}_{\mathcal{T}^R}(v') \mid v'$ *is a descendant of* v^R *in* $\mathcal{T}^R\}$.

For example, $W(\textsc{Root}_{\mathcal{ST}}, 2') = [1^*, 4^*]$, since this contains the $\textsc{Dfs}_{\mathcal{ST}}$ values for the nodes below $2', 3'$ in $\mathcal{ST}$ (see Figure 4.1). This can be computed in $O((\log d)/\epsilon)$ time. We perform two binary searches in the children of p, searching for $\textsc{LeftRank}_{\mathcal{T}^R}(v)$ and $\textsc{RightRank}_{\mathcal{T}^R}(v)$. Then $W(p, v^R) = [\textsc{LeftRank}_{\mathcal{ST}}(v''), \textsc{RightRank}_{\mathcal{ST}}(v''')]$, where v'' and v''' are the nodes found by the binary searches.

In order to find occurrences of strings across more than one block, we will need to store the relations across contiguous blocks. This motivates the following two definitions.

Definition 4.8. *The* ***head, tail*** *of the* $\mathcal{T}$*-maximal parsing are respectively sequence* $v_1, \ldots, v_i$ *and string* $v_{i+1} \ldots v_f$ *such that* $v_1, \ldots, v_i$ *is the smallest sequence for which* $v_{i+1} \ldots v_f$ *is a point in* $\mathcal{T}$.

We denote by $H(T)$ the translation of the head of the $\mathcal{T}$-maximal parsing of T. The head of the $\mathcal{T}$-maximal parsing of T is cbd, bd, d, cba, ba and the tail is the string ba. Hence $H(T)$ equals 74863. It may seem that *tail* is always just v_f. However that is not always the case. Consider a modification $\mathcal{TM}$ of tree $\mathcal{T}$ were node cbd is replaced by $cbde$ and nodes bde, de, e are added to complete the suffix tree. Note that cbd is not a node of $\mathcal{TM}$ it is only a point. The string *bcbd* is parsed as *b.cb.d* and the tail is *cb.d* and therefore it is not just the last block.

Next we define a set of points relating the leaves of $\mathcal{ST}$ with the points in $\mathcal{T}$.

Definition 4.9. *The **linking points set** of the $\mathcal{T}$-maximal parsing $v_1 \dots v_f$ of T is the following set:*

$$\mathcal{L} = \left\{ \langle \text{DFS}(R(V(v_1 \dots v_i))), \text{DFS'}(p_i) \rangle \,\middle|\, \begin{array}{l} p_i \textit{ is the largest prefix of } v_{i+1} \dots v_f \\ \textit{that is a point in } \mathcal{T}, \textit{ for } 0 < i \le f \end{array} \right\}$$

The set $\mathcal{L}$ is shown in Figure 4.1 (bottom-right) and consists of the following points:

- $\langle \text{DFS}(R(V(cbd, bd, d, cba, ba, ba))), \text{DFS'}(\epsilon) \rangle = \langle \text{DFS}(2'2'3'6'7'8'), 0 \rangle = \langle 2^*, 0 \rangle$
- $\langle \text{DFS}(R(V(cbd, bd, d, cba, ba))), \text{DFS'}(ba) \rangle = \langle \text{DFS}(2'3'6'7'8'), 3 \rangle = \langle 3^*, 3 \rangle$
- $\langle \text{DFS}(R(V(cbd, bd, d, cba))), \text{DFS'}(ba) \rangle = \langle \text{DFS}(3'6'7'8'), 3 \rangle = \langle 4^*, 3 \rangle$
- $\langle \text{DFS}(R(V(cbd, bd, d))), \text{DFS'}(cba) \rangle = \langle \text{DFS}(6'7'8'), 7 \rangle = \langle 5^*, 7 \rangle$
- $\langle \text{DFS}(R(V(cbd, bd))), \text{DFS'}(d) \rangle = \langle \text{DFS}(7'8'), 9 \rangle = \langle 6^*, 9 \rangle$
- $\langle \text{DFS}(R(V(cbd))), \text{DFS'}(bd) \rangle = \langle \text{DFS}(8'), 4 \rangle = \langle 7^*, 4 \rangle$

To compute orthogonal range queries we use the wavelet tree. As we mentioned before (see Subsection 3.2.2), this structure requires $f \log f'(1+o(1))$ bits and can compute orthogonal range queries in the space $[1, f] \times [1, f']$ in $O((1+occ') \log f')$ time. We need to store points in the $[0, d'-1] \times [0, t-1]$ space, where d' is the number of nodes of $\mathcal{ST}$. We only need to store f points. Therefore we must reduce the support space to the rank space.

The space $[0, d'-1]$ can be reduced to $[1, f]$ in $O(1)$ time, with Rank over a bitmap of $d' + o(d')$ bits. This bitmap contains a 1 for leaf of $\mathcal{ST}$, and the nodes are ordered by Dfs. In our example the leaves correspond to nodes $2^*, 3^*, 4^*, 5^*, 6^*$, i.e. the bitmap is 0011111. To determine which point corresponds to node v we compute $\text{Rank}(B, \text{DFS}_{\mathcal{ST}}(v)+1)$. For example for $v = 3'6'7'8'$ we have compute $\text{Rank}(B, 4+1) = 3$, which makes sense since the point associated with this node is the third if we start counting top-down in Figure 4.1 (bottom right).

The reduction of space $[0, t-1]$, is obtained by setting f' to t and, therefore, the time to report occurrences is $O((1 + occ') \log t)$.

We propose an index data structure composed of:

- A dictionary suffix tree $\mathcal{T}$, that is appropriate and chosen in a way that minimizes its space requirements. In Section 4.3 we show how to use the Lempel-Ziv algorithm to infer $\mathcal{T}$.
- The sparse suffix tree $\mathcal{ST}$, for searching translated sub-strings of P.
- The linking points, processed for orthogonal range queries to associate a translated suffix of P with its corresponding prefix.

Our searches for the pattern inside a blocks and spanning more than one block, in different ways. We refer to this as **type** 1 and **type** > 1 occurrences $(occ_1, occ_{>1})$.

4.2.3 Occurrences Lying Inside a Single Block

The algorithm for finding occurrences inside a single block starts by identifying all the words in the dictionary $\mathcal{T}$ that contain P as a sub-string. Since $\mathcal{T}$ is a suffix tree, it is possible to achieve this in a simple way.

- Descend by P in $\mathcal{T}$. If this is impossible then there are no type 1 occurrences of P.
- Start a depth-first traversal of the sub-tree below P.
- For each node v reached compute the range query $W(\textsc{Root}_{\mathcal{ST}}, v^R) : [0, t]$.

The search in $\mathcal{T}$ consists in considering words that start with P and appending some letters. The weak descend and the range query consist in prep-ending some letters to the words found on the search in $\mathcal{T}$. For example, consider $P = b$. By reading b, we reach node 2 of $\mathcal{T}$ (see Figure 4.1). The search on $\mathcal{T}$ returns nodes $2, 3, 4$, which leads us to consider words b, ba, bd. This originates the following weak descends: $W(\textsc{Root}_{\mathcal{ST}}, 4') = \emptyset$, $W(\textsc{Root}_{\mathcal{ST}}, 2') = [1^*, 4^*]$, $W(\textsc{Root}_{\mathcal{ST}}, 7') = [6^*, 7^*]$. We don't need to consider words that start with b, since they don't correspond to blocks; there may be occurrences of ba or cba

because of *ba*; there may be occurrences of *bd* and *cbd* because of *bd*. The range queries return no occurrences for b, occurrences 2^*, 3^* and 4^* for *ba* and occurrences 6^* and 7^* for *bd*. This corresponds to occurrences $cbd.bd.d.cba.ba.\underline{b}a$, $cbd.bd.d.cba.\underline{b}a.ba$, $cbd.bd.d.c\underline{b}a.ba.ba$ for *ba* and occurrences $cbd.\underline{b}d.d.cba.ba.ba$, $c\underline{b}d.bd.d.cba.ba.ba$, for *bd*.

Theorem 4.10. *The above procedure is correct and complete.*

Proof. (Correct) Clearly every reported block is $\alpha.P.\beta$ for some α,β and hence it contains an occurrence of P. (Complete) Suppose block $v_i = \alpha.P.\beta$. Hence $\alpha.P.\beta$ is a node in $\mathcal{T}$. Since $\mathcal{T}$ is a suffix tree, $P.\beta$ is also a node in $\mathcal{T}$. Node $P.\beta$ is reached by the search in $\mathcal{T}$, since it starts by P. Every node v of $\mathcal{ST}$ for which $v[0] = \textsc{Dfs}((\alpha.P.\beta)^R)$ has its $\textsc{Dfs}_{\mathcal{ST}}$ time in $W(\textsc{Root}_{\mathcal{ST}}, (P.\beta)^R)$, hence block v_i is found in the range query. □

This algorithm was essentially presented by Navarro [98], except that the range queries were computed as depth-first searches in a trie similar to $\mathcal{T}^R$. In Navarro's algorithm each node of that trie stored one block. Therefore, the time of these searches was bounded by the number of type 1 occurrences of p, denoted by occ_1. We do not have a direct correspondence between the nodes of $\mathcal{T}^R$ and the blocks of $\mathcal{T}$-maximal parsing, which means that this approach has no worst case guarantees. In essence the problem is that we may be executing more range queries than the number of occurrences found. To fix this problem we remove some nodes of $\mathcal{T}$ that are "useless" and responsible for this situation.

Definition 4.11. *A* ***spurious*** *entry for string T in the suffix tree $\mathcal{T}$ is a leaf v of $\mathcal{T}$ such that v^R is a leaf of $\mathcal{T}^R$ and v is not a block in the $\mathcal{T}$-maximal parsing of T.*

For a dictionary $\mathcal{T}$ without spurious entries, we can guarantee that enough orthogonal range queries must return occurrences. Note that in the definition of *spurious* we only considered leaves. However, removing those leaves may cause internal nodes to become leaves and spurious. When we refer to a dictionary without spurious entries we mean that we must have no spurious entries at all, i.e. the nodes that become spurious must also be removed, until there are no spurious entries at all.

Lemma 4.12. *Assuming $\mathcal{T}$ has no spurious entries for T and v is a leaf of $\mathcal{T}$, then the query $W(\textsc{Root}_{\mathcal{ST}}, v^R) : [0, t]$ returns at least one linking point.*

Proof. There is some α such that $(\alpha.v)^R$ is a leaf in $\mathcal{T}^R$. Since $\mathcal{T}$ is a suffix tree and v is a leaf of $\mathcal{T}$, then $\alpha.v$ is also a leaf of $\mathcal{T}$. Hence, at least one linking point will be found by $W(\textsc{Root}_{\mathcal{ST}}, v^R) : [0, t]$, since $\textsc{Dfs}_{\mathcal{ST}}((\alpha.v)^R) \in W(\textsc{Root}_{\mathcal{ST}}, v^R)$. □

Spurious entries may be safely removed from the dictionary. Removing spurious entries can be done by considering $\mathcal{T}$ and $\mathcal{T}^R$ as a DAG, i.e. a node w in the DAG represents simultaneously v and v^R; there is an edge from w to w' if that edge exists in $\mathcal{T}$ or in $\mathcal{T}^R$. To remove spurious entries we perform a DFS over this DAG. The nodes that are spurious are the ones that do not have blocks, and are either sinks or have only one outgoing edge that existed in $\mathcal{T}$. The nodes are checked and removed in their finishing time (see Cormen et al. [28] for definitions). This procedure runs in $O(d)$ time. Note that the resulting structure remains a suffix tree.

4.2.4 Occurrences Spanning more than a Single Block

In this Section we focus on finding occurrences that span two or more consecutive blocks, i.e. occurrences of type > 1. The ideas presented in this Section are similar to those of Kärkkäinen et al. [61] and related with the approach proposed by Ferragina et al. [37].

We are now faced with the problem of retrieving the words in our dictionary that appear concatenated in $\mathcal{T}(T)$ and have P as a sub-string. Suppose that $P = cbdbddc$ and that we split P into two as $cbdbdd$ and c. We will now search for c in $\mathcal{T}$ and for $cbdbdd$ in $\mathcal{ST}$. The point c in $\mathcal{T}$ induces the range $I(c) = [5, 8]$; on the other hand string $cbdbdd$ is parsed into cbd, bd, b and hence will be translated into 748. To search on the sparse suffix tree, we need $R(748) = 6'7'8'$. This will induce the range $[5^*, 5^*]$. Finally, to solve our problem we perform the orthogonal range query $[5^*, 5^*] : [5, 8]$ over the linking points $\mathcal{L}$. This corresponds to the question: is the string $cbdbdd$, parsed as $cbd.bd.d$, ever followed by a block that starts by

i	0	1	2	3	4	5	6	7
P[i]	c	b	d	b	d	d	c	$'
trace_left[i]	ϵ	c	cb	cbd	b	bd	d	c
DFS'(father_left[i])	0	0	6	8	2	4	9	0
DFS'(trace_left[i])	0	5	6	8	2	4	9	5
DFS'(child_left[i])	0	6	6	8	2	4	9	6
trace_right[i]	cbd	bd	d	bd	d	d	c	ϵ
DFS'(father_right[i])	8	4	9	4	9	9	0	0
DFS'(trace_right[i])	8	4	9	4	9	9	5	0
I(trace_right[i])	[8,8]	[4,4]	[9,9]	[4,4]	[9,9]	[9,9]	[5,8]	[0,9]
DFS'(child_right[i])	8	4	9	4	9	9	6	0
P[i..]	cbd.bd.d.c	bd.bd.d.c	d.bd.d.c	bd.d.c	d.d.c	d.c	c	ϵ
tail(P[i..])	c	c	c	c	c	c	c	ϵ
H(P[i..])	748	448	848	48	88	8	ϵ	ϵ
R(H(P[i..]))	6'7'8'	udef	udef	6'7'	6'6'	6'	ϵ	ϵ
|father_left[i]| == i		FALSE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE
W(R(H(P[i..])), R(father_left[i]))			$\emptyset$	[5*,5*]	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
I(tail(P[i..]))		[5,8]	[5,8]	[5,8]	[5,8]	[5,8]	[5,8]	[0,9]
occ'			0	1	0	0	0	0

Table 4.2. (Top) Descend and suffix walk of *cbdbddc* in $\mathcal{T}$. (Bottom) Values for locating type > 1 occurrences.

c? The answer is yes, since there is a linking point in $[5^*, 5^*] : [5, 8]$. This point corresponds to $\underline{cbd.bd.d.c}ba.ba.ba$. We will now explain how to determine in which points to break P. The pattern should be separated into the head and tail of $P[i..]$, for every $0 < i < m$, to account for every possible translation that can occur (see definition 4.8). These points can be determined using the following dynamic programming equations:

$$tail(P[i..]) = \begin{cases} trace\_right[i] & \text{, if } |trace\_right[i]| = m - i. \\ tail(P[i + |father\_right[i]|..]) & \text{, otherwise.} \end{cases}$$

Algorithm 4 Locate $R(H(P[i..]))$ Algorithm

procedure *Locate_HPI*
  for $i \leftarrow m-1, 0 < i$ **do**
    $R(H(P[i..])) \leftarrow \textsc{Root}_{\mathcal{ST}}$
    if $|\text{trace\_right}[i]| < m - i$ **then**
      $R(H(P[i..])) \leftarrow \textsc{Descend}_{\mathcal{ST}}(R(H(P[i+|father\_right[i]|..])), father\_right[i])$
    end if
  end for
end procedure

$$H(P[i..]) = \begin{cases} \epsilon & \text{, if } |trace\_right[i]| = m - i. \\ father\_right[i].H(P[i+|father\_right[i]|..]) & \text{, otherwise.} \end{cases}$$

We use Algorithm 4 to locate points $R(H(P[i..]))$ in $\mathcal{ST}$. Whenever it is not possible to descend by a letter, the $\textsc{Descend}_{\mathcal{ST}}$ procedure returns the $udef$ state. See Table 4.2 (bottom) for an example of this computation. Assume that the descend and suffix walk of P is already computed. Hence, the arguments of $\textsc{Descend}_{\mathcal{ST}}$ are available when $\textsc{Descend}_{\mathcal{ST}}$ is executed. Therefore Algorithm 4 runs in $O((m/\epsilon)\log d)$ time, since it runs m times the $\textsc{Descend}_{\mathcal{ST}}$ operation, which requires $O((\log d)/\epsilon)$ time. Having located $tail(P[i..])$ in $\mathcal{T}$ and $R(H(P[i..]))$ in $\mathcal{ST}$, we know where to break the pattern. Now all that we need are the ranges for the range query. The range for $\mathcal{T}$ is simply $I(tail(P[i..]))$. Whenever $P[..i-1]^R$ is a node of $\mathcal{T}^{\mathcal{R}}$ the range for $\mathcal{ST}$ is $W(R(H(P[i..])), P[..i-1]^R)$.

Let us consider for example the case of $i = 3$. We have that $H(P[3..]) = 48$ and $R(H(P[3..])) = 6'7'$. Hence $W(6'7', (cbd)^R) = [5^*, 5^*]$, since $8'$ is the only descendant of itself in $\mathcal{T}^R$. This means that, when we are extending *bd.d* to the left by prep-ending a word from our dictionary that terminates in *cbd*, the only such word is *cbd*. Therefore we end up considering only the node *cbd.bd.d*.

Our algorithm for finding type > 1 occurrences of P proceeds as follows:

- Compute the descend and suffix walk of P in $\mathcal{T}$.
- Compute $tail(P[i..])$ from the descend and suffix walk of P.

- Locate the $R(H(P[i..]))$ points in $\mathcal{ST}$.
- If $|father\_left[i]| = i$ then $P[..i-1]^R = R(father\_left[i])$,

  compute $W(R(H(P[i..])), R(father\_left[i]))$.
- Compute $I(tail(P[i..]))$ from $tail(P[i..])$.
- Compute the orthogonal range queries $W(R(H(P[i..])), R(father\_left[i])) : I(tail(P[i..]))$.

An example of our algorithm is shown in Table 4.2 (bottom). The only range query that finds occurrences (occ') is the $[5^*, 5^*] : [5, 8]$ query, as we have explained in this Section.

4.3 A Compressed Self-Index based on LZ78 Dictionaries

We found it interesting to present this work in a general form, since it seems relevant to explore other techniques for inferring dictionaries, given a text T. We will now give a concrete instantiation of the above algorithm, using the Lempel-Ziv 78 Algorithm [127]. See Subsection 3.1.3 for a description of the Lempel-Ziv data compression algorithm.

Given a string T, we proceed as follows: compute the LZ78 parsing of $T^R = Z_1 \ldots Z_n$, then consider the suffix tree for strings $\{Z_1^R, \ldots, Z_n^R\}$ as our dictionary, denoted by $\mathcal{T}_{78}$. In our example T^R is parsed into $a, b, ab, abc, d, db, dbc$ and the resulting dictionary can be seen in Figure 4.1 (top-right).

The following lemmas give bound the number elements in our index.

Lemma 4.13. *If the number of blocks of the LZ78 parsing of T is n then the $\mathcal{T}_{78}$ has at most $2n$ nodes, i.e. $d \leq 2n$.*

Proof. Observe that every suffix of a Z_i^R is a Z_j^R for some j. Therefore the set $\{Z_1^R, \ldots, Z_n^R\}$ is suffix closed. Hence a suffix tree based on $\{Z_1^R, \ldots, Z_n^R\}$ will have at most $2n$ nodes. □

Lemma 4.14. *If the number of blocks of the LZ78 parsing of T is n then the $\mathcal{T}_{78}$-maximal parsing of T has at most n blocks, i.e. $f \leq n$.*

Proof. The idea is to show that if a block v_i of the $\mathcal{T}_{78}$-maximal parsing is a sub-string of some Z_j^R then it is a suffix. Suppose that v_i is a sub-string of Z_j^R. We have that $Z_j^R = \alpha.v_i.\beta$. Since the dictionary is a suffix tree and Z_j^R is a node, $v_i\beta$ is also a node and hence a dictionary word. Since the parsing is maximal, we have that $v_i.\beta = v_i$, i.e. that v_i is a suffix of Z_j^R.

Recall that $T = v_1 \dots v_f = Z_n^R \dots Z_1^R$. From the property we demonstrated we conclude that $|v_1| \geq |Z_n^R|$. Moreover by induction it holds that $|v_1 \dots v_i| \geq |Z_n^R \dots Z_{n-i+1}^R|$. Hence $|Z_n^R \dots Z_1^R| = |T| = |v_1 \dots v_f| \geq |Z_n^R \dots Z_{n-f+1}^R|$, which means that $1 \leq n - f + 1$, i.e. $f \leq n$. □

4.3.1 Space and Time Complexity

With the previous results we will now determine the space and time complexity of our algorithm using an LZ78 dictionary.

Lemma 4.15. *The* DFS'$_{78}$ *operation can be supported over $\mathcal{T}_{78}$ in $O(1)$ time with $o(u \log \sigma)$ bits.*

Proof. This result is obtained from Lemma 4.3. Observe that t, the number of points of $\mathcal{T}_{78}$, can be at most u. Moreover the largest value that tH_0 assumes can be bounded by the value of uH_0 for a bitmap with u bits, $2n$ of which are 1's. Using the bound that $n \leq u/\log_\sigma u$, proved by Ziv et al. [127] we can show that the space occupied by this bitmap is at most $2u \log \sigma (\log \log u / \log u) + o(u \log \log u / \log u)$ bits, which is $o(u \log \sigma)$. □

We will refer to the index that uses LZ78 dictionaries as the Inverted-LZ-Index (ILZI). The next theorem gives an overview of the space/time complexity of this structure. A previous version of this result [106] required more space.

Theorem 4.16. *Let d and d' be the number of nodes of $\mathcal{T}_{78}$ and $\mathcal{ST}_{78}$ respectively. Let t be the number of points of $\mathcal{T}_{78}$. Let f be the size of the $\mathcal{T}_{78}$-maximal parsing of T. The space/time trade-off of the Inverted-LZ-Index can be summarized as follows:*

Space in bits	$[\frac{d}{n}(1+\epsilon)+\frac{d'}{n}(1+\epsilon)+\frac{f}{n}]uH_k+o(u\log\sigma)$
	$\leq (5+\epsilon)uH_k+o(u\log\sigma)$
Time to count	$O(((m/\epsilon)+occ)\log u)$
Time to locate	*free after counting*
Time to display l chars	$O(l/\epsilon)$, *improvable to* $O(l/(\epsilon\log_\sigma u))$
Conditions	$k=o(\log_\sigma u)$, $\sigma=O(u)$, $0<\epsilon\leq 1$, ϵ *is constant*

Proof. (Space) The space requirements come from adding up the space of $\mathcal{T}_{78}$, $\mathcal{ST}_{78}$ and the range data structure. The $\mathcal{T}_{78}$ suffix tree suffix tree requires at most $(1+\epsilon)d\log d+5d+o(d)$, according to Lemma 4.2. Moreover to support DFS'$_{78}$ we need $o(u\log\sigma)$ extra bits. The $\mathcal{ST}_{78}$ sparse suffix tree requires $(1+\epsilon)d'\log d'+5d'+o(d')$ bits, according to Lemma 4.2. The range data structure (wavelet tree) requires another $f\log f(1+o(1))$ bits. According to lemmas 4.13 and 4.14, $d,d'\leq 2n$, $f\leq n$, hence the dominate factor is $(5+\epsilon)\log u$. Ziv et al. [127] showed that $\sqrt{u}\leq n\leq u/\log_\sigma u$, and, therefore $n=o(u\log\sigma)$, which means that all remaining space is $o(u\log\sigma)$. The relation between n and H_k was established by Kosaraju et al. [66], who showed that $n\log u=uH_k+o(u\log\sigma)$ for $k=o(\log_\sigma u)$. Therefore the expression in the theorem accounts for the space requirements of the ILZI.

(Count/Locate) We have already seen that Algorithm 1 runs in $O((m/\epsilon)\log\sigma)$ time. The time to find occurrences of type 1 is $O((1+occ_1)\log u)$. Observe that the number of queries computed is less than or equal to twice the number of leaves below P. By Lemma 4.12 we know that the queries at the leaves must return occurrences. Therefore the total time amortizes to $O((1+occ_1)\log u)$. The time to find occurrences of type >1 is the time of Algorithm 4, plus m weak descents and m range queries. Therefore the total time for occurrences of type >1 is $O((occ_{>1}+m/\epsilon)\log u)$, where $occ_{>1}$ is the number of type >1 occurrences.

(Display) Observe that even though we do not store $R(\mathcal{T}_{78}(T))$ explicitly, we have $O(1/\epsilon)$ access time to it. The idea is to store a pointer to the leaf of $\mathcal{ST}_{78}$ with path-label $R(\mathcal{T}_{78}(T))$, denoted by $\text{FirstLeaf}_{\mathcal{ST}}$. Therefore $R(\mathcal{T}_{78}(T))[i] = \text{Letter}_{ST}(\text{FirstLeaf}_{\mathcal{ST}}, i)$. Hence we can compute the j-th letter of $R(\mathcal{T}_{78}(T))[i]$ as $\text{Letter}(\text{Letter}_{ST}(\text{FirstLeaf}_{\mathcal{ST}}, i), j)$, in $O(1/\epsilon)$ time. To achieve optimal $O(l/(\epsilon \log_\sigma u))$ time we use an approach based on the work of Sadakane [113], similar to Arroyuelo et al. [6]. We define a new bitmap D' similar to bitmap D used to retrieve the first $\log u$ bits of a node v instead of the first letter. This requires $d + o(d)$ bits. We also need a bitmap Q that indicates which sequences of $(\log u)/2$ bits do appear as the first bits of some v. By $(i)_2$ we denote the binary representation of i, with $(\log u)/2$ bits. The Q bitmap is defined as $Q[i] = 1$ iff $(i)_2$ is the prefix of some $(v)_2$ padded with zeros. Bitmap Q contains $2^{(\log u)/2} = \sqrt{u}$ bits and can therefore be stored in $o(u)$ bits. With these bitmaps we are able to retrieve $(\log u)/2$ bits from a block in $O(1)$ time, i.e. $\log_\sigma u/2$ letters. We repeat these bitmaps for $\mathcal{ST}_{78}$ and hence are able to retrieve $(\log u)/2$ bits from consecutive blocks. Finally we need another bitmap to be able to skip blocks. We use a bitmap V that marks the beginnings of the blocks in $R(\mathcal{T}_{78}(T))$. In our running example T is parsed as *cbd.bd.d.cba.ba.ba*. Therefore the V bitmap would be 100.10.1.100.10.10 (the dots are obviously not part of the bitmap). This can be represented in $o(u \log \sigma)$ bits using the argument of Lemma 4.15. As pointed out by Arroyuelo et al. [6], this bitmap can be used to report the occurrences of P as positions in T instead of as a block and an offset. This can be obtained with a simple Select query. For example to determine the starting position of the second block we compute $\text{Select}_1(V, 2) = 3$. $\square$

The worst case of the space expression is $(5+4\epsilon)H_k + o(u \log \sigma)$. However the worst example we were able to find, based on De Bruijn cycles, yielded $(4+3\epsilon)H_k + o(u \log \sigma)$ bits. In the next Section we show concrete values for the space expression. We will now explain this bad case. A De Bruijn Cycle for a given q is a binary string of size $2^q - q + 1$ whose sub-strings of size q are all the different strings of size q. For example the string 0000111101100101000 is a De Bruijn cycle for $q = 4$, see Figure 4.3. Our example consists in setting T to be the prefixes of a De Bruijn Cycle in such a way that the resulting LZ78-trie is the cycle itself. In our example this would yield 0.00.000.0000..., note that the dots are not part of

```
0000111101100101000
0000 1110 1001 1000
 0001 1101 0010
  0011 1011 0101
   0111 0110 1010
    1111 1100 0100
```

Fig. 4.3. Bruijn Cycle for $q = 4$

the string and are only present to make it easier to understand the example. This example is designed to make the $\mathcal{T}_{78}$ have as many nodes as possible and in fact the expression d/n converges to 2 in this example, with increasing q. This is easy to observe since $\mathcal{T}_{78}$ contains approximately contains one leaf for every node of the LZ-trie, therefore in total it will contain about $2n$ nodes. However we were unable to make d'/n converge to 2 and in fact it converges to 1 in this example. Since f/n also converges to 1 we obtain the $(4 + 3\epsilon)H_k + o(u \log \sigma)$ worst case we mentioned.

Let us make a brief parentheses to point out that some simpler alternatives to our approach fail. In fact it is really necessary to reverse T and to use the suffix tree $\mathcal{T}_{78}$. Suppose that instead of using the approach we have been described we decided to parse T forward and do a maximal parsing with the LZ78 trie of T instead of with $\mathcal{T}_{78}$. In this scenario consider that $T = aabaa$, the LZ78 parsing of T is $a.ab.aa$. Observe, however, that it is not possible to do a maximal parsing this way since the process would begin by selecting aa but it would stall at b since b is not a block of the LZ78 trie. This problem could be overcome but this is a symptom of a more important problem. If we decide to build our index this way we would loose any guarantees about its space requirements.

Another simplification that one may consider is to think that turning the LZ78 trie into a suffix tree by making it suffix closed should not add too many nodes and that we could use that structure instead. This, in fact, is not true and some experimental results show that we would end up with around $O(u)$ nodes.

4.4 Practical Issues and Testing

4.4.1 Practical Considerations

This Section presents some results obtained with a prototype that was implemented to test these ideas, more extensive empirical results are given in appendix A. Navarro [98] pointed out that, by using a naive search instead of the range data structure, it was possible to build a smaller index that was faster in practice. The naive way to compute an orthogonal range query is to choose the smallest range and, for each point of that range, check whether the point belongs to the other range. Suppose for example, that we wish to compute the range query $W(\textsc{Root}_{\mathcal{ST}}, 2') = [1^*, 4^*] : [0, 9] = [0, t-1]$. First observe that, when we refer to the smallest range, we are referring to the range in the $[1, f] \times [1, f]$ grid not in the $[0, d'-1] \times [0, t-1]$ space. Therefore we reduce the $[1^*, 4^*] : [0, 9]$ query to the $[1^{p*}, 3^{p*}] : [1^p, 6^p]$ query. Obviously the smallest range is the $[1^{p*}, 3^{p*}]$ one. Since, for this particular query, the second range covers the whole space, the result is $[1^{p*}, 3^{p*}]$, which corresponds to $\{2^*, 3^*, 4^*\}$. We have already seen that this type of queries is used for type 1 occurrences. Therefore, using this method, the time to compute the range queries for type 1 occurrences is $O(occ_1)$, since the verification is always trivially true.

For type > 1 occurrences this procedure has no worst case guarantees. However, in practice, this is acceptable and more efficient [98]. Therefore we did not implement the range data structure and used this approach instead. This immediately removes our capability of reducing $[0, t-1]$ to $[1, f]$, which means that we cannot use points of $\mathcal{T}$ to support the linking points. This means that there is no reason to use a compressed bitmap to support the Dfs' operation for points that are not nodes, as described in Lemma 4.3. Instead we store $\langle \textsc{Dfs}(R(V(v_1 \dots v_i))), \textsc{Dfs}(v_{i+1}) \rangle$ when $i < f$ and $\langle \textsc{Dfs}(R(V(v_1 \dots v_i))), 0 \rangle$ when $i = f$, since v_{i+1} is the largest prefix of $v_{i+1} \dots v_f$ that is a node in $\mathcal{T}$. Note that these points can be obtained by storing $\mathcal{T}_{78}(T)$, since two consecutive blocks represent a point.

Observe that the linking points in our example actually coincide exactly with this definition, (see Figure 4.1) (bottom-right). To find the linking points associated with a node v of $\mathcal{T}$, we find the leaves below point $R(v)$ in $\mathcal{ST}$. Moreover, to decide which range is smaller, we estimate the number of points in $I'(v)$ as the number of points in $W(\textsc{Root}_{\mathcal{ST}}, R(v))$.

In Navarro's approach, occurrences of type > 1 are further distinguished between type 2 and type > 2. Navarro did not use dynamic programming, because it is possible to guarantee that there are not too many occurrences of type > 2 (type > 2 occurrences span more than two blocks). The fundamental argument is that, since the LZ78-blocks are all distinct, a given Z_i occurs in at most one position. Therefore the $P[i..j]$ sub-strings of P occur in at most $O(m^2)$ positions. Hence there cannot be more than $O(m^2)$ type > 2 occurrences of P in the LZ78 parsing of T. For $\mathcal{T}_{78}(T)$ no such result exists. However, even though a word v may correspond to more than one block of $\mathcal{T}_{78}(T)$, in average it does not correspond to many. Therefore we do not use dynamic programming either. Instead, we use different procedures for type 2 and type > 2 occurrences.

There is not a very compelling reason to store $\mathcal{ST}_{78}$ as a suffix tree when not using dynamic programming. Inverted files store a list of occurrences for every dictionary word. These lists are usually ordered by the position in T of the occurrences of the words. This regularity is usually explored, for example, with delta coding, to store these lists in compressed form. This property is also important when searching for patterns because, since it scans the text sequentially, it provides better cache performance. Our implementation of $\mathcal{ST}_{78}$ is similar to a sparse suffix array, i.e. a suffix array for $R(\mathcal{T}_{78}(T))$. However, the suffixes of $R(\mathcal{T}_{78}(T))$ are only sorted by the first block. Suffixes that start with same block are ordered by position in $R(\mathcal{T}_{78}(T))$, just like in inverted files.

A very important aspect of our prototype is that the implementation of $\mathcal{T}_{78}$ differs considerably from the succinct representation we presented. The fundamental reason for this fact is that the succinct implementation could suffer from poor cache performance. Instead we opted for a more cache aware implementation. The $\mathcal{T}_{78}$ tree is implemented in a pointer like fashion. Every node is stored in a memory cell indexed by its breath-first time-stamp. For example, node cb will be stored in cell 3. The LETTER operation is replaced by a HEAD pointer, that, for every node v with father node $v[..i-1]$, points to node $v[i..]$. This information suffices to read edge-labels, by using suffix links. Every node v stores a CHILD pointer, its DFS time, the suffix link, the string depth, the HEAD pointer and pointers indicating $W(\text{ROOT}_{\mathcal{ST}_{78}}, v^R)$ over $\mathcal{ST}_{78}$. This provides better cache performance in several points. First, we store the information in the nodes and the topological structure of the tree together. Second, there is no need to traverse back and forth from $\mathcal{T}_{78}$ to $\mathcal{T}^{\mathcal{R}}{}_{78}$ to read edge-labels or compute suffix links. Third, the BFS ordering avoids some cache faults in branching.

Therefore, our implementation consisted in tree main components, the $\mathcal{T}_{78}$ suffix tree, implemented in a pointer like fashion, the $\mathcal{ST}_{78}$ sparse suffix tree, implemented in as a sort of suffix array and the string and the $\mathcal{T}_{78}$ maximal parsing ($\mathcal{T}_{78}(T)$). These structures are linked in the following way: $\mathcal{T}_{78}$ points to $\mathcal{ST}_{78}$ by storing, in every node v, two pointers that represent $W(\text{ROOT}_{\mathcal{ST}_{78}}, v^R)$; $\mathcal{ST}_{78}$ points to $\mathcal{T}_{78}(T)$, since it is a suffix array of $\mathcal{T}_{78}(T)$; $\mathcal{T}_{78}(T)$ points to $\mathcal{T}_{78}$ since it is a sequence of nodes of $\mathcal{ST}_{78}$.

The size of each of these components is dependent on n. However since $\mathcal{T}_{78}$ stores a considerable amount of information per node it requires a considerable amount of space. This constitutes a severe problem. In order to solve it, we infer a smaller dictionary, i.e. a $\mathcal{T}_{78}$ tree with fewer nodes. In practice, we use the following variation of the LZ78 parsing:

Definition 4.17. *The **LZ78 parsing with quorum l** of a string T is the sequence $Z_1, \ldots, Z_n$ of strings such that $T = Z_1 \ldots Z_n$ and, for every i, $Z_i = Z_j c$ where c is a letter and Z_j is the largest prefix of $Z_i \ldots Z_n$ that appears at least $l+1$ times among the $Z_1, \ldots, Z_{i-1}$.*

Clearly the LZ78 parsing with quorum 0 corresponds to the usual notion of LZ78 parsing. In practice a quorum of 2 compensates for the space requirements of $\mathcal{T}_{78}$ without affecting

	english.50MB								
l	$i/2^{23}$	$i/u8$	i/uH_k	d/n	d'/n	f/n	$\lvert\mathcal{T}\rvert$	$\lvert\mathcal{ST}\rvert$	$f(\log u)/2^{23}$
32	47.6	0.95	2.63	0.02	-	1.58	1.81	26.83	19.00
16	46.4	0.93	2.56	0.04	-	1.45	3.41	24.57	18.43
8	45.8	0.92	2.53	0.07	-	1.33	6.18	21.69	17.92
4	48.5	0.97	2.68	0.12	-	1.24	11.01	20.08	17.46
2	54.3	1.09	2.99	0.19	-	1.15	18.44	18.72	17.10
1	61.8	1.24	3.41	0.29	-	1.08	28.19	17.58	16.05
0	93.3	1.87	5.15	0.64	1.33	0.94	63.41	15.28	14.61

	dna.50MB								
l	$i/2^{23}$	$i/u8$	i/uH_k	d/n	d'/n	f/n	$\lvert\mathcal{T}\rvert$	$\lvert\mathcal{ST}\rvert$	$f(\log u)/2^{23}$
32	30.9	0.62	2.24	0.02	-	1.37	1.43	16.94	12.52
16	31.3	0.63	2.27	0.04	-	1.29	2.88	15.92	12.46
8	33.0	0.66	2.39	0.08	-	1.22	5.39	15.11	12.48
4	37.0	0.74	2.68	0.14	-	1.16	10.17	14.36	12.48
2	44.0	0.88	3.19	0.24	-	1.11	17.75	13.73	12.54
1	52.5	1.05	3.81	0.37	-	1.07	27.20	13.22	12.07
0	92.6	1.85	6.72	0.92	1.20	0.97	69.21	11.97	11.45

Table 4.3. Space requirements of the ILZI index for different quorum values. Variable l represents different quorum values. Variable i represents the size of the different indexes in bits. Therefore $i/2^{23}$ gives the size in Megabytes (MB), $i/u8$ gives the ratio with the original string, i/uH_k gives the ratio with a compressed string, where H_k is estimated as $(n \log u)/u$. Columns d/n, d'/n and f/n, shows empirical values for the space terms of our index. Most of the time we do not know the d'/n since we implemented the $\mathcal{ST}_{78}$ sparse suffix tree as a suffix array. Columns $|\mathcal{T}|$, $|\mathcal{ST}|$ and $f(\log u)/2^{23}$ show the size occupied by these data structures in Megabytes (MB).

performance too much. Table 4.3 and Figure 4.4 show the trade-off obtained using different quorum values in our prototype, for detailed exposition see Appendix A. Table 4.3 shows the size of the ILZI for different quorum values. Our results show that increasing the quorum value significantly reduces the space requirements of the ILZI while degrading the time performance only slightly. Observe that with a quorum of 2 our index has acceptable space requirements, in practice. Our results also show the ILZI has acceptable space requirements in theory. For example the results show that for the English file the practical value is

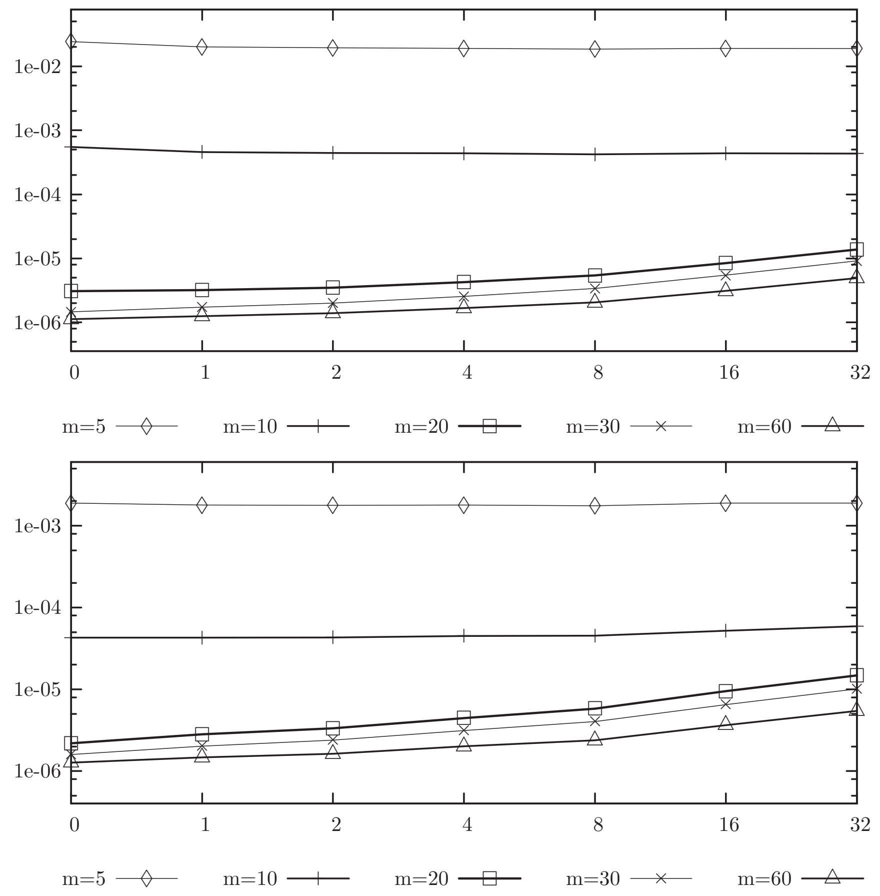


Fig. 4.4. Time performance of the ILZI index for counting. The x-axis represents different quorum values. The results are given in seconds.

$2.99uH_k$ bits with is close to the size estimated for quorum 0, i.e. $((0.64 + 1.33)(1 + \epsilon) + 0.94)uH_k + o(u \log \sigma)$ bits.

4.4.2 Experimental Results

We compared our implementation, Inverted-Lempel-Ziv-Index (ILZI), against the implementations provided in the Pizza&Chili corpus [128][2]. As texts, we used the following files from the Pizza&Chili corpus:

[2] Tested on Pentium 4, 3.2 GHz, 1 MB of L2, 1Gb of RAM, with Fedora Core 3, compiled with `gcc-3.4 -O9`.

- DNA (DNA sequences). This file is a sequence of newline-separated gene DNA sequences (without descriptions, just the bare DNA code) obtained from files 01hgp10 to 21hgp10, plus 0xhgp10 and 0yhgp10, from Gutenberg Project. Each of the 4 bases is coded as an uppercase letter A,G,C,T, and there are a few occurrences of other special characters. Downloaded on June 9, 2005.
- English (English texts). This file is the concatenation of English text files selected from etext02 to etext05 collections of Gutenberg Project. We deleted the headers related to the project so as to leave just the real text. Downloaded on May 4, 2005.

The files were trimmed to 50 Megabytes. The indexes were parametrized to occupy approximately the same space whenever possible. The indexes used were the following:

- Raw is the raw string with one character per byte.
- ILZI is the inverted Lempel-Ziv index described in this Chapter.
- LZI, is the Lempel-Ziv index proposed and implemented by Navarro [98].
- NFMI is an implementation of the FM-index by Navarro [98].
- CSAx8 is an implementation of Sadakane's compressed suffix array [110] by Sadakane.
- LZI-7 is a less space demanding variation of Navarro's Lempel-Ziv index, proposed and implemented by Arroyuelo et al. [6].
- SSA is an implementation of the Succinct Suffix Array [73] by Veli Mäkinen and Rodrigo González.
- AFFMI is an implementation of the alphabet friendly FM-Index [38] by Rodrigo González.
- FMI2 is an implementation of the FM-Index [37] by Paolo Ferragina and Rossano Venturini. This prototype corresponds to version 2.
- SAC is an implementation of the suffix array [79] using $\lceil \log u \rceil$ bits for each entry. Implemented by Veli Mäkinen and Rodrigo González.

		Raw	ILZI	LZI	NFMI	CSAx8	LZI-7	SSA	RL	AFFMI	FMI2	SAC
dna.50MB	$i/2^{23}$	50.0	44.0	60.9	63.4	44.6	54.5	44.3	44.1	43.5	47.1	212.5
	i/uH_k	3.63	3.19	4.42	4.60	3.23	3.95	3.21	3.19	3.16	3.42	15.41
english.50MB	$i/2^{23}$	50.0	54.3	81.1	66.8	56.3	72.3	54.1	52.2	54.6	53.2	212.5
	i/uH_k	2.76	2.99	4.47	3.69	3.11	3.99	2.99	2.88	3.01	2.94	11.73

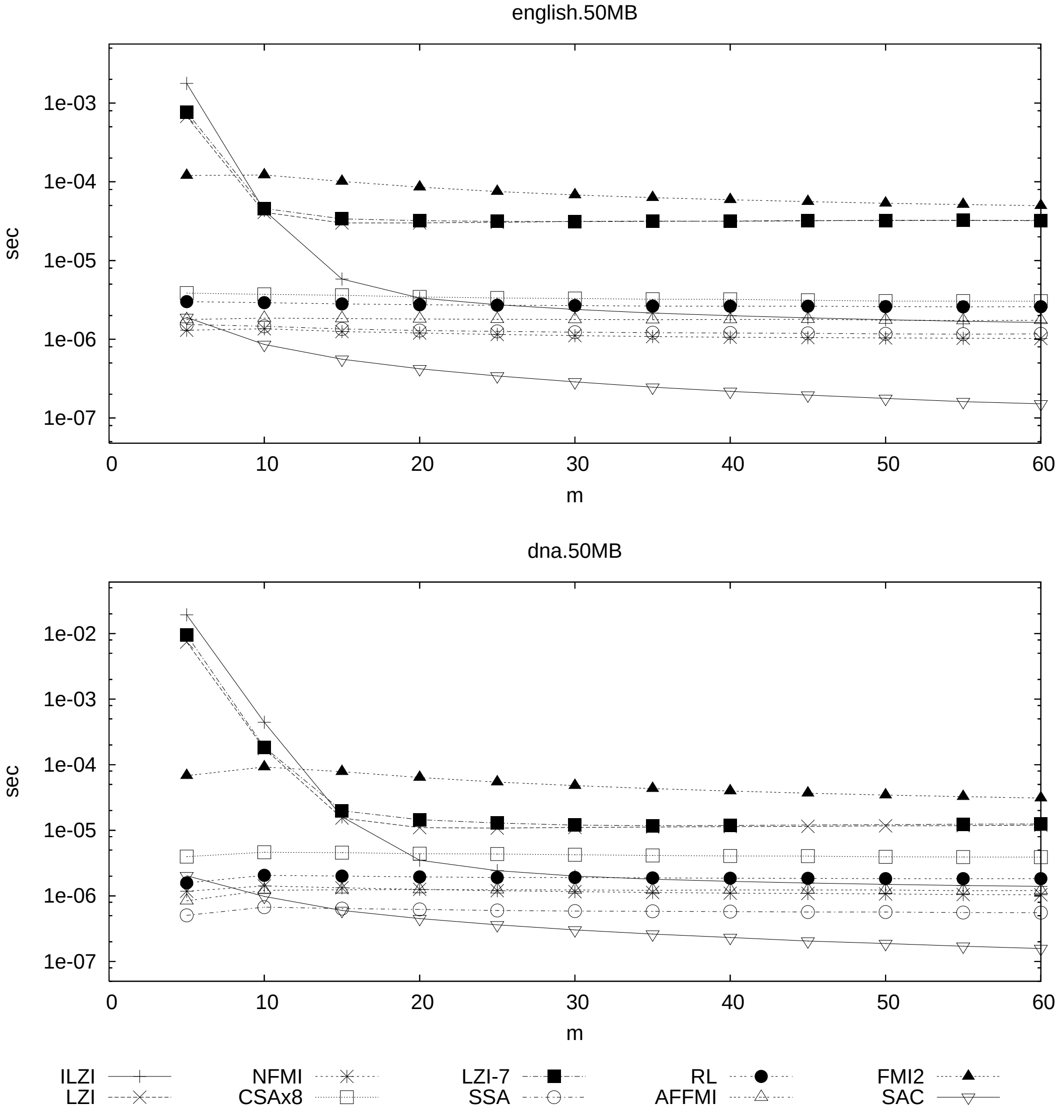
Table 4.4. Table with the size of different compressed indexes for sample files. It shows the space requirements of different indexes, the original string (Raw), the Inverted-LZ-Index (ILZI), Navarro's LZ-index (LZI), Navarro's implementation of the FM-index (NFMI), Sadakane's CSArray (CSAx8), smaller LZ-index (LZI-7), the succinct suffix array (SSA), the run-length FM-index (RL), the alphabet friendly FM-index (AFFMI), the second version of the FM-index (FMI2), SAC is a suffix arrays in uncompressed form, packed in bits. Variable i represents the size of the different indexes in bits. Therefore $i/2^{23}$ gives the size in Megabytes (MB), i/uH_k gives the ratio with a compressed string, where H_k is estimated as $(n \log u)/u$.

In Table 4.4 we show the space requirements of different compressed indexes for the sample files. The indexes were parametrized to occupy roughly the same space but within reasonable performance results.

Figures 4.5,4.6,4.7 show the time performance of different compressed indexes. The performance of compressed indexes can be described as $\Theta(m.C + occ.R + out.O)$, where out is the number of letters that we wish to display, C is the counting factor, R is the reporting factor and O is the outputting factor. For some compressed indexes it is possible to run the indexes in counting mode and the resulting time is $\Theta(m.C)$. However for Lempel-Ziv indexes this is not possible, and our index runs in $\Theta(m.C + occ.R)$ even for counting. To be precise it is not exactly occ. Instead it is something in between $occ_{>1}$ and occ, since some type 1 occurrences can be counted faster in practice. We determined the factors and overall query time for all the indexes.

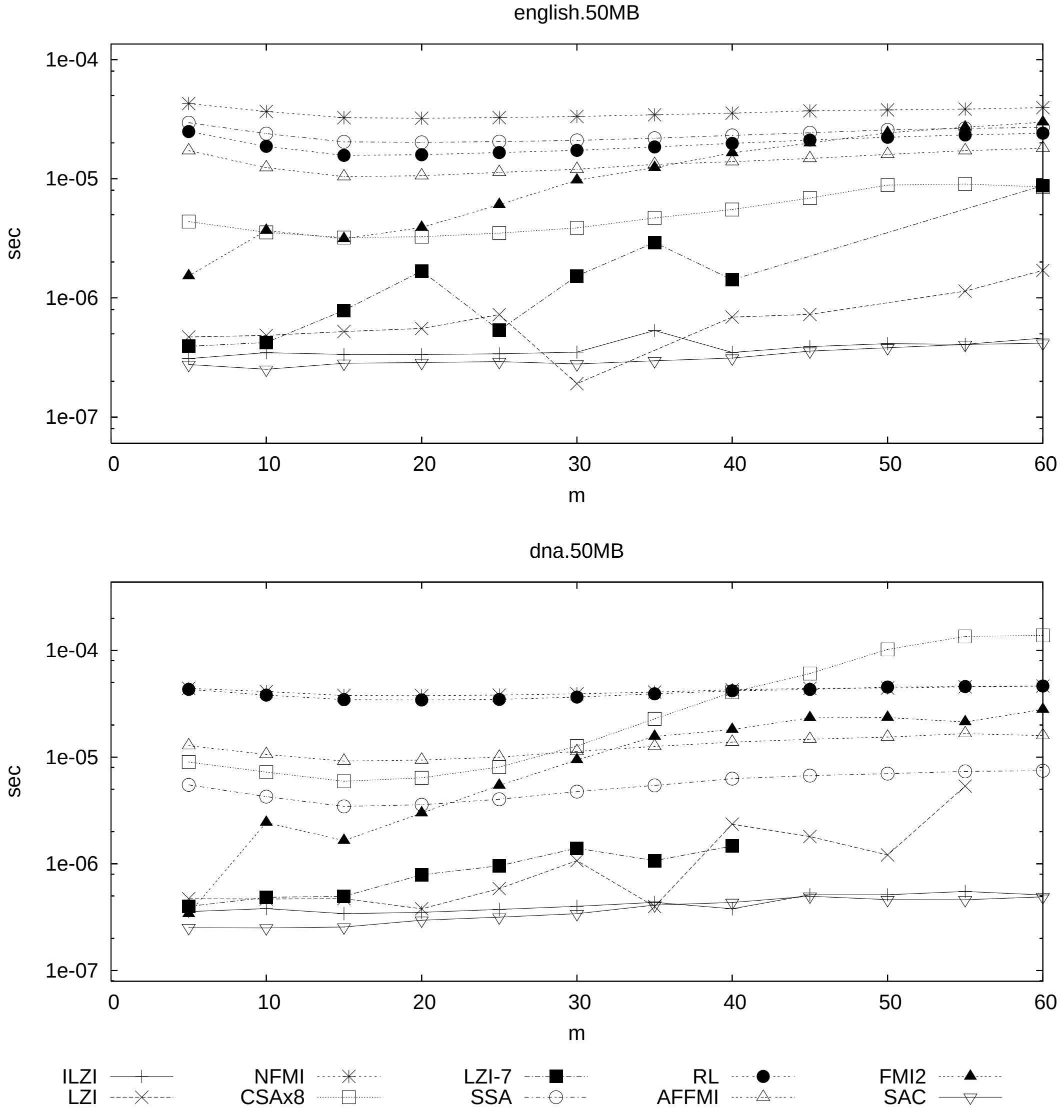
The fact that LZ-based indexes cannot operate in counting mode can be observed empirically since the time of these indexes is not constant in the graphs of Figure 4.5. As expected, when m increases occ decreases and the time also decreases. Eventually, the overall time becomes competitive with other compressed indexes. For most examples this happens when m is around 20. Figure 4.5 also shows that reducing the dependency on m

Fig. 4.5. Time results for counting. These graphs shows the impact of our improvement. This can be observed by comparing the ILZI and LZI indexes. The graphs also show the fact that LZ based indexed cannot count in optimal time. However they do become competitive when m increases, causing occ to decrease.



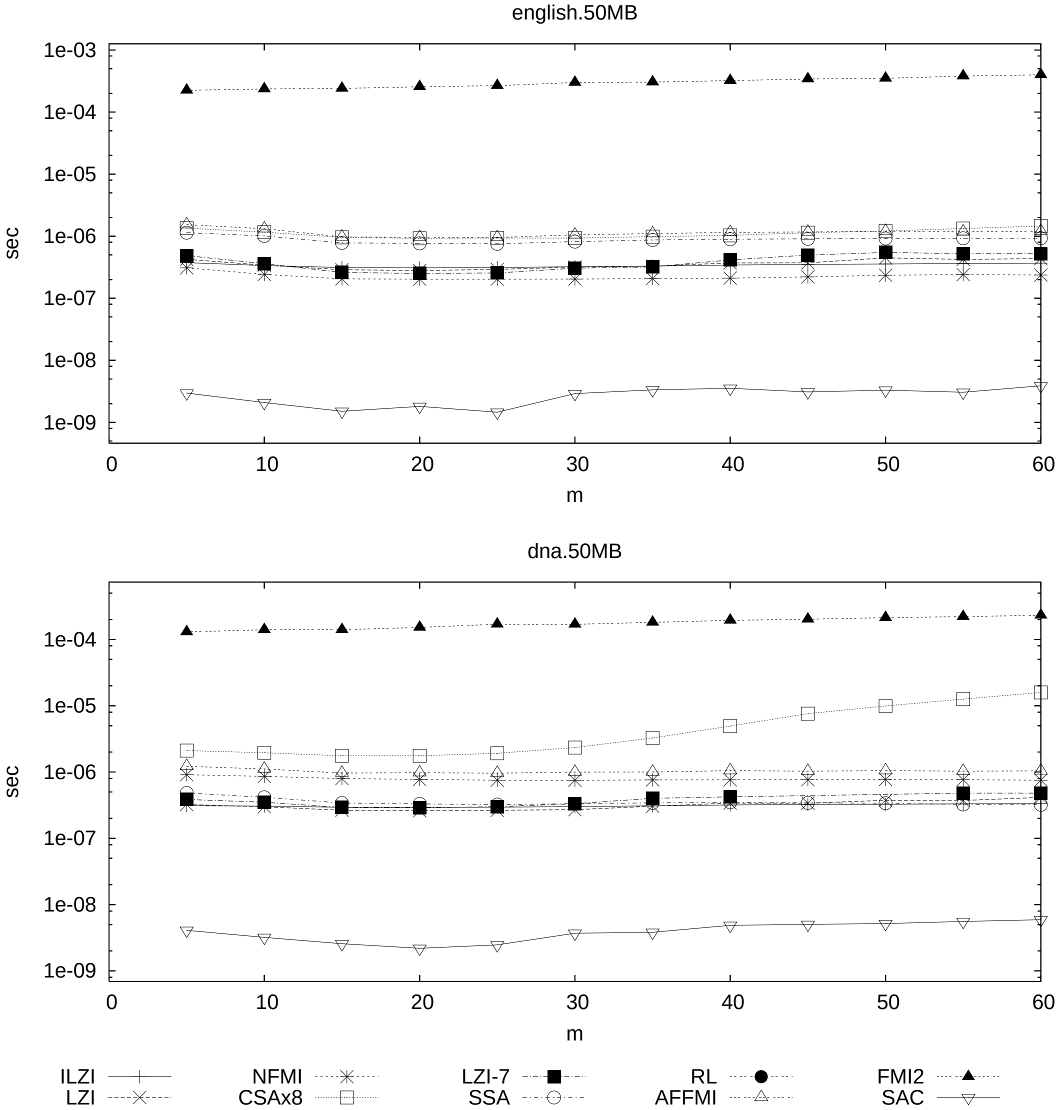
from $O(m^2)$ to $O(m)$ had significant impact in the query time. This makes our index up to an order of magnitude faster than LZI for counting when m is large. On the contrary, for small patterns ($m = 5$) it is up to 2.6 times slower than LZI and up to four orders of magnitude slower than the other compressed indexes.

Fig. 4.6. Time results for reporting factor (R). Theses graphs confirm that in fact LZ based indexes are the fastest at reporting occurrences. These results show that this factor is comparable to that of suffix arrays, being orders of magnitude faster than the alternatives.



On the other hand LZ-based indexes are extremely fast at reporting occurrences. In fact they are the only self-indexes using $O(uH_k)$ bits able to spend $O(\log u)$ time per occurrence. This is also visible, in the graphs of Figure 4.6, since the reporting factor of LZ-based indexes is around an order of magnitude smaller than that of other compressed indexes. Moreover, since, in practice, these indexes do not use range queries, the reporting

Fig. 4.7. Time results for outputting factor (O). These results show that the ILZI is among the fastest compressed indexes at outputting.



time is obtained directly from memory, which means that their real performance is closer to $O(1)$ than to $O(\log n)$. This explains why their performance is so close to that of suffix arrays. This comparison is not completely fair for LZI-7, since it reports positions instead of blocks and hence requires further computation.

The displaying time per character is not a very decisive factor to tell indexes apart since all of them are very fast (see Figure 4.7). The FM-index performed extremely well on natural language based files. The Lempel-Ziv compressed indexes had more stable performance and are among the fastest for all samples. The suffix arrays are around two orders of magnitude faster than the compressed indexes, most likely due to cache effects.

4.5 Conclusions

In this Chapter we presented a new compressed full-text self-index, the ILZI. We began by pointing out our objectives and motivation as clearly as possible, we then gave a detailed description of this index containing both theory and practice.

From a theoretical point of view we made several important contributions. First we observed that our dictionary, the $\mathcal{T}_{78}$ tree, is a suffix tree. This structure was first presented by Kärkkäinen [59], but his version required T to be present and since it was based on LZ77, it was not necessarily a suffix tree. In the work presented by Navarro [98] the structure is called RevTrie, but its suffix tree nature is not explored and, in fact, reading an edge-label requires $O(m^2)$ time. In the work presented by Ferragina and Manzini [37] it appears as an FM-Index of $T_{\$}^{R}$. They prove that its space requirements can be related to the entropy of the text T. However its suffix tree structure is also not explored. Second we observed that in the LZ78 the same string S may appear in $O(m)$ different ways as the concatenation of LZ78 blocks. This, in turn, forces algorithms based on the LZ78 parsing to have quadratic behavior. We solved this problem by discarding the original parsing and using a maximal parsing. In the maximal parsing, a string S appears in at most one way as the concatenation of blocks. Navarro uses the original LZ78 parsing. Ferragina and Manzini discard the parsing and solve the problem by using an FM-index, i.e. resorting to the Burrows-Wheeler transformation.

Our index is a significant contribution to LZ-based compressed indexes. We improved the counting time performance of LZ-based indexes to linear time on m, without resorting to other compressed indexes. At the same time, the structure we propose is smaller than

LZI, for all the files we tested. In theory, with the terms we obtained in Table 4.3, we can choose an ϵ to make the index smaller than $4uH_k + o(u \log \sigma)$. In practice it can be seen in Table 4.3 that ILZI is always smaller than LZI. However a new version of the LZ-index proposed by Arroyuelo et al. [6] requires only $(2+\epsilon)uH_k + o(u \log \sigma)$ with worst case guarantees. Without worst case guarantees it requires $(1+\epsilon)uH_k + o(u \log \sigma)$ bits and it has $O(m^2)$ average search time for $m \geq 2 \log_\sigma u$. It is interesting to notice that Arroyuelo et al. independently explored the suffix tree structure of $\mathcal{T}_{78}$ to reduce the time to read an edge-label to $O(m)$. We cannot achieve the reduced space requirements of Arroyuelo et al., essentially because we are storing more structures.

Another important theoretical contribution was a succinct representation of suffix trees (Lemma 4.2). This representation is not very competitive when compared to the compressed suffix trees presented by Sadakane [109]. Nevertheless, it is adequate for our goals. For suffix trees, in general, it requires more space than the representation of Sadakane. In fact, the problem is the space required to store R and R^{-1}, $(1+\epsilon)n \log n$ bits. Arroyuelo et al. [6] showed how to reduce the space requirements of R. However, even with such an improvement, it is still not comparable to Sadakane's approach in terms of space.

We also presented some important practical observations. The notion of spurious entries, although introduced for theoretical reasons, played an important role in reducing the space requirements of the ILZI, which became considerably smaller than the LZI (see Table 4.4). The most evident consequence of this fact can be seen in Table 4.3, which shows the theoretical values of d/n. Observe that, in theory, ignoring the effect of spurious entries, we should have that $1 \leq d/n \leq 2$, since this expression evaluates the ratio of nodes that $\mathcal{T}_{78}$ (d) should have when compared to the number of nodes in the LZ78 trie (n). However, all the values presented in this table are smaller than 1. This improvement is a consequence of removing spurious entries.

Another very important practical contribution was the notion of LZ78 parsing with quorum l. This allowed us to reduce the overall space requirements of the ILZI. By adjusting the value of l we can balance between the size of $\mathcal{T}_{78}$ and the size of $\mathcal{ST}_{78}$. The improvement this notion provided in terms of space was crucial and, moreover, the performance was degraded only slightly. This can be observed in Table 4.3 and Figure 4.4. We only show the impact of this notion in the counting performance since the reporting (R) and outputting (O) times are unaffected by this notion. A careful look at Table 4.3 shows an impressive improvement in space from quorum 0 to quorum 1, that becomes progressively less significant for increasing quorum values. The penalty in performance is not very severe. For small values of m, i.e. $m = 5$ or $m = 10$, it is practically zero. This is important since these are the values for which the ILZI performs worst and as we mentioned before for these values it actually is less efficient than the LZI.

5

Finding Longest Common Sub-Strings

In this chapter we explain how to use the Inverted-Lempel-Ziv-Index to solve the longest common substring problem, in linear time.

A longest common substring between string P and string T is one largest string that occurs both in P and T [1]. Determining a longest common substring is relevant in many DNA applications, were a given pattern P is compared against an already existing database T. Using a suffix tree for T, this problem can be solved in $O(m)$ time, for a pattern of size m. Suffix trees, however, require a considerable amount of space. Compressed indexes can be used to store T and require less space than suffix trees. However not all the functionality of suffix trees is supported by all compressed indexes. Moreover, some fully functional representation of suffix trees do not have optimal space requirements. In this chapter we extend the functionality of the Inverted Lempel-Ziv Index, to solve the longest common substring problem. Our algorithm runs in $O((m/\epsilon)\log^2 u)$ time requiring, $O(uH_k) + 5\epsilon u + o(u \log \sigma)$ bits, for any $0 < \epsilon \leq 1$.

The longest substring problem consists in searching a large text T for a longest substring of a given pattern string P. Usually T is known a priori and can be preprocessed. The search procedure starts when a pattern string P is given. This is the scenario when searching, for instance, DNA databases [82]. In essence, a large amount of DNA data is compiled into a database, and, when a new DNA string is obtained, it is useful to uncover its relations

[1] This string is not necessarily unique.

with the existing DNA. One important such relation, that gives a similarity measure, is a longest common substring.

A number of interesting applications for longest common substrings are known [52]. We describe two of these to illustrate the importance of the problem. The first application is the identification of organic samples. The idea is to use a database of human Mitochondrial DNA. This kind of DNA can be reliably identified by polymerase chain reaction. Moreover these strings are highly variable and can be used as a "nearly unique" identification of a person. The longest common substring can be used to identify to whom the DNA fragments of a given sample of blood or hair belong to.

Another application is related to the DNA contamination problem. Most of the processes used to manipulate DNA may contaminate a DNA sample with external DNA. This is a very serious problem that can lead to wasted sequencing. To solve this problem we can store a database of the DNA that is likely to contaminate an experience. A sequenced sample is then compared against this database. If there are common substrings, between a piece of sequenced DNA and the database, larger than a given threshold, then the sample is probably contaminated. In this example it may not be crucial to determine what caused the contamination and it may be enough to determine that it is contaminated.

5.1 Related Work

The classical solution for the longest common substring problem consists in preprocessing T into a suffix tree [52]. The problem with suffix trees is that they are very space demanding. We propose to solve this problem using the ILZI.

Many compressed indexes do not have full suffix tree functionality. In particular, they are only capable of solving the longest common substring problem in a naive way, by reducing it to the exact matching problem, which gives an $O(m^2)$ time complexity. A considerable amount of research has been done to extend the functionally of these structures, mainly by extending the functionality of compressed suffix arrays by adding a parentheses representation of the shape of the corresponding suffix tree [50, 51, 62, 87, 111, 112]. Of these representations, only the one given by Sadakane using the compressed suffix array of Grossi, Gupta and Vitter [50] is dependent on H_k. Using other compressed suffix arrays either depends on H_0 or it requires $O(u \log \sigma)$ bits. We propose, as far as we know, the first approach that solves the longest common substring problem with linear time complexity on m using a compressed index.

Our algorithm works as a sort of sliding window over P. The window starts out empty and is increased as we find larger common substrings between P and T. Once the window size increases it never decreases again. For every substring of P inside the window, we verify if that string occurs in T. If it does, we increase the window. Otherwise, we shift the window. In the end, the size of this window corresponds to the size of a longest common substring. This is essentially how the classical algorithm over suffix trees works. However, we are able to support this procedure over the inverted Lempel-Ziv index, even though this index does not have full suffix tree functionality.

5.2 Basic Concepts and Notation

For a description of basic concepts, the Inverted-Lempel-Ziv-Index and notation please refer to the previous chapters.

Definition 5.1. *A **longest common substring** is one largest string that is a substring of P and T, denoted by* LCSS(P,T).

As a running example we shall consider, again, string $T = cbdbddcbababa$ and $\mathcal{T}_{78}$ as the suffix tree in Figure 5.2 (top-right). This figure is similar to Figure 4.1 and is repeated here for convenience. For $P = cbddcbabd$, the longest common substring is *bddcbab*. Figure 5.1 gives a schematic representation of this solution.

```
               1
   012 3456789 012
T: cbd.bddcbab.aba
       |||||||
P:    cbddcbabd
```

Fig. 5.1. Schematic representation of the solution of the longest common substring problem for $P = cbddcbabd$.

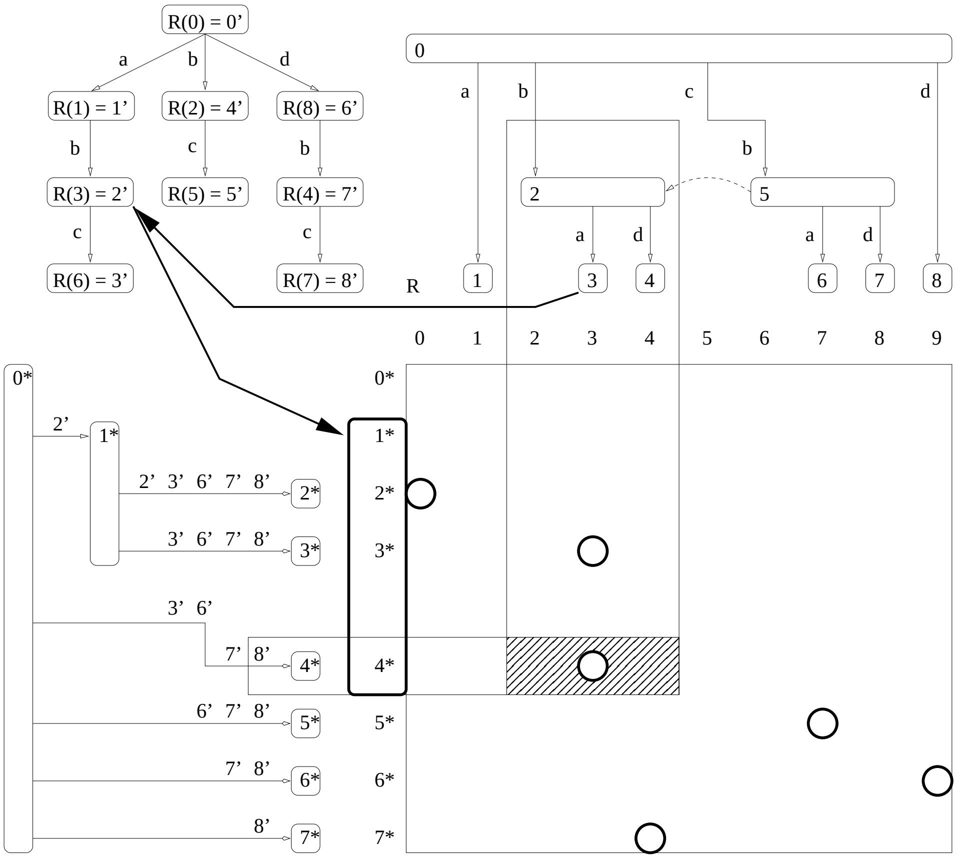


Fig. 5.2. (top-right) Suffix tree $\mathcal{T}_{78}$. (top-left) Reverse tree $\mathcal{T}^R_{78}$. (bottom-left), Sparse suffix tree $\mathcal{ST}_{78}$. (bottom-right) Linking points and orthogonal range query [4*,4*]:[2,4].

We need to add a new operation to the ILZI, the $\textsc{SdepAnc}(v, i)$, that returns the ancestor of node v that is at string-depth i. This can be computed using the $\textsc{Anc}(v, j)$ and $\textsc{Sdep}(v)$ operations. The idea is to perform a binary search in the elements $\textsc{Anc}(v, j)$, i.e. a binary search over the $\textsc{Sdep}(\textsc{Anc}(v, j))$ values. This requires $O(\log |v|)$ time.

Recall that d and d' are the number of nodes of $\mathcal{T}_{78}$ and $\mathcal{ST}_{78}$ respectively, that t is the number of points of $\mathcal{T}_{78}$ and that f is the size of the $\mathcal{T}_{78}$-maximal parsing of T.

5.3 Computing a Longest Common Sub-String

We now explain how to use the ILZI to solve the longest common substring problem. Similarly to what happens in the exact matching problem, the algorithm runs in two phases. In the first phase we find a longest substring of P that occurs inside a block of $\mathcal{T}_{78}(T)$. In the second phase, we find a longest substring of P that occurs as a sequence of blocks and later as a substring of a sequence of blocks.

5.3.1 LCSS Inside a Single Block

We start by explaining how to determine a longest common substring of P that is contained inside a block of $\mathcal{T}_{78}(T)$. This is essentially the classical solution to the problem, i.e. compute the descend and suffix walk of P in $\mathcal{T}_{78}$ and report a point with maximal string-depth. This string-depth is denoted by $lcss_1$. Assuming that $\mathcal{T}_{78}$ has no spurious entries (recall that those can be removed, Section 4.2.3), $lcss_1$ can be determined in $O((m/\epsilon)\log\sigma)$ time. In our example the resulting LCSS would be string *cbd* or *cba* (see $trace\_right[0]$, and $trace\_right[4]$ in Table 5.1, top).

5.3.2 Aligned Longest Common Sub-String

In order to explain the computation of the LCSS of P that spans more than a single block, we start by introducing the concept of sparse descend and suffix walk. To solve the LCSS problem, we would like to compute the descend and suffix walk of P in the suffix tree of T. However we do not have the suffix tree of T. Instead, we have a sparse suffix tree $\mathcal{ST}_{78}$. We will start by solving the longest common substring in the sparse suffix tree $\mathcal{ST}_{78}$. This will provide some insight and tools to find the LCSS. This motivates the following problem:

Definition 5.2. *The* ***aligned longest common substring problem*** *for the $\mathcal{T}_{78}$-maximal parsing of T consists in finding a longest string that appears as a concatenation of blocks in $\mathcal{T}_{78}(T)$ and is a substring of P, i.e. $P[i..j] = v_{i'} \dots v_{j'}$.*

This restriction of the longest common substring problem allows us to introduce the solution without paying attention to the blocks at the extremes of the substring. The solution for this problem is essentially the same as performing a descend and suffix walk of P in $\mathcal{ST}_{78}$ and reporting the lowest point. Recall that $\mathcal{ST}_{78}$ is, in a way, indexing T^R, so we must read P backwards. First, we must translate the pattern P, since $\mathcal{ST}_{78}$ indexes $R(\mathcal{T}_{78}(T))$. This must be done in a maximal way. For this we use the *trace_right* of the descend and suffix walk of P in $\mathcal{T}_{78}$. Algorithm 5 shows how to compute this procedure. We will refer to Algorithm 5 as the sparse descend and suffix walk of P. Note that for the sparse descend and suffix walk of P, we assume that $\textsc{Suffix\_Link}_{\mathcal{ST}_{78}}(\textsc{Root}_{\mathcal{ST}_{78}})$ does not exist. Therefore if the sparse descend and suffix walk reaches this point, it should return the $udef$ state.

The aligned longest common substring problem can be solved by computing the sparse descend and suffix walk of P in $\mathcal{ST}_{78}$ and outputting the lowest node, i.e. the one with the largest string-depth. In our example, that point corresponds to string *bddcba*, as shown in Table 5.1 (bottom).

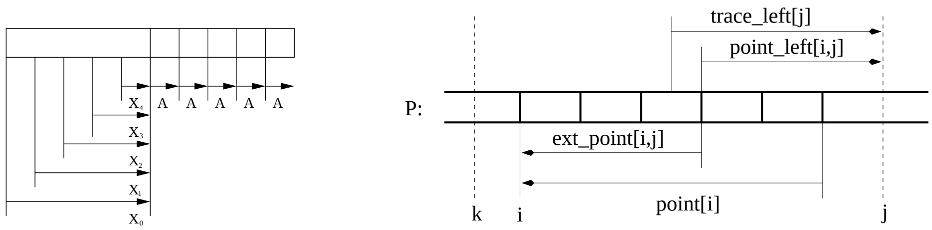
The fundamental problem with Algorithm 5 is that it can require $O(m^2)$ time. The quadratic behavior occurs in the following example: suppose $\mathcal{ST}_{78}$ is the suffix tree of string A^n. Then, $trace\_right[i] = A$ for $m/2 < i < m$ and $trace\_right[i] = X_i$, for $0 \leq i \leq m/2$, where $|A| = 1$ and $|X_i| = m/2 - i + 1$ (see fig. 5.3(a)). In this example, the **while** cycle in the sparse descend and suffix walk runs $O(m/2)$ times for $O(m/2)$ cases, giving a total of $O(m^2/4)$ operations.

The solution for this problem consists in observing that the **while** cycle in the sparse descend and suffix walk is a naive scan over a list of elements to find the first one with a given property. The property is that $\textsc{Descend}?(\text{point}[i], \text{father\_right}[i])$ is true. Replacing

Algorithm 5 Sparse Descend and Suffix Walk

```
procedure Sparse_Descend_and_Suffix(P)
    point[|P|] ← ROOT
    for i ← |P| − 1, i ≥ 0 do
        point[i] ← point[i + |father_right[i]|]
        while NOT DESCEND?(point[i], father_right[i]) do
            point[i] ← SUFFIX_LINK(point[i])
        end while
        i − −
    end for
end procedure
```

this search by a binary search will allow us to compute the sparse descend and suffix walk of P in $\mathcal{ST}_{78}$ without a quadratic dependency on m. Let us start by solving the problem



(a) (b)

Fig. 5.3. (left) Pattern with $O(m^2)$ sparse descend and suffix walk. (right) Schematic representation of $ext\_point[i,j]$ and $point\_left[j]$.

when $point[i]$ is a node of $\mathcal{ST}_{78}$. A chain of suffix links across nodes corresponds to part of a branch of $\mathcal{ST}_{78}^R$. In fact, following a suffix link from a node, in $\mathcal{ST}_{78}$, corresponds in $\mathcal{ST}_{78}^R$ to moving up to the father node. This is the way $\text{SUFFIX\_LINK}(v)$ is computed, i.e. $\text{SUFFIX\_LINK}(v) = R^{-1}(\text{ANC}_{\mathcal{ST}_{78}^R}(R(v), 1))$. In order to do a binary search, all that we need is to be able to access an arbitrary ancestor of a node of $\mathcal{ST}_{78}^R$ in constant time, i.e. $\text{ANC}_{\mathcal{ST}_{78}^R}(R(v), j)$. This is already possible in the representation of suffix trees. Therefore, by replacing the while cycle in Algorithm 5 by a binary search, the algorithm runs in $O((m/\epsilon)(\log m) \log n)$ time when $point[i]$ is a node.

i	0	1	2	3	4	5	6	7	8	9
$P[i]$	c	b	d	d	c	b	a	b	d	$\$'$
$trace\_left[i]$	ϵ	c	cb	cbd	d	c	cb	cba	b	bd
$\text{DFS}'(father\_left[i])$	0	0	6	8	9	0	6	7	2	4
$\text{DFS}'(trace\_left[i])$	0	5	6	8	9	5	6	7	2	4
$\text{DFS}'(child\_left[i])$	0	6	6	8	9	6	6	7	2	4
$trace\_right[i]$	cbd	bd	d	d	cba	ba	a	bd	d	ϵ
$\text{DFS}'(father\_right[i])$	8	4	9	9	7	3	1	4	9	0
$\text{DFS}'(trace\_right[i])$	8	4	9	9	7	3	1	4	9	0
$\text{DFS}'(child\_right[i])$	8	4	9	9	7	3	1	4	9	0
$P[i..]$	$cbd.d.cba.bd$	$bd.d.cba.bd$	$d.d.cba.bd$	$d.cba.bd$	$cba.bd$	$ba.bd$	$a.bd$	bd	d	ϵ
$V^{-1}R^{-1}(point[i])$	cbd	$bd.d.cba$	d	$d.cba$	cba	ba	$udef$	bd	d	ϵ
$point[i]$	$8'$	$3'6'7'$	$6'$	$3'6'$	$3'$	$2'$	$udef$	$7'$	$6'$	ϵ

Table 5.1. (Top) Descend and suffix walk of *cbddcbabd* in $\mathcal{T}_{78}$. (Bottom) Sparse descend and suffix walk of *cbddcbabd* in $\mathcal{ST}_{78}$.

When $point[i]$ is not a node and the DESCEND?(point[i], trace-right[i]) is false, then it will remain false until $point[i]$ becomes a node. Hence there is no need to test the DESCEND? predicate for the in-between points. Therefore we can skip the in-between points. A point is represented by storing its father node, $father\_point[i]$, its string-depth and its child node $child\_point[i]$. Suppose we wish to determine if SUFFIX_LINK(point[i]) is a node or a point. One way to compute this result is to determine if there is an ancestor of SUFFIX_LINK($child\_point[i]$) at the string-depth $\text{SDEP}(point[i])-1$. This can be computed by using the *SdepAnc* operation and verifying whether the resulting node has the desired string-depth. The *SdepAnc* operation over the $\mathcal{ST}_{78}$ tree takes $O(\log f)$ time, since it is a binary search. Therefore, verifying whether there is an ancestor of $child\_point[i]$ at a given string-depth takes $O(\log n)$ time. Hence, computing the first node in the trail of suffix links of $point[i]$ can be done in $O(((\log m)/\epsilon)\log f)$ time, using a binary search.

Overall, the sparse descend and suffix walk takes $O((m/\epsilon)(\log m)\log u)$ time, including the $O((m/\epsilon)\log\sigma)$ time to compute the descend and suffix walk of P in $\mathcal{T}_{78}$. We assume that we are able to compute the size of the substring found by using an operation **represented string-depth** and denoted by $||.||$. For example the represented string depth of $3'6'7'$ is 6 since it corresponds to the size of string *abc.d.bd*. We will now make a slight detour to explain, in detail, how to compute represented string depth.

5.3.3 Represented String Depth

In this subsection we explain how to compute the represented string depth. The problem is that the string-depth in $\mathcal{ST}_{78}$ does not correspond to the size of the substring of P that it may represent. For example, for point $3'6'7'$, its string-depth is 3, i.e. $|3'6'7'| = 3$. The size of the string that is translated into $3'6'7'$ is different. Observe that $R(V(bd, d, cba)) = 3'6'7'$ and $|bddcba^R| = 6$. This value will be denominated by **represented string-depth** and is denoted as $|V^{-1}(R^{-1}(3'6'7'))|$, or $||3'6'7'||$.

This value can be computed in two ways: with the V bitmap (see the display portion of the proof of Theorem 4.16); or by preprocessing a dependency tree;

The first solution consists in using Select_1 over V. In our example, we have that $V = 100.10.1.100.10.10$. We start by determining a leaf that is a descendant of $p = 3'6'7'$. In this case this is trivial since 4^*, the first node below p, is already a leaf. In general, we obtain this result as $v' = \text{RightRank}_{\mathcal{ST}_{78}}(v)$, where v is the first node below p. We now compute $\text{Sdep}_{\mathcal{ST}_{78}}(v')$. In our example we have that $\text{Sdep}_{\mathcal{ST}_{78}}(4^*) = 4$. The represented string-depth can now be computed as $\text{Select}_1(V, 4+1) - \text{Select}_1(V, 4+1-|3'6'7'|) = \text{Select}_1(V, 5) - \text{Select}_1(V, 2) = 9 - 3 = 6$. The general expression is therefore $\text{Select}_1(V, 1 + \text{Sdep}(v')) - \text{Select}_1(V, 1 + \text{Sdep}(v') - |p|)$. This requires only $O(1)$ time.

The problem with this bitmap is that it adds $u + o(u)$ bits to the index. A compressed version requires only $o(u \log \sigma)$ bits, as noted in Lemma 4.15.

We will now give a version that is less space demanding and, moreover, provides some insight into the sparse descend and suffix walk. We give a solution that does not increase the index size, but requires more space at query time. Still, the query space remains $O(m)$

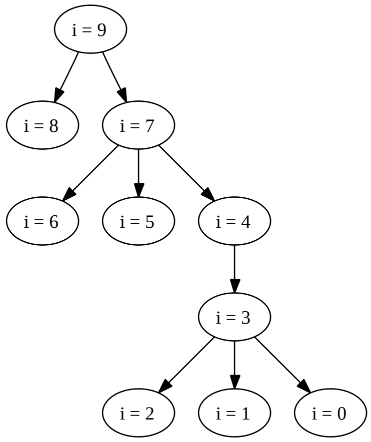


Fig. 5.4. Dependency tree

computer words, i.e. $O(m \log m)$ bits. Our solution does not allow us to compute $||p||$ for every point in $\mathcal{ST}_{78}$. However, we are able to compute it for all necessary points, i.e. prefixes of $point[i]$. The idea is to associate to the descend and suffix walk of P, in $\mathcal{T}_{78}$, a dependency tree. The dependency tree shows which $father\_right[i]$ can be concatenated to give substrings of P.

Definition 5.3. *The* ***dependency tree*** *of the descend and suffix walk of P in $\mathcal{T}_{78}$ is a tree with $m+1$ nodes, labeled by $0, \ldots, m$, such that for every i, node $i + |father\_right[i]|$ is a child of node i.*

The dependency tree for our example is shown in Figure 5.3.3. The dependency tree represents the concatenations of blocks that build substrings of P. These concatenations correspond to portions of branches in the tree. That doesn't necessarily mean that all those configurations must appear in T. In fact, the concatenations that do appear in the text are given by the sparse descend and suffix walk. For example, consider the branch from node 1 to the root. This branch consists of nodes $1, 3, 4, 7, 9$, that have the following $father\_right[i]$ strings bd, d, cba, bd, ϵ. Concatenating these blocks gives a suffix of pattern P. However, the string $bd.d.cba.bd$ does not occur in T. This can be seen because the corresponding point in $\mathcal{ST}_{78}$ does not exist, i.e. point $7'3'6'7' = R(4, 8, 6, 4) = R(V(bd, d, cba, bd))$

does not exist in $\mathcal{ST}_{78}$. In fact $point[1] = 3'6'7'$, which represents that $bd.d.cba$ is a substring of T but $bd.d.cba.bd$ is not. To determine the represented string-depth of $point[1]$ all we need to do is to observe that it corresponds to the branch $1, 3, 4, 7$. Hence, the value we want can be obtained from the first and last nodes of this branch as $|bd.d.cba| = 6 = 7 - 1$. Observe that node "$i = 7$" is the third ancestor of node "$i = 1$" in the dependency tree. Therefore the represented string-depth can be computed as $|V^{-1}(R^{-1}(point[i]))| = \textsc{Anc}(i, \textsc{Sdep}(point[i])) - i$, were $\textsc{Anc}(i, j)$ gives the j-th ancestor of node i in the dependency tree. In our example this is $|bd.d.cba| = |V^{-1}(R^{-1}(point[1]))| = \textsc{Anc}(1, \textsc{Sdep}(point[1])) - 1 = \textsc{Anc}(1, 3) - 1 = 7 - 1 = 6$. The operation $\textsc{Anc}$ can be supported in $O(1)$ time by preprocessing the dependency tree in $O(m)$ time, using the algorithms given by Dietz [31], Berkman and Vishkin [15] and Bender and Farach-Colton [13].

5.3.4 LCSS Spanning More than a Single Block

We have now completely solved the aligned longest common substring problem. In our example, the aligned longest common substring is $bd.d.cba$. The size of an aligned longest common substring is denoted by $lcss_{>1}$. We will now be concerned with a longest substring of P that appears in T spanning across more than one block. The solution consists in trying to extend the aligned substrings found by the previous algorithm. The idea is to try to add a block in the left and one block in the right and finding a maximal substring.

Adding a block to the right of an aligned longest common substring turns out to be trickier than expected. The problem is that the block in the end may extend over some blocks that already existed in the end, i.e. simply concatenating a block will not solve the problem. Consider the example where node cbd of $\mathcal{T}_{78}$ is replaced by $cbde$ and nodes bde, de, e are added to complete the suffix tree. Note that cbd is no longer a node, it is only a point. Consider that $P = \underline{b.cb.d.d.d}$ and $T = b.cb.d.d.\underline{b.cbddd}e$. The aligned longest common substring is $b.cb.d.d$. However the real longest common substring is $b.cbddd$ and this string cannot be obtained by concatenating a block in front of $b.cb.d.d$. Instead in this case the second block must be extended. The information of how many letters, in the end of P, can be potentially joined into one block is obtained from $trace\_left[i]$.

The algorithm we propose consists in successively extending the longest common substring known so far. This value is denoted by l. We start with $l = \max(lcss_1, lcss_{>1})$. Then, we try to extend every substring of P with size l. When we find a substring of P with size $l' > l$, we update the value of l to l' and the rest of the search uses this new value. This search would require $O(m)$ verifications if the substrings of P always appeared with the same configuration in $\mathcal{T}_{78}(T)$. However, this is not the case.

Let us start by explaining the solution for the blocks in the right. Suppose we choose two indexes i and j. We must start by finding the smallest suffix[2], $ext\_point[i,j]$, of $point[i]$ that can be extended by $trace\_left[j]$ until position j (see Figure 5.3(b)). This can be computed in $O((\log n)/\epsilon)$ time by doing a binary search using the $\text{Anc}_{\mathcal{ST}_{78}^R}$ operation in $\mathcal{ST}_{78}^R$, between $R(child\_point[i])$ and $\text{Root}_{\mathcal{ST}_{78}}$, searching for the first point p in $\mathcal{ST}_{78}^R$ such that $||R(child\_point[i])|| - ||p|| \leq |trace\_left[j]|$.

Alternatively $ext\_point[i,j]$ can be computed in $O(\log m)$ time with the dependency tree. The idea is to do a binary search in the dependency tree using the *Anc* operation, searching for the first ancestor i' such that $i' - i + trace\_left[j] >= j$

Observe (see Figure 5.3(b)) that $trace\_left[j]$ and $ext\_point[i,j]$ may overlap. Therefore $V^{-1}(R^{-1}(ext\_point[i,j])).trace\_left[j]$ may be different from $P[i..j]$. To solve this problem we use $point\_left[i,j]$, that can be obtained as an ancestor point of $trace\_right[i + ||ext - point[i,j]|| + 1]$, i.e. $point\_left[i,j] = \text{SdepAnc}(trace\_right[i + ||ext - point[i,j]|||], j + 1 - (i + ||ext - point[i,j]|||))$. This requires $O(\log m)$ time. Overall, for every j, the time to compute the $ext\_point[i,j]$ and $point\_left[i,j]$ is $O(\log m)$.

The problem of adding a block at the left consists basically in extending the case in Figure 5.3(b) by adding an incomplete block before position i. In other words we wish to search for string $P[k..j]$, (see Figure 5.3(b)). A node in $\mathcal{T}_{78}$ representing string $P[k..i]$

[2] Recall that $point[i]$ represents a string oriented from left to right.

can be obtained from $trace\_right[k]$ by using the SdepAnc operation, i.e. $P[k..i-1] =$ SdepAnc$(father_right[k], i-k)$. This requires $O(\log m)$ time. The information from $P[k..i]$ and $P[i..i+||ext\_point[i,j]||]$ can be combined with the weak descent operation.

We now have all the pieces to search for string $P[k..j]$ assuming that the configuration is determined by point i. We know that $P[k..j] = P[k..i].P[i+1..i+||ext-point[i,j]||].P[i+||ext-point[i,j]||+1..j]$ and that $P[k..i] =$ SdepAnc$(father_right[k], i-k)$, that $P[i..i+||ext\_point[i,j]||] = V^{-1}(R^{-1}(ext\_point[i,j]))$ and that $P[i+||ext\_point[i,j]||+1..j] = point\_left[i,j]$. Therefore checking whether string $P[k..j]$ occurs in T with a configuration determined by i can be performed with the following range query: $W(ext\_point[i,j], R(\text{Dfs}(P[k..i]))) : I(point\_left[i,j])$

This query is used only to determine whether there are points in this range and not to output them. The result is denoted by Check(P,k,i,j) and can be computed in $O(\log u)$ time, including the time to compute all necessary values and points and assuming that the sparse descend and suffix walk is already computed. For example Check$(P,1,1,7)$ will originate the range query $W(3'6'7',0') : I(b) = [4^*,4^*] : [2,4]$ that does find a point and corresponds to the LCSS of our example, i.e. *bd.d.cba.b*.

5.3.5 Space and Time Complexity

Algorithm 6 shows our search procedure. Let us now study its complexity. First, observe that the internal **while** guard evaluates to true at most $|LCSS(T,P)|$ times and that $|LCSS(T,P)| < m$. Therefore, the Check$(P,k,i,k+l+1)$ operations that return true require at most $O(m \log n)$ time. The Check operations that return false are actually the most time consuming ones. Observe that the number of times that the external **while** cycle executes depends on $|father\_left[i-1]|$ and that $|father\_left[i-1]| \leq lcss_1$. Therefore the number of times that the Check$(P,k,i,k+l+1)$ operation returns false is at most $O(m.lcss_1)$. Hence, overall, the external **for** cycle requires $O(m.lcss_1 \log u)$ time. Adding

Algorithm 6 LCSS search

```
procedure LCSS(P)
    DESCEND_AND_SUFFIX(P)
    SPARSE_DESCEND_AND_SUFFIX(P)
    l ← max(lcss_1, lcss_{>1})
    for i ← |P|, i ≥ 0 do
        k ← i
        while |father_left[i − 1]| ≤ i − k do
            while CHECK(P, k, i, k + l + 1) do
                l + +
            end while
            k − −
        end while
    end for
    return l
end procedure
```

up the time of the descend and suffix walk and the sparse descend and suffix walk we have that our algorithm takes $O(m(lcss_1 + (\log m)/\epsilon) \log u)$ time.

Our final concern has to do with the value of $lcss_1$. Observe that $lcss_1$ must be smaller than the largest block in the LZ78 parsing and therefore this value is not hiding an $O(m)$ time dependency. Even so it would be preferable to have a bound on $lcss_1$. We will achieve this by preventing the Lempel-Ziv blocks from growing beyond $(\log u)/\epsilon$.

Definition 5.4. *The ϵ **bounded LZ78 parsing** of a string T and $0 < \epsilon \leq 1$ is the sequence $Z_1, \ldots, Z_n$ of strings such that $T = Z_1 \ldots Z_n$. For every i, Z_j is the largest prefix of $Z_i \ldots Z_n$ among the $Z_1, \ldots, Z_{i-1}$ and $Z_i = Z_j c$ if $|Z_j| < (\log u)/\epsilon$ or $Z_i = Z_j$ otherwise.*

We can use this parsing, instead of the original LZ78, to generate the Lempel-Ziv suffix tree $\mathcal{T}_{78}$. With this variation our algorithm requires $O((m/\epsilon) \log^2 u)$ time, since $lcss_1 < (\log u)/\epsilon$.

Theorem 5.5. *Assume that we use the ϵ bounded LZ78 parsing of string T. Let d and d' be the number of nodes of $\mathcal{T}_{78}$ and $\mathcal{ST}_{78}$ respectively. Let t be the number of points of*

$\mathcal{T}_{78}$. Let f be the size of the $\mathcal{T}_{78}$-maximal parsing of T. The space/time trade-off of the Inverted-LZ-Index for the longest common substring problem is summarized as follows:

Space in bits	$[\frac{d}{n}(1+\epsilon)]uH_k + [\frac{d'}{n}(1+\epsilon)+\frac{f}{n}]u(\epsilon + H_k) + o(u \log \sigma)$
	$\leq (2+\epsilon)uH_k + (3+\epsilon)u(\epsilon + H_k) + o(u \log \sigma)$
Time to compute lcss	$O((m/\epsilon)\log^2 u)$
Conditions	$k = o(\log_\sigma u)$, $\sigma = O(n)$, $0 < \epsilon \leq 1$, ϵ *is constant*

Proof. (Space) Kosaraju et al [66] showed that $n \log u = uH_k + o(u \log \sigma)$ for $k = o(\log_\sigma u)$ for the LZ78 parsing. In order to determine $n \log u$ for the bounded LZ78 parsing, just separate T into two strings, one with no repeated blocks and one in which every block is of size $(\log u)/\epsilon$. For the first string use the previous bound. For the second string observe that we have at most $\epsilon u/\log u$ blocks. Therefore, this part will require at most ϵu bits, and $n \log u = u(\epsilon + H_k) + o(u \log \sigma)$. An important detail to notice is that the dictionary tree $\mathcal{T}_{78}$ does not need to store repeated blocks and therefore has a smaller bound.

5.4 Practical Issues and Testing

We implemented a prototype for testing these ideas and compared it against naive LCSS algorithm implemented with other compressed indexes.

We start by testing all possible longest common substrings that span across more than two blocks. Since we have no range data structure (see Subsection 4.4.1), there is no way to avoid searching all this space. Usually a long common substring will be found in this space. This allows us to prune a large amount of the search space for substrings that are contained in two blocks, which is important because it is a very large space. Moreover, since our scanning technique basically consists in trying to extend substrings of P in T, there is little point in bounding the LZ78 parsing in practice. To test our algorithm, we implemented a naive LCSS algorithm. The idea is, for every suffix of P, to determine its largest prefix that occurs in T. The largest of these prefixes are LCSS's. To determine these

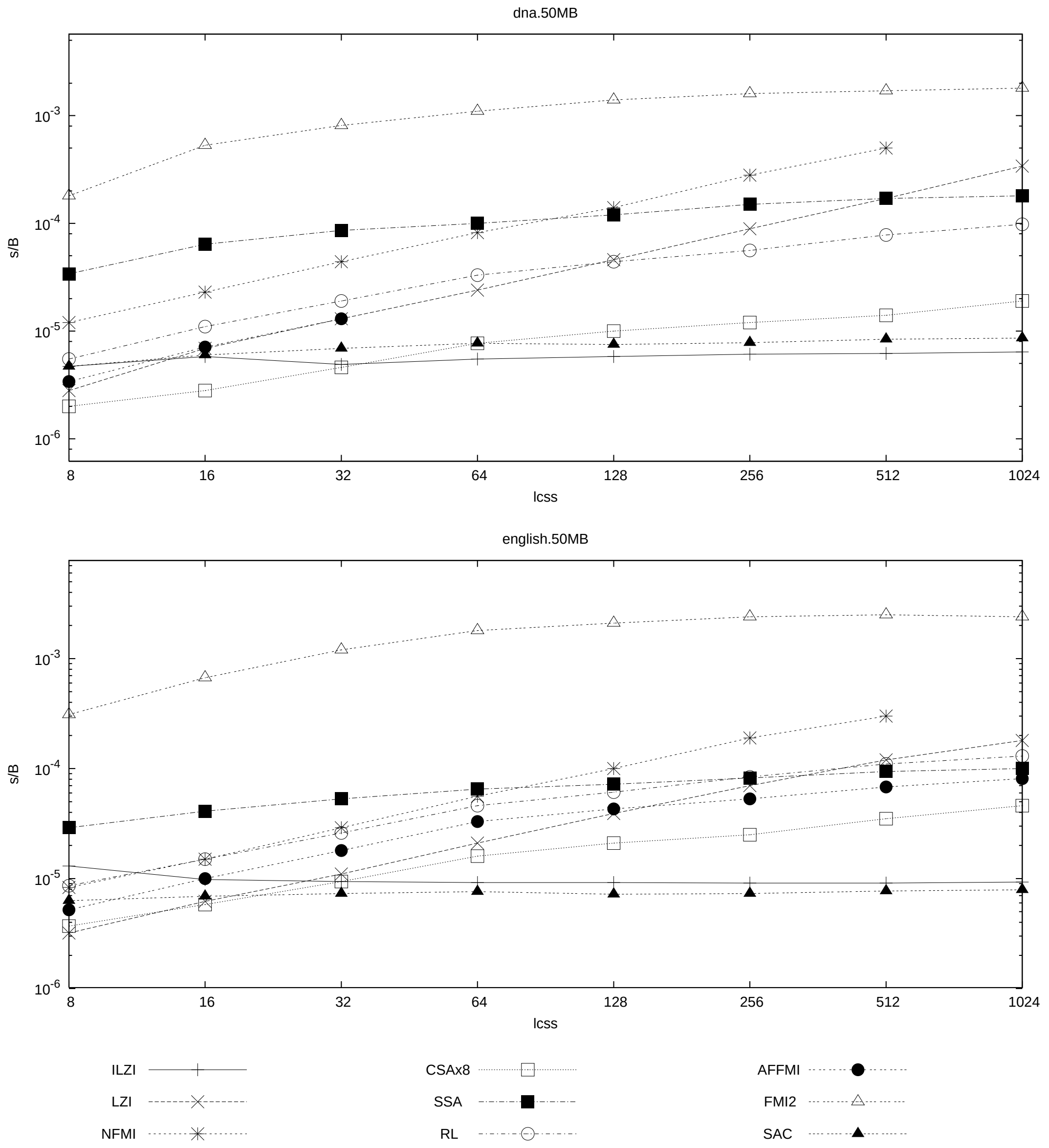


Fig. 5.5. Average user time of processing a byte for different lcss values.

prefixes, we use a compressed index. We used all the compressed indexes in the Pizza&Chili that were either suffix arrays or FM-indexes. In practice, we reduced the problem to the exact matching problem by determining these prefixes a priori and using them to do exact matching with the compressed indexes. As texts, we used the files in the Pizza&Chili corpus.

Table 6.2 shows the size of different indexes. The time results are shown in Figure 5.5. The pattern strings were generated randomly in such a way that the average size of the above prefixes is $lcss/2$. This is in fact the dominant factor for the naive, algorithm, that has a running time close to $O(lcss.m/2)$. In the y-axes, we show the time each index takes to process a byte, i.e. seconds per byte. This gives curves similar to $O(lcss/2)$ for the naive method. The x-axes represent the LCSS value. Note that both axes use logarithmic scales. Tests used patterns of roughly $1MB$ of size and ran for 60 seconds each. The tests were performed on a Pentium 4, 3.2 GHz, 1 MB of L2, 1Gb of RAM, with Fedora Core 3, and the prototype was compiled with `gcc-3.4 -O9`.

5.5 Conclusions and Future Work

In this chapter we extended the functionality of the Inverted-Lempel-Ziv-Index to be able to solve the longest common substring problem. This allowed us to show how flexible this index is. Note that the ability to perform pattern matching and compute the longest common substring are two of the main virtues of suffix trees.

As far as we are aware, this is the only existing method to compute the longest common substring using a compressed index. However, there is a data structure, by Sadakane, that gives compressed suffix trees (CST) with full functionality [109] in compressed space $uH_k+6u+o(u \log \sigma)$ bits. Using CST's it is possible to compute a longest common substring in $O((m(\log \sigma) \min(\sigma, \log u) \log^2 u)/ \log\log u)$ time. Sadakane pointed out that the u terms in the space requirements may become a bottleneck, which happens for $H_k < 1$. Grossi et al. [50] solved this problem by adding an $O(\log u)$ slowdown factor. Our extension of the ILZI provides the best of two worlds, since we can eliminate our $3u$ bits dependence by choosing any $o(1)$ function. In particular, if we take $\epsilon = 1/\log\log u$ then the ILZI requires at most $5uH_k + o(u \log \sigma)$ bits and computes LCSS in $O(m(\log^2 u) \log\log u)$ time. This is competitive with the solution presented by Grossi et al, especially because, in practice, the

uH_k factor is much smaller than 5, it is usually 3 (see the column i/uH_k of Table 4.3). If we take $\epsilon = 1$ the resulting ILZI is competitive against Sadakane's CST, since CST's is still slower than the ILZI by an $O((\log \sigma) \min(\sigma, \log u)/ \log \log u)$ factor.

We implemented a prototype to test this algorithm that showed a linear dependence on m. This is in fact the fundamental problem associated with the longest common substring. In fact our extension of the ILZI performs worse for small $LCSS$ sizes, mainly because it is forced to check for $LCSS$ that are spread across only two blocks and this is time consuming. We also concluded that, interestingly, the naive method is in fact efficient for a considerable interval of $LCSS$ sizes.

6

Approximate String Matching

Despite the explosion of interest on self-indexes in recent years, there has not been much progress on search functionality beyond the basic exact search. In this chapter we focus on indexed approximate string matching (ASM), which is of great interest in many applications. We present an ASM algorithm that works on top of the inverted Lempel-Ziv index. We build on top of a hybrid approach, which is the best in practice for this problem. We show that the ILZI can be seen as an extension of the classical q-samples index. We give new insights on this type of index, which can be of independent interest, and then apply them to the ILZI. We show experimentally that our algorithm has a competitive performance and provides a useful space-time tradeoff compared to classical indexes.

Approximate string matching (ASM) is an important problem that arises in applications related to text searching, pattern recognition, signal processing, and computational biology, to name a few. It consists in locating all the occurrences of a given pattern string P in a larger text string T, letting the occurrences be at edit distance at most k from P. In this chapter, we focus on edit distance, that is, the minimum number of character insertions, deletions, and substitutions of single characters to convert one string into the other.

6.1 Related Work

In the on-line version of the problem we can preprocess the pattern but not the text [97]. The classical sequential search solution runs in $O(um)$ worst-case time (see [97]). An optimal average-case algorithm requires time $O(u(k + \log_\sigma m)/m)$ [21, 40], where σ is the size of the alphabet Σ. Those good average-case algorithms are called *filtration* algorithms: they traverse the text fast while checking for a simple necessary condition, and only when this holds do they verify the text area using a classical ASM algorithm. For long texts, however, sequential searching may be impractical because it must scan all the text.

There exist indexes specifically devoted to ASM [20, 25, 26, 71], but these are oriented to worst-case performance. There seems to exist an unbreakable space-time barrier with indexed ASM: either one obtains exponential times (in m or k), or one obtains exponential index space (e.g. $O(u \log^k u)$). Another trend is to reuse an index designed for exact searching, all of which are linear-space, and try to do ASM over it. Indexes based on suffix trees [121], suffix arrays [78], q-grams and q-samples, have been used. There exist several algorithms, based on suffix trees or arrays, which focus on worst-case performance [24, 45, 119]. Given the mentioned time-space barrier, they achieve a search time independent of u but exponential on m or k. Essentially, they simulate the sequential search over all the possible text suffixes, taking advantage of the fact that similar substrings are factored out in suffix trees or arrays.

Indexes based on q-grams (indexing all text substrings of length q) or q-samples (indexing non-overlapping text substrings of length q) are appealing because they require less space than suffix trees or arrays. The algorithms on those indexes do not offer any relevant worst-case guarantee, but perform well on average when the *error level* $\alpha = k/m$ is low enough, say $O(1/\log_\sigma u)$. Those indexes basically simulate an on-line filtration algorithm, such that the "necessary condition" checked involves exact matching of pattern substrings, and as such can be verified with any exact-searching index. Such filtration indexes, e.g. [90, 116], cease to be useful for moderate k values, which are still of interest in many applications.

The most successful approach, in practice, is in between the two techniques described above, and is called "hybrid" indexing. The index determines the text positions requiring verification using not an exact, but an approximate-matching condition. Those are checked with a technique of the first kind (whose time is exponential on the length of the string or the number of errors). Since these searches are done over short strings and allowing few errors, the exponential cost is controlled. Indexes of this kind offer average-case guarantees of the form $O(mu^{\lambda})$ for some $0 < \lambda < 1$, and work well for higher error levels. They have been implemented over q-gram indexes [88], suffix arrays [92] and over q-sample indexes [101].

Yet, many of those linear-space indexes are very large anyway. For example, suffix arrays require 4 times the text size and suffix trees require at the very least 10 times [67]. Therefore, it does make sense to use compressed indexes for this problem.

Despite the great success of self-indexes, they have been mainly used for exact searching. Only very recently some indexes taking $O(u)$ or $O(u\sqrt{\log u})$ *bits* have appeared [20, 55, 68]. Yet, those are again of the worst-case type, and thus all their times are exponential in k.

We present a practical algorithm that runs on a compressed self-index and belongs to the most successful class of hybrid algorithms. More details are given next.

6.2 Our Contribution in Context

One can easily use *any* compressed self-index to implement a filtration ASM method that relies on looking for exact occurrences of pattern substrings, as this is what all self-indexes provide. The details of how to divide P are given by Lemma 6.1, using $j = k + 1$, $A = P$ and $B = O$. Indeed, this has been already attempted [83] using the FM-index [37] and a Lempel-Ziv index [98]. The Lempel-Ziv index worked better because it is faster to extract the text to verify (recall that in self-indexes the text is not directly available). The specific structure of the Lempel-Ziv index used allowed several interesting optimizations (such as factoring out the work of several text extractions) that we will not discuss further here.

Lempel-Ziv indexes are based on splitting the text into a sequence of so-called *phrases* of varying length. They are rather efficient to find the (exact) occurrences that lie within phrases, but those that span two or more phrases are more costly.

Our goal in this chapter is to have efficient approximate searching over a small and practical self-index. Based on the described previous experiences we want:

1. An algorithm of the hybrid type, which implies that the self-index should do approximate search for pattern pieces.
2. A Lempel-Ziv-based index, so that the extraction of text to verify is fast.
3. A way to avoid the problems derived from pieces spanning several Lempel-Ziv phrases.

We will focus on our index of Chapter 4 whose suffix-tree-like structure is useful for this approximate searching.

Mimicking q-sample indexes is particularly useful for our goals. Consider that the text is partitioned into contiguous q-samples. Any occurrence O of P is of length at least $m-k$. Wherever an occurrence lies, it must contain at least $j = \lfloor (m-k-q+1)/q \rfloor$ complete q-samples. The following lemma, simplified from [94], gives the connection to use approximate searching for pattern substrings with a q-samples index [101].

Lemma 6.1. *Let A and B be strings such that $ed(A,B) \leq k$. Let $A = A_1 A_2 \ldots A_j$, for strings A_i and for any $j \geq 1$. Then there is a substring B' of B and an i such that $ed(B', A_i) \leq \lfloor k/j \rfloor$.*

Consider for example that $k = 9$, $A = A_1.A_2.A_3$, $A_1 = ab$, $A_2 = cdefghijklm$, $A_3 = no$ and $B = axbcxdxexfgxhixjkxlxmnxo$, see Table 6.1. We can conclude that there is a substring B' of B and an i such that $ed(B', A_i) \leq \lfloor 9/3 \rfloor = 3$. In particular $ed(A_1, axb) = 1$. In this case we also have that $ed(A_3, nxo) = 1$.

Therefore, if we interpret $B = P$ and A contained in O, we index all the different text q-samples into, say, a trie data structure. Then the trie is traversed to find q-samples that match within P with at most $\lfloor k/j \rfloor$ errors. All the contexts around all occurrences of the matching q-samples are examined for full occurrences of P. Note, in passing, that we could also take $A = P$ and B contained in O, in which case we choose how to partition P but we must be able to find any text substring with the index (exactly [90] or approximately [88, 92], depending on j). Thus, we must use a suffix tree or array [92], or even a q-gram index if we never use pieces of P longer than q [88, 90].

A Lempel-Ziv parsing can be regarded as an irregular sampling of the text, and therefore our goal in principle is to adapt the techniques of [101] to an irregular parsing (thus we must stick to the interpretation $B = P$). As desired, we would not need to consider occurrences spanning more than one phrase. Moreover, the trie of phrases stored by all Lempel-Ziv self-indexes is the exact analogous of the trie of q-samples. Thus, we could search without requiring further structures in the index.

The irregular parsing poses several challenges, however. There is no way to ensure that there will be a minimum number j of phrases contained in an occurrence. Occurrences could even be fully contained in a phrase!

We develop several tools to face those challenges. First, we point out a variation of Lemma 6.1 that distributes the errors in a convenient way when the samples are of varying length. Second, we introduce a new filtration technique where the samples that overlap the occurrence (not only those contained in the occurrence) can be considered. This is of interest even for classical q-sample indexes. Third, we search for q-samples within long phrases to detect occurrences even if they are within a phrase. This technique also includes novel insights.

We implement our scheme and compare it experimentally with the best technique in practice over classical indexes [92], and with the previous developments over compressed self-indexes [83]. The experiments show that our technique is practical and provides a relevant space-time tradeoff for indexed ASM.

Table 6.1. The shortest path in the edit graph is shown in bold, we do not show inactive cells. (Left) Dynamic programming table D for strings, $B = P = axbcxdxexfgxhixjkxlxmnxo$ (vertical) and $A = O = abcdefghijklmno$ (horizontal) with $k = 9$. (Right) Dynamic programming table D for string $axbcxdxexfgxhixjkxlxmnxo$ and A divided into strings $A_1 = ab$, $A_2 = cdefghijklm$ and $A_3 = no$ and searched for with $k = 3$, $k = 7$ and $k = 3$ respectively.

		a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
	0	1	2	3	4	5	6	7	8	9						
a	1	**0**	1	2	3	4	5	6	7	8	9					
x	2	**1**	1	2	3	4	5	6	7	8	9					
b	3	2	**1**	2	3	4	5	6	7	8	9					
c	4	3	2	**1**	2	3	4	5	6	7	8	9				
x	5	4	3	**2**	2	3	4	5	6	7	8	9				
d	6	5	4	3	**2**	3	4	5	6	7	8	9				
x	7	6	5	4	**3**	3	4	5	6	7	8	9				
e	8	7	6	5	4	**3**	4	5	6	7	8	9				
x	9	8	7	6	5	**4**	4	5	6	7	8	9				
f		9	8	7	6	5	**4**	5	6	7	8	9				
g			9	8	7	6	5	**4**	5	6	7	8	9			
x				9	8	7	6	**5**	5	6	7	8	9			
h					9	8	7	6	**5**	6	7	8	9			
i						9	8	7	6	**5**	6	7	8	9		
x							9	8	7	**6**	6	7	8	9		
j								9	8	7	**6**	7	8	9		
k									9	8	7	**6**	7	8	9	
x										9	8	**7**	7	8	9	
l											9	8	**7**	8	9	
x												9	**8**	8	9	
m													9	**8**	9	
n														9	**8**	9
x															**9**	9
o																**9**

		a	b		c	d	e	f	g	h	i	j	k	l	m		n	o
	0	1	2															
a	1	**0**	1															
x	2	**1**	1															
b	3	2	**1**	**0**	1	2	3	4	5	6	7							
c				1	**0**	1	2	3	4	5	6	7						
x				2	**1**	1	2	3	4	5	6	7						
d				3	2	**1**	2	3	4	5	6	7						
x				4	3	**2**	2	3	4	5	6	7						
e				5	4	3	**2**	3	4	5	6	7						
x				6	5	4	**3**	3	4	5	6	7						
f				7	6	5	4	**3**	4	5	6	7						
g					7	6	5	4	**3**	4	5	6	7					
x						7	6	5	**4**	4	5	6	7					
h							7	6	5	**4**	5	6	7					
i								7	6	5	**4**	5	6	7				
x									7	6	**5**	5	6	7				
j										7	6	**5**	6	7				
k											7	6	**5**	6	7			
x												7	**6**	6	7			
l													7	**6**	7			
x														**7**	7			
m															**7**	**0**	1	2
n																1	**0**	1
x																2	**1**	1
o																3	2	**1**

6.3 An Improved q-samples Index

In this Section we extend classical q-sample indexes by allowing samples to overlap the pattern occurrences. This is of interest by itself, and will be used for an irregular sampling index later. Remind that a q-samples index stores the locations, in T, of all the substrings $T[qi..qi+q-1]$.

6.3.1 Varying the Error Distribution

We will need to consider parts of samples in the sequel, as well as samples of different lengths. Lemma 6.1 gives the same number of errors to all the samples, which is disadvantageous when pieces are of different lengths. The next lemma generalizes Lemma 6.1 to allow different numbers of errors in each piece.

Lemma 6.2. *Let A and B be strings, let $A = A_1 A_2 \dots A_j$, for strings A_i and some $j \geq 1$. Let $k_i \in \mathbb{R}$ such that $\sum_{i=1}^{j} k_i > ed(A, B)$. Then there is a substring B' of B and an i such that $ed(A_i, B') < k_i$.*

Proof. The edit distance between A and B corresponds to the shortest path in the edit graph of A and B. The partition of A induces a partition in this graph and in particular splits the shortest path. Let B_i be the substring of B that shares the shortest path with A_i. This means that $\sum_{i=1}^{j} ed(A_i, B_i) = ed(A, B)$.

Suppose, by absurd, that for every B' substring of B and every i we have that $ed(A_i, B') \geq k_i$. Therefore, this is also true for the B_i's, i.e., $ed(A_i, B_i) \geq k_i$. This is absurd since that way $ed(A, B) = \sum_{i=1}^{j} ed(A_i, B_i) \geq \sum_{i=1}^{j} k_i > ed(A, B)$. Therefore, there must exist an i such that $ed(A_i, B_i) < k_i$. Note that B_i is the substring B' we mention in the lemma. $\square$

Recall the application of Lemma 6.1 with $k = 9$, $A = A_1.A_2.A_3$, $A_1 = ab$, $A_2 = cdefghijklm$, $A_3 = no$ and $B = axbcxdxexfgxhixjkxlxmnxo$ (see Table 6.1). We can conclude that there is a substring B' of B and an i such that $ed(B', A_i) \leq \lfloor 9/3 \rfloor = 3$. In particular, we have that $ed(A_1, axb) = 1$ and $ed(A_3, nxo) = 2$. This number of errors (3) is excessively high for strings a as small as A_1 and A_3. If, instead, we use Lemma 6.2 with $k_1 = k_3 = 1.2$ and $k_2 = 6.7$, note that $k_1 + k_2 + k_3 = 9.1 > 9$. Using this lemma, we only allow for 1 error for A_1 and A_3. In this case we have that $ed(A_1, axb) = 1 < 1.2$. We also have that $ed(A_3, nxo) = 1 < 1.2$. Lemma 6.1 is a particular case of Lemma 6.2: set $k_i = k/j + \epsilon$ for a sufficiently small $\epsilon > 0$.

An early version of this lemma was given by Navarro et al. [93] in the context of hierarchical verification. This result was later improved by Navarro et al. [94], that presented the following lemma:

Lemma 6.3. *Let A and B be strings, let $A = A_1A_2 \ldots A_j$, for strings A_i and some $j \geq 1$. Let $k_i \in \mathbb{N}_0$ such that $j \geq 1$ and $\sum_{i=1}^{j} k_i \geq ed(A, B) - j + 1$. Then there is a substring B' of B and an i such that $ed(A_i, B') \leq k_i$.*

Proof. We will prove this lemma using Lemma 6.2. Let k_i be the numbers referred in this lemma. Consider $k'_i = k_i + 1$. The k'_i numbers satisfy the conditions of Lemma 6.2, i.e. $\sum_{i=1}^{j} k'_i = j + \sum_{i=1}^{j} k_i \geq j - j + 1 + k = k + 1 > k$. Therefore, by Lemma 6.2, we can conclude that there is a substring B' of B and an i such that $ed(A_i, B') < k'_i = k_i + 1$. Since $ed(A_i, B')$ and k_i are integers we have that $ed(A_i, B') \leq k_i$. $\square$

In our version the k_i's can be real numbers. However, this is not the case with the version given by Navarro et al. For example with $k = 3$, $k_i = 1/3$ and $j = 3$, where $\sum_{i=1}^{j} k_i = 1 = 3 - 3 + 1 = k - j + 1$, we conclude that some A_i must occur with at most $1/3$ errors. Since the edit distance between two strings is always an integer, the conclusion would be that $ed(A_i, B') \leq 0$, which is not correct. This conclusion is the main reason for Lemma 6.2. In this way, we can expose our results without being forced to work with integers, i.e. having to deal with floors and ceilings. Also, we do not need to know j a priori, which is important since we do not know the value of j before locating O in T. The proof of Lemma 6.3 shows that it is a consequence of Lemma 6.2. In fact, Lemma 6.2 and Lemma 6.3 are equivalent. To prove that Lemma 6.3 implies Lemma 6.2 use the transformation $k'_i = \lceil k_i \rceil - 1$ and a reasoning similar to the one above.

Lemma 6.2 can be used to adapt the error levels to the length of the pieces. For example, it is appropriate to try to maintain a constant error level, by taking $k_i = (1+\epsilon)\ k \cdot |A_i|/|A|$ for any $\epsilon > 0$.

In our example, this would yield $k_1 = k_3 = (1+\epsilon)\ 9 \times 2/15 = (1+\epsilon)1.2$ and $k_2 = (1+\epsilon)\ 9 \times 11/15 = (1+\epsilon)6.6$. This gives essentially the same result we presented before.

6.3.2 Partial q-sample Matching

Contrary to all previous work, let us assume that A in Lemma 6.2 is not only that part of an approximate occurrence O formed by full q-samples, but instead that $A = O$, so that A_1 is the suffix of a sample and A_j is the prefix of a sample. An advantage of this is that now the number of involved q-samples is at least $j = \lceil (m-k)/q \rceil$, and therefore we can permit fewer errors per piece (e.g. $\lfloor k/j \rfloor$ using Lemma 6.1). On the other hand, we would like to allow fewer errors for the pieces A_1 and A_j. Yet, notice that any text q-sample can participate as A_1, A_j, or as a fully contained q-sample in different occurrences at different text positions. Lemma 6.2 tells us that we could allow $k_i = (1+\epsilon)\ k \cdot |A_i|/|A|$ errors for A_i, for any $\epsilon > 0$. Conservatively, this is $k_i = (1+\epsilon)\ k \cdot q/(m-k)$ for $1 < i < j$, and less for the extremes.

In order to adapt the trie searching technique to those partial q-samples, we should not only search all the text q-samples with $(1+\epsilon)\ k \cdot q/(m-k)$, but also all their prefixes and suffixes with fewer errors. This includes, for example, verifying all the q-samples whose first or last character appears in P (cases $|A_1| = 1$ and $|A_j| = 1$). This is unaffordable. Our approach will be to redistribute the errors across A using Lemma 6.2 in a different way to ensure that only sufficiently long q-sample prefixes and suffixes are considered.

Let v be a non-negative integer parameter. We associate to every letter of A a weight: the first and last v letters have weight 0 and the remaining letters have weight $(1+\epsilon)/(|A|-2v)$. We define $|A_i|_v$ as the sum of the weights of the letters of A_i. For example if A_i is within the first v letters of A then $|A_i|_v = 0$; if it does not contain any of the first or last v letters then $|A_i|_v = (1+\epsilon)\ |A_i|/(|A|-2v)$.

We can now apply Lemma 6.2 with $k_i = k \cdot |A_i|_v$, provided that $k > 0$. Note that $\sum_{i=1}^{j} k_i = (1+\epsilon)\ k > k$. In this case, if $|A_1| \leq v$ we have that $k_1 = 0$ and, therefore, A_1 can never be found with *strictly less* than zero errors. The same holds for A_j. This effectively relieves us from searching for any q-sample prefix or suffix of length at most v.

Parameter v is thus doing the job of discarding q-samples that have very little overlap with the occurrence $O = A$, and maintaining the rest. It balances between two exponential costs: one due to verifying all the occurrences of too short prefixes/suffixes, and another due to permitting too many errors when searching for the pieces in the trie. In practice, tuning this parameter will have a very significant impact on performance.

Recall the application of Lemma 6.2 with $k = 9$, $A = A_1.A_2.A_3$, $A_1 = ab$, $A_2 = cdefghijklm$, $A_3 = no$, $k_1 = k_3 = 1.2$, $k_2 = 6.7$ and $B = axbcxdxexfgxhixjkxlxmnxo$. In this case we would need to search for all the q-samples that end in a string that is at edit distance 1 from $A_1 = ab$. This, in particular, includes all the q-samples that contain the suffix a, all the q-samples that contain the suffix b and so on. This yields an excessive number of samples to verify. If we consider $v = 1$ in this example we will instead obtain $k_1 = k_3 = (1+\epsilon)\ 9 \times 1/13 \approx 0.7$ and $k_2 = (1+\epsilon)\ 9 \times 11/13 \approx 7.7$. Note that $k_1 + k_2 + k_3 = 9.1 > 9$. In our example we have that $ed(A_2, cxdxexfgxhixjkxlxm) = 7 < 7.7$. Note that, in this example, we moved the errors to A_2 since A_1 and A_3 are very small.

6.3.3 A Hybrid q-samples Index

We have explained all the ideas necessary to describe a hybrid q-samples index. The algorithm works in two steps. First we determine all the q-samples O_i for which $ed(O_i, P') <$

$k \cdot |O_i|_v$ for some substring P' of P. In this phase we also determine the q-samples that contain a suffix O_1 for which $ed(O_1, P') < k \cdot |O_1|_v$ for some prefix P' of P (note that we do not need to consider substrings of P, just prefixes). Likewise we also determine the q-samples that contain a prefix O'_j for which $ed(O'_j, P') < k \cdot |O_j|_v$ for some suffix P' of P (similar observation). The q-samples that classify are potentially contained inside an approximate occurrence of P, i.e. O_i may be a substring of a string O such that $ed(O, P) \leq k$. In order to verify whether this is the case, in the second phase we scan the text context around O_i, with a sequential algorithm.

As the reader might have noticed, the problem of verifying conditions such as $ed(O_i, P') < k \cdot |O_i|_v$ is that we cannot know a priori which i does a given text q-sample correspond to. Different occurrences of the q-sample in the text could participate in different positions of an O, and even a single occurrence in T could appear in several different O's. We do not know either the size $|O|$, as it may range from $m - k$ to $m + k$.

A simple solution is as follows. Conservatively assume $|O| = m - k$ (in Section 6.5.1 we give a more sophisticated approach). Then, search P for each different text q-sample in three roles: (1) as a q-sample contained in O, so that $|O_i| = q$, assuming pessimistically $|O_i|_v = (1 + \epsilon)\ \min(q/(m - k - 2v), 1)$; (2) as an O_1, matching a prefix of P for each of the q-sample suffixes of lengths $v < \ell < q$, assuming $|O_1| = \ell$ and thus $|O_1|_v = (1 + \epsilon)\ \min((\ell - v)/(m - k - 2v), 1)$; (3) as an O_j, matching a suffix of P for each of the q-sample prefixes, similarly to case (2) (that is, $|O_j|_v = |O_1|_v$). For the q-samples index we assume that $q < m - k$ and therefore the case of O contained inside a q-sample does not occur.

In practice, one does not search for each q-sample in isolation, but rather factors out the work due to common q-gram prefixes by backtracking over the trie and incrementally computing the dynamic programming matrix between every different q-sample and any substring of P.We note that the trie of q-samples is appropriate for role (3), but not particularly efficient for roles (1) and (2) (finding q-samples with some specific suffix). In our application to a Lempel-Ziv index this will not be a problem, because we will have also a trie of the reversed phrases, that will replace the q-grams.

6.4 Using a Lempel-Ziv Self-Index

We now adapt our technique to the irregular parsing of phrases produced by a Lempel-Ziv-based index. We will focus on the ILZI although the results can be carried over other similar indexes.

Recall that the ILZI partitions the text into phrases such that every suffix of a phrase is also a phrase (similarly to LZ78 compressors [125], where every prefix of a phrase is also a phrase). It uses two tries, one storing the phrases and another storing the reverse phrases. In addition, it stores a mapping that facilitates moving from one trie to the other, and it stores the compressed text as a sequence of phrase identifiers.

6.4.1 Handling Different Lengths

As explained, the main idea is to use the phrases instead of q-samples. For this sake, Lemma 6.2 solves the problem of distributing the errors homogeneously across phrases. However, other problems arise, especially for long phrases. For example, an occurrence could be completely inside a phrase. In general, backtracking over long phrases is too costly.

Even when an occurrence is not completely contained inside a phrase, it is not viable to start searching for an occurrence with too many errors, even if the occurrence is long. This is essentially the argument in favor of filtration and hybrid indexes. Recall our example with $k = 9$, $A = A_1.A_2.A_3$, $A_1 = ab$, $A_2 = cdefghijklm$, $A_3 = no$, $k_1 = k_3 = 0.7$, $k_2 = 7.7$ and $B = axbcxdxexfgxhixjkxlxmnxo$. Recall also that we are considering $P = B$ and $O = A$. It is not a viable approach to search for all the samples A' such that $ed(A', cxdxexfgxhixjkxlxm) \leq 7$, by using neighborhood generation.

The problem is essentially that, in this way, we allow for all the seven errors to concentrate in the beginning of A',i.e., we have to potentially consider for σ^7 strings. Only a very small percentage of these strings will end up extending to a sample A' that verifies $ed(A', cxdxexfgxhixjkxlxm) \leq 7$. In order to avoid this problem we adopt a hybrid approach.

We resort again to q-samples, this time within phrases. We choose two non-negative integer parameters q and $s < q$. We will look for any q-gram of P that appears with less than s errors within any phrase. All phrases spotted along this process must be verified. Still, some phrases not containing any pattern q-gram with $< s$ errors can participate in an occurrence of P (e.g. if $\lfloor (m - k - q + 1)/q \rfloor \cdot s \leq k$ or if the phrase is shorter than q). Next, we show that those remaining phrases have a specific structure that makes them easy to find.

Lemma 6.4. *Let A and B be strings and q and s be integers such that $0 \leq s < q \leq |A|$ and for any substrings B' of B and A' of A with $|A'| = q$ we have that $ed(A', B') \geq s$. Then for every prefix A' of A there is a substring B' of B such that $ed(A', B') \leq ed(A, B) - s\lfloor (|A| - |A'|)/q \rfloor$.*

Proof. This lemma is an application of Lemma 6.2 and it is important because it gives a good description of the restrictions obtained by the filtration.

Apply Lemma 6.2 with $A_1 = A'$, $|A_j| < q$ and the remaining A_i's of size q, i.e. $|A_i| = q$. Consider $k_1 = ed(A, B) - s\lfloor (|A| - |A'|)/q \rfloor + \epsilon$ with $0 < \epsilon < 1$, $k_j = 0$. The remaining k_i's are equal to s. Note that $j = \lfloor (|A| - |A'|)/q \rfloor + 1$ and therefore $\sum_{i=1}^{j} k_i = ed(A, B) + \epsilon - s\lfloor (|A| - |A'|)/q \rfloor + s(j - 1) = ed(A, B) + \epsilon - s\lfloor (|A| - |A'|)/q \rfloor + s\lfloor (|A| - |A'|)/q \rfloor = ed(A, B) + \epsilon > ed(A, B)$.

Therefore we conclude that there is a substring B' of B and an i such that $ed(A_i, B') < k_i$. We will prove that it must be $i = 1$. Suppose that $i = j$ then $ed(A_j, B') < k_j = 0$, which is impossible. Suppose that $1 < i < j$, then $ed(A_i, B') < k_i = s$ with $|A_i| = q$, which contradicts the hypotheses of the lemma. Therefore $i = 1$ and $ed(A_1, B') = ed(A', B') < k_1 = ed(A, B) - s\lfloor(|A| - |A'|)/q\rfloor + \epsilon$, which means that $ed(A', B') \leq ed(A, B) - s\lfloor(|A| - |A'|)/q\rfloor$. □

The lemma implies that, if a phrase is close to a substring of P, but none of its q-grams are sufficiently close to any substring of P, then the errors must be distributed uniformly along the phrase. Therefore we can check the phrase progressively (for increasing prefixes), so that the number of errors permitted grows slowly. This severely limits the necessary backtracking to find those phrases that escape from the q-gram-based search.

Let us consider $q = 4$ in our running example. Note that for a substring A' of A of size 4 we have that $k \cdot |A'|_v = (1 + \epsilon)\ 9 \times 4/13 \approx 2.8$. Therefore we choose $s = 2$. Note that we cannot use Lemma 6.4 with $A = A_2 = cdefghijklm$ and $B = cxdxexfgxhixjkxlxm$, since $ed(fghi, fgxhi) = 1 < 2 = s$ and $|fghi| = 4$. This means that there is an exceptionally well preserved area of B in A where we only deleted 1 letter out of 5 consecutive characters. Consider instead that $A = cdefghijklm$ and that $B = cxdxexfgxhxijxkxlm$. In this case we can use Lemma 6.4 and obtain the bound $ed(cdefghijklm[..i], B') \leq 7 - 2\lfloor(11 - i - 1)/4\rfloor$ for some substrings B' of B. Note in particular that this is tight for $i = 6$, i.e. $ed(cdefghijklm[..6], B') = ed(cdefghi, cxdxexfgxhxi) = 5 = 7 - 2 \times \lfloor 4/4 \rfloor$.

Parameter s allows us to balance between two search costs. If we set it low, then the q-gram-based search will be stricter and faster, but the search for the escaping phrases will be costlier. If we set it high, most of the cost will be related with the q-gram search.

6.4.2 A Hybrid Lempel-Ziv Index

The following lemma describes the way we combine previous results to perform approximate search using the ILZI.

Lemma 6.5. *Let A and B be strings such that $0 < ed(A, B) \leq k$. Let $A = A_1 A_2 \ldots A_j$, for strings A_i and some $j \geq 1$. Let q, s and v be integers such that $0 \leq s < q \leq |A|$ and $0 \leq v < |A|/2$. Then there is a substring B' of B and an i such that either:*

1. *there is a substring of A' of A_i such that $|A'| = q$ and $ed(A', B') < s$, or*
2. *$ed(A_i, B') < k \cdot |A_i|_v$ in which case for any prefix A' of A_i there exists a substring B'' of B' such that $ed(A', B'') < k \cdot |A_i|_v - s\lfloor(|A_i| - |A'|)/q\rfloor$.*

Proof. As we have explained, our approach first consists in applying Lemma 6.2 considering $k_i = k \cdot |A_i|_v$ for $0 \le i \le j$. Observe that, by the definition of $|A_i|_v$, we have that $\sum_{i=1}^{j} k \cdot |A_i|_v = k(\epsilon + (|A| - 2v)/(|A| - 2v)) = k(\epsilon + 1) > k$. Now, we classify the resulting A_i into one the two classes we defined before. The first class justifies the first condition of this lemma. If A_i belongs to the second class, then we apply Lemma 6.4 with the resulting k_i. $\square$

As before, the search runs in two phases. In the first phase, we find the phrases whose text context must be verified. In the second phase we verify those text contexts for an approximate occurrence of P. Lemma 6.5 gives the key to carry out the first phase. We find the relevant phrases via two searches:

- We look for any q-gram contained in a phrase which matches within P with less than s errors. We backtrack in the trie of phrases for every $P[y_1..]$, descending in the trie and advancing y_2 in $P[y_1..y_2]$ while computing the dynamic programming matrix between the current trie node and $P[y_1..y_2]$. We look for all trie nodes at depth q that match some $P[y_1..y_2]$ with less than s errors. Since every suffix of a phrase is a phrase in the ILZI, every q-gram within any phrase can be found starting from the root of the trie of phrases. All the phrases Z that descend from each q-gram trie node found must be verified (those are the phrases that start with that q-gram). We must also spot the phrases suffixed by each such Z. For this sake, we map each phrase Z to the trie of reverse phrases and also verify all the descent of the reverse trie nodes. This search covers case 1 in Lemma 6.5.

This is precisely the case in our example, since we have that $q = 4$ and $ed(fghi, P[9..13]) = ed(fghi, fgxhi) = 1 < 2 = s$. We then determine that the phrase $cdefghijklm$ contains the string $fghi$ and locate the occurrence by searching the context around that phrase.

- We look for any phrase A_i matching a portion of P with less than $k \cdot |A_i|_v$ errors. This is done over the trie of phrases. Yet, as we go down in the trie (thus considering longer phrases), we can enforce that the number of errors found up to depth d must be less than $k \cdot |A_i|_v - s\lfloor(|A_i| - d)/q\rfloor$. This covers case 2 in Lemma 6.5, where the equations vary according to the roles described in Section 6.3.3 (that is, depending on i):
  - $1 < i < j$, in which case we are considering a phrase contained inside O that is not a prefix nor a suffix. The $k \cdot |A_i|_v$ formula (both for the matching condition and the backtracking limit) can be bounded by $(1 + \epsilon)\ k \cdot \min(|A_i|/(m - k - 2v), 1)$, which depends on $|A_i|$. Since A_i may correspond to any trie node that descends from the current one, we determine a priori which $|A_i| \leq m - k$ maximizes the backtracking limit. We apply the backtracking for each $P[y_1..]$.

    This is occurs in the variation with $P = axbcxdxexfgxhxijxkxlmnxo$, that obtains the limit $ed(cdefghijklm[..d + 1], B') \leq 7.7 - 2\lfloor(11 - d)/4\rfloor$ for some substring B' of $P[3..13] = cxdxexfgxhxijxkxlm$. Note that, in particular, this limit is valid for string $cdefghijklm$. Any string for which this is not the case is abandoned by the search.
  - $i = j$, in which case we are considering a phrase that starts by a suffix of O. Now $k \cdot |A_i|_v$ can be bounded by $(1 + \epsilon)\ k \cdot \min((d - v)/(m - k - 2v), 1)$, yet still the limit depends on $|A_i|$ and must be maximized a priori. This time we are only interested in suffixes of P, that is, we can perform m searches with $y_2 = m$ and different y_1. If a node verifies the condition, we must consider also those that descend from it, to get all the phrases that start with the same suffix of P.
  - $i = 1$, in which case we are considering a phrase that ends in a prefix of O. This search is as the case $i = j$, with similar formulas. We are only interested in prefixes of

P, that is $y_1 = 0$. As the phrases are suffix-closed, we can conduct a single search for $P[0..]$ from the trie root, finding all phrase suffixes that match each prefix of P. Each such suffix node must be mapped to the reverse trie and the descent there must be included. The case $i = j = 1$ is different, as it includes the case where O is contained inside a phrase. In this case we do not require the matching trie nodes to be suffixes, but also prefixes of suffixes. That is, we include the descent of the trie nodes and map each node in that descent to the reverse trie, just as in case 1.

A simplified version of the previous searches is shown in Algorithm 7. The **return** statement terminates the SEARCH procedure, while the REPORT procedure does not. The elements obtained by **return** and REPORT are used in the second phase of the algorithm. Note that depending on whether $i' = 0$ or $j' = m - 1$ the REPORT procedure includes the Z phrases relative to cases $i = 1$ or $i = j$ respectively. The C variable does not represent a specific string. Instead it represents a generic string, which means that bound in line 13 is generic.

6.4.3 Homogeneous Lempel-Ziv Phrases

In this section we give further insight into the structure of the homogeneous Lempel-Ziv phrases, i.e. those that verify the conditions of Lemma 6.4. The objective of this section is to analyze the complexity of the search for the escaping phrases, i.e. those that are not found in the q-gram based search. The search for these homogeneous phrases is described in the second point of the previous subsection.

We need a couple of lemmas that give further insight into the structure of these homogeneous phases. Lemma 6.4 explained that by restricting the minimal number of errors that a substring of A, of size q, may have we are also restricting the maximal number of errors that it can have. In fact this condition also restricts the spacing between consecutive errors. The next lemmas explain this property.

Algorithm 7 Simplified first phase

procedure SEARCH(A', i')
    $j' \leftarrow i'$
    for $j' < m$ **do**
        $B' \leftarrow P[i'..j']$
        if $|A'| = q$ and $ed(A', B') < s$ **then**
            return all Z's that contain A'
        end if
        $A_i \leftarrow A'$
        if $ed(A_i, B') < k \cdot |A_i|_v$ **then**
            REPORT(A_i, i', j')
        end if
        $A_i \leftarrow A'.C$
        if $ed(A', B') < k \cdot |A_i|_v - s\lfloor(|A_i| - |A'|)/q\rfloor$ **then**
            for $c \in \Sigma$ when $A'.c$ is a Z phrase **do**
                SEARCH ($A'.c$, i')
            end for
        end if
        $j'++$
    end for
end procedure
$i' \leftarrow 0$
for $i' < m$ **do**
    SEARCH (ϵ, i')
    $i'++$
end for

Lemma 6.6. *Let A and B be strings and q and s be integers such that $0 \leq s \leq q \leq |A|$ and for any substrings B' of B and A' of A with $|A'| = q$ we have that $ed(A', B') \geq s$. Then for any prefix of A' of A such that for any substring B' of B, $ed(A', B') \geq 1$ we can conclude that $|A'| > |A| - q\lceil(ed(A, B) - 1 + \epsilon)/s\rceil$, for any $\epsilon > 0$.*

Proof. Suppose by absurd that there is a prefix A' of A such that for any substring B' of B we have $ed(A', B') \geq 1$ and $|A'| \leq |A| - q\lceil(ed(A, B) - 1 + \epsilon)/s\rceil$ for some $\epsilon > 0$.

Now apply Lemma 6.2 with $A_1 = A'$, $|A_j| < q$ and the remaining A_i's of size q, i.e. $|A_i| = q$ and $j = 2 + \lceil (ed(A,B) - 1 + \epsilon)/s \rceil$. Consider $k_1 = 1$, $k_j = 0$ and the remaining k_i's equal to s. Therefore $\sum_{i=1}^{j} k_i = 1 + s(j-2) = 1 + s\lceil (ed(A,B) - 1 + \epsilon)/s \rceil \geq 1 + ed(A,B) - 1 + \epsilon = ed(A,B) + \epsilon > ed(A,B)$.

Therefore we conclude that there is a substring B' of B and an i such that $ed(A_i, B') < k_i$. This, however, is absurd because no value of i may verify this condition. Suppose that $i = j$. Then $ed(A_j, B') < k_j = 0$, which is impossible. Suppose that $1 < i < j$. Then $ed(A_i, B') < k_i = s$ with $|A_i| = q$, which contradicts the hypotheses of the lemma. Finally, it cannot be that $i = 1$ either. That would mean that $ed(A_1, B') = ed(A', B') < k_1 = 1$ for some B' and this contradicts our initial hypotheses that $ed(A', B') \geq 1$ for any B'. □

Lemma 6.7. *Let A and B be strings and q and s be integers such that $0 \leq s \leq q \leq |A|$ and for any substrings B' of B and A' of A with $|A'| = q$ we have that $ed(A', B') \geq s$. Then for any prefix of A' of A for which there is a substring B' of B, $ed(A', B') \leq 1$ we can conclude that $|A'| < q\lceil (1 + \epsilon)/s \rceil$, for any $\epsilon > 0$.*

Proof. Suppose by absurd that there is a prefix A' of A for which there is a substring B' of B such that $ed(A', B') \leq 1$ and $|A'| \geq q\lceil (1 + \epsilon)/s \rceil$, for some $\epsilon > 0$.

Consider the suffix A'' that results from A by removing A'. Consider also the suffix B'' that results when removing B' from B. Note that B' is not necessarily a prefix of B, but we are only interested in the remaining suffix.

We must have $ed(A, B) \leq ed(A', B') + ed(A'', B'')$, because of the shortest path interpretation. Therefore $ed(A'', B'') \geq ed(A, B) - ed(A', B') \geq ed(A, B) - 1$. The suffix A'' however is too small to contain this number of errors. To see this, apply Lemma 6.6, replacing 1 by $ed(A, B) - 1$ and using A'' and B''. We conclude that we must have $|A''| > |A| - q\lceil (1 + \epsilon)/s \rceil$. On the other hand, by our initial hypotheses, we conclude that $|A''| = |A| - |A'| \leq |A| - q\lceil (1 + \epsilon)/s \rceil$. Therefore we have reached an absurd condition and the lemma is proved. □

Ukkonen studied the structure of the set of O strings such that $ed(P, O) = k$, denoted as $U_k(P)$ [118], showing that $|U_k(P)| \leq (12/5)(m + 1)^k(\sigma + 1)^k$ (see Subsection 2.5.2). We will now count the number of strings for which the errors are spread homogeneously. We define as $U_{k,q,s}(P)$ the set of O strings such that $ed(P, O) = k$, and for any substring O' of O, with $|O'| = q$, and substring P' of P we have that $ed(O', P') \geq s$.

Lemma 6.8. $|U_{k,q,s}| \leq (2\sigma + 1)^k(2k + 1)[q(2 + k/s) + k - m - 1]^k$

Proof. From the previous lemmas we conclude that, for the strings in $U_{k,q,s}$ the spacing from an error to the next varies from $|O| - q\lceil(k - 1 + \epsilon)/s\rceil$ to $q\lceil(1 + \epsilon)/s\rceil$. Therefore, counting the number of strings in $U_{k,q,s}$ is a matter of choosing these spacings and the type of error to use. The total number of error types is $2\sigma+1$. One σ counts insertions, the other counts substitutions, and the 1 counts the deletions. The number of spacings available is given by the following expression:

$$q\lceil(1 + \epsilon)/s\rceil - (|O| - q\lceil(k - 1 + \epsilon)/s\rceil) - 1 \leq q(2 + k/s) - |O| - 1.$$

Therefore, the number of strings in $U_{k,q,s}$ can be bounded as follows:

$$\begin{aligned} |U_{k,q,s}| &\leq (2\sigma + 1)^k \sum_{|O|=m-k}^{m+k} [q(2 + k/s) - |O| - 1]^k \\ &= (2\sigma + 1)^k[q(2 + k/s) + k - m - 1]^k(2k + 1). \end{aligned}$$

□

This bound is somewhat loose. In fact, the $(2\sigma + 1)$ and $[q(2 + k/s) + k - m - 1]$ never occur together. For example when $[q(2 + k/s) + k - m - 1]$ occurs, the other factor is 1.

He have therefore shown that, by choosing q and s properly, $U_{k,q,s}$ is much smaller than U_k. Consider for example the optimal case where $m.s = q.k$. Then $U_{k,q,s} = O((2\sigma+1)^k(k - 1 + 2(m.s/k))^k(2k + 1))$. This shows that the dependency in m can be reduced to $m.s/k$, which is considerably smaller, especially when we take $s = 1$. To obtain this optimal result we choose the q and s parameters the same way as the hybrid index, by using $q = \lfloor m/h\rfloor$

and $s = \lfloor k/h \rfloor + 1$ for an appropriate parameter h. This parameter plays the role of the j parameter in the hybrid index of Navarro et al. [99].

6.5 Practical Issues and Testing

We implemented a prototype to test our algorithm on the ILZI compressed index [106]. As a baseline, we used efficient sequential bit-parallel algorithms, namely BPM, the bit-parallel dynamic programming matrix of Myers [89], and EXP, the exact pattern partitioning by Navarro and Baeza-Yates [91].

We also included in the comparison an implementation of a filtration index using the simple approach of Lemma 6.1 with $A = P$ and $B = O$, as briefly described in the beginning of Section 6.2 [83]. The indexes used in that implementation are the LZ-index [98] (LZI) and Navarro's implementation of the FM-index [37]. We also compare an improved variant of the approximate LZ-index (DLZI [83]). Note that the FM-Index does not need to divide the text into blocks. However it takes longer to locate occurrences.

The machine was a Pentium 4, 3.2 GHz, 1 MB L2 cache, 1GB RAM, running Fedora Core 3, and compiling with `gcc-3.4 -O9`. For the texts we used the files in the Pizza&Chili corpus (`http://pizzachili.dcc.uchile.cl`), with 50 MB of English and DNA and 64 MB of proteins. The pattern strings were sampled randomly from the text and each character was distorted with 10% of probability. All the patterns had length $m = 30$. Every configuration was tested during at least 60 seconds using at least 5 repetitions. Hence, the numbers of repetitions varied between 5 and 130,000. To parametrize the hybrid index we tested all the j values from 1 to $k + 1$ and reported the best time. For the ILZI we choose $q = \lfloor m/h \rfloor$ and $s = \lfloor k/h \rfloor + 1$ for some convenient h, since Lemma 6.8 shows this is the best approach and it was corroborated by our experiments. To determine the value of h and v, we also tested the viable configurations and reported the best results. In our examples, choosing v and h such that $2v$ is slightly smaller than q yielded the best configuration.

Table 6.2. Memory peaks, in Megabytes, for the different approaches when $k = 6$.

	ILZI	Hybrid	LZI	DLZI	FMIndex
English	55	257	145	178	131
DNA	45	252	125	158	127
Proteins	105	366	217	228	165

Figure 6.1 shows how the parameter v affects the average query time. The average query time, in seconds, is shown in Figure 6.2 and the respective memory heap peaks for indexed approaches are shown in Table 6.2. The incomplete results are due to limitations of the respective prototype. The hybrid index provides the fastest approach to the problem. However it also requires the most space. Aside from the hybrid index, our approach is always either the fastest or within reasonable distance from the fastest approach. For low error levels, $k = 1$ or $k = 2$, our approach is significantly faster, up to an order of magnitude better. This is very important since the compressed approaches seem to saturate at a given performance for low error levels: in English $k = 1$ to 3, in DNA $k = 1$ to 2, and in proteins $k = 1$ to 5. This means that indexed approaches are the best alternative only for low error levels. In fact, the sequential approaches outperform the compressed indexed approaches for higher error levels. In DNA this occurs at $k = 4$ (BPM) and in English at $k = 5$ (EXP).

Our index performed particularly well on proteins, as did the hybrid index. This could happen due to the fact that proteins behave closer to random text, and this means that the parametrization of ours and the hybrid index indeed balances between exponential worst cases.

In terms of space, the ILZI is also very competitive, as it occupies almost the same space as the plain text, except for proteins, that are not very compressible. We presented the space that the algorithms need to operate and not just the index size, since the other approaches need intermediate data structures to operate.

6.5.1 A Tight Backtracking Bound

For the real prototype we used a stricter backtracking than what was explained in previous sections. For each pattern substring $P[y_1..y_2]$ to be matched, we computed the maximum number of errors that could occur when matching it in the text, taking into consideration the position $O[x_1..x_2]$ where it would be matched, and maximizing over the possible areas of O where the search would be necessary. For example, the extremes of P can be matched with fewer errors than the middle. This process involves precomputing tables that depend on m and k.

For each $O_i = O[x_1..x_2 - 1]$, we can compute the value of $|O_i|_v$ as $(v(x_2) - v(x_1))(1 + \epsilon)/(|O| - 2v)$, where $v(x) = \max(v, \min(|O| - v, x))$. The problem is that, as we mentioned in the previous section, we perform searches for $P' = P[y_1..y_2 - 1]$ and therefore the values of x_1 and x_2 are not known. Therefore, we will compute a table that, for each pair y_1, $y_2 - 1$, stores the maximal possible value of $|O_i|_v$. To compute this maximal value we can use linear programming. Consider that I_1, D_1, S_1, I_2, D_2, S_2 and I_3, D_3, S_3 are variables. The D variables represent deletions, the S variables represent substitutions and the I variables represent insertions in the process that transforms O into P. The variables indexed by 1 (I_1, D_1, S_1) are relative to the transformation of $O[0..x_1 - 1]$ into $P[0..y_1 - 1]$. The variables indexed by 2 (I_2, D_2, S_2) are relative to the transformation of $O[x_1..x_2 - 1]$ into $P[y_1..y_2 - 1]$. The variables indexed by 3 (I_3, D_3, S_3) are relative to the transformation of $O[x_2..|P| - 1]$ into $P[y_2..|O| - 1]$.

The following equations describe the relations between P and O, the first three equations describe the effect of the errors and the last two reflect the bounds on the number of errors, namely that $ed(P, O) \leq k$ and $ed(O_i, P') < k.|O_i|_v$.

$$y_1 + I_1 - D_1 = x_1$$

$$y_2 - y_1 + I_2 - D_2 = x_2 - x_1$$

$$|P| - y_2 + I_3 - D_3 = |O| - x_2$$

$$I_1 + I_2 + I_3 + D_1 + D_2 + D_3 + S_1 + S_2 + S_3 \leq k$$

$$I_2 + S_2 + D_2 < k.|O_i|_v$$

For q-samples, we have the restriction that $x_2 - x_1 = q$. Our goal is, therefore, to compute a table $MO(y_1, y_2)$ that stores the maximum $|O_i|_v$ restricted by the relations above, for predetermined values of m and v. This can be solved by dividing this problem into linear programming problems, considering $|O|$ constant. A naive solution simply scans the possible values of $I_1, I_2, I_3, D_1, D_2, D_3, S_1, S_2, S_3, |O|$, requiring $O(k^{10})$ operations for each pair (y_1, y_2). This table depends on m and k, however it is completely independent of T, hence it can be computed a priori. Therefore the time to compute this table is not considered at query time.

We also need to maximize the limit condition $k.|O_i|_v - s\lfloor(|O_i| - |O'|)/q\rfloor$. For this purpose we add variables x_3, I_4, D_4, S_4 and the following equations:

$$I_4 \leq I_3$$

$$D_4 \leq D_3$$

$$S_4 \leq S_3$$

$$x_2 + D_4 \leq x_3 \leq |O|$$

In this scenario $O_i = O[x_1..x_3 - 1]$ and $O' = O[x_1..x_2 - 1]$. The formula for $|O_i|_v$ becomes $(v(x_3) - v(x_1))(1 + \epsilon)/(|O| - 2v)$ and the expression $s\lfloor(|O_i| - |O'|)/q\rfloor$ is computed as $s\lfloor(x_3 - x_2)/q\rfloor$. We store the maximal value of expression $k.|O_i|_v - s\lfloor(|O_i| - |O'|)/q\rfloor$ in a table $MOSQ(y_1, y_2, |O'|)$. Therefore, we can verify that condition $ed(O', P') < k.|O_i|_v -$

$s\lfloor(|O_i| - |O'|)/q\rfloor$ is verified by checking $ed(O', P[y_1..y_2 - 1]) < MOSQ(y_1, y_2, |O'|)$, where $|O'|$ is the current string depth of the search, i.e. d. This table can also be precomputed, since it does not depend on the letters of P. However it does depend on q and s.

6.6 Conclusions and Future Work

We presented an adaptation of the hybrid index for Lempel-Ziv compressed indexes. We started by addressing the problem of approximate matching with q-samples indexes, where we described a new approach to this problem. We then adapted our algorithm to the irregular parsing produced by Lempel-Ziv indexes. Our approach was flexible enough to be used as a hybrid index instead of an exact-searching-based filtration index. We implemented our algorithm and compared it with the simple filtration approach built over different compressed indexes, with sequential algorithms, and with a good uncompressed index.

Our results show that our index provides a good space/time tradeoff, using a small amount of space (at best 0.9 times the text size, which is 5.6 times less than a classical index) in exchange for searching from 6.2 to 33 times slower than a classical index, for $k = 1$ to 3. This is better than the existing compressed approaches for low error levels, which is significant since indexed approaches are most valuable, if compared to sequential approaches, when the error level is low. Therefore our work significantly improves the usability of compressed indexes for approximate matching.

A crucial part of our work was our approach to the prefixes/suffixes of O. This approach is in fact not essential for q-samples indexes. However it can improve previous results [101] (see Table A.14). However for a Lempel-Ziv index it is essential. In practice, we verified that $2v \leq q$. This means that we are sort of sacrificing q letters from the pattern and distribute the errors by the remaining letters, the same way that q-samples sacrifice at least $q - 1$ letters.

An interesting idea we presented was to associate error weights to the letters of O. This was done in a uniform fashion, except for the edges. We believe that tuning these weights may lead to considerable improvements. For example, assigning smaller weights to least frequent letters will force the algorithm to search for less frequent strings and, therefore, decrease the number of positions to verify. This weighting, however, cannot be precomputed and therefore requires further research.

Finally, our implementation can be further improved since we do no secondary filtering, that is, we do not apply any sequential filter over the text contexts before fully verifying them.

Fig. 6.1. Average user time, in seconds, that the ILZI takes to find occurrences of patterns of size 30 with k errors, using different v's.

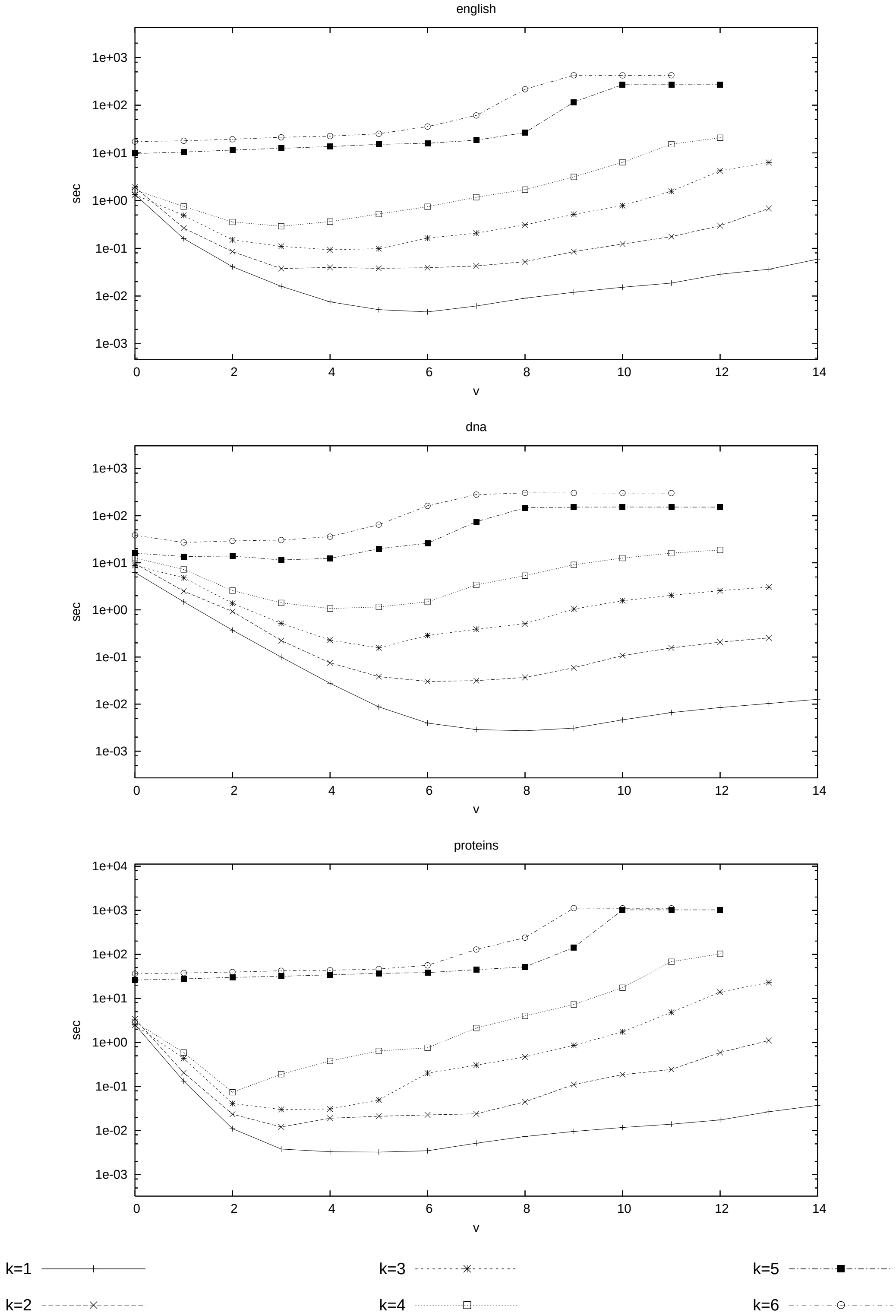
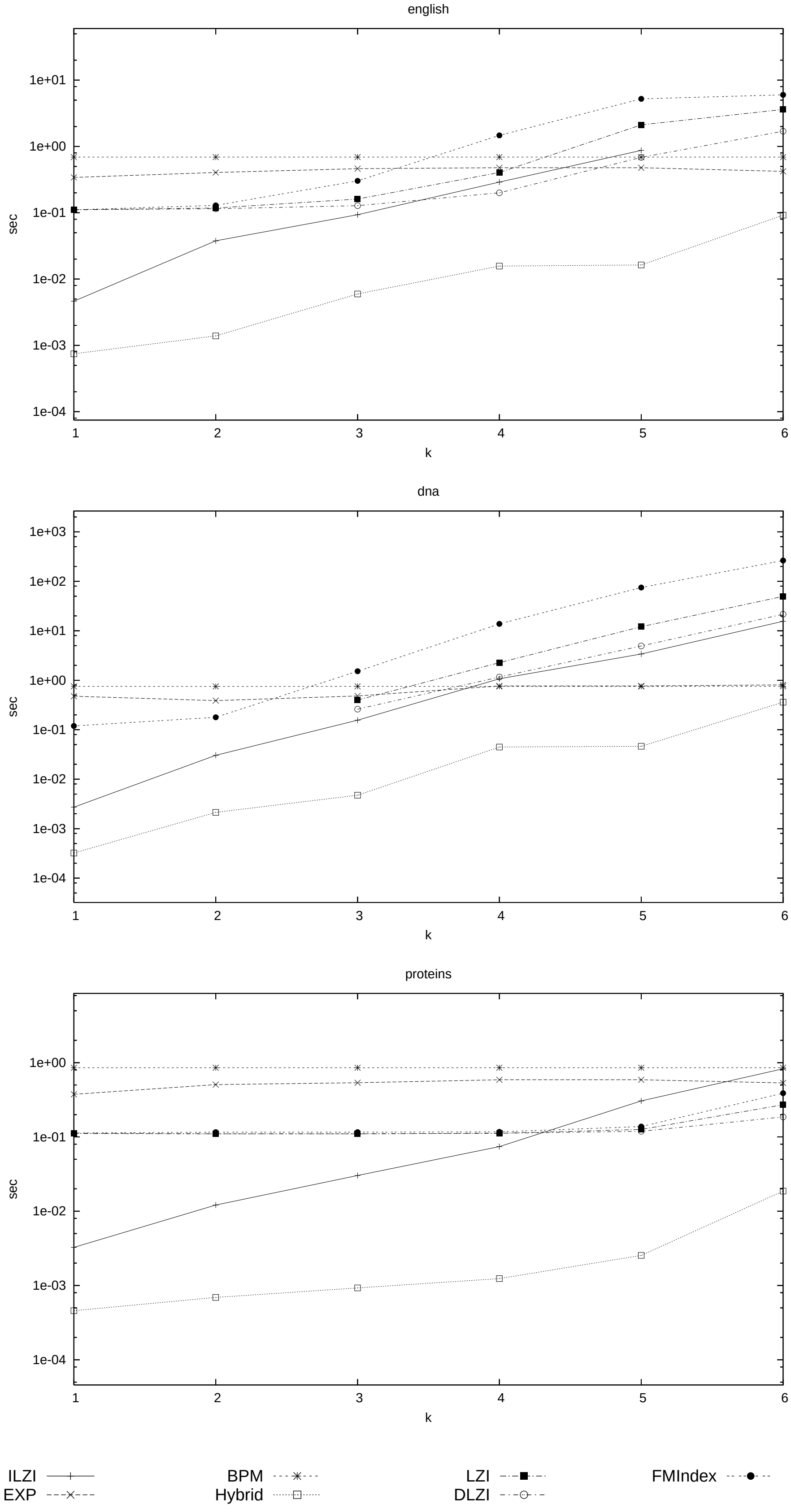


Fig. 6.2. Average user time, in seconds, for finding the occurrences of patterns of size 30 with k errors.



7

Conclusions

The central focus of the work described in this thesis is the study of Lempel-Ziv based compressed self-indexes. Compressed indexes are a relatively new form of full-text index that are becoming popular due to their remarkable performance. Moreover, index data structures are also the focus of a great deal of attention, due to potential applications in computational biology and information retrieval. In this chapter, we present a resume of what has been obtained and what is left to do.

7.1 Results Obtained

In this thesis, we proposed a new Lempel-Ziv compressed index, the ILZI. In Chapter 4 we note that the search time of Lempel-Ziv compressed indexes is affected by an $O(m^2)$ dependency on m. We explain that the Lempel-Ziv index of Ferragina and Manzini [37] does not completely address this problem, since, even though it obtains an $O(m)$ search time, it does so by using the FM-Index.

We clearly identify the reasons that cause the $O(m^2)$ time dependency. We propose the inverted Lempel-Ziv index that obtains an $O(m)$ time dependency by addressing these causes. The main cause for the $O(m^2)$ complexity is that the same string may appear in $O(m)$ different ways as the concatenation of LZ78 blocks and this is an undesirable property when searching for p. We proposed the maximal parsing to solve this problem.

We also pointed out that our dictionary $\mathcal{T}_{78}$ is a suffix tree. This property was previously unknown and hence unexplored.

Our approach reduces the dependency from $O(m^2)$ to $O(m)$, which represents a significant improvement. We argued theoretically that the underlying improvement should be around $(\log u)/H_k$. We implemented a prototype of the inverted Lempel-Ziv index and demonstrated empirically that our improvement was significant. We also introduced some important practical notions such as spurious entries (this concept is also of theoretical importance) and the LZ78 parsing with quorum, which were important to reduce the space requirements of the ILZI. In fact in practice the ILZI is smaller than the LZI (see Table 4.4).

We also proposed a new succinct representation of suffix trees (see lemma 4.2) that is efficient for our index. Our representation is based on the duality between suffix trees and tries and on the succinct representation of trees by Geary et al. [43] that is able to access the j-th ancestor of a node in constant time. This representation may be of independent interest. Note, however, that a naive implementation will not be by itself competitive, not even against classical approaches. Suppose we assume a naive implementation that uses a wavelet tree for storing R. This representation will require $(d \log d + 5d)(1 + o(1))$ bits. If we use the implementation of Rank and Select by Gonzalez et al. [47], we have that $(1 + o(1)) = 1,375$. Also if we assume that $\log d = 32$ and that $d = 2u$ this representation requires around $12.7u$ bytes. This is less competitive than the representation of Kurtz [67] and than representations of enhanced suffix arrays by Abouelhoda et al. [1]. However, assuming that $d = 2u$ may be too pessimistic. A simple test with 10 megabytes of English and DNA (from Pizza&Chili) shows empirically that $d = 1.7u$. Using these values this representation requires $10.8u$ bytes, which is competitive with Kurtz's representation but not so much with the representation of Abouelhoda et al.

In Chapter 5 we show that the functionality of the ILZI can be extended to solve the longest common substring problem (see Chapter 5). This shows that, in fact, compressed indexes based on the Lempel-Ziv data compression are extremely flexible. Note that this result is the best known algorithm for this problem using only $O(uH_k)$ bits. We also implemented a prototype to show that, in practice, we were able to obtain a linear dependency on m.

Chapter 6 describes a hybrid approach to perform approximate pattern matching with the ILZI. Our approach focused on adapting filtration techniques to the structure of the ILZI, instead of first using filtration and then using the ILZI as a black-box index. We demonstrated empirically that our approach was more efficient than this simpler approach, particularly for low error levels. This includes most typical searches. To achieve this objective, we introduced new algorithmic ideas. We abstracted some of the structure of the ILZI and explained it as a variable q-samples index. We pointed a variation of the filtration lemma that distributes errors according to the size of the samples. We introduced a new filtration technique where the samples that overlap the occurrence (not only those contained in the occurrence) can be considered. This is of interest even for classical q-sample indexes. We search for q-samples within long phrases to detect occurrences even if they are within a phrase. This technique also includes novel insights.

We believe that this thesis makes a valuable contribution to the field. We clearly pointed out a crucial problem in the performance of Lempel-Ziv based compressed indexes, explaining the causes of the phenomenon and proposing a way to solve it without using another data compression technique. Our work contributes to the overall understanding of Lempel-Ziv based indexes. Throughout this thesis we recurrently pointed out the parallelism between LZ based compressed indexes and inverted indexes. This relation also accounts for much of the flexibility of LZ based compressed indexes. In fact, we believe that it is possible to extend the functionality of the ILZI by importing results from inverted indexes. Inverted indexes have a number of desirable properties: they perform well in secondary memory and some work on dynamic inverted indexes already exists. Moreover, from our point of view, this parallelism may contribute greatly to a larger awareness of the inverted-LZ-index. From a high level point of view, the ILZI can simply be described as “an inverted index with a dictionary inferred by the LZ78 algorithm". Due to the popularity of inverted indexes and of the LZ data compression, this should immediately give a rough idea to a vast amount of computer science professionals and scientists. The practical approach we used should also be easy to grasp.

7.2 Future Work

Compressed indexes, in particular Lempel-Ziv based compressed indexes, are still being heavily researched. We believe a number of problems are important enough to require further research. The first is related with word based indexes. Adapting the Lempel-Ziv indexes to consider words as indivisible, instead of characters, may provide indexes that are more effective, both in terms of space and time, for natural language applications. In particular it is interesting to study if it can effectively replace inverted indexes.

Another interesting area of research is related with compressed suffix trees. Designing a Lempel-Ziv based compressed suffix tree representation is an important open problem in this area. Recall that Sadakane [109] proposed a representation of suffix trees (see Chapter 5), that supports all the operations that suffix trees support with reasonable performance, based in the compressed suffix array of Grossi, Gupta and Vitter [50]. However, it seems very challenging to obtain a representation that requires $O(uH_k)$ bits and allows moving from one node to another in less than $O(m)$ time. The main problem that we faced when trying to design such a representation is how to represent nodes, since a node in the suffix tree will be associated with $O(m)$ range queries in the ILZI. In Chapter 5 we were able to move through $O(m)$ nodes in $O(1)$ amortized time per transition, because the sequence of nodes to visit had a very regular structure. However, moving arbitrarily from one node to another requires $O(m)$ time in general.

The construction process also requires further attention. Compressed indexes are usually derived from an uncompressed one. Although this is usually a simple process, it is also a very space consuming approach. Arroyuelo et al. [7] have recently proposed a way to construct the LZI in a space efficient way. It is necessary to apply this type of research to the ILZI, mainly addressing the problems caused by spurious entries and maximal parsing.

Designing dynamic indexes is also an interesting challenge. Most indexes are static, meaning that they must be rebuilt from scratch upon text changes. This is currently a problem even on uncompressed full-text indexes. Note that this problem is related to the previous point, since having a dynamic compressed representation also permits us to construct the index with small space requirements. There is some recent work on dynamic compressed indexes [19, 35, 54, 77] and some interesting older work for inverted files [42, 70].

Finally obtaining indexes that operate efficiently in secondary memory is crucial for many applications. Secondary memory is a largely overlooked issue in compressed indexes. However, it is crucial to address this issue to obtain indexes that can handle very large text collections, as is the case of information retrieval systems. This is also a problem for uncompressed full-text indexes. A survey on full-text indexes in secondary memory was given by Kärkkäinen et al. [60]. Recently Arroyuelo et al. [5] proposed a secondary memory aware Lempel-Ziv index. We expect our analogy with inverted files to prove very useful in improving this kind of results.

A

Experimental Results

In this appendix we give more extensive experimental results.

The texts used for the experiments were obtained from the Pizza&Chili corpus:

- Sources (program source code). This file is formed by C/Java source code obtained by concatenating all the .c, .h, .C and .java files of the linux-2.6.11.6 and gcc-4.0.0 distributions. Downloaded on June 9, 2005.
- Pitches (MIDI pitch values). This file is a sequence of pitch values (bytes in 0-127, plus a few extra special values) obtained from a myriad of MIDI files freely available on Internet. The MIDI files were processed using semex 1.29 tool by Kjell Lemstrom, so as to convert them to IRP format. This is a human-readable tuple format, where the 5th column is the pitch value. Then the pitch values were coded in one byte each and concatenated. Downloaded during April 2005.
- Proteins (protein sequences). This file is a sequence of newline-separated protein sequences (without descriptions, just the bare proteins) obtained from the Swissprot database. Each of the 20 amino acids is coded as one uppercase letter. Updated on December 15, 2006.
- DNA (DNA sequences). This file is a sequence of newline-separated gene DNA sequences (without descriptions, just the bare DNA code) obtained from files 01hgp10 to 21hgp10, plus 0xhgp10 and 0yhgp10, from Gutenberg Project. Each of the 4 bases is coded as an

uppercase letter A,G,C,T, and there are a few occurrences of other special characters. Downloaded on June 9, 2005.

- English (English texts). This file is the concatenation of English text files selected from etext02 to etext05 collections of Gutenberg Project. We deleted the headers related to the project so as to leave just the real text. Downloaded on May 4, 2005.
- XML (structured text). This file is an XML that provides bibliographic information on major computer science journals and proceedings and it is obtained from dblp.uni-trier.de. Downloaded on September 27, 2005.

Some files were trimmed to 50 Megabytes. The other files already occupied approximately the same space. The files that were trimmed are referred with a 50MB suffix, as for example sources.50MB. The size of the other files can be consulted in table A.6 under Raw.

The information about the test files in Tables A.1,A.2 and A.3 was obtained from the Pizza&Chili site. The inverse probability of matching is 1 divided by the probability that two text characters chosen at random match. This is a measure of the effective alphabet size (on a uniformly distributed text, it is precisely the alphabet size). In Table A.6 we

Table A.1. Alphabet size and Inverse matching probability for the collections of the Pizza&Chili corpus.

Collection	Alphabet size	Inverse matching probability
Sources	230	24.77
Pitches	133	39.75
Proteins	27	17.02
DNA	16	3.91
English	239	15.25
XML	97	28.73

estimate the empirical entropy by using the number of blocks in the LZ78 parsing of T. However in Table A.2 we show the compressibility of the available texts using other methods. Compression ratios are all expressed as the percentage of the compressed text size divided by size of the original text. In particular, Gzip (v.1.3.3) gives an idea of compressibility via dictionaries, Bzip2 (v.1.0.3) gives an idea of block-sorting compressibility, and PPMDi gives an idea of Partial-Match-based compressors. Table A.3 gives the true value of the empirical entropy for different k's measured in number of bits per input symbol. In parentheses we give the corresponding compression ratio.

Table A.2. Compression ratios for the collections of the Pizza&Chili corpus.

Collection	Size (Mb)	gzip -1	gzip -9	bzip -1	bzip -9	ppmdi -l 0	ppmdi -l 9
SOURCES	50.00	28.95%	23.29%	22.18%	19.79%	19.35%	16.71%
PITCHES	53.25	30.59%	30.24%	34.62%	35.73%	29.81%	30.31%
PROTEINS	50.00	49.71%	47.39%	46.25%	45.56%	43.77%	42.05%
DNA	50.00	32.50%	27.05%	26.55%	25.98%	23.82%	24.31%
ENGLISH	50.00	44.90%	37.52%	32.85%	28.40%	29.91%	24.35%
XML	50.00	21,46%	17.23%	14.31%	11.22%	12.10%	9.21%

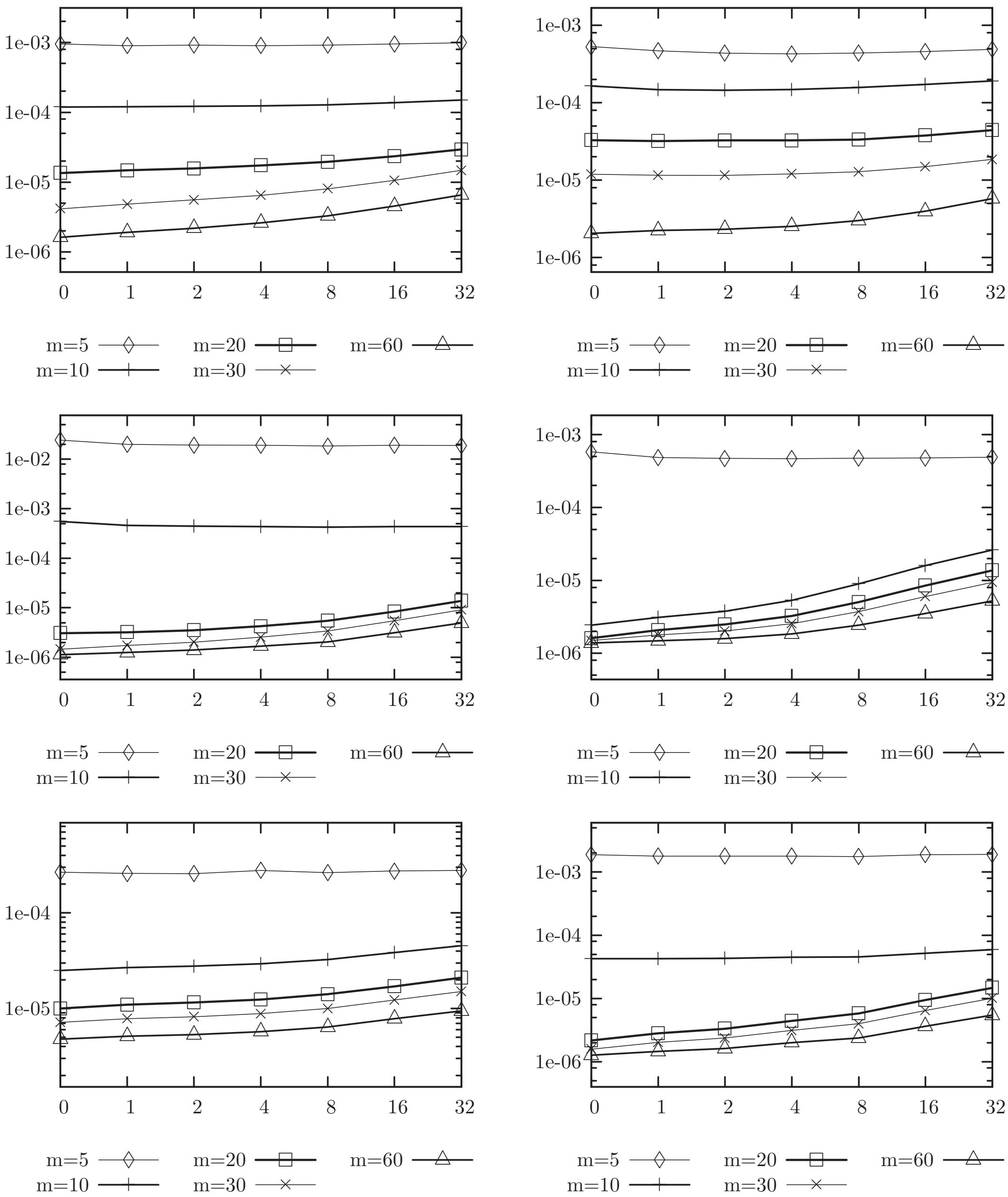
Table A.3. Empirical entropy for the collections of the Pizza&Chili corpus.

Collection	Size (Mb)	H0	H1	H2	H3	H4	H5
SOURCES	50.00	5.537 (69.21%)	4.038 (50.48%)	3.012 (37.65%)	2.214 (27.68%)	1.714 (21.43%)	1.372 (17.15%)
PITCHES	53.25	5.628 (70.35%)	4.716 (58.95%)	4.119 (51.49%)	3.443 (43.04%)	2.341 (29.26%)	1.275 (15.94%)
PROTEINS	50.00	4.195 (52.44%)	4.173 (52.16%)	4.146 (51.82%)	4.034 (50.43%)	3.728 (46.61%)	2.705 (33.81%)
DNA	50.00	1.982 (24.78%)	1.935 (24.19%)	1.925 (24.06%)	1.920 (24.00%)	1.913 (23.91%)	1.903 (23.79%)
ENGLISH	50.00	4.529 (56.61%)	3.606 (45.08%)	2.922 (36.53%)	2.386 (29.83%)	2.013 (25.16%)	1.764 (22.05%)
XML	50.00	5.230 (65.37%)	3.294 (41.17%)	2.007 (25.09%)	1.322 (16.53%)	0.956 (11.95%)	0.735 (9.19%)

A.1 The Inverted Lempel-Ziv Index

The results in this section are related to the algorithms described in chapter 4.

Fig. A.1. Time performance of the ILZI index for counting. The x-axis represents different quorum values. The results are given in seconds.



In Table A.6 we show the space requirements of different compressed indexes for the sample files. Variable i represents the size of the different indexes in bits. Therefore $i/2^{23}$ gives the size in Megabytes (MB), $i/u8$ gives the ratio with the original string, i/uH_k gives the ratio with a compressed string, where H_k is estimated as $(n \log u)/n$. The *par* line gives the parameters used for indexes that require it. The parameters were chosen so that the resulting index occupied approximately the same size as the ILZI. However, some minimal values were used for performance reasons. For the CSArray we give the D value, for CSAx8 we have that $L = 8 \times D$.

Figures A.2,A.3,A.4,A.5,A.6 show the time performance of different compressed indexes, the same results can be observed in Tables A.7 to A.12.

	sources.50MB			*dblp.xml.50MB*			*dna.50MB*		
l	$i/2^{23}$	$i/u8$	i/uH_k	$i/2^{23}$	$i/u8$	i/uH_k	$i/2^{23}$	$i/u8$	i/uH_k
32	50.0	1.00	2.80	26.9	0.54	2.73	30.9	0.62	2.24
16	48.0	0.96	2.69	24.8	0.50	2.52	31.3	0.63	2.27
8	46.6	0.93	2.61	24.2	0.48	2.46	33.0	0.66	2.39
4	48.6	0.97	2.72	24.1	0.48	2.45	37.0	0.74	2.68
2	53.5	1.07	3.00	26.1	0.52	2.65	44.0	0.88	3.19
1	59.6	1.19	3.34	28.9	0.58	2.93	52.5	1.05	3.81
0	87.4	1.75	4.90	42.5	0.85	4.32	92.6	1.85	6.72
	d/n	d'/n	f/n	d/n	d'/n	f/n	d/n	d'/n	f/n
	0.60	1.19	0.90	0.54	1.08	0.87	0.92	1.20	0.97
	total			*total*			*total*		
	2.70	+	1.79ϵ	2.49	+	1.62ϵ	3.09	+	2.12ϵ

	proteins			*pitches*			*english.50MB*		
l	$i/2^{23}$	$i/u8$	i/uH_k	$i/2^{23}$	$i/u8$	i/uH_k	$i/2^{23}$	$i/u8$	i/uH_k
32	85.9	1.35	2.54	75.8	1.42	2.83	47.6	0.95	2.63
16	85.6	1.34	2.53	73.9	1.39	2.76	46.4	0.93	2.56
8	87.5	1.37	2.59	72.6	1.36	2.71	45.8	0.92	2.53
4	93.7	1.47	2.77	76.4	1.43	2.85	48.5	0.97	2.68
2	102.8	1.61	3.04	84.7	1.59	3.16	54.3	1.09	2.99
1	120.4	1.89	3.56	97.9	1.84	3.65	61.8	1.24	3.41
0	226.5	3.55	6.69	161.7	3.04	6.04	93.3	1.87	5.15
	d/n	d'/n	f/n	d/n	d'/n	f/n	d/n	d'/n	f/n
	0.85	1.22	0.98	0.76	1.25	0.94	0.64	1.33	0.94
	total			*total*			*total*		
	3.04	+	2.07ϵ	2.95	+	2.01ϵ	2.90	+	1.96ϵ

Table A.4. Space requirements of the ILZI index for different quorum values. Variable l represents different quorum values. Variable i represents the size of the different indexes in bits. Therefore $i/2^{23}$ gives the size in Megabytes (MB), $i/u8$ gives the ratio with the original string, i/uH_k gives the ratio with a compressed string, where H_k is estimated as $(n \log u)/n$. The bottom part of the column shows empirical values for the space terms of our index, d/n, d'/n and f/n. Below total we show the empirical value of the space expression. Note that the factor associated with the ϵ counts the space of all the structures except the range data structure.

	dblp.xml.50MB							pitches						
$(\downarrow l)(m \rightarrow)$	5	10	20	30	40	50	60	5	10	20	30	40	50	60
32	4.91e-4	1.91e-4	4.41e-5	1.85e-5	1.17e-5	7.68e-6	5.78e-6	2.77e-4	4.53e-5	2.10e-5	1.51e-5	1.24e-5	1.06e-5	9.43e-6
16	4.58e-4	1.73e-4	3.76e-5	1.49e-5	8.64e-6	5.40e-6	3.96e-6	2.74e-4	3.85e-5	1.71e-5	1.23e-5	1.01e-5	8.84e-6	7.81e-6
8	4.38e-4	1.57e-4	3.33e-5	1.28e-5	7.43e-6	4.26e-6	3.01e-6	2.64e-4	3.25e-5	1.41e-5	9.98e-6	8.27e-6	7.21e-6	6.42e-6
4	4.25e-4	1.48e-4	3.26e-5	1.20e-5	6.49e-6	3.68e-6	2.53e-6	2.77e-4	2.94e-5	1.24e-5	8.83e-6	7.30e-6	6.38e-6	5.74e-6
2	4.37e-4	1.45e-4	3.25e-5	1.15e-5	6.18e-6	3.42e-6	2.32e-6	2.57e-4	2.77e-5	1.16e-5	8.20e-6	6.78e-6	5.92e-6	5.34e-6
1	4.69e-4	1.47e-4	3.19e-5	1.16e-5	6.10e-6	3.33e-6	2.23e-6	2.59e-4	2.67e-5	1.10e-5	7.83e-6	6.49e-6	5.67e-6	5.14e-6
0	5.30e-4	1.64e-4	3.28e-5	1.19e-5	6.01e-6	3.07e-6	2.05e-6	2.66e-4	2.49e-5	9.99e-6	7.15e-6	5.98e-6	5.25e-6	4.79e-6

	dna.50MB							proteins						
$(\downarrow l)(m \rightarrow)$	5	10	20	30	40	50	60	5	10	20	30	40	50	60
32	1.89e-2	4.35e-4	1.38e-5	9.12e-6	7.00e-6	5.75e-6	4.92e-6	4.89e-4	2.62e-5	1.37e-5	9.42e-6	7.31e-6	6.05e-6	5.22e-6
16	1.90e-2	4.37e-4	8.43e-6	5.46e-6	4.30e-6	3.59e-6	3.12e-6	4.79e-4	1.60e-5	8.49e-6	6.01e-6	4.75e-6	4.01e-6	3.50e-6
8	1.85e-2	4.24e-4	5.44e-6	3.41e-6	2.71e-6	2.31e-6	2.05e-6	4.73e-4	8.96e-6	5.04e-6	3.74e-6	3.09e-6	2.70e-6	2.43e-6
4	1.90e-2	4.37e-4	4.25e-6	2.56e-6	2.09e-6	1.83e-6	1.68e-6	4.66e-4	5.33e-6	3.24e-6	2.54e-6	2.19e-6	1.98e-6	1.84e-6
2	1.93e-2	4.44e-4	3.50e-6	2.01e-6	1.67e-6	1.50e-6	1.40e-6	4.72e-4	3.76e-6	2.48e-6	2.01e-6	1.80e-6	1.67e-6	1.59e-6
1	2.00e-2	4.59e-4	3.20e-6	1.74e-6	1.46e-6	1.33e-6	1.26e-6	4.85e-4	3.10e-6	2.08e-6	1.78e-6	1.63e-6	1.54e-6	1.48e-6
0	2.42e-2	5.49e-4	3.07e-6	1.46e-6	1.24e-6	1.16e-6	1.13e-6	5.81e-4	2.42e-6	1.61e-6	1.49e-6	1.44e-6	1.40e-6	1.38e-6

	english.50MB							sources.50MB						
$(\downarrow l)(m \rightarrow)$	5	10	20	30	40	50	60	5	10	20	30	40	50	60
32	1.90e-3	5.93e-5	1.48e-5	1.01e-5	7.76e-6	6.39e-6	5.46e-6	9.87e-4	1.50e-4	2.95e-5	1.48e-5	1.03e-5	8.01e-6	6.56e-6
16	1.89e-3	5.20e-5	9.49e-6	6.49e-6	5.07e-6	4.22e-6	3.66e-6	9.54e-4	1.38e-4	2.36e-5	1.06e-5	7.13e-6	5.53e-6	4.52e-6
8	1.75e-3	4.54e-5	5.82e-6	4.02e-6	3.18e-6	2.69e-6	2.38e-6	9.13e-4	1.28e-4	1.96e-5	8.05e-6	5.30e-6	4.06e-6	3.30e-6
4	1.79e-3	4.50e-5	4.45e-6	3.13e-6	2.55e-6	2.22e-6	2.01e-6	9.01e-4	1.24e-4	1.73e-5	6.45e-6	4.23e-6	3.21e-6	2.61e-6
2	1.78e-3	4.32e-5	3.34e-6	2.39e-6	1.99e-6	1.77e-6	1.63e-6	9.13e-4	1.22e-4	1.58e-5	5.58e-6	3.60e-6	2.71e-6	2.18e-6
1	1.79e-3	4.29e-5	2.82e-6	2.03e-6	1.73e-6	1.57e-6	1.47e-6	9.01e-4	1.20e-4	1.48e-5	4.85e-6	3.12e-6	2.36e-6	1.90e-6
0	1.89e-3	4.28e-5	2.19e-6	1.59e-6	1.42e-6	1.32e-6	1.27e-6	9.48e-4	1.19e-4	1.35e-5	4.14e-6	2.62e-6	2.00e-6	1.63e-6

Table A.5. Time performance of the ILZI for counting in seconds. Variable l represents different quorum values.

		Raw	ILZI	LZI	NFMI	CSAx8	LZI-7	SSA	RL	AFFMI	FMI2	SAC
sources.50MB	$i/2^{23}$	50.0	53.5	80.9	68.1	54.6	73.6	57.2	53.2	54.1	53.7	212.5
	$i/8u$	1.00	1.07	1.62	1.36	1.09	1.47	1.14	1.06	1.08	1.07	4.25
	i/uH_k	2.80	3.00	4.53	3.81	3.06	4.13	3.20	2.98	3.03	3.01	11.91
	par				5	7		512	26	36	0.17	
dblp.xml.50MB	$i/2^{23}$	50.0	26.1	44.5	64.9	25.8	40.4	54.6	33.6	32.9	28.7	212.5
	$i/8u$	1.00	0.52	0.89	1.30	0.52	0.81	1.09	0.67	0.66	0.57	4.25
	i/uH_k	5.08	2.65	4.52	6.60	2.62	4.11	5.55	3.42	3.35	2.91	21.60
	par				5	19		512	512	512	0.1	
dna.50MB	$i/2^{23}$	50.0	44.0	60.9	63.4	44.6	54.5	44.3	44.1	43.5	47.1	212.5
	$i/8u$	1.00	0.88	1.22	1.27	0.89	1.09	0.89	0.88	0.87	0.94	4.25
	i/uH_k	3.63	3.19	4.42	4.60	3.23	3.95	3.21	3.19	3.16	3.42	15.41
	par				5	11		24	64	24	0.17	
proteins	$i/2^{23}$	63.7	102.8	152.9	100.9	100.1	136.3	108.2	104.1		109.0	270.8
	$i/8u$	1.00	1.61	2.40	1.58	1.57	2.14	1.70	1.63		1.71	4.25
	i/uH_k	1.88	3.04	4.52	2.98	2.96	4.03	3.20	3.08		3.22	8.00
	par				10	6		10	12		0.27	
pitches	$i/2^{23}$	53.2	84.7	124.8	86.8	86.1	114.4	87.5	84.8		92.9	226.3
	$i/8u$	1.00	1.59	2.34	1.63	1.62	2.15	1.64	1.59		1.74	4.25
	i/uH_k	1.99	3.16	4.66	3.24	3.21	4.27	3.27	3.17		3.47	8.44
	par				9	5		16	12		0.26	
english.50MB	$i/2^{23}$	50.0	54.3	81.1	66.8	56.3	72.3	54.1	52.2	54.6	53.2	212.5
	$i/8u$	1.00	1.09	1.62	1.34	1.13	1.45	1.08	1.04	1.09	1.06	4.25
	i/uH_k	2.76	2.99	4.47	3.69	3.11	3.99	2.99	2.88	3.01	2.94	11.73
	par				5	7		64	30	24	0.17	

Table A.6. Table with the size of different compressed indexes for sample files. It shows the space requirements of different indexes, the original string (Raw), the Inverted-LZ-Index (ILZI), Navarro's LZ-index (LZI), Navarro's implementation of the FM-index (NFMI), Sadakane's CSArray (CSAx8), smaller LZ-index (LZI-7), the succinct suffix array (SSA), the run-length FM-index (RL), the alphabet friendly FM-index (AFFMI), the second version of the FM-index (FMI2), SAC is a suffix array in uncompressed form, packed in bits.

Fig. A.2. Time results for counting. These graphs shows the impact of our improvement. This can be observed by comparing the ILZI and LZI indexes. The graphs also show the fact that LZ based indexed cannot count in optimal time. However they do become competitive when m increases, causing occ to decrease.

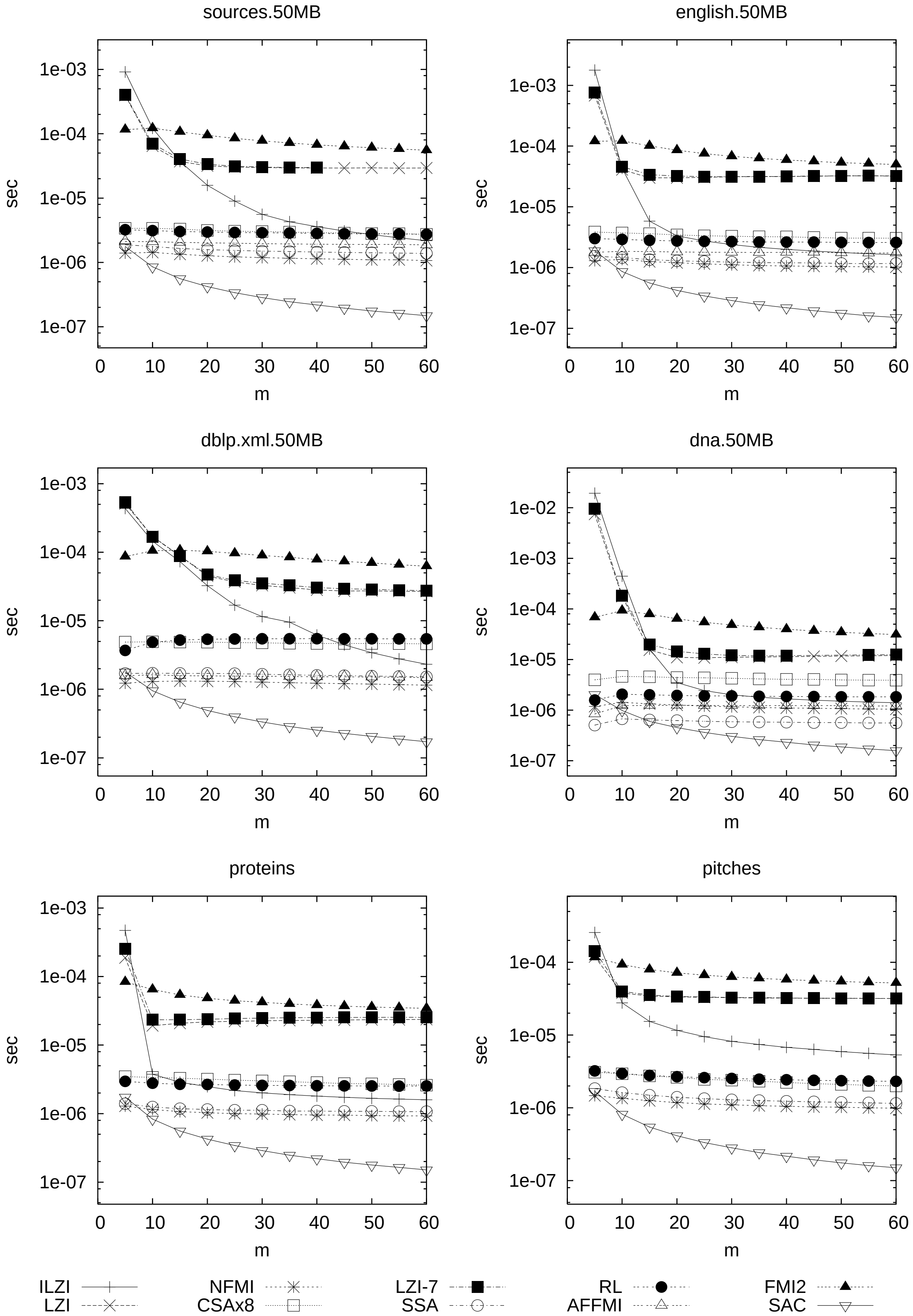


Fig. A.3. Time results for the total reporting time.

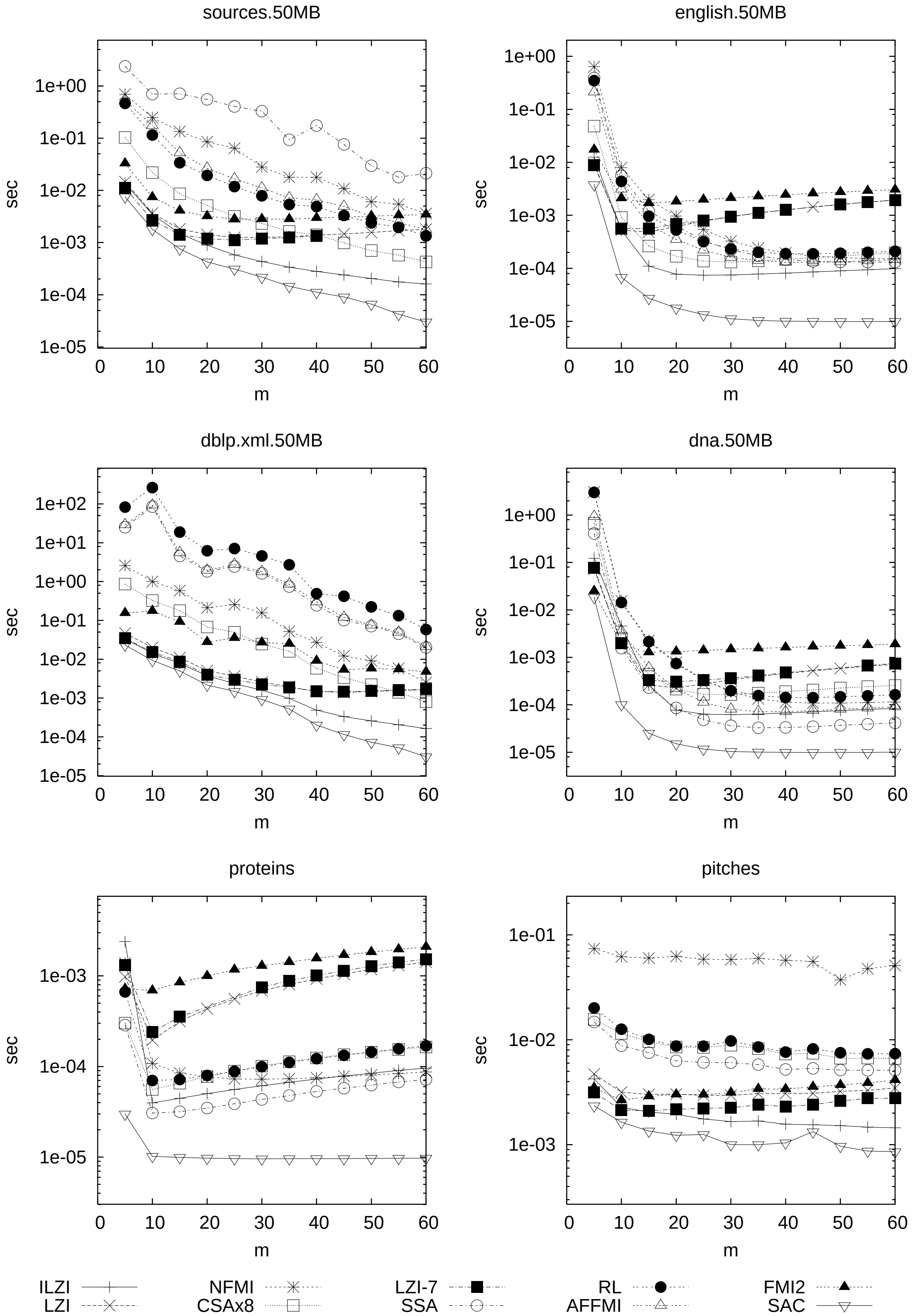


Fig. A.4. Time results for reporting factor (R). Theses graphs confirm that in fact LZ based indexes are the fastest at reporting occurrences. These results show that this factor is comparable to that of suffix arrays, being orders of magnitude faster than the alternatives.

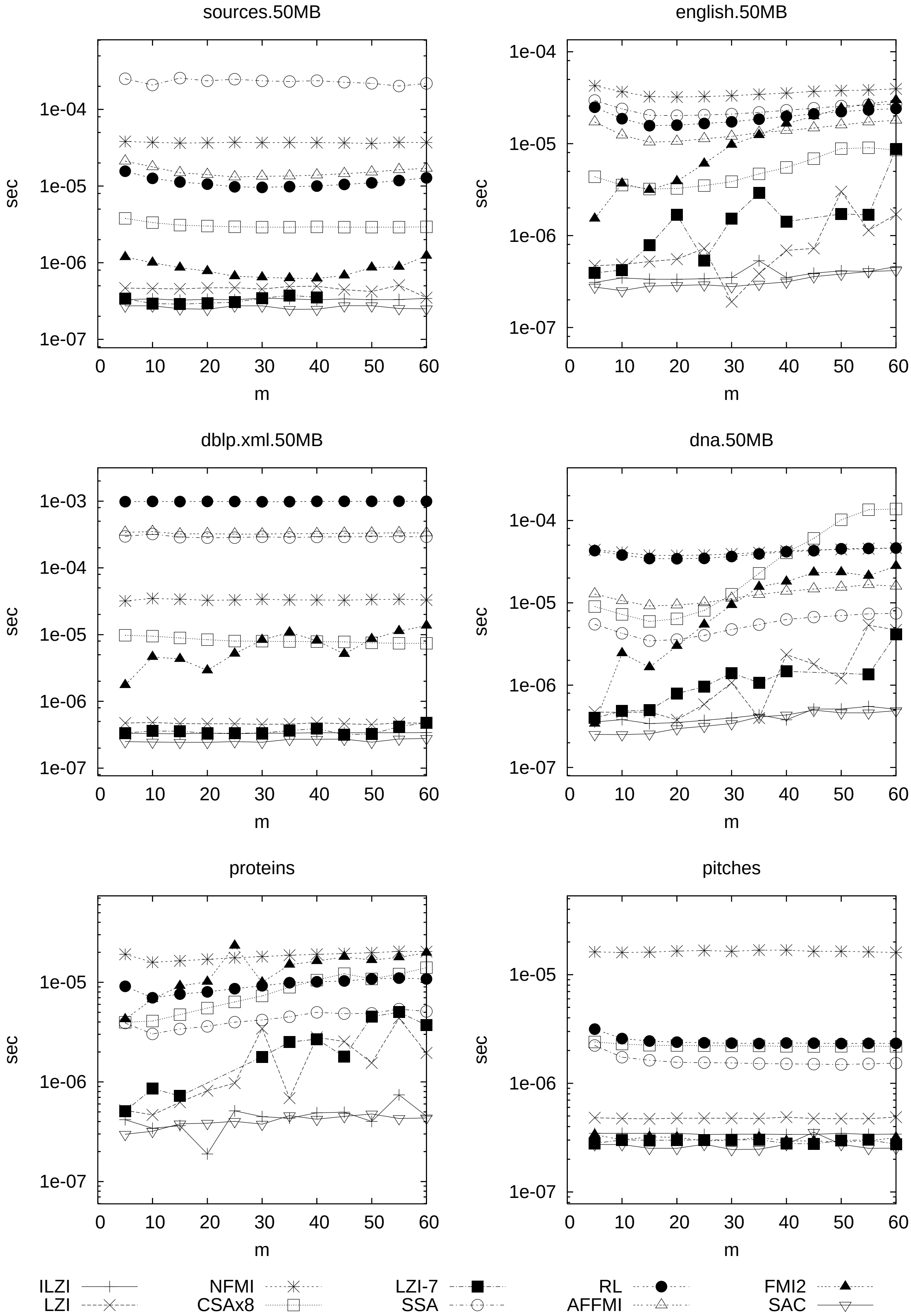


Fig. A.5. Time results for outputting.

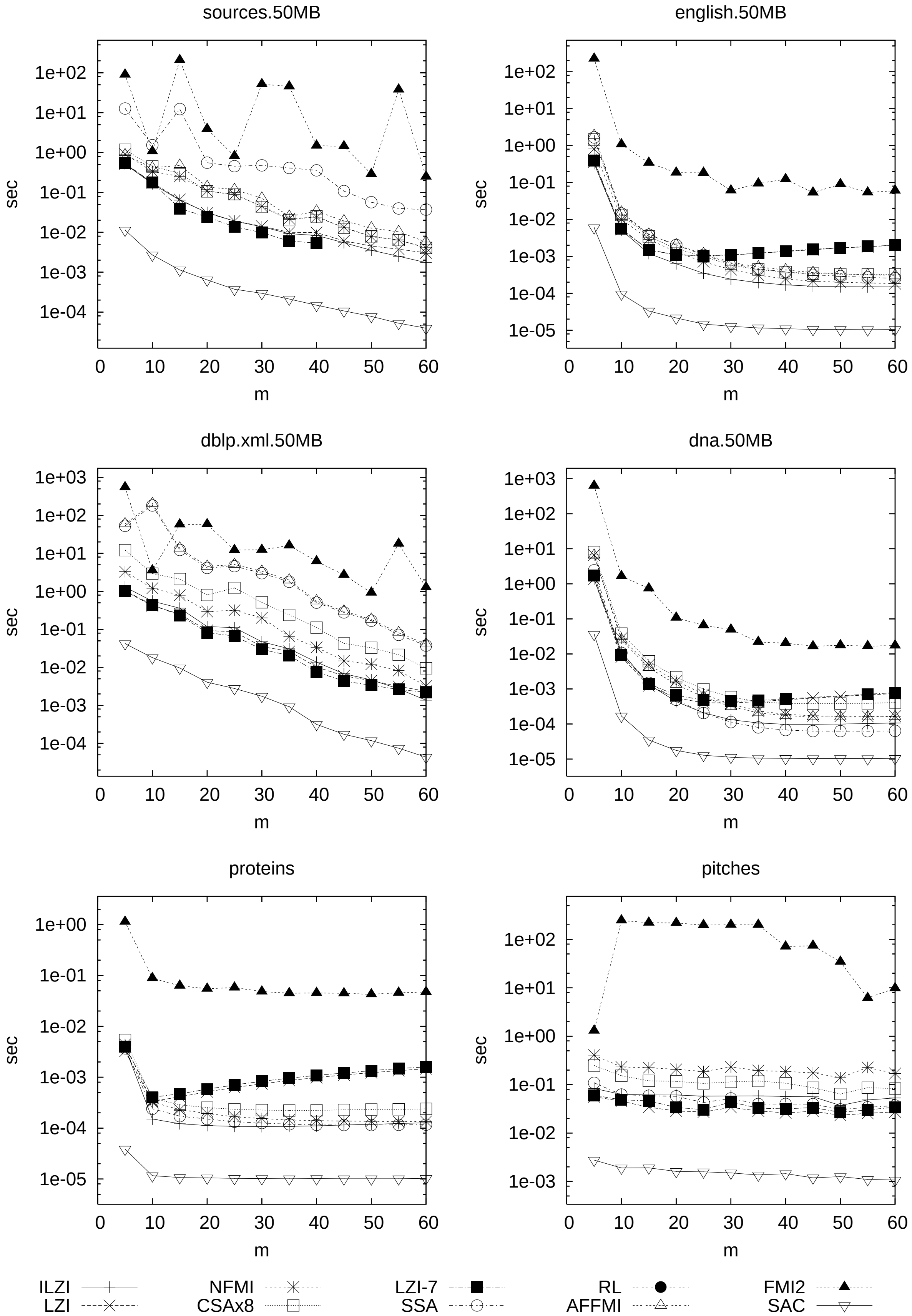


Fig. A.6. Time results for outputting factor (O). These results show that the ILZI is among the fastest compressed indexes at outputting.

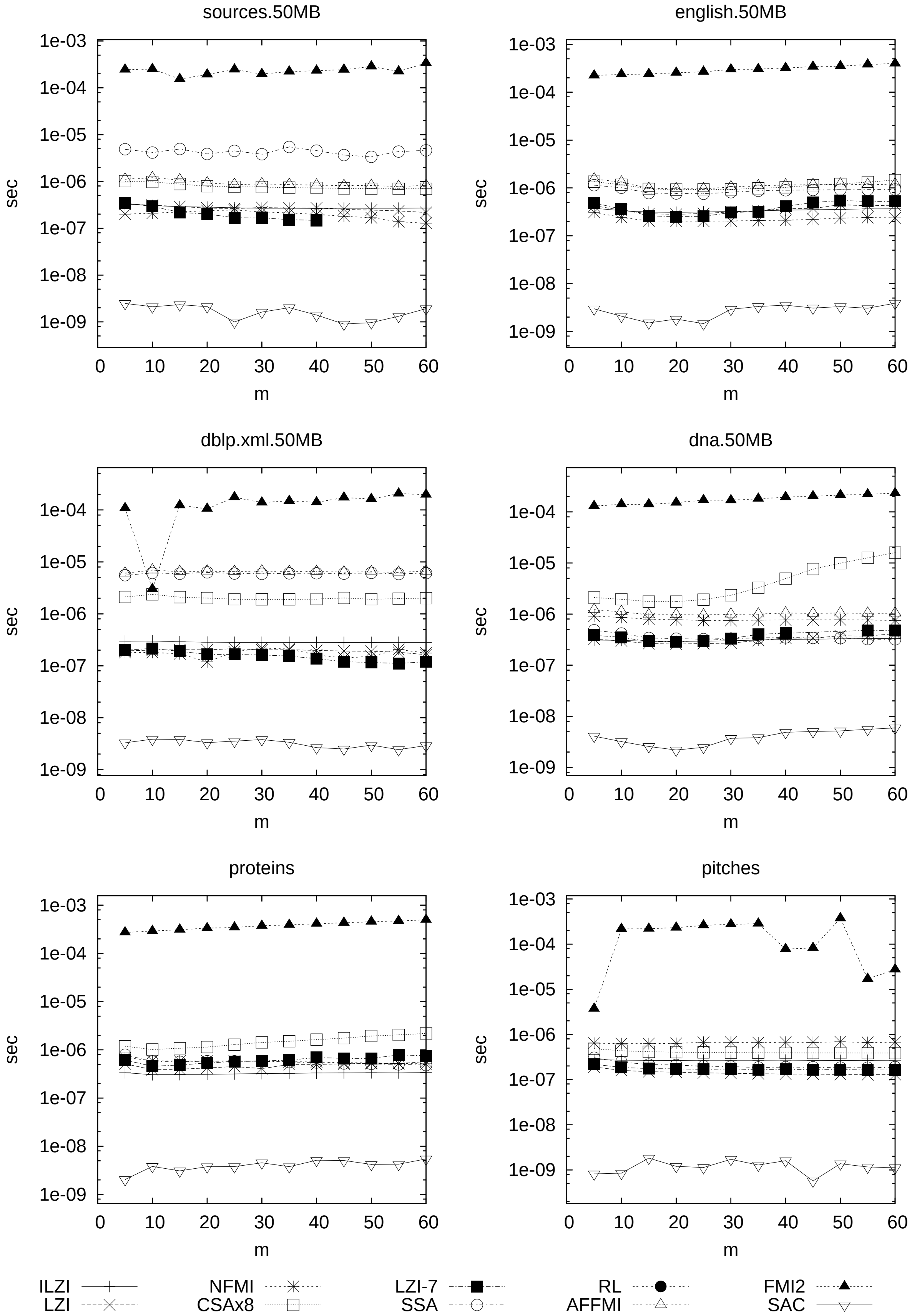


Table A.7. Time results for file dblp.xml.50MB in seconds. C is counting, R is the reporting factor, TR it the total reporting time, O is the outputting factor and TO is the total outputting time.

	m	ILZI	LZI	NFMI	CSAx8	LZI-7	SSA	RL	AFFMI	FMI2	SAC
C	5	(8) 4e-4	(9) 5e-4	(1) 1e-6	(6) 5e-6	(10) 5e-4	(3) 2e-6	(5) 4e-6	(2) 2e-6	(7) 9e-5	(4) 2e-6
	10	(8) 1e-4	(10) 2e-4	(2) 1e-6	(6) 5e-6	(9) 2e-4	(4) 2e-6	(5) 5e-6	(3) 2e-6	(7) 1e-4	(1) 1e-6
	20	(7) 3e-5	(8) 4e-5	(2) 1e-6	(5) 5e-6	(9) 5e-5	(4) 2e-6	(6) 5e-6	(3) 2e-6	(10) 1e-4	(1) 5e-7
	30	(7) 1e-5	(8) 3e-5	(2) 1e-6	(5) 5e-6	(9) 4e-5	(4) 2e-6	(6) 5e-6	(3) 2e-6	(10) 9e-5	(1) 3e-7
	40	(7) 6e-6	(8) 3e-5	(2) 1e-6	(5) 5e-6	(9) 3e-5	(4) 2e-6	(6) 5e-6	(3) 2e-6	(10) 8e-5	(1) 3e-7
	50	(5) 3e-6	(8) 3e-5	(2) 1e-6	(6) 5e-6	(9) 3e-5	(4) 2e-6	(7) 5e-6	(3) 2e-6	(10) 7e-5	(1) 2e-7
	60	(5) 2e-6	(8) 3e-5	(2) 1e-6	(6) 5e-6	(9) 3e-5	(4) 2e-6	(7) 5e-6	(3) 1e-6	(10) 6e-5	(1) 2e-7
R	5	(2) 3e-7	(4) 5e-7	(7) 3e-5	(6) 1e-5	(3) 3e-7	(8) 3e-4	(10) 1e-3	(9) 3e-4	(5) 2e-6	(1) 3e-7
	10	(2) 3e-7	(4) 5e-7	(7) 3e-5	(6) 9e-6	(3) 4e-7	(8) 3e-4	(10) 1e-3	(9) 3e-4	(5) 5e-6	(1) 2e-7
	20	(3) 3e-7	(4) 5e-7	(7) 3e-5	(6) 8e-6	(2) 3e-7	(8) 3e-4	(10) 1e-3	(9) 3e-4	(5) 3e-6	(1) 2e-7
	30	(2) 3e-7	(4) 5e-7	(7) 3e-5	(5) 8e-6	(3) 3e-7	(8) 3e-4	(10) 1e-3	(9) 3e-4	(6) 8e-6	(1) 2e-7
	40	(2) 3e-7	(4) 5e-7	(7) 3e-5	(5) 8e-6	(3) 4e-7	(8) 3e-4	(10) 1e-3	(9) 3e-4	(6) 8e-6	(1) 3e-7
	50	(3) 3e-7	(4) 5e-7	(7) 3e-5	(5) 8e-6	(2) 3e-7	(8) 3e-4	(10) 1e-3	(9) 3e-4	(6) 9e-6	(1) 2e-7
	60	(2) 3e-7	(3) 5e-7	(7) 3e-5	(5) 7e-6	(4) 5e-7	(8) 3e-4	(10) 1e-3	(9) 3e-4	(6) 1e-5	(1) 3e-7
TR	5	(2) 3e-2	(4) 5e-2	(7) 3e+0	(6) 9e-1	(3) 3e-2	(8) 3e+1	(10) 8e+1	(9) 3e+1	(5) 2e-1	(1) 2e-2
	10	(2) 1e-2	(4) 2e-2	(7) 1e-0	(6) 3e-1	(3) 2e-2	(8) 9e+1	(10) 3e+2	(9) 9e+1	(5) 2e-1	(1) 9e-3
	20	(2) 4e-3	(4) 5e-3	(7) 2e-1	(6) 7e-2	(3) 4e-3	(8) 2e+0	(10) 6e+0	(9) 2e+0	(5) 3e-2	(1) 2e-3
	30	(2) 2e-3	(4) 3e-3	(7) 2e-1	(5) 2e-2	(3) 2e-3	(8) 2e+0	(10) 5e+0	(9) 2e+0	(6) 3e-2	(1) 9e-4
	40	(2) 5e-4	(3) 1e-3	(7) 3e-2	(5) 6e-3	(4) 2e-3	(8) 2e-1	(10) 5e-1	(9) 3e-1	(6) 9e-3	(1) 2e-4
	50	(2) 3e-4	(3) 1e-3	(7) 9e-3	(5) 2e-3	(4) 2e-3	(8) 7e-2	(10) 2e-1	(9) 8e-2	(6) 6e-3	(1) 7e-5
	60	(2) 2e-4	(4) 2e-3	(6) 3e-3	(3) 8e-4	(5) 2e-3	(8) 2e-2	(10) 6e-2	(9) 2e-2	(7) 5e-3	(1) 3e-5
O	5	(5) 3e-7	(3) 2e-7	(2) 2e-7	(6) 2e-6	(4) 2e-7	(7) 6e-6		(8) 6e-6	(9) 1e-4	(1) 3e-9
	10	(5) 3e-7	(3) 2e-7	(2) 2e-7	(6) 2e-6	(4) 2e-7	(8) 6e-6		(9) 7e-6	(7) 3e-6	(1) 4e-9
	20	(5) 3e-7	(4) 2e-7	(2) 1e-7	(6) 2e-6	(3) 2e-7	(7) 6e-6		(8) 7e-6	(9) 1e-4	(1) 3e-9
	30	(5) 3e-7	(3) 2e-7	(4) 2e-7	(6) 2e-6	(2) 2e-7	(7) 6e-6		(8) 7e-6	(9) 1e-4	(1) 4e-9
	40	(5) 3e-7	(4) 2e-7	(3) 2e-7	(6) 2e-6	(2) 1e-7	(7) 6e-6		(8) 7e-6	(9) 1e-4	(1) 3e-9
	50	(5) 3e-7	(4) 2e-7	(3) 2e-7	(6) 2e-6	(2) 1e-7	(7) 6e-6		(8) 6e-6	(9) 2e-4	(1) 3e-9
	60	(5) 3e-7	(3) 2e-7	(4) 2e-7	(6) 2e-6	(2) 1e-7	(7) 6e-6		(8) 7e-6	(9) 2e-4	(1) 3e-9
TO	5	(4) 1e+0	(2) 1e+0	(5) 3e+0	(6) 1e+1	(3) 1e+0	(7) 5e+1		(8) 6e+1	(9) 6e+2	(1) 4e-2
	10	(4) 6e-1	(2) 4e-1	(5) 1e+0	(6) 3e+0	(3) 4e-1	(8) 2e+2		(9) 2e+2	(7) 4e+0	(1) 2e-2
	20	(4) 1e-1	(3) 9e-2	(5) 3e-1	(6) 8e-1	(2) 8e-2	(7) 4e+0		(8) 5e+0	(9) 6e+1	(1) 4e-3
	30	(4) 5e-2	(3) 4e-2	(5) 2e-1	(6) 5e-1	(2) 3e-2	(7) 3e+0		(8) 3e+0	(9) 1e+1	(1) 2e-3
	40	(4) 1e-2	(3) 1e-2	(5) 3e-2	(6) 1e-1	(2) 8e-3	(7) 5e-1		(8) 6e-1	(9) 6e+0	(1) 3e-4
	50	(4) 5e-3	(3) 4e-3	(5) 1e-2	(6) 3e-2	(2) 3e-3	(7) 2e-1		(8) 2e-1	(9) 9e-1	(1) 1e-4
	60	(2) 1e-3	(4) 2e-3	(5) 3e-3	(6) 9e-3	(3) 2e-3	(7) 4e-2		(8) 4e-2	(9) 1e+0	(1) 4e-5

Table A.8. Time results for file dna.50MB in seconds. C is counting, R is the reporting factor, TR it the total reporting time, O is the outputting factor and TO is the total outputting time.

	m	ILZI	LZI	NFMI	CSAx8	LZI-7	SSA	RL	AFFMI	FMI2	SAC
C	5	(10) 2e-2	(8) 7e-3	(3) 1e-6	(6) 4e-6	(9) 1e-2	(1) 5e-7	(4) 2e-6	(2) 8e-7	(7) 7e-5	(5) 2e-6
	10	(10) 4e-4	(8) 2e-4	(4) 1e-6	(6) 5e-6	(9) 2e-4	(1) 7e-7	(5) 2e-6	(3) 1e-6	(7) 9e-5	(2) 1e-6
	20	(6) 4e-6	(8) 1e-5	(4) 1e-6	(7) 4e-6	(9) 1e-5	(2) 6e-7	(5) 2e-6	(3) 1e-6	(10) 6e-5	(1) 5e-7
	30	(6) 2e-6	(8) 1e-5	(3) 1e-6	(7) 4e-6	(9) 1e-5	(2) 6e-7	(5) 2e-6	(4) 1e-6	(10) 5e-5	(1) 3e-7
	40	(5) 2e-6	(8) 1e-5	(3) 1e-6	(7) 4e-6	(9) 1e-5	(2) 6e-7	(6) 2e-6	(4) 1e-6	(10) 4e-5	(1) 2e-7
	50	(5) 2e-6	(8) 1e-5	(3) 1e-6	(7) 4e-6		(2) 6e-7	(6) 2e-6	(4) 1e-6	(9) 3e-5	(1) 2e-7
	60	(5) 1e-6	(8) 1e-5	(3) 1e-6	(7) 4e-6	(9) 1e-5	(2) 6e-7	(6) 2e-6	(4) 1e-6	(10) 3e-5	(1) 2e-7
R	5	(3) 4e-7	(5) 5e-7	(10) 4e-5	(7) 9e-6	(4) 4e-7	(6) 5e-6	(9) 4e-5	(8) 1e-5	(2) 3e-7	(1) 3e-7
	10	(2) 4e-7	(3) 5e-7	(10) 4e-5	(7) 7e-6	(4) 5e-7	(6) 4e-6	(9) 4e-5	(8) 1e-5	(5) 2e-6	(1) 3e-7
	20	(2) 4e-7	(3) 4e-7	(10) 4e-5	(7) 6e-6	(4) 8e-7	(6) 4e-6	(9) 3e-5	(8) 9e-6	(5) 3e-6	(1) 3e-7
	30	(2) 4e-7	(3) 1e-6	(10) 4e-5	(8) 1e-5	(4) 1e-6	(5) 5e-6	(9) 4e-5	(7) 1e-5	(6) 9e-6	(1) 3e-7
	40	(1) 4e-7	(4) 2e-6	(10) 4e-5	(8) 4e-5	(3) 1e-6	(5) 6e-6	(9) 4e-5	(6) 1e-5	(7) 2e-5	(2) 4e-7
	50	(2) 5e-7	(3) 1e-6	(7) 4e-5	(9) 1e-4		(4) 7e-6	(8) 5e-5	(5) 2e-5	(6) 2e-5	(1) 5e-7
	60	(2) 5e-7	(4) 5e-6	(8) 5e-5	(10) 1e-4	(3) 4e-6	(5) 7e-6	(9) 5e-5	(6) 2e-5	(7) 3e-5	(1) 5e-7
TR	5	(5) 1e-1	(3) 7e-2	(10) 3e+0	(7) 7e-1	(4) 8e-2	(6) 4e-1	(9) 3e+0	(8) 9e-1	(2) 3e-2	(1) 2e-2
	10	(8) 5e-3	(4) 2e-3	(10) 2e-2	(6) 3e-3	(5) 2e-3	(2) 2e-3	(9) 1e-2	(7) 4e-3	(3) 2e-3	(1) 1e-4
	20	(2) 8e-5	(6) 2e-4	(9) 8e-4	(4) 2e-4	(7) 3e-4	(3) 9e-5	(8) 7e-4	(5) 2e-4	(10) 1e-3	(1) 2e-5
	30	(3) 6e-5	(8) 3e-4	(6) 2e-4	(5) 2e-4	(9) 4e-4	(2) 4e-5	(7) 2e-4	(4) 8e-5	(10) 1e-3	(1) 1e-5
	40	(3) 7e-5	(8) 5e-4	(5) 1e-4	(7) 2e-4	(9) 5e-4	(2) 3e-5	(6) 1e-4	(4) 7e-5	(10) 2e-3	(1) 1e-5
	50	(3) 8e-5	(8) 6e-4	(5) 1e-4	(7) 2e-4		(2) 4e-5	(6) 1e-4	(4) 8e-5	(9) 2e-3	(1) 1e-5
	60	(3) 8e-5	(8) 7e-4	(5) 1e-4	(7) 3e-4	(9) 7e-4	(2) 4e-5	(6) 2e-4	(4) 9e-5	(10) 2e-3	(1) 1e-5
O	5	(2) 3e-7	(3) 3e-7	(6) 9e-7	(8) 2e-6	(4) 4e-7	(5) 5e-7		(7) 1e-6	(9) 1e-4	(1) 4e-9
	10	(3) 3e-7	(2) 3e-7	(6) 9e-7	(8) 2e-6	(4) 3e-7	(5) 4e-7		(7) 1e-6	(9) 1e-4	(1) 3e-9
	20	(4) 3e-7	(2) 3e-7	(6) 8e-7	(8) 2e-6	(3) 3e-7	(5) 3e-7		(7) 1e-6	(9) 2e-4	(1) 2e-9
	30	(3) 3e-7	(2) 3e-7	(6) 8e-7	(8) 2e-6	(4) 3e-7	(5) 3e-7		(7) 1e-6	(9) 2e-4	(1) 4e-9
	40	(2) 3e-7	(3) 3e-7	(6) 8e-7	(8) 5e-6	(5) 4e-7	(4) 3e-7		(7) 1e-6	(9) 2e-4	(1) 5e-9
	50	(2) 3e-7	(4) 4e-7	(5) 8e-7	(7) 1e-5		(3) 3e-7		(6) 1e-6	(8) 2e-4	(1) 5e-9
	60	(3) 3e-7	(4) 4e-7	(6) 8e-7	(8) 2e-5	(5) 5e-7	(2) 3e-7		(7) 1e-6	(9) 2e-4	(1) 6e-9
TO	5	(2) 1e+0	(3) 1e+0	(7) 7e+0	(8) 8e+0	(4) 2e+0	(5) 2e+0		(6) 6e+0	(9) 6e+2	(1) 4e-2
	10	(5) 1e-2	(2) 8e-3	(7) 3e-2	(8) 4e-2	(3) 9e-3	(4) 1e-2		(6) 3e-2	(9) 2e+0	(1) 2e-4
	20	(2) 4e-4	(4) 6e-4	(7) 2e-3	(8) 2e-3	(5) 7e-4	(3) 5e-4		(6) 1e-3	(9) 1e-1	(1) 2e-5
	30	(3) 1e-4	(6) 4e-4	(5) 4e-4	(8) 6e-4	(7) 4e-4	(2) 1e-4		(4) 3e-4	(9) 5e-2	(1) 1e-5
	40	(3) 1e-4	(7) 5e-4	(5) 2e-4	(6) 4e-4	(8) 5e-4	(2) 7e-5		(4) 2e-4	(9) 2e-2	(1) 1e-5
	50	(3) 1e-4	(7) 6e-4	(5) 2e-4	(6) 4e-4		(2) 6e-5		(4) 2e-4	(8) 2e-2	(1) 1e-5
	60	(3) 1e-4	(7) 7e-4	(5) 2e-4	(6) 4e-4	(8) 8e-4	(2) 6e-5		(4) 2e-4	(9) 2e-2	(1) 1e-5

Table A.9. Time results for file english.50MB in seconds. C is counting, R is the reporting factor, TR it the total reporting time, O is the outputting factor and TO is the total outputting time.

	m	ILZI	LZI	NFMI	CSAx8	LZI-7	SSA	RL	AFFMI	FMI2	SAC
C	5	(10) 2e-3	(8) 7e-4	(1) 1e-6	(6) 4e-6	(9) 8e-4	(2) 2e-6	(5) 3e-6	(3) 2e-6	(7) 1e-4	(4) 2e-6
	10	(8) 4e-5	(7) 4e-5	(2) 1e-6	(6) 4e-6	(9) 5e-5	(3) 1e-6	(5) 3e-6	(4) 2e-6	(10) 1e-4	(1) 9e-7
	20	(6) 3e-6	(8) 3e-5	(2) 1e-6	(7) 3e-6	(9) 3e-5	(3) 1e-6	(5) 3e-6	(4) 2e-6	(10) 9e-5	(1) 4e-7
	30	(5) 2e-6	(9) 3e-5	(2) 1e-6	(7) 3e-6	(8) 3e-5	(3) 1e-6	(6) 3e-6	(4) 2e-6	(10) 7e-5	(1) 3e-7
	40	(5) 2e-6	(8) 3e-5	(2) 1e-6	(7) 3e-6	(9) 3e-5	(3) 1e-6	(6) 3e-6	(4) 2e-6	(10) 6e-5	(1) 2e-7
	50	(5) 2e-6	(9) 3e-5	(2) 1e-6	(7) 3e-6	(8) 3e-5	(3) 1e-6	(6) 3e-6	(4) 2e-6	(10) 5e-5	(1) 2e-7
	60	(4) 2e-6	(9) 3e-5	(2) 1e-6	(7) 3e-6	(8) 3e-5	(3) 1e-6	(6) 3e-6	(5) 2e-6	(10) 5e-5	(1) 2e-7
R	5	(2) 3e-7	(4) 5e-7	(10) 4e-5	(6) 4e-6	(3) 4e-7	(9) 3e-5	(8) 2e-5	(7) 2e-5	(5) 2e-6	(1) 3e-7
	10	(2) 3e-7	(4) 5e-7	(10) 4e-5	(5) 4e-6	(3) 4e-7	(9) 2e-5	(8) 2e-5	(7) 1e-5	(6) 4e-6	(1) 3e-7
	20	(2) 3e-7	(3) 6e-7	(10) 3e-5	(5) 3e-6	(4) 2e-6	(9) 2e-5	(8) 2e-5	(7) 1e-5	(6) 4e-6	(1) 3e-7
	30	(3) 4e-7	(1) 2e-7	(10) 3e-5	(5) 4e-6	(4) 2e-6	(9) 2e-5	(8) 2e-5	(7) 1e-5	(6) 1e-5	(2) 3e-7
	40	(2) 3e-7	(3) 7e-7	(10) 4e-5	(5) 6e-6	(4) 1e-6	(9) 2e-5	(8) 2e-5	(6) 1e-5	(7) 2e-5	(1) 3e-7
	50	(2) 4e-7	(4) 3e-6	(10) 4e-5	(5) 9e-6	(3) 2e-6	(9) 3e-5	(7) 2e-5	(6) 2e-5	(8) 2e-5	(1) 4e-7
	60	(2) 5e-7	(3) 2e-6	(10) 4e-5	(4) 9e-6	(5) 9e-6	(8) 3e-5	(7) 2e-5	(6) 2e-5	(9) 3e-5	(1) 4e-7
TR	5	(4) 1e-2	(3) 9e-3	(10) 6e-1	(6) 5e-2	(2) 9e-3	(9) 4e-1	(8) 3e-1	(7) 2e-1	(5) 2e-2	(1) 4e-3
	10	(2) 5e-4	(3) 5e-4	(10) 8e-3	(5) 9e-4	(4) 6e-4	(9) 6e-3	(8) 4e-3	(7) 3e-3	(6) 2e-3	(1) 7e-5
	20	(2) 8e-5	(6) 6e-4	(9) 1e-3	(3) 2e-4	(8) 7e-4	(7) 6e-4	(5) 5e-4	(4) 3e-4	(10) 2e-3	(1) 2e-5
	30	(2) 7e-5	(8) 9e-4	(7) 3e-4	(3) 1e-4	(9) 9e-4	(5) 2e-4	(6) 2e-4	(4) 2e-4	(10) 2e-3	(1) 1e-5
	40	(2) 8e-5	(8) 1e-3	(7) 2e-4	(5) 1e-4	(9) 1e-3	(4) 1e-4	(6) 2e-4	(3) 1e-4	(10) 2e-3	(1) 1e-5
	50	(2) 9e-5	(9) 2e-3	(5) 2e-4	(6) 2e-4	(8) 2e-3	(3) 1e-4	(7) 2e-4	(4) 1e-4	(10) 3e-3	(1) 1e-5
	60	(2) 1e-4	(8) 2e-3	(5) 2e-4	(6) 2e-4	(9) 2e-3	(3) 1e-4	(7) 2e-4	(4) 1e-4	(10) 3e-3	(1) 1e-5
O	5	(3) 4e-7	(4) 4e-7	(2) 3e-7	(7) 1e-6	(5) 5e-7	(6) 1e-6		(8) 2e-6	(9) 2e-4	(1) 3e-9
	10	(3) 3e-7	(4) 3e-7	(2) 2e-7	(7) 1e-6	(5) 4e-7	(6) 1e-6		(8) 1e-6	(9) 2e-4	(1) 2e-9
	20	(5) 3e-7	(4) 3e-7	(2) 2e-7	(7) 9e-7	(3) 3e-7	(6) 8e-7		(8) 1e-6	(9) 3e-4	(1) 2e-9
	30	(5) 3e-7	(4) 3e-7	(2) 2e-7	(7) 9e-7	(3) 3e-7	(6) 8e-7		(8) 1e-6	(9) 3e-4	(1) 3e-9
	40	(3) 3e-7	(4) 4e-7	(2) 2e-7	(7) 1e-6	(5) 4e-7	(6) 9e-7		(8) 1e-6	(9) 3e-4	(1) 4e-9
	50	(3) 4e-7	(4) 4e-7	(2) 2e-7	(8) 1e-6	(5) 5e-7	(6) 9e-7		(7) 1e-6	(9) 4e-4	(1) 3e-9
	60	(3) 4e-7	(4) 4e-7	(2) 2e-7	(8) 1e-6	(5) 5e-7	(6) 9e-7		(7) 1e-6	(9) 4e-4	(1) 4e-9
TO	5	(2) 3e-1	(3) 4e-1	(5) 8e-1	(6) 1e+0	(4) 4e-1	(7) 2e+0		(8) 2e+0	(9) 2e+2	(1) 6e-3
	10	(2) 5e-3	(3) 5e-3	(5) 1e-2	(6) 1e-2	(4) 6e-3	(7) 1e-2		(8) 2e-2	(9) 1e+0	(1) 1e-4
	20	(2) 6e-4	(3) 1e-3	(5) 1e-3	(6) 2e-3	(4) 1e-3	(7) 2e-3		(8) 2e-3	(9) 2e-1	(1) 2e-5
	30	(2) 2e-4	(7) 1e-3	(3) 4e-4	(4) 6e-4	(8) 1e-3	(5) 6e-4		(6) 7e-4	(9) 6e-2	(1) 1e-5
	40	(2) 2e-4	(7) 1e-3	(3) 2e-4	(4) 4e-4	(8) 1e-3	(5) 4e-4		(6) 4e-4	(9) 1e-1	(1) 1e-5
	50	(2) 2e-4	(7) 2e-3	(3) 2e-4	(5) 3e-4	(8) 2e-3	(4) 3e-4		(6) 3e-4	(9) 9e-2	(1) 1e-5
	60	(2) 1e-4	(7) 2e-3	(3) 2e-4	(6) 3e-4	(8) 2e-3	(4) 3e-4		(5) 3e-4	(9) 6e-2	(1) 1e-5

Table A.10. Time results for file pitches in seconds. C is counting, R is the reporting factor, TR it the total reporting time, O is the outputting factor and TO is the total outputting time.

	m	ILZI	LZI	NFMI	CSAx8	LZI-7	SSA	RL	AFFMI	FMI2	SAC
C	5	(9) 3e-4	(7) 1e-4	(1) 1e-6	(4) 3e-6	(8) 1e-4	(3) 2e-6	(5) 3e-6		(6) 1e-4	(2) 2e-6
	10	(6) 3e-5	(7) 4e-5	(2) 1e-6	(4) 3e-6	(8) 4e-5	(3) 2e-6	(5) 3e-6		(9) 9e-5	(1) 8e-7
	20	(6) 1e-5	(7) 3e-5	(2) 1e-6	(4) 3e-6	(8) 3e-5	(3) 1e-6	(5) 3e-6		(9) 7e-5	(1) 4e-7
	30	(6) 8e-6	(7) 3e-5	(2) 1e-6	(4) 2e-6	(8) 3e-5	(3) 1e-6	(5) 3e-6		(9) 6e-5	(1) 3e-7
	40	(6) 7e-6	(7) 3e-5	(2) 1e-6	(4) 2e-6	(8) 3e-5	(3) 1e-6	(5) 2e-6		(9) 6e-5	(1) 2e-7
	50	(6) 6e-6	(8) 3e-5	(2) 1e-6	(4) 2e-6	(7) 3e-5	(3) 1e-6	(5) 2e-6		(9) 5e-5	(1) 2e-7
	60	(6) 5e-6	(8) 3e-5	(2) 1e-6	(4) 2e-6	(7) 3e-5	(3) 1e-6	(5) 2e-6		(9) 5e-5	(1) 1e-7
R	5	(4) 3e-7	(5) 5e-7	(9) 2e-5	(7) 2e-6	(2) 3e-7	(6) 2e-6	(8) 3e-6		(3) 3e-7	(1) 3e-7
	10	(4) 3e-7	(5) 5e-7	(9) 2e-5	(7) 2e-6	(3) 3e-7	(6) 2e-6	(8) 3e-6		(2) 3e-7	(1) 3e-7
	20	(4) 3e-7	(5) 5e-7	(9) 2e-5	(7) 2e-6	(2) 3e-7	(6) 2e-6	(8) 2e-6		(3) 3e-7	(1) 3e-7
	30	(4) 3e-7	(5) 5e-7	(9) 2e-5	(7) 2e-6	(3) 3e-7	(6) 2e-6	(8) 2e-6		(2) 3e-7	(1) 2e-7
	40	(4) 3e-7	(5) 5e-7	(9) 2e-5	(7) 2e-6	(2) 3e-7	(6) 2e-6	(8) 2e-6		(3) 3e-7	(1) 3e-7
	50	(4) 3e-7	(5) 5e-7	(9) 2e-5	(7) 2e-6	(3) 3e-7	(6) 1e-6	(8) 2e-6		(2) 3e-7	(1) 3e-7
	60	(4) 3e-7	(5) 5e-7	(9) 2e-5	(7) 2e-6	(2) 3e-7	(6) 2e-6	(8) 2e-6		(3) 3e-7	(1) 3e-7
TR	5	(4) 4e-3	(5) 5e-3	(9) 7e-2	(7) 2e-2	(2) 3e-3	(6) 1e-2	(8) 2e-2		(3) 3e-3	(1) 2e-3
	10	(3) 2e-3	(5) 3e-3	(9) 6e-2	(7) 1e-2	(2) 2e-3	(6) 9e-3	(8) 1e-2		(4) 3e-3	(1) 2e-3
	20	(2) 2e-3	(5) 3e-3	(9) 6e-2	(7) 9e-3	(3) 2e-3	(6) 6e-3	(8) 9e-3		(4) 3e-3	(1) 1e-3
	30	(2) 2e-3	(4) 3e-3	(9) 6e-2	(7) 9e-3	(3) 2e-3	(6) 6e-3	(8) 1e-2		(5) 3e-3	(1) 1e-3
	40	(2) 2e-3	(4) 3e-3	(9) 6e-2	(7) 7e-3	(3) 2e-3	(6) 5e-3	(8) 8e-3		(5) 3e-3	(1) 1e-3
	50	(2) 2e-3	(4) 3e-3	(9) 4e-2	(7) 7e-3	(3) 3e-3	(6) 5e-3	(8) 8e-3		(5) 4e-3	(1) 1e-3
	60	(2) 1e-3	(4) 3e-3	(9) 5e-2	(7) 7e-3	(3) 3e-3	(6) 5e-3	(8) 7e-3		(5) 4e-3	(1) 9e-4
O	5	(4) 3e-7	(2) 2e-7	(7) 6e-7	(6) 5e-7	(3) 2e-7	(5) 3e-7			(8) 4e-6	(1) 8e-10
	10	(5) 3e-7	(2) 2e-7	(7) 6e-7	(6) 4e-7	(3) 2e-7	(4) 2e-7			(8) 2e-4	(1) 8e-10
	20	(5) 3e-7	(2) 1e-7	(7) 6e-7	(6) 4e-7	(3) 2e-7	(4) 2e-7			(8) 2e-4	(1) 1e-9
	30	(5) 3e-7	(2) 1e-7	(7) 7e-7	(6) 4e-7	(3) 2e-7	(4) 2e-7			(8) 3e-4	(1) 2e-9
	40	(5) 3e-7	(2) 1e-7	(7) 7e-7	(6) 4e-7	(3) 2e-7	(4) 2e-7			(8) 8e-5	(1) 2e-9
	50	(5) 3e-7	(2) 1e-7	(7) 7e-7	(6) 4e-7	(3) 2e-7	(4) 2e-7			(8) 4e-4	(1) 1e-9
	60	(5) 3e-7	(2) 1e-7	(7) 7e-7	(6) 4e-7	(3) 2e-7	(4) 2e-7			(8) 3e-5	(1) 1e-9
TO	5	(4) 8e-2	(2) 6e-2	(7) 4e-1	(6) 2e-1	(3) 6e-2	(5) 1e-1			(8) 1e+0	(1) 3e-3
	10	(5) 6e-2	(2) 5e-2	(7) 2e-1	(6) 2e-1	(3) 5e-2	(4) 6e-2			(8) 2e+2	(1) 2e-3
	20	(5) 6e-2	(2) 3e-2	(7) 2e-1	(6) 1e-1	(3) 3e-2	(4) 6e-2			(8) 2e+2	(1) 2e-3
	30	(5) 6e-2	(2) 3e-2	(7) 2e-1	(6) 1e-1	(3) 4e-2	(4) 5e-2			(8) 2e+2	(1) 2e-3
	40	(5) 6e-2	(2) 3e-2	(7) 2e-1	(6) 1e-1	(3) 3e-2	(4) 4e-2			(8) 7e+1	(1) 1e-3
	50	(5) 4e-2	(2) 2e-2	(7) 1e-1	(6) 6e-2	(3) 3e-2	(4) 3e-2			(8) 3e+1	(1) 1e-3
	60	(5) 5e-2	(2) 3e-2	(7) 2e-1	(6) 8e-2	(3) 3e-2	(4) 4e-2			(8) 1e+1	(1) 1e-3

Table A.11. Time results for file proteins in seconds. C is counting, R is the reporting factor, TR it the total reporting time, O is the outputting factor and TO is the total outputting time.

	m	ILZI	LZI	NFMI	CSAx8	LZI-7	SSA	RL	AFFMI	FMI2	SAC
C	5	(9) 5e-4	(7) 2e-4	(1) 1e-6	(5) 3e-6	(8) 3e-4	(2) 1e-6	(4) 3e-6		(6) 8e-5	(3) 2e-6
	10	(6) 4e-6	(7) 2e-5	(2) 1e-6	(5) 3e-6	(8) 2e-5	(3) 1e-6	(4) 3e-6		(9) 6e-5	(1) 8e-7
	20	(4) 2e-6	(7) 2e-5	(2) 1e-6	(6) 3e-6	(8) 2e-5	(3) 1e-6	(5) 3e-6		(9) 5e-5	(1) 4e-7
	30	(4) 2e-6	(7) 2e-5	(2) 1e-6	(6) 3e-6	(8) 2e-5	(3) 1e-6	(5) 3e-6		(9) 4e-5	(1) 3e-7
	40	(4) 2e-6	(7) 2e-5	(2) 1e-6	(6) 3e-6	(8) 3e-5	(3) 1e-6	(5) 3e-6		(9) 4e-5	(1) 2e-7
	50	(4) 2e-6	(7) 2e-5	(2) 9e-7	(6) 3e-6	(8) 3e-5	(3) 1e-6	(5) 3e-6		(9) 4e-5	(1) 2e-7
	60	(4) 2e-6	(7) 2e-5	(2) 9e-7	(6) 3e-6	(8) 3e-5	(3) 1e-6	(5) 3e-6		(9) 3e-5	(1) 2e-7
R	5	(2) 4e-7	(4) 5e-7	(9) 2e-5	(6) 4e-6	(3) 5e-7	(5) 4e-6	(8) 9e-6		(7) 4e-6	(1) 3e-7
	10	(2) 3e-7	(3) 5e-7	(9) 2e-5	(6) 4e-6	(4) 9e-7	(5) 3e-6	(8) 7e-6		(7) 7e-6	(1) 3e-7
	20	(1) 2e-7	(3) 8e-7	(8) 2e-5	(5) 6e-6		(4) 4e-6	(6) 8e-6		(7) 1e-5	(2) 4e-7
	30	(2) 4e-7	(4) 3e-6	(9) 2e-5	(6) 7e-6	(3) 2e-6	(5) 4e-6	(7) 9e-6		(8) 1e-5	(1) 4e-7
	40	(2) 5e-7	(4) 3e-6	(9) 2e-5	(7) 1e-5	(3) 3e-6	(5) 5e-6	(6) 1e-5		(8) 2e-5	(1) 4e-7
	50	(1) 4e-7	(3) 2e-6	(9) 2e-5	(6) 1e-5	(4) 5e-6	(5) 5e-6	(7) 1e-5		(8) 2e-5	(2) 5e-7
	60	(2) 5e-7	(3) 2e-6	(9) 2e-5	(7) 1e-5	(4) 4e-6	(5) 5e-6	(6) 1e-5		(8) 2e-5	(1) 4e-7
TR	5	(9) 2e-3	(6) 1e-3	(8) 1e-3	(3) 3e-4	(7) 1e-3	(2) 3e-4	(4) 7e-4		(5) 7e-4	(1) 3e-5
	10	(3) 4e-5	(7) 2e-4	(6) 1e-4	(4) 6e-5	(8) 2e-4	(2) 3e-5	(5) 7e-5		(9) 7e-4	(1) 1e-5
	20	(3) 5e-5	(7) 4e-4	(4) 8e-5	(5) 8e-5		(2) 3e-5	(6) 8e-5		(8) 1e-3	(1) 1e-5
	30	(3) 6e-5	(7) 7e-4	(4) 7e-5	(6) 1e-4	(8) 7e-4	(2) 4e-5	(5) 1e-4		(9) 1e-3	(1) 1e-5
	40	(3) 7e-5	(7) 9e-4	(4) 8e-5	(6) 1e-4	(8) 1e-3	(2) 5e-5	(5) 1e-4		(9) 2e-3	(1) 1e-5
	50	(4) 8e-5	(7) 1e-3	(3) 8e-5	(5) 1e-4	(8) 1e-3	(2) 6e-5	(6) 1e-4		(9) 2e-3	(1) 1e-5
	60	(4) 1e-4	(7) 1e-3	(3) 9e-5	(5) 2e-4	(8) 2e-3	(2) 7e-5	(6) 2e-4		(9) 2e-3	(1) 1e-5
O	5	(2) 3e-7	(3) 5e-7	(5) 7e-7	(7) 1e-6	(4) 6e-7	(6) 8e-7			(8) 3e-4	(1) 2e-9
	10	(2) 3e-7	(3) 4e-7	(6) 6e-7	(7) 1e-6	(4) 5e-7	(5) 6e-7			(8) 3e-4	(1) 4e-9
	20	(2) 3e-7	(3) 4e-7	(6) 6e-7	(7) 1e-6	(4) 5e-7	(5) 6e-7			(8) 3e-4	(1) 4e-9
	30	(2) 3e-7	(3) 4e-7	(5) 6e-7	(7) 1e-6	(6) 6e-7	(4) 6e-7			(8) 4e-4	(1) 5e-9
	40	(2) 3e-7	(3) 5e-7	(5) 6e-7	(7) 2e-6	(6) 7e-7	(4) 5e-7			(8) 4e-4	(1) 5e-9
	50	(2) 3e-7	(4) 5e-7	(5) 5e-7	(7) 2e-6	(6) 7e-7	(3) 5e-7			(8) 5e-4	(1) 4e-9
	60	(2) 3e-7	(5) 6e-7	(4) 5e-7	(7) 2e-6	(6) 8e-7	(3) 5e-7			(8) 5e-4	(1) 5e-9
TO	5	(4) 4e-3	(2) 3e-3	(6) 4e-3	(7) 5e-3	(5) 4e-3	(3) 4e-3			(8) 1e+0	(1) 4e-5
	10	(2) 2e-4	(5) 3e-4	(4) 3e-4	(6) 4e-4	(7) 4e-4	(3) 2e-4			(8) 9e-2	(1) 1e-5
	20	(2) 1e-4	(6) 5e-4	(4) 2e-4	(5) 3e-4	(7) 6e-4	(3) 2e-4			(8) 5e-2	(1) 1e-5
	30	(2) 1e-4	(6) 7e-4	(4) 2e-4	(5) 2e-4	(7) 8e-4	(3) 1e-4			(8) 5e-2	(1) 1e-5
	40	(2) 1e-4	(6) 1e-3	(4) 1e-4	(5) 2e-4	(7) 1e-3	(3) 1e-4			(8) 5e-2	(1) 1e-5
	50	(3) 1e-4	(6) 1e-3	(4) 1e-4	(5) 2e-4	(7) 1e-3	(2) 1e-4			(8) 4e-2	(1) 1e-5
	60	(3) 1e-4	(6) 1e-3	(4) 1e-4	(5) 2e-4	(7) 2e-3	(2) 1e-4			(8) 5e-2	(1) 1e-5

Table A.12. Time results for file sources.50MB in seconds. C is counting, R is the reporting factor, TR it the total reporting time, O is the outputting factor and TO is the total outputting time.

	m	ILZI	LZI	NFMI	CSAx8	LZI-7	SSA	RL	AFFMI	FMI2	SAC
C	5	(10) 9e-4	(8) 4e-4	(1) 1e-6	(6) 3e-6	(9) 4e-4	(3) 2e-6	(5) 3e-6	(4) 2e-6	(7) 1e-4	(2) 2e-6
	10	(10) 1e-4	(7) 6e-5	(2) 1e-6	(6) 3e-6	(8) 7e-5	(3) 2e-6	(5) 3e-6	(4) 2e-6	(9) 1e-4	(1) 9e-7
	20	(7) 2e-5	(8) 3e-5	(2) 1e-6	(6) 3e-6	(9) 3e-5	(3) 2e-6	(5) 3e-6	(4) 2e-6	(10) 9e-5	(1) 4e-7
	30	(7) 6e-6	(8) 3e-5	(2) 1e-6	(6) 3e-6	(9) 3e-5	(3) 2e-6	(5) 3e-6	(4) 2e-6	(10) 8e-5	(1) 3e-7
	40	(7) 4e-6	(8) 3e-5	(2) 1e-6	(6) 3e-6	(9) 3e-5	(3) 1e-6	(5) 3e-6	(4) 2e-6	(10) 7e-5	(1) 2e-7
	50	(5) 3e-6	(8) 3e-5	(2) 1e-6	(7) 3e-6		(3) 1e-6	(6) 3e-6	(4) 2e-6	(9) 6e-5	(1) 2e-7
	60	(5) 2e-6	(8) 3e-5	(2) 1e-6	(7) 3e-6		(3) 1e-6	(6) 3e-6	(4) 2e-6	(9) 6e-5	(1) 1e-7
R	5	(2) 3e-7	(4) 5e-7	(9) 4e-5	(6) 4e-6	(3) 3e-7	(10) 3e-4	(7) 2e-5	(8) 2e-5	(5) 1e-6	(1) 3e-7
	10	(3) 3e-7	(4) 5e-7	(9) 4e-5	(6) 3e-6	(2) 3e-7	(10) 2e-4	(7) 1e-5	(8) 2e-5	(5) 1e-6	(1) 3e-7
	20	(3) 3e-7	(4) 5e-7	(9) 4e-5	(6) 3e-6	(2) 3e-7	(10) 2e-4	(7) 1e-5	(8) 1e-5	(5) 8e-7	(1) 2e-7
	30	(3) 3e-7	(4) 5e-7	(9) 4e-5	(6) 3e-6	(2) 3e-7	(10) 2e-4	(7) 1e-5	(8) 1e-5	(5) 6e-7	(1) 3e-7
	40	(2) 3e-7	(4) 5e-7	(9) 4e-5	(6) 3e-6	(3) 4e-7	(10) 2e-4	(7) 1e-5	(8) 1e-5	(5) 6e-7	(1) 2e-7
	50	(2) 3e-7	(3) 4e-7	(8) 4e-5	(5) 3e-6		(9) 2e-4	(6) 1e-5	(7) 2e-5	(4) 9e-7	(1) 3e-7
	60	(2) 3e-7	(3) 4e-7	(8) 4e-5	(5) 3e-6		(9) 2e-4	(6) 1e-5	(7) 2e-5	(4) 1e-6	(1) 2e-7
TR	5	(3) 1e-2	(4) 1e-2	(9) 7e-1	(6) 1e-1	(2) 1e-2	(10) 2e+0	(7) 5e-1	(8) 5e-1	(5) 3e-2	(1) 7e-3
	10	(3) 3e-3	(4) 4e-3	(9) 2e-1	(6) 2e-2	(2) 3e-3	(10) 7e-1	(7) 1e-1	(8) 2e-1	(5) 7e-3	(1) 2e-3
	20	(2) 9e-4	(4) 1e-3	(9) 9e-2	(6) 5e-3	(3) 1e-3	(10) 6e-1	(7) 2e-2	(8) 3e-2	(5) 3e-3	(1) 4e-4
	30	(2) 4e-4	(4) 1e-3	(9) 3e-2	(5) 2e-3	(3) 1e-3	(10) 3e-1	(7) 8e-3	(8) 1e-2	(6) 3e-3	(1) 2e-4
	40	(2) 3e-4	(4) 1e-3	(9) 2e-2	(5) 1e-3	(3) 1e-3	(10) 2e-1	(7) 5e-3	(8) 7e-3	(6) 3e-3	(1) 1e-4
	50	(2) 2e-4	(4) 2e-3	(8) 6e-3	(3) 7e-4		(9) 3e-2	(5) 2e-3	(6) 3e-3	(7) 3e-3	(1) 7e-5
	60	(2) 2e-4	(6) 2e-3	(8) 4e-3	(3) 4e-4		(9) 2e-2	(4) 1e-3	(5) 2e-3	(7) 3e-3	(1) 3e-5
O	5	(4) 3e-7	(3) 3e-7	(2) 2e-7	(6) 1e-6	(5) 3e-7	(8) 5e-6		(7) 1e-6	(9) 2e-4	(1) 2e-9
	10	(4) 3e-7	(5) 3e-7	(2) 2e-7	(6) 1e-6	(3) 3e-7	(8) 4e-6		(7) 1e-6	(9) 3e-4	(1) 2e-9
	20	(5) 3e-7	(4) 3e-7	(3) 2e-7	(6) 8e-7	(2) 2e-7	(8) 4e-6		(7) 9e-7	(9) 2e-4	(1) 2e-9
	30	(4) 3e-7	(5) 3e-7	(3) 2e-7	(6) 8e-7	(2) 2e-7	(8) 4e-6		(7) 9e-7	(9) 2e-4	(1) 2e-9
	40	(4) 3e-7	(5) 3e-7	(3) 2e-7	(6) 7e-7	(2) 1e-7	(8) 5e-6		(7) 8e-7	(9) 2e-4	(1) 1e-9
	50	(4) 3e-7	(3) 2e-7	(2) 2e-7	(5) 7e-7		(7) 3e-6		(6) 8e-7	(8) 3e-4	(1) 1e-9
	60	(4) 3e-7	(3) 2e-7	(2) 1e-7	(5) 7e-7		(7) 5e-6		(6) 8e-7	(8) 3e-4	(1) 2e-9
TO	5	(3) 5e-1	(2) 5e-1	(6) 9e-1	(7) 1e+0	(4) 5e-1	(8) 1e+1		(5) 9e-1	(9) 9e+1	(1) 1e-2
	10	(2) 2e-1	(3) 2e-1	(5) 3e-1	(7) 4e-1	(4) 2e-1	(9) 2e+0		(6) 4e-1	(8) 1e+0	(1) 3e-3
	20	(4) 3e-2	(3) 3e-2	(6) 1e-1	(5) 1e-1	(2) 2e-2	(8) 6e-1		(7) 1e-1	(9) 4e+0	(1) 6e-4
	30	(3) 1e-2	(4) 1e-2	(6) 4e-2	(5) 4e-2	(2) 1e-2	(8) 5e-1		(7) 7e-2	(9) 5e+1	(1) 3e-4
	40	(3) 8e-3	(4) 1e-2	(5) 2e-2	(6) 2e-2	(2) 5e-3	(8) 4e-1		(7) 3e-2	(9) 1e+0	(1) 1e-4
	50	(2) 4e-3	(3) 5e-3	(4) 8e-3	(5) 8e-3		(7) 6e-2		(6) 1e-2	(8) 3e-1	(1) 8e-5
	60	(2) 2e-3	(3) 3e-3	(5) 4e-3	(4) 4e-3		(7) 4e-2		(6) 6e-3	(8) 2e-1	(1) 4e-5

Table A.13. Average user time, in seconds, that the ILZI takes to find occurrences of patterns of size 30 with k errors, using different v's.

english.50MB

$k \downarrow, \rightarrow v$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1e+0	2e-1	4e-2	2e-2	8e-3	5e-3	5e-3	6e-3	9e-3	1e-2	2e-2	2e-2	3e-2	4e-2	6e-2
2	2e+0	3e-1	9e-2	4e-2	4e-2	4e-2	4e-2	4e-2	5e-2	9e-2	1e-1	2e-1	3e-1	7e-1	
3	1e+0	5e-1	1e-1	1e-1	9e-2	1e-1	2e-1	2e-1	3e-1	5e-1	8e-1	2e+0	4e+0	6e+0	
4	2e+0	8e-1	4e-1	3e-1	4e-1	5e-1	7e-1	1e+0	2e+0	3e+0	6e+0	2e+1	2e+1		
5	1e+1	1e+1	1e+1	1e+1	1e+1	2e+1	2e+1	2e+1	3e+1	1e+2	3e+2	3e+2	3e+2		
6	2e+1	2e+1	2e+1	2e+1	2e+1	3e+1	4e+1	6e+1	2e+2	4e+2	4e+2	4e+2			

dna.50MB

$k \downarrow, \rightarrow v$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	6e+0	1e+0	4e-1	1e-1	3e-2	9e-3	4e-3	3e-3	3e-3	3e-3	5e-3	7e-3	8e-3	1e-2	1e-2
2	1e+1	3e+0	9e-1	2e-1	8e-2	4e-2	3e-2	3e-2	4e-2	6e-2	1e-1	2e-1	2e-1	3e-1	
3	9e+0	5e+0	1e+0	5e-1	2e-1	2e-1	3e-1	4e-1	5e-1	1e+0	2e+0	2e+0	3e+0	3e+0	
4	1e+1	7e+0	3e+0	1e+0	1e+0	1e+0	1e+0	3e+0	5e+0	9e+0	1e+1	2e+1	2e+1		
5	2e+1	1e+1	1e+1	1e+1	1e+1	2e+1	3e+1	7e+1	1e+2	2e+2	2e+2	2e+2	2e+2		
6	4e+1	3e+1	3e+1	3e+1	4e+1	6e+1	2e+2	3e+2	3e+2	3e+2	3e+2	3e+2			

proteins

$k \downarrow, \rightarrow v$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3e+0	1e-1	1e-2	4e-3	3e-3	3e-3	3e-3	5e-3	7e-3	1e-2	1e-2	1e-2	2e-2	3e-2	4e-2
2	3e+0	2e-1	2e-2	1e-2	2e-2	2e-2	2e-2	2e-2	5e-2	1e-1	2e-1	2e-1	6e-1	1e+0	
3	2e+0	4e-1	4e-2	3e-2	3e-2	5e-2	2e-1	3e-1	5e-1	9e-1	2e+0	5e+0	1e+1	2e+1	
4	3e+0	6e-1	7e-2	2e-1	4e-1	6e-1	8e-1	2e+0	4e+0	7e+0	2e+1	7e+1	1e+2		
5	3e+1	3e+1	3e+1	3e+1	3e+1	4e+1	4e+1	5e+1	5e+1	1e+2	1e+3	1e+3	1e+3		
6	4e+1	4e+1	4e+1	4e+1	4e+1	5e+1	6e+1	1e+2	2e+2	1e+3	1e+3	1e+3			

A.2 Finding Longest Common Sub-Strings

Figure A.7 gives more extensive results for the algorithms described in chapter 5.

A.3 Approximate String Matching

The results in this section are related to algorithms described in chapter 6.

Table A.14. Tables for $m = 24$, $k = 9$, $v = 1$ and $q = d = x_2 - x_1 = 4$. The table on the left refers to blocks of type O_1, i.e. prefixes of O that are suffixes of samples, and the table on the right to blocks of type O_i, i.e. samples in the middle of O. Note that using the approach of Navarro et al. [101] in this example yields 3 errors per sample.

$\downarrow y_2 - y_1$	$\lceil k \cdot \lvert O_1 \rvert_v \rceil - 1$					$\downarrow y_2 - y_1$	$\lceil k \cdot \lvert O_i \rvert_v \rceil - 1$																								
$\lvert O_1 \rvert \rightarrow$	0	1	2	3	4	$y_1 \rightarrow$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	-1	-1	-1	-1	-1	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	0	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	-1	-1
3	-1	-1	-1	1	2	3	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	-1	-1	-1
4	-1	-1	-1	1	2	4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	-1	-1	-1	-1
5	-1	-1	-1	-1	2	5	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	-1	-1	-1	-1	-1
6	-1	-1	-1	-1	2	6	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	-1	-1	-1	-1	-1	-1
7	-1	-1	-1	-1	-1	7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Table A.15. Tables for $m = 24$, $k = 9$, $v = 2$ and $q = d = x_2 - x_1 = 4$. The table on the left refers to blocks of type O_1, i.e. prefixes of O that are suffixes of samples, and the table on the right to blocks of type O_i, i.e. samples in the middle of O. Note that using $v = 2$ is excessive since too many errors from the extremes are pushed into the middle of O.

$\downarrow y_2 - y_1$	$\lceil k \cdot \lvert O_1 \rvert_v \rceil - 1$					$\downarrow y_2 - y_1$	$\lceil k \cdot \lvert O_i \rvert_v \rceil - 1$																								
$\lvert O_1 \rvert \rightarrow$	0	1	2	3	4	$y_1 \rightarrow$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	-1	-1	-1	-1	-1	1	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	-1	-1
3	-1	-1	-1	0	1	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	-1	-1	-1
4	-1	-1	-1	-1	1	4	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2	2	-1	-1	-1	-1
5	-1	-1	-1	-1	1	5	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2	-1	-1	-1	-1	-1
6	-1	-1	-1	-1	-1	6	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	-1	-1	-1	-1	-1	-1
7	-1	-1	-1	-1	-1	7	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	-1	-1	-1	-1	-1	-1	-1	-1
8	-1	-1	-1	-1	-1	8	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Tables A.14 and A.15 show examples of the tight backtracking bound values for q-samples, i.e. when $q = x_2 - x_1 = 4$.

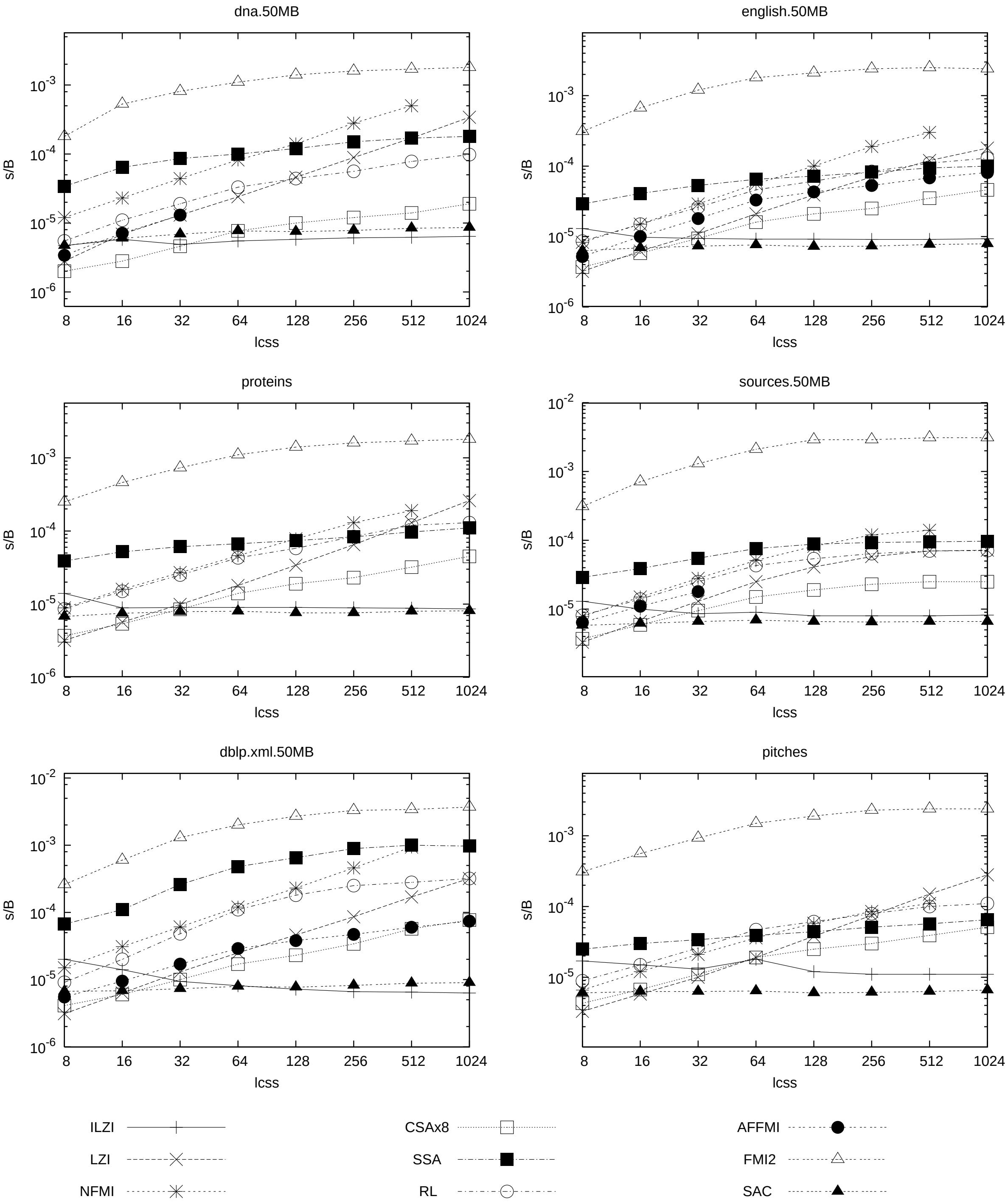


Fig. A.7. Average user time of processing a byte for different lcss values.

References

[1] M. Abouelhoda, S. Kurtz, E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] M. I. Abouelhoda, E. Ohlebusch, S. Kurtz. Optimal exact string matching based on suffix arrays. In Proceedings of *SPIRE*, volume 2476 of *LNCS*, pages 31–43, 2002.

[3] S. Alstrup, G. Brodal, T. Rauhe. New data structures for orthogonal range searching. In Proceedings of *FOCS*, pages 198–207, 2000.

[4] A. Andersson, S. Nilsson. Efficient implementation of suffix trees. *Softw., Pract. Exper.*, 25(2):129–141, 1995.

[5] D. Arroyuelo, G. Navarro. A lempel-ziv text index on secondary storage. In Proceedings of *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM 2007)*, LNCS 4580, pages 83–94, 2007.

[6] D. Arroyuelo, G. Navarro, K. Sadakane. Reducing the space requirement of LZ-index. In Proceedings of *CPM*, LNCS 4009, pages 319–330, 2006.

[7] D. Arroyuelo, G. Navarro. Space-efficient construction of LZ-index. In Proceedings of *ISAAC*, volume 3827 of *LNCS*, pages 1143–1152, 2005.

[8] R. A. Baeza-Yates. Text-retrieval: Theory and practice. In Proceedings of *IFIP Congress (1)*, volume A-12 of *IFIP Transactions*, pages 465–476, 1992.

[9] R. A. Baeza-Yates. A unified view to string matching algorithms. In Proceedings of *SOFSEM*, volume 1175 of *LNCS*, pages 1–15, 1996.

[10] R. A. Baeza-Yates, G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

[11] H. Bannai, S. Inenaga, A. Shinohara, M. Takeda. Inferring strings from graphs and arrays. In Proceedings of *MFCS*, volume 2747 of *LNCS*, pages 208–217, 2003.

[12] T. Bell, J. Cleary, I. Witten. *Text compression.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1990.

[13] M. A. Bender, M. Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004.

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

[15] O. Berkman, U. Vishkin. Finding level-ancestors in trees. *J. Comput. Syst. Sci.*, 48(2):214–230, 1994.

[16] A. Blumer, J. Blumer, D. Haussler, R. M. McConnell, A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, 1987.

[17] S. Burkhardt, J. Kärkkäinen. Fast lightweight suffix array construction and checking. In Proceedings of *CPM*, volume 2676 of *LNCS*, pages 55–69, 2003.

[18] M. Burrows, D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, DEC SRC, 1994.

[19] H.-L. Chan, W.-K. Hon, T. W. Lam. Compressed index for a dynamic collection of texts. In Proceedings of *CPM*, volume 3109 of *LNCS*, pages 445–456, 2004.

[20] H.-L. Chan, T. W. Lam, W.-K. Sung, S.-L. Tam, S.-S. Wong. A linear size index for approximate pattern matching. In Proceedings of *CPM*, volume 4009 of *LNCS*, pages 49–59, 2006.

[21] W. I. Chang, T. G. Marr. Approximate string matching and local similarity. In Proceedings of *CPM*, volume 807 of *LNCS*, pages 259–273, 1994.

[22] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988.

[23] D. Clark, J. Munro. Efficient suffix trees on secondary storage. *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 383–391, 1996.

[24] A. L. Cobbs. Fast approximate matching using suffix trees. In Proceedings of *CPM*, volume 937 of *LNCS*, pages 41–54, 1995.

[25] L. P. Coelho, A. L. Oliveira. Dotted suffix trees a structure for approximate text indexing. In Proceedings of *SPIRE*, volume 4209 of *LNCS*, pages 329–336, 2006.

[26] R. Cole, L. A. Gottlieb, M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In Proceedings of *STOC*, pages 91–100, 2004.

[27] L. Colussi, A. D. Col. A time and space efficient data structure for string searching on large texts. *Inf. Process. Lett.*, 58(5):217–222, 1996.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*. McGraw, second edition, 2001.

[29] M. Crochemore, J. Désarménien, D. Perrin. A note on the Burrows-Wheeler transformation. *Theor. Comput. Sci.*, 332(1-3):567–572, 2005.

[30] M. Crochemore, R. Vérin. Direct construction of compact directed acyclic word graphs. In Proceedings of *CPM*, volume 1264 of *LNCS*, pages 116–129, 1997.

[31] P. F. Dietz. Finding level-ancestors in dynamic trees. In Proceedings of *WADS*, volume 519 of *LNCS*, pages 32–40, 1991.

[32] J.-P. Duval, A. Lefebvre. Words over an ordered alphabet and suffix permutations. *ITA*, 36(3):249–259, 2002.

[33] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[34] P. Ferragina, R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.

[35] P. Ferragina, G. Manzini. Opportunistic data structures with applications. In Proceedings of *FOCS*, pages 390–398, 2000.

[36] P. Ferragina, G. Manzini. On compressing and indexing data. Technical Report TR-02-01, Dipartimento di Informatica, January 22 2002. Tue 22 January 2002 7:45:41 GMT.

[37] P. Ferragina, G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[38] P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro. An alphabet-friendly FM-index. In Proceedings of *SPIRE*, volume 3246 of *LNCS*, pages 150–160, 2004.

[39] J. Fischer, V. Heun. A new succinct representation of rmq-information and improvements in the enhanced suffix array. In Proceedings of *ESCAPE'07*, volume 4614 of *LNCS*, pages 459–470, 2007.

[40] K. Fredriksson, G. Navarro. Average-optimal single and multiple approximate string matching. *ACM Journal of Experimental Algorithmics (JEA)*, 9(1.4), 2004.

[41] T. Gagie. Large alphabets and incompressibility. *Information Processing Letters*, 99(6):246–251, 2006.

[42] L. Galambos. Dynamization in IR systems. In Proceedings of *Intelligent Information Systems*, Advances in Soft Computing, pages 297–310, 2004.

[43] R. F. Geary, R. Raman, V. Raman. Succinct ordinal trees with level-ancestor queries. In Proceedings of *SODA*, pages 1–10, 2004.

[44] R. Giegerich, S. Kurtz, J. Stoye. Efficient implementation of lazy suffix trees. *Softw., Pract. Exper.*, 33(11):1035–1049, 2003.

[45] G. Gonnet. A tutorial introduction to Computational Biochemistry using Darwin. Technical report, Informatik E.T.H., Zuerich, Switzerland, 1992.

[46] G. H. Gonnet, R. A. Baeza-Yates, T. Snider. New indices for text: Pat trees and pat arrays. In Proceedings of *Information Retrieval: Data Structures & Algorithms*, pages 66–82. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1992.

[47] R. Gonzalez, S. Grabowski, V. Makinen, G. Navarro. Practical implementation of rank and select queries. *Poster Proc. WEA*, 5:27–38, 2005.

[48] S. Grabowski, V. Mäkinen, G. Navarro. First Huffman, then Burrows-Wheeler: A simple alphabet-independent FM-index. In Proceedings of *SPIRE*, volume 3246 of *LNCS*, pages 210–211, 2004.

[49] R. Grossi, A. Gupta, J. S. Vitter. High-order entropy-compressed text indexes. In Proceedings of *SODA*, pages 841–850, 2003.

[50] R. Grossi, A. Gupta, J. S. Vitter. When indexing equals compression: experiments with compressing suffix arrays and applications. In Proceedings of *SODA*, pages 636–645, 2004.

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

[52] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1999.

[53] T. Hagerup. Sorting and Searching on the Word RAM. *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, pages 366–398, 1998.

[54] W.-K. Hon, T. W. Lam, K. Sadakane, W.-K. Sung, S.-M. Yiu. Compressed index for dynamic text. In Proceedings of *Data Compression Conference*, pages 102–111, 2004.

[55] T. Huynh, W. Hon, T. Lam, W. Sung. Approximate string matching using compressed suffix arrays. In Proceedings of *CPM*, pages 434–444, 2004.

[56] H. Hyyrö, G. Navarro. A practical index for genome searching. In Proceedings of *SPIRE*, volume 2857 of *LNCS*, pages 341–349, 2003.

[57] G. Jacobson. Space-efficient static trees and graphs. In Proceedings of *FOCS*, pages 549–554, 1989.

[58] J. Karkkainen. *Repetition-based text indexes.* PhD thesis, Report A-1999-4, Department of Computer Science, University of Helsinki, Finland, 1999.

[59] J. Kärkkäinen, E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In Proceedings of *3rd South American Workshop on String*

Processing, pages 141–155, 1996.

[60] J. Kärkkäinen, S. S. Rao. Full-text indexes in external memory. In Proceedings of *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*, pages 149–170, 2002.

[61] J. Kärkkäinen, E. Ukkonen. Sparse suffix trees. In Proceedings of *COCOON*, volume 1090 of *LNCS*, pages 219–230, 1996.

[62] D. K. Kim, H. Park. A new compressed suffix tree supporting fast search and its construction algorithm using optimal working space. In Proceedings of *CPM*, volume 3537 of *LNCS*, pages 33–44, 2005.

[63] D. K. Kim, J. S. Sim, H. Park, K. Park. Linear-time construction of suffix arrays. In Proceedings of *CPM*, volume 2676 of *LNCS*, pages 186–199, 2003.

[64] D. Knuth. *The art of computer programming, volume 3: sorting and searching*. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 1998.

[65] P. Ko, S. Aluru. Space efficient linear time construction of suffix arrays. In Proceedings of *CPM*, volume 2676 of *LNCS*, pages 200–210, 2003.

[66] S. R. Kosaraju, G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM J. Comput.*, 29(3):893–911, 1999.

[67] S. Kurtz. Reducing the space requirement of suffix trees. *Softw., Pract. Exper.*, 29(13):1149–1171, 1999.

[68] T. Lam, W. Sung, S. Wong. Improved approximate string matching using compressed suffix data structures. In Proceedings of *Algorithms and Computation (ISAAC)*, pages 339–348, 2005.

[69] Li, Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, 1993.

[70] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, R. C. Agarwal. Dynamic maintenance of web indexes using landmarks. In Proceedings of *WWW*, pages 102–111, 2003.

[71] M. Maaß, J. Nowak. Text indexing with errors. In Proceedings of *CPM*, pages 21–32, 2005.

[72] M. G. Maaß. Linear bidirectional on-line construction of affix trees. *Algorithmica*, 37(1):43–74, 2003.

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

[74] V. Mäkinen, G. Navarro. Implicit compression boosting with applications to self-indexing. In Proceedings of *SPIRE*, LNCS, 2007, To appear.

[75] V. Mäkinen. Compact suffix array. In Proceedings of *CPM*, volume 1848 of *LNCS*, pages 305–319, 2000.

[76] V. Mäkinen, G. Navarro. Compressed compact suffix arrays. In Proceedings of *CPM*, volume 3109 of *LNCS*, pages 420–433, 2004.

[77] V. Mäkinen, G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. In Proceedings of *CPM*, volume 4009 of *LNCS*, pages 306–317, 2006, Extended version to appear in ACM TALG.

[78] U. Manber, E. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, pages 935–948, 1993.

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

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

[81] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[82] H.-W. Mewes, K. Heumann. Genome analysis: Pattern search in biological macromolecules. In Proceedings of *CPM*, pages 261–285, 1995.

[83] P. Morales. Solución de consultas complejas sobre un indice de texto comprimido (solving complex queries over a compressed text index). Undergraduate thesis, Dept.

of Computer Science, University of Chile, 2005. G. Navarro, advisor.

[84] J. I. Munro. Tables. In Proceedings of *FSTTCS*, volume 1180 of *LNCS*, pages 37–42, 1996.

[85] J. I. Munro, R. Raman, V. Raman, S. S. Rao. Succinct representations of permutations. In Proceedings of *ICALP*, volume 2719 of *LNCS*, pages 345–356, 2003.

[86] J. I. Munro, V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.

[87] J. I. Munro, V. Raman, S. S. Rao. Space efficient suffix trees. *J. Algorithms*, 39(2):205–222, 2001.

[88] E. W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, 1994.

[89] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)*, 46(3):395–415, 1999.

[90] G. Navarro, R. Baeza-Yates. A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2), 1998.

[91] G. Navarro, R. Baeza-Yates. Very fast and simple approximate string matching. *Information Processing Letters*, 72:65–70, 1999.

[92] G. Navarro, R. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms (JDA)*, 1(1):205–239, 2000.

[93] G. Navarro, R. Baeza-Yates. Improving an algorithm for approximate pattern matching. *Algorithmica*, 30(4):473–502, 2001.

[94] G. Navarro, R. Baeza-Yates, E. Sutinen, J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.

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

[96] G. Navarro, M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press,

2002.

[97] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

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

[99] G. Navarro, R. A. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, 1(1):205–239, 2000.

[100] G. Navarro, R. A. Baeza-Yates, E. Sutinen, J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Eng. Bull.*, 24(4):19–27, 2001.

[101] G. Navarro, E. Sutinen, J. Tarhio. Indexing text with approximate q-grams. *J. Discrete Algorithms*, 3(2-4):157–175, 2005.

[102] R. Pagh. Low redundancy in static dictionaries with o(1) worst case lookup time. In Proceedings of *ICALP*, volume 1644 of *LNCS*, pages 595–604, 1999.

[103] R. Raman, V. Raman, S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In Proceedings of *SODA*, pages 233–242, 2002.

[104] L. M. S. Russo, A. L. Oliveira. An efficient algorithm for generating super condensed neighborhoods. In Proceedings of *CPM*, volume 3537 of *LNCS*, pages 104–115, 2005.

[105] L. M. S. Russo, A. L. Oliveira. Faster generation of super condensed neighbourhoods using finite automata. In Proceedings of *SPIRE*, volume 3772 of *LNCS*, pages 246–255, 2005.

[106] L. M. S. Russo, A. L. Oliveira. A compressed self-index using a Ziv-Lempel dictionary. In Proceedings of *SPIRE*, volume 4209 of *LNCS*, pages 163–180, 2006.

[107] L. M. S. Russo, A. L. Oliveira. Efficient generation of super condensed neighborhoods. *Journal of Discrete Algorithms*, 5(3):501–513, September 2007.

[108] K. Sadakane. Space-Efficient Data Structures for Flexible Text Retrieval Systems. *Proc. ISAAC*, pages 14–24, 2002.

[109] K. Sadakane. Compressed suffix trees with full functionality. to appear in Theory of Computing Systems.

[110] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In Proceedings of *ISAAC*, volume 1969 of *LNCS*, pages 410–421, 2000.

[111] K. Sadakane. Succinct representations of LCP information and improvements in the compressed suffix arrays. In Proceedings of *SODA*, pages 225–232, 2002.

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

[113] K. Sadakane, R. Grossi. Squeezing succinct data structures into entropy bounds. In Proceedings of *SODA*, pages 1230–1239, 2006.

[114] K.-B. Schürmann, J. Stoye. Counting suffix arrays and strings. In Proceedings of *SPIRE*, volume 3772 of *LNCS*, pages 55–66, 2005.

[115] J. Stoye. Index structures for large sequence data: Suffix trees and affix trees. In Proceedings of *GI Jahrestagung (Ergänzungsband)*, volume 20 of *LNI*, pages 67–71, 2002.

[116] E. Sutinen, J. Tarhio. Filtration with q-samples in approximate string matching. In Proceedings of *CPM*, volume 1075 of *LNCS*, pages 50–63, 1996.

[117] W. Szpankowski. Asymptotic properties of data compression and suffix trees. *IEEE Transactions on Information Theory*, 39(5):1647–1659, 1993.

[118] E. Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6(1):132–137, 1985.

[119] E. Ukkonen. Approximate string-matching over suffix trees. In Proceedings of *CPM*, volume 684 of *LNCS*, pages 228–242, 1993.

[120] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

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

[122] I. Witten, T. Bell, A. Moffat. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann, 1999.

[123] S. Wu, U. Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.

[124] M. Zhang, L. Hu, Q. Li, J. Ju. Weighted directed word graph. In Proceedings of *CPM*, volume 3537 of *LNCS*, pages 156–167, 2005.

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

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

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

[128] http://pizzachili.dcc.uchile.cl/.

[129] http://www.bzip.org/.