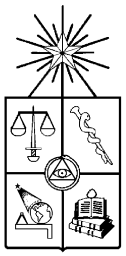


Universidad de Chile

Facultad de Ciencias Físicas y Matemáticas

Departamento de Ciencias de la Computación

Autoíndices Comprimidos para Texto Basados en Lempel-Ziv

Por

Diego Arroyuelo Billiardi

Tesis para optar al grado de

Doctor en Ciencias Mención Computación

Profesor Guía :		Gonzalo Navarro
Comité	:	Jérémy Barbay
	:	Benjamin Bustos
	:	Mauricio Marin
	:	Juha Kärkkäinen (Profesor Externo, Universidad de Helsinki, Finlandia)

Esta tesis ha recibido el apoyo del programa de Becas de Doctorado de CONICYT, Chile; Yahoo! Research Latin America; y de los Proyectos Fondecyt 1-050493 y 1-080019.

SANTIAGO - CHILE

Marzo 2009

RESUMEN DE LA TESIS
PARA OPTAR AL GRADO DE
DOCTOR EN CIENCIAS, MENCIÓN COMPUTACIÓN
POR: DIEGO GASTON ARROYUELO BILLIARDI
FECHA: 30/03/2009
PROF. GUIA: Prof. GONZALO NAVARRO BADINO

AUTOÍNDICES COMPRIMIDOS PARA TEXTO BASADOS EN LEMPEL-ZIV

Debido a las grandes colecciones de texto disponibles hoy en día, el problema de *búsqueda en texto* es común en diversas aplicaciones, las que necesitan proveer acceso eficiente a esas colecciones. Los índices clásicos para texto requieren demasiado espacio (además de almacenar el texto), y entonces podrían no caber en memoria principal, aún para textos de tamaño moderado que sí caben. La tendencia actual en búsqueda en texto indexado es la de los *autoíndices comprimidos*, los cuales reemplazan el texto con una representación que requiere espacio proporcional al del texto comprimido y proveen acceso indexado al texto.

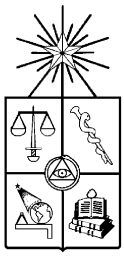
Los autoíndices comprimidos basados en arreglos de sufijos son los más conocidos en la literatura. Existen, sin embargo, otros tipos de autoíndices que han recibido menos atención, a pesar de su eficiencia. En esta tesis realizamos un estudio de los autoíndices comprimidos basados en el algoritmo de compresión Lempel-Ziv (LZ-índices para abreviar), contribuyendo con nuevos desarrollos. Basamos nuestros estudios en el LZ-índice de Navarro (o simplemente LZ-índice). Previo a nuestro trabajo, los LZ-índices eran una alternativa atractiva, aunque con algunas deficiencias: eran usualmente más grandes que otros esquemas existentes, no permitían compromisos entre espacio y tiempo de búsqueda (para ajustarse a la memoria disponible), requerían demasiado espacio de construcción, y no funcionaban eficientemente en memoria secundaria (para textos muy grandes). Contribuimos a aliviar todas esas deficiencias.

Nuestra primera contribución es un enfoque práctico para reducir el espacio del LZ-índice, alcanzando hasta dos tercios de su espacio, y manteniendo sus buenas cualidades. Aunque nuestros índices sólo garantizan buen tiempo promedio de búsqueda, son las alternativas más eficientes para las operaciones claves de extraer subcadenas del texto y mostrar los contextos de las ocurrencias, suponiendo que disponemos del espacio que requieren nuestros índices.

Luego estudiamos maneras de reducir el espacio que requiere el LZ-índice, esta vez con garantías de peor caso en tiempo de búsqueda, obteniendo los LZ-índices más pequeños que existen, a la vez que mejoramos el costo de búsqueda original. Exploramos diversas maneras para lograr nuestro objetivo, y obtenemos una familia completa de LZ-índices, con diferentes requerimientos de espacio y tiempos de búsqueda. De esta manera, somos capaces de competir en casos en los que el LZ-índice original no podía hacerlo, debido a limitaciones de espacio.

Estudiamos luego la construcción con poco espacio de nuestra familia de LZ-índices. Concluimos que nuestros LZ-índices pueden ser construidos sin espacio extra sobre el del índice final, lo que mejora su aplicabilidad ya que si disponemos del espacio para almacenar el índice final, seremos capaces de construirlo sin acceder a la memoria secundaria. En la práctica, nuestros LZ-índices son los autoíndices más rápidos en construirse con espacio proporcional al del texto comprimido.

Finalmente, estudiamos cómo adaptar el LZ-índice en memoria secundaria. Obtenemos el índice para texto en memoria secundaria más pequeño que existe, siendo a la vez muy competitivos en cantidad de accesos a disco.



University of Chile
Faculty of Physics and Mathematics
Department of Computer Science

Lempel-Ziv Compressed Full-Text Self-Indexes

By

Diego Arroyuelo Billiardi

A thesis submitted to the University of Chile
in fulfillment of the thesis requirements for the degree of

Ph.D. in Computer Science

Advisor	:	Gonzalo Navarro
Committee	:	Jérémy Barbay
	:	Benjamin Bustos
	:	Mauricio Marin
	:	Juha Kärkkäinen (External Professor, University of Helsinki, Finland)

This work has been supported by CONICYT PhD. Scholarship, Chile; Yahoo! Research Latin America; and Fondecyt Grants 1-050493 and 1-080019.

SANTIAGO - CHILE
March 2009

Abstract

Because of the many growing text collections available from different sources, the *full-text search* problem arises in a wealth of applications, which need to provide efficient access to large text collections. Classical text indexes require much space (besides storing the text itself), and thus might not fit in main memory even for moderate texts which do. The current trend in indexed text search is that of *compressed full-text self-indexes*, which replace the text with a representation that takes space proportional to that of the compressed text, and provide indexed access to it. The most prominent approaches include indexes based on *Compressed Suffix Arrays*, on the *Burrows-Wheeler transform*, and on *Lempel-Ziv* compression.

Though the attention of researchers has been biased towards suffix-array-based compressed indexes, other kinds of compressed self-indexes deserve to be considered, because of their efficiency. In this thesis we carry out a deep study of compressed full-text self-indexes based on Lempel-Ziv compression (LZ-indexes for short), contributing with new developments in this area. We base our study on Navarro's LZ-index (or simply LZ-index). Before our work, the LZ-indexes were an attractive alternative, though with some drawbacks: they were usually larger than competing schemes, did not allow for any space/time tradeoff (to fit different amounts of main memory available), required too much construction space, and could not be handled efficiently on secondary storage (in cases where the text is very large). We contribute to alleviate all these drawbacks.

Our first contribution is a practical approach to reduce the space of the LZ-index, achieving up to 2/3rds of its space, and still maintaining its good features. Although they can provide only good average-case search complexity, given the space they need, our LZ-indexes are the most efficient alternatives in practice for *extracting* text substrings and *displaying* the occurrence contexts, which are key operations for compressed self-indexes.

Then, we go one step further and study ways to reduce the space required by the LZ-index, this time providing worst-case guarantees at search time. As a result, we obtain the smallest existing LZ-indexes, also improving the original search complexity. We explore several ways to achieve our goal, and obtain a novel and complete family of LZ-indexes, providing competitive space/time trade-offs. Thus, we are now able to compete in cases where the original LZ-index was unable because of space limitations.

We then study the space-efficient construction of our family of LZ-indexes. We conclude that our indexes can be constructed without any extra space on top of that required by the final index. This enhances the applicability of our LZ-indexes, since wherever these can be used, we will be able to build them without accessing secondary storage. In practice, our LZ-indexes are the fastest compressed self-indexes to be constructed within compressed space.

Finally, we study how to adapt the LZ-index to work on secondary storage. We obtain the smallest existing index on secondary storage, with a very competitive performance.

Contents

Chapter 1

Introduction

1.1 Problem Definition and Applications

Text searching is a classical problem in Computer Science. Given a sequence of symbols $T[1..u]$, the *text*, over an alphabet $\Sigma = \{1, \ldots, \sigma\}$ of size σ, and given another (short) sequence $P[1..m]$, the *search pattern*, also over Σ, the *full-text search problem* consists in finding all the *occ* occurrences of P in T. The aim of this thesis is to design and analyze algorithms and data structures for the full-text search problem.

With the huge amount of text data available nowadays, which need to be searched to find patterns of interest, the full-text search problem plays a fundamental role in modern computer applications, which include text databases in general. Unlike *word-based* text searching (which is typical in application areas like *Information Retrieval*), we wish to find any *text substring*, not only whole words or phrases. This has applications in texts where the concept of *word* is not well defined (e.g., Oriental languages), or texts where words do not exist at all (e.g., DNA, protein, and MIDI pitch sequences), or where one wishes to retrieve more than words (e.g., program code), etc.

One of the most important challenges of a system supporting text search is that of providing fast access to the occurrences of user-entered patterns.

1.1.1 Application Domains

Nowadays, many applications are highly related to the full-text search problem, not only because of the vast collections of *traditional* texts available either in digital libraries, emails, government or corporate documents, and even the World Wide Web, but also because of the many other, at first sight not so related, areas that generate data in form of text: genomic and biological research, sensors and electronic devices (e.g., data from satellites

and MIDI pitch sequences), XML and software databases, etc. We illustrate here some application domains of full-text search.

(1) Computational Biology: Since the advent of the DNA sequencing technologies, biological applications have become highly related to text processing, since biological sequences (like DNA and proteins) are conceptualized as strings of symbols, or texts. In the case of DNA, we need a four-symbol alphabet to distinguish among the four possible *nucleotides*, or *bases* (`A`=Adenine, `C`=Cytosine, `G`=Guanine, and `T`=Thymine). In the case of proteins, a twenty-symbol alphabet is used, as there are twenty known amino acids.

A few years ago, the *Human Genome Project* [Hum05, Fre91] finished the task of sequencing the Human Genome, which aimed at identifying the genes that make up the entire Human Genome, as well as developing and improving the sequencing technologies. The result is a sequence of about 3×10^9 base pairs. It is usual that biologists need to search for DNA subsequences in this genome [Fre91], for instance in order to find genes of interest [HTSW03]. In this case, we want to be able to search for any text substring (since there are no word boundaries in DNA data), and hence the full-text search problem is appropriate.

(2) Music Processing: The *Musical Instrument Digital Interface* (MIDI) format is a standard protocol that enables the communication between computers and musical instruments. A MIDI file consists of a sequence of events, usually coded as text, which allows us to reconstruct an audio signal.

Although the full-text search problem may not be enough for the kind of processing that one usually needs to carry out on MIDI pitch sequences, this is used as a first filter to reduce the number of candidate subsequences that are further evaluated with a more costly process [FMN06].

(3) Digital Libraries: these play an important role in many aspects of life nowadays, since many text documents are represented in digital form, and stored in digital text repositories. This includes government and corporate documents, scientific papers, news wires and digital newspapers, etc. In order to find and use the information stored in a repository, we must provide the user with search capabilities.

(4) XML Documents: The *eXtensible Markup Language* (XML) [w3c94] has become the standard for exchange of data between applications, in particular on the World Wide Web, among many other uses. Thus, vast collections of XML documents are automatically generated by computer applications, and hence nowadays it is quite common to find very large XML databases. Since an XML document can contain text, it is natural to allow for full-text search capabilities (see, for example, `http://www.w3.org/TR/xpath-full-text-10-use-cases/`).

(5) Source Code Repositories: A *source code repository* stores large amounts of source code, aiding developers to control the different versions of their software. Some large repositories are public, as for instance *Google Code* or *Source Forge*. In the former case, the user is provided with search capabilities on the source code (`http://www.google.com/codesearch`). Thus, text searching plays an important role in this application.

1.1.2 Approaches for Solving the Full-Text Search Problem

There are two general approaches for solving the full-text search problem:

Sequential Text Searching. We search for the pattern P directly on the plain representation of T. That is, we do not construct any data structure on the text, mainly because the text is small, highly dynamic, or it is not available in advance. The main problem here is to reduce as much as possible the number of comparisons among text and pattern symbols, which is $O(mn)$ if we use a naive approach. See the book by Navarro and Raffinot [NR02] for a complete review on sequential text searching. This is the approach used by the widely-used Unix search tool `grep`.

Indexed Text Searching. Unlike the previous approach, we build a data structure (or *index*) on the text to restrict the search to a small portion of the text, avoiding the sequential scan. In this way we improve the search time at the expense of increasing the space requirement to solve the problem, since we need to store the index. This approach is used when the text is so large that a sequential scan is prohibitively costly, many searches (using different patterns) must be performed on the same text, the text does not change so frequently, and there is sufficient storage space to maintain the index and provide efficient access to it.

In this thesis we assume that the text is large and known in advance to queries, and we need to perform several queries on it. Therefore, we focus on indexed text searching, thus allowing efficient access to the pattern occurrences, yet increasing the space requirement since we need to store the index along with the textual database. Therefore, in this thesis we will study space-efficient text indexes, with the aim of reducing the space requirement to solve the problem.

1.1.3 Typical Queries

There exist three typical kinds of queries, which arise in different types of applications, namely:

— `exists`(P), which tell us whether pattern P exists in T or not;

— `count`(P), which counts the number of occurrences of pattern P in T; and

— `locate`(P), which reports the starting position of all the occurrences of pattern P in T.

These correspond to *existential*, *cardinality*, and *locating* queries respectively. From now on, let *occ* be the total number of occurrences of pattern P in T. In the case of `locate` queries, one can also be interested in finding only a certain number of pattern occurrences, such that the remaining ones are located while the user processes the first ones, or on user demand (think for example of the 10 result pages shown by a Web search engine). In other cases we can just need, for example, to find a fixed number K of (arbitrary) occurrences. Let us think again of Web search engines, where the answer to a user-entered query usually consists of a ranking with the (hopefully) most relevant Web pages for the query, plus an arbitrary (short) context (or snippet) where the pattern occurs within every such page; this snippet is displayed in order to help users decide whether a page is relevant or not for their needs. Therefore, in this example we must be able to quickly find just one arbitrary pattern occurrence in the text and display its context. We call *partial* `locate` queries those that ask to locate K arbitrary occurrences. We insist that the occurrences found are arbitrary, so this is different to the better known problem of finding the K first occurrences.

1.2 Full-Text Indexing

Since in this thesis we focus on indexed full-text searching, we review the classical full-text indexes, as well as the new trends in this area.

1.2.1 Classical Full-Text Indexing

Classical full-text indexes, like *suffix trees* [Wei73, McC76, Apo85] and *suffix arrays* [MM93, GBYS92], are among the best known and used data structures. Suffix trees allow us to locate the *occ* occurrences of pattern P in text T in $O(m + occ)$ time. Suffix arrays, on the other hand, have a basic search time of $O(m \log u + occ)$, which can be dropped to $O(m + \log u + occ)$ time if extra information is stored [MM93].

However, these indexes require lots of space: $O(u \log u)$ and $u \log u$ bits respectively, which in practice is about 10-20 and 4 times the text size respectively, apart from the text itself. Thus, we can have large texts which still fit into main memory, but whose corresponding suffix tree (or array) cannot be handled in main memory. A typical example is that of the Human Genome, which can be represented in less than 1 gigabyte of memory (if we use just 2 bits per symbol), yet the space of the corresponding suffix tree and array is about 40 gigabytes [Kur99] and 12 gigabytes respectively.

Using secondary storage for the indexes is several orders of magnitude slower, and has a significant influence on the running time of an application [Vit08, KR03]. Therefore

one looks for ways to reduce the index size, with the main motivation of maintaining the indexes of very large texts entirely in main memory. The modern trend is to use the compressibility of the text to reduce the space of the index.

A general objective of this thesis will be *to provide fast access to the text using as little space as possible*. In recent years there has been much research on *compressed text databases*, focusing on techniques to represent the text and the index using little space, yet permitting efficient text searching [NM07].

1.2.2 Compressing and Indexing Texts

As we have seen, classical full-text indexes (like suffix trees and suffix arrays) require $O(u \log u)$ bits of space, which is space demanding in practice. As a result, we can hold in main memory only the indexes of relatively small texts. As we said before, using secondary storage for the indexes is several orders of magnitude slower, hence we look to store the indexes in main memory as much as we can.

An important question is whether a full-text index can require only $O(u \log \sigma)$ bits of space (i.e., space proportional to the size of the uncompressed text) or, even better, whether we can profit from the compressibility of the text to reduce the space of the index (e.g., an index requiring space proportional to the size of the compressed text, $O(uH_k(T))$ bits [1]).

The Former Attempts to Reduce the Index Space. Trying to answer these questions, there are some (mainly practical) attempts to reduce the space requirement of the indexes, as for example the work of Kärkkäinen [Kär95], which presents the *suffix cactus*, a hybrid between suffix trees and suffix arrays that requires about 10 times the text size of main memory to operate; Kurtz [Kur99] presents a more compact representation of suffix trees requiring about 10 times the text size; and Abouelhoda et al. [AOK02], which define the so-called *enhanced suffix arrays*, with a space requirement of 6 times the text size.

These approaches have been mainly practical, in the sense that the improvement was mainly due to good engineering, with remarkable, though not spectacular, results: they reduce the space requirement of classical indexes, providing some compromises of search time and space requirement, yet not profiting from text compression. Also, they still need the text to operate.

[1] $uH_k(T)$ is a lower bound to the number of bits used to represent T by any k-th order compressor, where $H_k(T)$ denotes the k-th order empirical entropy of T. See Section 2.4 for more details.

Compressed Full-Text Self-indexes. To provide fast access to the text using little space, the current (more principled) trend is to use *compressed full-text self-indexes*, which are defined in what follows.

Definition 1.1. A *compressed full-text index* is one whose space requirement is proportional to the compressed text size under some compression model (e.g., $O(uH_k(T))$ bits of space).

Therefore, the space of the index can be reduced when the text is compressible (for this reason, Ferragina and Manzini [FM00, FM01] called *opportunistic* this kind of indexes).

This track was started by Kärkkäinen and Ukkonen [KU96a, Kär99], which studied text indexes based on repetitions, defining the first indexes based on the Lempel-Ziv compression algorithms [LZ76]. Later, Grossi and Vitter [GV00, GV05] defined the *Compressed Suffix Arrays*, based on regularities of suffix arrays to reduce their space. Independently, Mäkinen [Mäk00, Mäk03] obtained a compact representation of suffix arrays, also profiting from some regularities in the runs of suffix arrays to reduce the space in practice to about 2.5 times the text size. However, and just like classical indexes, all these indexes still need the text to operate. An important space saving could be achieved if we do not need the text to operate.

Definition 1.2. A full-text *self-index* allows one to search and extract any part of the text without storing the text itself.

This introduces important savings of memory space. Sadakane [Sad00, Sad02, Sad03], and Ferragina and Manzini [FM00, FM01, FM05] defined the first existing self-indexes. The former is a representation of Compressed Suffix Arrays [GV05] which does not need the text to operate. The index of Ferragina and Manzini, on the other hand, is based on the close relation between suffix arrays and the *Burrows-Wheeler* transform [BW94]. Later, many other compressed self-indexes were defined, like the ones by Grossi et al. [GGV03], Foschini et al. [GGV04, FGGV06], Navarro [Nav04, Nav08], Mäkinen and Navarro [MN05], Ferragina et al. [FMMN04, FMMN07], and Russo and Oliveira [RO06, RO07], among others. An important survey about compressed self-indexes is given by Navarro and Mäkinen [NM07].

To summarize, a compressed full-text self-index *replaces* the text with a more space-efficient representation of it (profiting from text compressibility to get smaller indexes), at the same time providing indexed access to the text. Taking space proportional to the compressed text, replacing it, and providing efficient indexed access to it is an unprecedented breakthrough in text indexing and compression.

Compressed full-text self-indexes are not only useful to reduce the space requirement of text indexes, but also they have application in cases where accessing the text is so

expensive that the index must search without having the text at hand, as occurs with most Web search engines.

Extending the Set of Queries, with Applications. Since compressed full-text self-indexes replace the text, we are also interested in supporting operations:

— `display`(P, ℓ), which displays a context of ℓ symbols surrounding the *occ* occurrences of pattern P in T; and

— `extract`(i, j), which decompresses the substring $T[i..j]$, for any text positions $i \leqslant j$.

Thus we can see compressed self-indexes as full-text indexes compressing the text, or as compressors allowing efficient text extraction and indexed full-text searching.

In the scenario of compressed full-text self-indexes, where we *replace the text* with a representation allowing indexed-search capabilities, being able to efficiently `extract` arbitrary text substrings is one of the most basic and important problems that indexes must solve efficiently.

While `locate` queries are important in classical full-text indexing (since we have the text at hand to access the occurrences and their contexts, as needed by many applications), they are usually not enough for compressed self-indexes, since we obtain just text positions, and no clue about the text surrounding these occurrences. In many applications the context surrounding an occurrence is as important as (and sometimes more important than) the occurrence position itself. For example, a user might be interested in the context surrounding the occurrences to decide whether the answer is interesting or not; think, for example, of the widely-used Unix search tool `grep`, which by default shows the text lines containing the occurrences. The relevance of this information is witnessed by the fact that most modern Web search engines display, along with the answers to a query, a context surrounding a pattern occurrence within each document. Therefore, in our scenario it is usually more important to obtain the contexts surrounding the pattern occurrences (i.e., `display` queries), than just text positions (i.e., `locate` queries). The latter can be interesting in specific cases, for example if one wants to take statistics about the positions of the occurrences (for instance, the average difference between successive occurrences, e.g., for linguistic, data mining, or motif discovering applications), but `display` queries are more frequently used in general.

Finally, `count` and `exists` queries have much more specific applications, and they usually compose the internal machinery of more complex tasks, for example, for *approximate pattern matching* (where one wants to find the occurrences of a pattern in a text, allowing differences between the pattern and the occurrences) and *text categorization* (where a document is assigned a class or category depending on the frequency of appearance

of given keywords). However, this is not enough for many other applications. For example, after categorizing the documents, a user might want to see a document, or a portion of it, so we have to ask the index to reproduce it, or may want to search for a given pattern and display the occurrence contexts in order to decide whether the document is of interest or not. Some *pattern discovery* tasks may use the frequency of certain strings to decide that they are important patterns. Another example is *selective dissemination of information*, where user profiles are formed by keywords of interest and the system is interested in the presence or absence of those keywords to send or not the document to the user.

As with text compression, where a text (or a part of it) must be uncompressed to process it, using compressed indexes increases processing time. This increase depends on the index and its performance. However, given the relation between main and secondary memory access times, it is preferable to handle compressed indexes entirely in main memory, rather than handling them in uncompressed form on secondary storage. In our work we look for indexes that can be efficiently queried at search time. When the compressed index is so large that it does not fit in main memory, we will try to reduce the cost of transmission between secondary and main memory, since a smaller index potentially requires both a smaller transfer time and smaller disk seek time, which is the main component of the time to access a disk.

Families of Compressed Self-indexes. The main types of compressed self-indexes [NM07] are *Compressed Suffix Arrays* [GV05, Sad03], indexes based on *backward search* [FM05, FMMN07, MN05] (which are alternative ways to compress suffix arrays, known as the *FM-index* family), and the indexes based on *Lempel-Ziv* compression algorithms [LZ76, ZL78] (LZ-indexes for short) [KU96a, Nav04, FM05, RO07].

It is usual that indexes of a given family are suitable for a given kind of query. For instance, indexes based on suffix arrays are very competitive for `exists` and `count` queries, since the occurrences lie in a contiguous interval of the suffix array, which can be found usually in time proportional to the pattern length. LZ-indexes, on the other hand, support efficient `locate` and `extract` queries. In Table 1.1 we show the most efficient existing compressed self-indexes, where the different families are separated by horizontal lines. Some of the indexes obtained in this thesis are also shown; we give more details of these later in this chapter.

In this thesis we are interested in LZ-indexes, since they have shown to be effective for locating occurrences and extracting text, outperforming other compressed indexes. What characterizes the particular niche of LZ-indexes is the $O(uH_k(T))$ space combined with $O(\log u)$ time per located occurrence. Moreover, and as we shall see in this thesis, in practice many pattern occurrences can be actually found in constant time per occurrence, which makes the LZ-indexes a very appealing alternative. Also, fast displaying of text substrings is very important in self-indexes, as the text is not available otherwise.

Table 1.1: Space and time complexities for the most efficient existing compressed self-indexes. The different families (Compressed Suffix Arrays, FM-indexes, and LZ-indexes, respectively) are separated by horizontal lines in the tables. We assume $\sigma = O(\text{polylog}(u))$ in all cases, except for the FM-index, where $\sigma = o(\log u / \log\log u)$ is assumed [FM05]. The time for `locate` is after counting the pattern occurrences. We also assume $0 < \epsilon < 1$.

Index		Space in bits
Sadakane's Compressed Suffix Array (SAD-CSA)	[Sad03]	$(1+\epsilon)uH_0(T) + O(u \log\log\sigma)$
Grossi, Gupta and Vitter's CSA (GGV-CSA)	[GGV03]	$(2+\epsilon)uH_k(T) + o(u\log\sigma)$
FM-index (FMI)	[FM05]	$5uH_k(T) + o(n\log\sigma)$
Alphabet Friendly FM-index (AF-FMI)	[FMMN07]	$uH_k(T) + o(u\log\sigma)$
Navarro's LZ-index (NAV-LZI)	[Nav04]	$4uH_k(T) + o(u\log\sigma)$
Inverted LZ-index (ILZI)	[RO07]	$(5+\epsilon)uH_k(T) + o(u\log\sigma)$
This thesis a	[ANS06, ANS08]	$(1+\epsilon)uH_k(T) + o(u\log\sigma)$
This thesis b	[ANS06, ANS08]	$(2+\epsilon)uH_k(T) + o(u\log\sigma)$
This thesis c	[AN07b, ANS08]	$(3+\epsilon)uH_k(T) + o(u\log\sigma)$

Index	`count`	`locate`	`extract`
SAD-CSA	$O(m\log u)$	$O(occ\log^{\frac{1}{1+\epsilon}} u)$	$O(\ell + \log^{\frac{1}{1+\epsilon}} u)$
GGV-CSA	$O(\frac{m}{\log_\sigma u} + \frac{\log^{\frac{3+\epsilon}{1+\epsilon}} u}{\log^{\frac{1-\epsilon}{1+\epsilon}} \sigma})$	$O(occ\log u\log^{\epsilon}_{\sigma} u)$	$O(\ell/\log_\sigma u + \log u\log^{\epsilon}_{\sigma} u)$
FMI	$O(m)$	$O(occ\log^{1+\epsilon} u)$	$O(\ell + \log^{1+\epsilon} u)$
AF-FMI	$O(m)$	$O(occ\log^{1+\epsilon} u)$	$O(\ell + \log^{1+\epsilon} u)$
NAV-LZI	$O(m^3\log\sigma + m\log u + occ)$	$O(occ\log u)$	$O(\ell\log\sigma)$
ILZI	$O(\frac{m}{\epsilon}\log u + occ)$	$O(occ\log u)$	$O(\frac{\ell}{\epsilon\log_\sigma u})$
This thesis a	$O(\frac{m^2}{\epsilon})$ on average	$O(\frac{m^2}{\epsilon})$ on average	$O(\frac{\ell}{\epsilon\log_\sigma u})$
This thesis b	$O(\frac{m^2}{\epsilon} + m\log u + \frac{occ}{\epsilon})$	$O(occ\log u)$	$O(\frac{\ell}{\epsilon\log_\sigma u})$
This thesis c	$O(m)$	$O((m+\frac{occ}{\epsilon})\log u)$	$O(\frac{\ell}{\epsilon\log_\sigma u})$

Historically, the first compressed index based on Lempel-Ziv compression was that of Kärkkäinen and Ukkonen [KU96a, Kär99], the KU-LZI for short, which has a locating time $O(m^2 + (m+occ)\log u)$ and a space requirement of $O(uH_k(T))$ bits, plus the text (as it is needed to operate) [NM07].

Ferragina and Manzini [FM05] present an index based on Lempel-Ziv compression, the FM-LZI for short, although combined with Burrows-Wheeler compression [BW94] to support locating the pattern occurrences in optimal $O(m+occ)$ time, without restrictions on m or occ. This is the only existing compressed full-text self-index with optimal search time, requiring $O(uH_k(T)\log^\gamma u) + o(u\log\sigma\log^\gamma u) = o(u\log u)$ bits of space, for any constant $\gamma > 0$. However, the $O(\log^\gamma u)$ extra factor in the space usage can make the space of this index excessive.

Navarro's LZ-index [Nav04, Nav08], the NAV-LZI for short, or simply the LZ-index

throughout this thesis, on the other hand, is a compressed full-text self-index based on the Lempel-Ziv 1978 [ZL78] (LZ78 for short) parsing of the text. The LZ-index takes about 4 times the size of the compressed text, that is $4uH_k(T) + o(u \log \sigma)$ bits, for any $k = o(\log_\sigma u)$, and answers queries in $O(m^3 \log \sigma + (m + occ) \log u)$ worst-case time. The index also replaces the text (i.e., it is a *self-index*): it can display a text context of phrases of length ℓ around an occurrence found (and in fact any sequence of LZ78 phrases) in $O(\ell \log \sigma)$ time, or obtain the whole text in time $O(u \log \sigma)$. The index is built in $O(u \log \sigma)$ time.

Russo and Oliveira [RO07] discard the LZ78 parsing of T and use a so-called maximal parsing instead, which is performed for the reverse text. In this way they avoid the $O(m^2)$ checks for the different pattern substrings, needed both by the KU-LZI and NAV-LZI. The resulting LZ-index, the *Inverted* LZ-index (ILZI for short), requires $(5 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits of space, for any $0 < \epsilon < 1$ and any $k = o(\log_\sigma u)$. The locating time of the index is $O((\frac{m}{\epsilon} + occ) \log u)$.

Current Technologies for Compressed Text Indexing. The practical implementation and evaluation of compressed self-indexes is a very important issue, since many indexes are proposed in theory but never implemented. Implementing the indexes and making them available motivates the practical use of these new techniques, and encourages technology transfer. Aiming at this, the *Pizza&Chili* corpus [FN05] provides practical implementations of compressed indexes, as well as some testbed texts. The current state of the art of practical compressed indexes in the corpus is surveyed in [FGNV08].

We are interested both in theoretical and practical results in this thesis. On the practical side, we shall make available in the *Pizza&Chili* corpus the practical prototypes described in this thesis, so that they can be used by the scientific community as well as by practitioners looking for particular solutions to practical problems in Computational Biology, Digital Libraries, and full-text databases in general.

1.3 Contributions of this Thesis

As we said before, the most basic problems for compressed self-indexes are that of reproducing the text (or any text substring) and searching (locating the occurrences and displaying a context, or counting occurrences), while requiring little space. However, there are many other functionalities that a compressed self-index must provide in order to be fully useful. Many of those have been obtained separately in the indexed text searching literature. For example, there are

— indexes like the suffix trees, allowing to search for a pattern in optimal $O(m + occ)$ time, yet requiring $O(u \log u)$ bits of space; or the LZ-index of Ferragina and Manzini

[FM05], also achieving optimal $O(m + occ)$ time yet requiring $o(u \log u)$ bits of space if σ is small;

— the suffix arrays, which are one of the most used text indexes in practice and can be built in $O(u)$ time [KSPP03, KA03, KS03], yet they require basically $u \log u$ bits of space, which can be excessive for large texts;

— compressed indexes using space proportional to the text entropy (and many times including the text) [NM07];

— indexes requiring little space to be built [HLS+07, HSS03a, NP07, GN08b]: compressed indexes are usually derived from a classical one. Although it is usually simple to build a classical index and then derive its compressed version, there might not be enough space to build the classical index first. Secondary memory might be available, but many classical indexes are costly to build on secondary memory. Therefore, an important problem is how to build compressed indexes without building their classical versions first;

— indexes supporting efficient construction and search on secondary memory [FG99, CM96]: although their small space requirements might permit compressed indexes fit in main memory, there will always be cases where they have to operate on disk. There is not much work yet on this important issue. A good survey on full-text indexes on secondary memory is by Kärkkäinen and Rao [KR03];

— and others supporting efficient insertion and deletion of texts [FG99, FM00, HLS+04a, CHLS07, MN08b, GN08b]: most indexes in the literature are static, in the sense that they have to be rebuilt from scratch upon text changes. This is currently a problem even on uncompressed full-text indexes, and not much has been done.

Some existing indexes accomplish several of these points [GN08b]. However, no existing data structure for text searching fits all the above requirements.

As we said before, we are mainly interested in Lempel-Ziv-based compressed full-text self-indexes, since they have shown to be effective in many aspects: They are fast for locating the pattern occurrences, displaying occurrence contexts, and extracting the text [Nav04, Nav08], which is very important in the track of compressed self-indexes. However, the current state of the art on these indexes still needs further attention in order to solve important existing problems. Therefore, in this thesis we carry out a deep study of compressed full-text self-indexes based on the Lempel-Ziv compression algorithm of 1978 [ZL78].

Specifically, we will focus our studies on Navarro's LZ-index. Before this thesis, Navarro's LZ-index had the following properties: fast full-text searching, fast text recovery,

uses little space for operation (yet this space is relatively high compared to other compressed full-text self-indexes), does not allow insertion nor deletion of text, only operates in main memory, and it needs much construction space. Many of the weak points of the LZ-index were already supported by compressed indexes based on suffix arrays, like FM-indexes [FM05] and Compressed Suffix Arrays [GGV03, FGGV06, Sad03], which are usually smaller (though sometimes slower) than LZ-indexes, provide time/space trade-offs, can be constructed within little space [HLS$^+$07, NP07, GN08b], support dynamism [FM00, HLS$^+$04a, CHLS07, MN08b, GN08b], and can be stored on secondary storage [MNS04].

We aim at adding new features to the LZ-index, as well as improving the existing ones (e.g., improve the time to locate pattern occurrences, improve the time to extract text substrings, etc.), from a theoretical and practical point of view. The idea is to compete with compressed indexes based on suffix arrays. The result are LZ-indexes which are smaller than the original LZ-index [Nav04], provide time/space trade-offs, are very competitive in practice, can be constructed within little space, and can be handled on secondary storage.

This is a summary of the main results of this thesis:

1.3.1 Reducing the Space Requirement of Lempel-Ziv Text Indexes

Navarro's LZ-index has shown to be an attractive alternative in practice [Nav04], outperforming competing schemes in many practical scenarios. However, the space requirement of the LZ-index is relatively large compared with competing schemes [Nav04]: 1.2–1.6 times the text size versus 0.6–0.7 and 0.3–0.8 times the text size of the *Compressed Suffix Array* [Sad03] and the *FM-index* [FM05], respectively. In addition, the LZ-index does not offer space/time trade-offs, which limits its applicability.

In Chapter 4 we aim at reducing the space requirement of the LZ-index, mainly from a practical point of view. We introduce a scheme to study the redundancy of the original LZ-index. As a result, we obtain indexes requiring $3uH_k(T) + o(u \log \sigma)$ and $(2+\epsilon)uH_k(T)+o(u \log \sigma)$ bits of space, for any $0 < \epsilon < 1$, which is less than the space of the original LZ-index. Our indexes cannot provide worst-case guarantees at search time, yet they support `locate` queries for patterns of length m in $O(\frac{m}{\epsilon} + \frac{n}{\epsilon\sigma^{m/2}})$ average time, which is $O(\frac{m^2}{\epsilon})$ for $m \geqslant \log_\sigma u$. In practice, our data structure uses about 2/3 of the space of the original LZ-index, while retaining a very good extracting and displaying performance: our indexes offer an attractive alternative for full-text searching, outperforming competing schemes in many practical scenarios.

We conclude that our LZ-indexes are an interesting alternative for highly compressible texts, outperforming competing schemes while requiring little space.

The main results of this chapter have been published in [AN08].

1.3.2 Stronger Lempel-Ziv Text Indexes

Despite the fact that in the previous part we reduce the space requirement of the LZ-index, with relevant results in practice, that is a mainly practical result which only gives us average-case guarantees at search time. The challenge of Chapter 5 is to reduce the space requirement of the LZ-index, while retaining its good theoretical features. We develop a theoretical method to support the reduction of space requirement of the LZ-index while retaining worst-case guarantees at search time. Moreover, we show how to reduce the original time complexity of the LZ-index.

We first show that the connection between the tries conforming the LZ-index (the so-called *LZTrie* and *RevTrie*) can be supported within little space. This connection is very important for the search algorithm of the LZ-index and, before our work, it was thought that $uH_k(T) + o(u \log \sigma)$ bits were necessary for such a connection [Nav04]. This result leads us to obtain an LZ-index requiring $(1 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits of space, yet without worst-case guarantees at search time within this space. Building upon the previous result, we obtain new compressed full-text self-indexes requiring about half the space of the original LZ-index, $(2+\epsilon)uH_k(T)+o(u \log \sigma)$ bits of space, while simultaneously improving the locating complexity to $O(\frac{m^2}{\epsilon} + (m + occ) \log u + \frac{occ}{\epsilon})$ in the worst case.

Then, we build on these approaches to show how the locating time can be dropped to $O((m+occ) \log u)$ in the worst case, with an index requiring only $(3+\epsilon)uH_k(T)+o(u \log \sigma)$ bits of space, which is about half of the space required by competing schemes achieving the same time complexity [RO07]. We finally show how to achieve optimal extracting complexity, by adapting to our indexes a data structure of Grossi and Sadakane [GS06] to extract text substrings.

As a result we obtain the smallest existing LZ-indexes. Notice that if ϵ is a constant, we also achieve the same locating complexity $O(m^2 + (m + occ) \log u)$ as the KU-LZI [KU96a, Kär99, NM07], yet ours do not need the text to operate. Our technique has shown to be useful also to reduce the space of the RO-LZI [RO07], yet they cannot achieve the smaller overall space requirement that we achieve.

The main results of this chapter have been published in [ANS06, AN07b, ANS08]. In Table 1.1 we show our resulting indexes, compared against the most efficient known compressed full-text self-indexes.

1.3.3 Space-Efficient Construction of Lempel-Ziv Text Indexes

As we said before, the space-efficient construction of compressed full-text self-indexes is a very important aspect, regarding the practicality of the indexes. Therefore,

researchers have focused on this topic, obtaining relevant results [HLS+07, HSS03a, NP07, MN08b, GN08b]. All these works, however, define space-efficient algorithms to construct Compressed Suffix Arrays, leaving the problem of the space-efficient construction of LZ-indexes still open. It has been shown that the space required to build the LZ-index can be excessive [Nav04], although comparable to that required to build suffix arrays. This is mainly because, at construction time, a non-space-efficient representation of the tries that compose the LZ-index is used. Thus, the space-efficient construction of the LZ-index is highly related to the maintenance of succinct dynamic σ-ary trees (or tries), which was an open problem posed by Munro et al. [MRS01].

In Chapter 6 we develop a space-efficient algorithm for constructing the LZ-index. We replace the non-space-efficient intermediate representations of the tries that form the LZ-index by space-efficient counterparts. Our algorithm for the original LZ-index requires $4uH_k(T) + o(u \log \sigma)$ bits of space, which is as much space as the final LZ-index needs to operate. The construction time is $O(u(\log \sigma + \log \log u))$ in the worst case, which is an improvement over the $O(\sigma u)$ time of the preliminary algorithm presented in [AN05] (which is excessive for large alphabets). Thus, we conclude that wherever the LZ-index can be used (i.e., it can be held in main memory), we can build it without the need of accessing secondary storage.

We also show how to adapt this algorithm to build our smaller versions of the LZ-index described in Section 1.3.2, requiring $(1+\epsilon)uH_k(T)+o(u \log \sigma)$, $(2+\epsilon)uH_k(T)+o(u \log \sigma)$, and $(3+\epsilon)uH_k(T)+o(u \log \sigma)$ bits of space.

We present an alternative model to construct the indexes, in which we assume that the available main memory to carry out the indexing process is smaller than the space required by the final index. This model has applications in cases where the indexing process must be carried out in a computer that is not powerful enough to maintain the whole index in main memory, leaving a more powerful equipment exclusively to answer user queries. We show that under this model, the LZ-indexes can be constructed within $(1+\epsilon)uH_k(T)+o(u \log \sigma)$ bits of space, for any $0 < \epsilon < 1$, in $O(u(\log \sigma + \log \log u))$ time. This means that the LZ-indexes can be built within slightly more space than that required by the compressed text.

Our experimental results show that our indexing algorithm needs in practice about the same memory as that required by the final LZ-index. We compare our proposal against existing space-efficient algorithms to construct Compressed Suffix Arrays [Hon04, HLS+04b]. We find out that our algorithm is much faster than competing schemes: for example, we are able to construct the LZ-index for the whole Human Genome in about 5 hours on a 3 GHz CPU, versus 24 hours of Compressed Suffix Arrays and 28 hours of FM-index [Hon04, HLS+04b] (on a 1.7 GHz CPU). This introduces an important practical result regarding applications to biological research. Under the reduced-memory

scenario, our experimental results show that the LZ-index for the Human Genome can be constructed within 1.6 GB of main memory, which is about half of the space required by the uncompressed genome (assuming the symbols are represented by bytes).

It is important to note that our algorithm can be adapted to construct alternative LZ-indexes [RO07], and also that by the time of its preliminary introduction [AN05], ours was the first construction algorithm for a compressed full-text self-index requiring space proportional to $H_k(T)$ instead of $H_0(T)$. Nowadays, however, there exists a construction algorithm for the FM-index requiring space proportional to $H_k(T)$ [MN08b, GN08b].

The first results of this chapter have been published in [AN05], others are submitted [AN09]. In Table 1.2 we show a comparison among the best known algorithms for constructing full-text indexes.

Table 1.2: Comparison of different algorithms for constructing text indexes.

Index		Indexing space (in bits)	Indexing time
Suffix Arrays (in-place suffix sorting)	[FM07]	$u \log u$	$O(u \log u)$
Suffix Arrays (optimal space)	[HSS03a]	$O(u \log \sigma)$ (*)	$O(u \log \log \sigma)$
Compressed Suffix Arrays	[HLS$^+$07]	$u(H_0(T) + 2 + \epsilon) + o(u \log \sigma)$ (¶)	$O(u \log u)$
Compressed Suffix Arrays	[NP07]	$O(u \log \sigma \log_\sigma^{\log_3 2} u)$ (*)	$O(u)$
Alphabet Friendly FM-index	[GN08b]	$uH_k(T) + o(u \log \sigma)$ (§)	$O(u \log u(1 + \frac{\log \sigma}{\log \log u}))$
Original LZ-index (**this thesis**)		$4uH_k(T) + o(u \log \sigma)$	$O(u(\log \sigma + \log \log u))$
This thesis a		$(1 + \epsilon)uH_k(T) + o(u \log \sigma)$ (‡)	$O(u(\log \sigma + \log \log u))$
This thesis b		$(2 + \epsilon)uH_k(T) + o(u \log \sigma)$	$O(u(\log \sigma + \log \log u))$
This thesis c		$(3 + \epsilon)uH_k(T) + o(u \log \sigma)$	$O(u \log u(1 + \frac{\log \sigma}{\log \log u}))$

(*) this is $o(u \log u)$ bits for $\log \sigma = o(\log u)$.
(¶) this is $O(u \log \sigma)$ bits of space, in the worst case.
(§) for any $k \leqslant \alpha \log_\sigma u$ and any constant $0 < \alpha < 1$.
(‡) for any $0 < \epsilon < 1$ and $k = o(\log_\sigma u)$, applies to all LZ-index variants.

1.3.4 A Lempel-Ziv Text Index on Secondary Storage

Although compressed self-indexes are smaller and thus the indexes of larger texts could be accommodated in main memory, there exists always the need of indexing larger and larger texts, whose index cannot be maintained in main memory, nor even compressed. Therefore, we must be able to manage compressed self-indexes on secondary storage. We aim here to reduce the number of I/Os incurred by secondary-storage devices, since a smaller index can be potentially read faster.

We propose in Chapter 7 a representation of the LZ-index which can be efficiently managed on secondary storage. Our idea is to use the studies of previous chapters to

add some (controlled) redundancy to the index, which shall allow us to exchange many random disk accesses to index components into (equivalent) local accesses, which is more adequate for disk memories. The result is a representation of the LZ-index that requires $O(uH_k(T)) + o(u \log \sigma)$ bits of space, but that can be efficiently managed on secondary storage. Just as for the index of Clark and Munro [CM96], our index does not provide worst-case guarantees at search time. However, our experimental results show that our proposal offers an attractive trade-off for text searching on secondary-storage, being the smallest existing alternative in practice, with a good performance at search time: in some cases we are able to report up to 600 pattern occurrences per disk access, assuming a disk page of 32 KB.

From our research we conclude that our LZ-index is very adequate for applications where it is important to find quickly just a few pattern occurrences, so as to find the remaining while processing the first ones, or on user demand (think for example of Web search engines). Our work also shows that compressed indexes are suitable to be used on secondary storage. The previous result on this line was the Compressed Suffix Array on disk of Mäkinen et al. [MNS04], which requires the excessive amount of $O(\log u)$ disk accesses *per occurrence* reported.

The main results of this chapter have been published in [AN07a].

Chapter 2

Preliminary Concepts and Data Structures

2.1 Model of Computation

Unless otherwise stated, in most of this thesis we assume the standard *word* RAM (*Random Access Machine*) model of computation [AHU74], in which we can access any memory word of length $w = \Theta(\log u)$ in constant time ($\log x$ means $\lceil \log_2 x \rceil$ in this thesis). Standard arithmetic and logical operations (like additions, bit-wise operations, etc.) are assumed to take constant time under this model. We measure the size of our data structures in bits. At the beginning of each chapter we shall make extra assumptions on the model.

2.2 Tries and some Properties of Strings

We introduce some notation and representation for basic concepts in this thesis: strings and string sets.

2.2.1 Representing Strings

We model a string $S[1..\ell]$ over an alphabet $\Sigma = \{1, \ldots, \sigma\}$ as a sequence of alphabet symbols. We say that $|S| = \ell$ is the length of string S, and refer to the i-th symbol of S as s_i. Given string $S = s_1 \ldots s_\ell$, we use:

— $S[i..j]$, for $1 \leqslant i \leqslant j \leqslant \ell$, to denote any *substring* of S.

— $S[1..i]$, for $1 \leqslant i \leqslant \ell$, to denote any *prefix* of S. If $1 \leqslant i < \ell$, we say that $S[1..i]$ is a *proper prefix* of S.

— $S[j..\ell]$, for $1 \leqslant j \leqslant \ell$, to denote any *suffix* of S. If $1 < j \leqslant \ell$, we say that $S[j..\ell]$ is a *proper suffix* of S.

— $S^r = s_\ell \dots s_1$ to denote its reverse string. Moreover, $S^r[i..j]$ will actually mean $(S[i..j])^r = s_j \dots s_i$.

As every symbol s_i is represented using $\log \sigma$ bits, string S requires $\ell \log \sigma$ bits to be represented. We use ε to denote the *empty string*, which is the string containing no symbols.

2.2.2 Representing Set of Strings

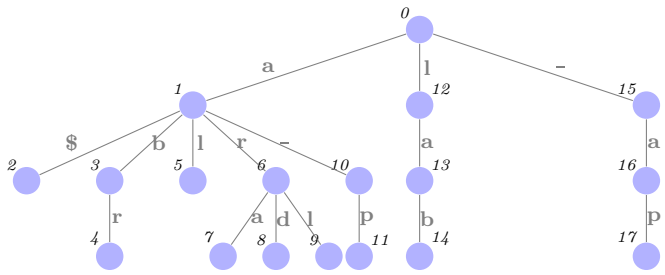
Given a set of strings $\mathcal{S}$, a *digital tree*, *cardinal tree*, or simply *trie* [Knu73] for $\mathcal{S}$ is a tree where every string in $\mathcal{S}$ is represented by a node in the tree. Let v be the node representing string $S_i \in \mathcal{S}$. This means that the trie edges in the root-to-v path are labeled with the symbols of S_i, in order (every edge is labeled with only one symbol). Notice that all nodes descending from v in the trie represent strings which have S_i as a prefix. The trie root represents the empty string ε.

Example 2.1. As an example, consider the set of strings $\mathcal{S}_1 = \{\texttt{a}, \texttt{l}, \texttt{ab}, \texttt{ar}, \texttt{\_}, \texttt{a\_}, \texttt{la}, \texttt{\_a}, \texttt{lab}, \texttt{ard}, \texttt{a\_p}, \texttt{ara}, \texttt{\_ap}, \texttt{al}, \texttt{abr}, \texttt{arl}, \texttt{a\$}\}$. The corresponding trie for this set is shown in Fig. 2.1(a). As another example, let us consider the set of string $\mathcal{S}_2 = \{\texttt{a}, \texttt{l}, \texttt{ba}, \texttt{ra}, \texttt{\_}, \texttt{\_a}, \texttt{al}, \texttt{a\_}, \texttt{bal}, \texttt{dra}, \texttt{p\_a}, \texttt{ara}, \texttt{pa\_}, \texttt{la}, \texttt{rba}, \texttt{lra}, \texttt{\$a}\}$. The resulting trie is shown in Fig. 2.1(b). In both cases, nodes representing strings in the set are shown in gray. We call *empty nodes* to those nodes not representing any string in the set; these are shown in black.

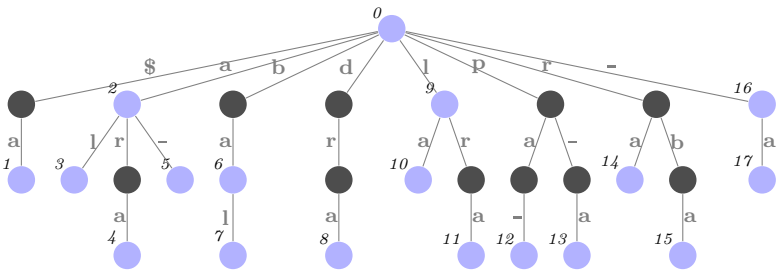
It is important to note that a trie representing a set of string $\mathcal{S}$ represents a *prefix closure* of $\mathcal{S}$, meaning that any prefix of every string in the set is represented by a node in the trie (the empty string is represented by the trie root). We say that a set of string $\mathcal{S}$ is *prefix closed* when for every string $S_i \in \mathcal{S}$, any prefix of S_i is also in $\mathcal{S}$. In the same way, we say that a set of string $\mathcal{S}$ is *suffix closed* when for every string $S_i \in \mathcal{S}$, any suffix of S_i is also in $\mathcal{S}$.

Example 2.2. The set $\mathcal{S}_1$ defined in Example 2.1 is prefix closed, whereas set $\mathcal{S}_2$ is not. This difference can be noted in the tries for these sets: the trie for $\mathcal{S}_2$ has empty nodes, whereas the trie for $\mathcal{S}_1$ has not.

If we want to search the set for a given string P, we spell out the symbols of P and use them to guide the descent in the trie, starting from the root. Let v be the trie node representing P. Since all the descendants of v have P as a prefix, then we are actually using the trie to support *prefix searches*, which we define as follows.



(a) Trie data structure representing the set of strings $\mathcal{S}_1 = \{\texttt{a}, \texttt{l}, \texttt{ab}, \texttt{ar}, \texttt{\_}, \texttt{a\_}, \texttt{la}, \texttt{\_a}, \texttt{lab}, \texttt{ard}, \texttt{a\_p}, \texttt{ara}, \texttt{\_ap}, \texttt{al}, \texttt{abr}, \texttt{arl}, \texttt{a\$}\}$.



(b) Trie data structure representing the set of strings $\mathcal{S}_2 = \{\texttt{a}, \texttt{l}, \texttt{ba}, \texttt{ra}, \texttt{\_}, \texttt{\_a}, \texttt{al}, \texttt{a\_}, \texttt{bal}, \texttt{dra}, \texttt{p\_a}, \texttt{ara}, \texttt{pa\_}, \texttt{la}, \texttt{rba}, \texttt{lra}, \texttt{\$a}\}$.

Figure 2.1: Trie data structures for different set of strings. Preorder (or depth-first) numbers are shown outside each node.

Definition 2.1. Given a set of strings $\mathcal{S} = \{S_1, \ldots, S_N\}$, the *prefix search problem* consists of finding all the strings $S_i \in \mathcal{S}$ such that S_i has a given string $P[1..m]$ as a prefix.

We introduce next some traditional (pointer-based) trie representations, assuming that n is the total number of trie nodes:

(1) The first one consists in storing, for each trie node, an array of size σ with pointers to its children (the i-th pointer points to the child corresponding to the i-th alphabet symbol, or *null* if such a child does not exist). This allows us to descend to any child in constant time. However the space requirement of this representation is $O(\sigma n \log n)$ bits. This can be excessive in cases where the average number of children of a node is much smaller than σ, which is usual in practice (except, perhaps, for small alphabets).

(2) Another alternative is to use a linked list in each node, storing the pointers to children (along with the corresponding symbol by which every child descends). This requires $O(n \log n)$ bits of storage, yet finding a given child now takes $O(\sigma)$ worst-case time. Instead, we can store these pointers in an array, requiring asymptotically the same

space, yet allowing to find a given child in $O(\log \sigma)$ worst-case time (by binary-searching the symbols labeling the children). However, adding a new child under this scheme can be costly. Finally, we can use a balanced binary search tree (sorted by the symbols), supporting to search for any child in $O(\log \sigma)$ time, as well as supporting efficient trie updates.

(3) Finally, we can use perfect hashing to store the children of a node, achieving $O(1)$ worst-case time to find a given child, and requiring $O(n \log n)$ bits of space [Ram96]. The construction takes $O(n)$ expected time if we use the classical randomized result by Fredman et al. [FKS84, CLRS01]. In the worst case, we have that the n trie edges distribute over $\Theta(n/\sigma)$ nodes, so these nodes have $\Theta(\sigma)$ children. Thus, the cost is $O(\sigma^2)$ per node in the worst case, which is $O(\sigma n)$ overall when added over those $\Theta(n/sigma)$ nodes.

In order to reduce the number of nodes in the trie representation of a set of strings, one can compress the empty unary paths (i.e., the non-branching paths) in the trie. Now, the edges are conceptually labeled with strings (formed by the concatenation of the symbols in the compressed path), rather than with single symbols. To avoid storing the whole strings labeling the edges, one stores only the first symbol and a number (which is called the *skip* value) indicating how many symbols to skip in the search string P in order to descend to a child node. This trie representation is known as *Patricia tree* [Mor68]. It is important to note that all internal nodes in a Patricia tree have degree at least 2. Also, the leaves always correspond to a string in the set we are representing. Then, it can be easily proved that the total number of nodes in a Patricia tree is at most $2N$.

2.2.3 Some Properties of Strings and Tries

Given an ordered alphabet Σ, the *lexicographic order* on Σ^* is defined by $x \leqslant y$ if and only if either string x is a prefix of string y or $x = uav$ and $y = ubw$, with $a, b \in \Sigma$, $a < b$, and $u, v, w \in \Sigma^*$. Thus, given a set of strings $\mathcal{S}$, it can be lexicographically sorted. We remind that, in lexicographic order, for any strings $x, y \in \Sigma^*$ and symbols $a, b \in \Sigma$, the following properties hold:

(1) $\varepsilon \leqslant x$;

(2) $ax \leqslant by$ if $a < b$; and

(3) $ax \leqslant ay$ if $x \leqslant y$.

Moreover, if the set is arranged in a trie data structure, then all the strings represented in a subtree form a lexicographical interval of the universe. If we number the non-empty nodes of a trie in preorder, then the subtree of a node forms a preorder interval.

Example 2.3. In the tries of Fig. 2.1 we show preorder numbers (i.e., lexicographic rank of the corresponding strings) outside each trie node (we only count non-empty nodes in the enumeration). In the trie of Fig. 2.1(a), the preorder interval [1..11] corresponds to all strings in the set $\mathcal{S}_1$ that start with '`a`', and the preorder interval [6..9] corresponds to all strings in $\mathcal{S}_1$ that start with '`ar`'.

2.3 Classical Indexes for Full-Text Searching

We review in this section the classical indexes for full-text searching, namely suffix trees and suffix arrays. Given a text T over an alphabet Σ, we assume that a special end-of-string symbol '\$' $\notin \Sigma$ is appended to T. We assume that symbol '\$' is smaller than any other symbol in the alphabet.

2.3.1 Suffix Trees

The *suffix tree* [Wei73, McC76, Apo85] of a text $T[1..u]$, denoted by ST_T, is one of the most known and elegant data structures in the algorithmic literature. It allows us, among many other applications, to search for a given pattern $P[1..m]$ based on a simple idea: every occurrence of P in T is the prefix of some suffix of T. Thus, we represent all the text suffixes in a data structure supporting prefix searches, for instance a trie. If we carry out a prefix search with P on this trie, we get the set of text suffixes that have P as a prefix, thus finding all the occurrences of P in T. This trie is called the *suffix trie* of T.

To get ST_T we prune the trie at a node as soon as there is only a unary path from the node to a leaf. We store in that leaf a pointer to the text position where the corresponding suffix starts. Moreover, every unary path in the trie is converted into a single edge, in the same way as in a Patricia tree, which is conceptually labeled with the substring that represents the replaced path. Thus, ST_T has $O(u)$ nodes [Mor68, Apo85], because we are representing u text suffixes in a Patricia tree. The edge labels are actually represented by storing the starting position in the text for the edge label, plus the length of the label. To descend in ST_T we have to access the text at the corresponding position and compare these symbols against that of the pattern we are looking for.

To search for the *occ* occurrences of a pattern P in T, we descend in ST_T using the symbols of P, up to:

(1) reaching a node v representing P; or

(2) we reach a leaf without consuming all of P.

If at some node we cannot descend anymore, we know that the pattern does not occur in T. For case (1), this takes $O(m)$ time. For counting the number of occurrences, we must

count the number of leaves within the subtree of node v. This can be done in constant time if we also store the subtree size of each node. For locating the text positions of the occurrences, we traverse the subtree of v, for example in preorder, and report the values stored in the leaves of this subtree. Notice that since this subtree has *occ* leaves and we are compressing unary paths, this subtree has $O(occ)$ nodes. Thus, the overall search time is optimal $O(m + occ)$. For case (2), we are sure that there is at most one occurrence of P in T. Then, we use the leaf to access the corresponding text position, and compare with the pattern (to determine whether it exists in T or not).

2.3.2 Suffix Arrays

The *suffix array* [MM93, GBYS92] of a text $T[1..u]$, denoted by $SA_T[1..u]$, is a lexicographically sorted array of the text suffixes. Given i, $SA_T[i]$ stores the starting text position for the i-th suffix in the lexicographic order. Suffix arrays are one of the most used data structures in practice for full-text searching. A suffix array requires $u \log u$ bits of space if no auxiliary data structure is used [MM93], which is typically 4 times the size of the text (assuming that text symbols are implemented in 1 byte, while integers require 4 bytes)

In such a case, given a pattern $P[1..m]$ we are capable to find the suffix-array interval $[i_1, i_2]$ containing the starting positions of the pattern occurrences in $O(m \log u)$ time, by binary searching SA_T (as it is sorted). This corresponds to counting the pattern occurrences, as the size of the interval equals the number of occurrences. To find the starting positions, we traverse the interval $[i_1, i_2]$ and report the positions stored in it, in $O(1)$ time per occurrence, resulting in the total search time of $O(m \log u + occ)$.

Example 2.4. In Fig. 2.2 we show the suffix array for our running example text, $T =$"`alabar_a_la_alabarda_para_apalabrarla`", where for clarity we replace blanks by '_', which is assumed to be lexicographically larger than any other symbol in the alphabet. We also show the corresponding (cyclically shifted) text suffix in each case. If we search, for example, for the pattern '`ala`', we will get the interval [6..8] in the suffix array (we underline the prefix '`ala`' of the corresponding suffixes).

Property 2.1. Given a suffix starting at position $SA_T[i]$ in T, its longest proper suffix has position $SA_T[i] + 1$ in T.

Notice that SA_T is a permutation, and that SA_T^{-1} denotes its inverse permutation.

Property 2.2. Given a suffix starting at position j of the text, its lexicographic rank among all suffixes is $SA_T^{-1}[j]$.

i	$SA_T[i]$	suffix (cyclically shifted)	$btw(T)$	$\Psi[i]$
1	38	$alabar_a_la_alabarda_para_apalabrarla	a	7
2	37	a$alabar_a_la_alabarda_para_apalabrarl	l	1
3	15	abarda_para_apalabrarla$alabar_a_la_al	l	18
4	3	abar_a_la_alabarda_para_apalabrarla$al	l	19
5	31	abrarla$alabar_a_la_alabarda_para_apal	l	20
6	13	alabarda_para_apalabrarla$alabar_a_la_	_	23
7	1	alabar_a_la_alabarda_para_apalabrarla$	$	24
8	29	alabrarla$alabar_a_la_alabarda_para_ap	p	25
9	27	apalabrarla$alabar_a_la_alabarda_para_	_	27
10	23	ara_apalabrarla$alabar_a_la_alabarda_p	p	30
11	17	arda_para_apalabrarla$alabar_a_la_alab	b	31
12	34	arla$alabar_a_la_alabarda_para_apalabr	r	32
13	5	ar_a_la_alabarda_para_apalabrarla$alab	b	33
14	11	a_alabarda_para_apalabrarla$alabar_a_l	l	34
15	25	a_apalabrarla$alabar_a_la_alabarda_par	r	35
16	8	a_la_alabarda_para_apalabrarla$alabar_	_	37
17	20	a_para_apalabrarla$alabar_a_la_alabard	d	38
18	16	barda_para_apalabrarla$alabar_a_la_ala	a	11
19	4	bar_a_la_alabarda_para_apalabrarla$ala	a	13
20	32	brarla$alabar_a_la_alabarda_para_apala	a	29
21	19	da_para_apalabrarla$alabar_a_la_alabar	r	17
22	36	la$alabar_a_la_alabarda_para_apalabrar	r	2
23	14	labarda_para_apalabrarla$alabar_a_la_a	a	3
24	2	labar_a_la_alabarda_para_apalabrarla$a	a	4
25	30	labrarla$alabar_a_la_alabarda_para_apa	a	5
26	10	la_alabarda_para_apalabrarla$alabar_a_	_	14
27	28	palabrarla$alabar_a_la_alabarda_para_a	a	8
28	22	para_apalabrarla$alabar_a_la_alabarda_	_	10
29	33	rarla$alabar_a_la_alabarda_para_apalab	b	12
30	24	ra_apalabrarla$alabar_a_la_alabarda_pa	a	15
31	18	rda_para_apalabrarla$alabar_a_la_alaba	a	21
32	35	rla$alabar_a_la_alabarda_para_apalabra	a	22
33	6	r_a_la_alabarda_para_apalabrarla$alaba	a	36
34	12	_alabarda_para_apalabrarla$alabar_a_la	a	6
35	26	_apalabrarla$alabar_a_la_alabarda_para	a	9
36	7	_a_la_alabarda_para_apalabrarla$alabar	r	16
37	9	_la_alabarda_para_apalabrarla$alabar_a	a	26
38	21	_para_apalabrarla$alabar_a_la_alabarda	a	28

Figure 2.2: Suffix array SA_T for the running example text. The text suffix indexed in each row is shaded. We also show two concepts related to suffix arrays: the *Burrows-Wheeler Transform* and the Ψ function of *Compressed Suffix Arrays*.

2.4 Text Compression

Text compression is a technique used to *represent a text using less space*, taking advantage of the regularities of non-random texts. It has two main advantages [BCW90]:

Advantage 1: It reduces the space requirement of the text, and

Advantage 2: It reduces the cost and increases the effective speed of text transmission, both between computers in a network and from secondary to main memory (where a compressed text is read/transferred faster than its uncompressed form).

The main disadvantage of compression is that processing time is increased, as we have to uncompress (a part of) the text in order to process it. Yet, the number of accesses to secondary memory are reduced (that is, there is a trade-off between CPU time and secondary memory accesses). More and more, it is advantageous to pay more CPU time for less I/O.

A concept related to text compression, and that we shall use in this thesis to model the compressibility of a text, is that of the k-th order empirical entropy of a sequence of symbols T over an alphabet of size σ, denoted by $H_k(T)$ [Man01]. The value $uH_k(T)$ provides a lower bound to the number of bits needed to compress T using any compressor that encodes each symbol considering only the context of k symbols that precedes it in T. Formally, we have:

Definition 2.2. Given a text $T[1..u]$ over an alphabet Σ, the *zero-order empirical entropy* of T is defined as

$$H_0(T) = \sum_{c \in \Sigma} \frac{n_c}{u} \log \frac{u}{n_c}$$

where n_c is the number of occurrences of symbol c in T. The sum includes only those symbols c that do occur in T, so that $n_c > 0$.

Definition 2.3. Given a text $T[1..u]$ over an alphabet Σ, the *k-th order empirical entropy* of T is defined as

$$H_k(T) = \sum_{s \in \Sigma^k} \frac{|T^s|}{u} H_0(T^s)$$

where T^s is the subsequence of T formed by all the symbols that occur preceded by the context s. Again, we consider only contexts s that do occur in T.

Property 2.3. Given a text T over an alphabet of size σ, it holds that $0 \leqslant H_k(T) \leqslant H_{k-1}(T) \leqslant \cdots \leqslant H_0(T) \leqslant \log \sigma$, for any $k \geqslant 0$.

This means that with longer contexts we get more compression. We remember now some bounds on the zero-order entropy of a binary string.

Property 2.4. Let $\mathcal{B}[1..u]$ be a binary string of length u, with $n \leqslant u$ bits set. Then:

(1) $\log \binom{u}{n} \leqslant uH_0(\mathcal{B}) \leqslant \log \binom{u}{n} + O(\log n)$.

(2) $n \log \frac{u}{n} \leqslant uH_0(\mathcal{B}) \leqslant n \log \frac{u}{n} + n \log e$.

Next we review two known techniques for text compression, which allow for indexed text searching: *LZ78 Compression* [ZL78] and the *Burrows-Wheeler transform* [BW94]. We also introduce the first concepts about searching in compressed texts.

2.4.1 Lempel-Ziv Compression

The general idea of *Lempel-Ziv compression* is to replace substrings in the text by a pointer to a previous occurrence of them [LZ76]. We obtain compression whenever the pointer requires less space than the string it is replacing. There exist different variants based on this type of compression [ZL77, ZL78, BCW90], yet we are particularly interested in the LZ78 format, mainly because LZ78 is the base of many practical compression tools, and because its properties make it more amenable for full-text searching [NR04] and full-text self-indexing (for example, the fast extraction of text substrings, a very important aspect for full-text self-indexes). Another interesting choice could be to focus on the LZ77 compression algorithm. However, using this for self-indexing is still full of challenges and problems that need to be solved (as for example, the fast extraction of text substrings). However, our results can also be applied to, for instance, the LZW compression algorithm [Wel84], since this is just a coding variant of LZ78. LZW is commonly used in practice, for example by Unix's `compress` tool.

The Lempel-Ziv compression algorithm of 1978 (usually named LZ78 [ZL78]) is based on a *dictionary of phrases*, in which we add every new *phrase* computed. At the beginning of the compression, the dictionary contains a single phrase b_0 of length 0 (i.e., the empty string). The current step of the compression is as follows: If we assume that a prefix $T[1..j]$ of T has been already compressed into a sequence of phrases $Z = b_1 \dots b_r$, all of them in the dictionary, then we look for the longest prefix of the rest of the text $T[j+1..u]$ which is a phrase of the dictionary. Once we have found this phrase, say b_s of length ℓ_s, we construct a new phrase $b_{r+1} = (s, T[j+\ell_s+1])$, write the pair at the end of the compressed file Z, that is, $Z = b_1 \dots b_r b_{r+1}$, and add the phrase to the dictionary.

We will call B_i the string represented by phrase b_i, thus $B_{r+1} = B_s T[j+\ell_s+1]$. Then, the LZ78 algorithm compresses the text T into a set of $n+1$ phrases $B_0, \dots, B_n$, such that $T = B_0 \dots B_n$, and $B_0 = \varepsilon$ (the empty string). We say that i is the *phrase identifier* corresponding to B_i, for $0 \leqslant i \leqslant n$.

Property 2.5. For all $1 \leqslant t \leqslant n$, there exists $\ell < t$ and $c \in \Sigma$ such that $B_t = B_\ell \cdot c$.

That is, every phrase B_t (except B_0) is formed by a previous phrase B_ℓ plus a symbol c at the end. This implies that the set of phrases is *prefix closed*, meaning that any prefix of a phrase B_t is also an element of the dictionary. Therefore, an alternative way to represent the set of strings $B_0, \ldots, B_n$ is a trie, which we call the *LZTrie*.

Property 2.6. Every phrase B_i, $0 \leqslant i < n$, represents a different text substring.

The only exception to this property is the last phrase B_n. We deal with the exception by appending to T a special symbol '\$' $\notin \Sigma$, assumed to be smaller than any other symbol in the alphabet. The last phrase will contain this symbol and thus will be unique too.

Example 2.5. In Fig. 2.3(a) we show the LZ78 phrase decomposition for our running example text. We show the phrase identifiers above each corresponding phrase in the parsing. In Fig. 2.3(c) we show the corresponding *LZTrie*. Inside each *LZTrie* node we show the corresponding phrase identifier.

Definition 2.4. Let $b_r = (r_1, c_1)$, $b_{r_1} = (r_2, c_2)$, $b_{r_2} = (r_3, c_3)$, and so on until $r_k = 0$ be phrases of the LZ78 parsing of T. The sequence of phrase identifiers $r, r_1, r_2, \ldots$ is called the *referencing chain* starting at phrase r.

The referencing chain starting at phrase r reproduces the way phrase b_r is formed from previous phrases and it is obtained by successively moving to the parent in the *LZTrie*. This concept will be useful in order to get the text corresponding to a given phrase.

Example 2.6. For example, the referencing chain of phrase 9 in Fig. 2.3(c) is $r = 9$, $r_1 = 7$, $r_2 = 2$, and $r_3 = 0$.

Alternative Encodings of the LZ78 Parsing. We study here two alternative encodings for the LZ78 parsing of text T:

Variant 1. The LZ78 phrases are represented by using two arrays, $S[1..n]$ of $n \log \sigma$ bits and $A[1..n]$ of $n \log n$ bits. If $B_i = B_j \cdot c$ is a phrase in the LZ78 parsing of T, then we represent the i-th phrase by storing the reference $A[i] \leftarrow j$ and the symbol $S[i] \leftarrow c$. If a phrase is composed just by a single symbol, then we store a value 0 in the corresponding position of A. This representation requires $n(\log n + \log \sigma)$ bits of space. See Fig. 2.3(b) for an illustration of variant 1.

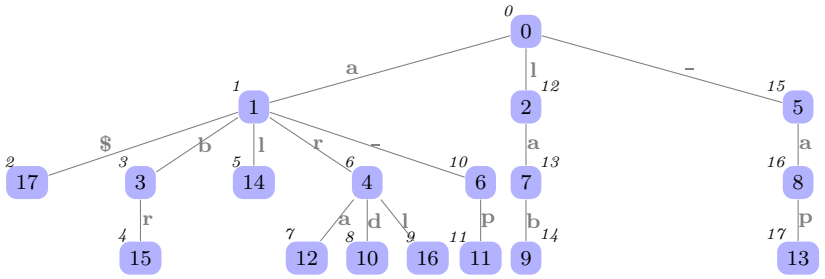
For the decompression, to compute the string B_i corresponding to the i-th LZ78 phrase we must follow the referencing chain for the phrase, obtaining the symbols that compose B_i and stopping the procedure as soon as we get a 0 in A. This procedure takes $O(|B_i|)$ time.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
a	l	ab	ar	_	a_	la	_a	lab	ard	a_p	ara	_ap	al	abr	arl	a$

(a) LZ78 phrase decomposition for the running example text T = "alabar_a_la_alabarda_para_apalabrarla", and the corresponding phrase identifiers.

phrase id.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
A	0	0	1	1	0	1	2	5	7	4	4	4	8	1	3	4	1
S	a	l	b	r	_	_	a	a	b	d	l	a	p	l	r	l	$

(b) Array representation for the LZ78 parsing of the running-example text.



(c) Lempel-Ziv Trie (*LZTrie*) for the running example. Preorder numbers are shown outside each node.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$Node[i]$	①	⑫	③	⑥	⑮	⑩	⑬	⑯	⑭	⑧	⑪	⑦	⑰	⑤	④	⑨	②

(d) *Node* data structure for the running example. The notation ⓙ represents a pointer to the *LZTrie* node with preorder j.

Figure 2.3: LZ78 parsing and the corresponding $LZTrie$ for our running example text.

We can uncompress the whole text (and in fact any set of phrases) by extracting B_1, B_2, and so on, in $O(u)$ overall time. However, this representation does not support quickly finding the longest prefix of the rest of the text that matches a phrase already in the dictionary, and thus it does not support the efficient construction of the LZ78 parsing.

Variant 2. We represent the set of LZ78 phrases with the *LZTrie*, plus a mapping *Node*$[1..n]$ such that *Node*$[i]$ stores the *LZTrie* node corresponding to phrase B_i. The *LZTrie* requires $n \log \sigma$ bits of space to represent the symbols labeling the trie, plus the space needed to represent the trie topology, which is $2n + o(n)$ bits if we use a succinct encoding [MR01, BDM$^+$05] (see Section 2.5.2 of this thesis for a review of succinct encodings of trees). The space used by *Node* is basically $n \log n$ bits if we store the preorders of the nodes (many succinct encodings of trees allows one to retrieve a node given its preorder, see Section 2.5.2). Thus, the space requirement of this representation is basically $n(\log n + \log \sigma + 2)$ bits.

Example 2.7. See Fig. 2.3(c) and Fig. 2.3(d) for an illustration of *LZTrie* and *Node* data structures respectively.

To extract a given phrase B_i, we first access to node *Node*$[i]$ in *LZTrie*, and then extract the symbols labeling the upward path, by going to the parent until reaching the trie root. Once again, many succinct-tree encodings support going to the parent of a node in constant time, while still requiring little space. So this procedure takes $O(|B_i|)$ time.

As for *Variant 1*, we can uncompress the whole text by extracting B_1, B_2, and so on, in $O(u)$ time overall. The compression algorithm takes also $O(u)$ time and it is efficient in practice provided we use the *LZTrie* for the construction, which allows rapid searching of the new text prefix (for each symbol of T we move once in the trie).

Thus, both representations require basically $n(\log n + \log \sigma)$ bits of space to represent the output of the LZ78 compression algorithm. In our work, however, we shall use *Variant 2*, since the *LZTrie* allows for search capabilities on the compressed text. Moreover, Grossi and Sadakane [GS06] use this representation to support extracting ℓ arbitrary contiguous symbols of T in optimal $O(\frac{\ell}{\log_\sigma u})$ time (compare this against the $O(\ell)$ time of the method in Variant 2).

Some Properties of the LZ78 Parsing. We review here some key properties of the LZ78 parsing of a text T.

Property 2.7 ([ZL78])**.** It holds that $\sqrt{u} \leqslant n \leqslant \frac{u}{\log_\sigma u}$. Thus, $\log n = \Theta(\log u)$ and $n \log u \leqslant u \log \sigma$ always hold.

For strings $T[1..u]$ generated from a *stationary ergodic source* (i.e., a model where an indefinitely long string is a representative of the entire source [WMB99]), we have that the ratio $\frac{n \log n}{u}$ achieved by the LZ78 parsing tends to H, the entropy of the source [BCW90]. Since the LZ78 representation of T uses basically $n(\log n + \log \sigma)$ bits of space (as we have seen before), the compression ratio of LZ78 is asymptotically optimal, since the $n \log n$ term dominates.

Thus, LZ78 is able to compress an indefinitely long string within the optimal space, according to the entropy model. This differentiates LZ78 compression from many other compression techniques not achieving this important property [BCW90]. However, optimality is achieved when $u \to \infty$. It is well known that LZ78 converges to the optimal very slowly as the text length grows. Despite of this, LZ78 has a reasonable performance in practice (for instance, the well-known `compress` compressor is based on LZ78), and its nice combinatorial properties lead to effective indexing approaches [Kär99, Nav04].

We shall use the following result of Kosaraju and Manzini [KM99] to bound the output of the LZ78 parsing of text T in terms of the k-th order empirical entropy of T. The extra big-oh space term reflects, for finite texts, the slow asymptotic convergence.

Lemma 2.1 ([KM99])**.** *It holds that* $n \log n = uH_k(T) + O(u\frac{1+k \log \sigma}{\log_\sigma u})$ *for any* k.

In our work we assume $k = o(\log_\sigma u)$ (and hence $\log \sigma = o(\log u)$ to allow for $k > 0$, i.e., high-order compression); so that, $n \log n = uH_k(T) + o(u \log \sigma)$.

By using Property 2.4, we can prove the following:

Lemma 2.2. *Let* $T[1..u]$ *be a text over an alphabet of size* σ*, and let* n *be the number of phrases in the LZ78 parsing of* T*. Let* $\mathcal{B}[1..u]$ *be a binary string of length* u *and* n *bits set. Then, it holds that* $nH_0(\mathcal{B}) = o(u \log \sigma)$.

Proof. Because of Property 2.4 (2) we have that $uH_0(\mathcal{B}) \leqslant n \log \frac{u}{n} + n \log e$. Since $\log \frac{u}{n}$ grows with n and since $n \leqslant \frac{u}{\log_\sigma u}$ (Property 2.7), replacing n by $\frac{u}{\log_\sigma u}$ yields $n \log \frac{u}{n} + n \log e \leqslant \frac{u}{\log_\sigma u}(\log \log_\sigma u + \log e)$, which is $o(u \log \sigma)$. □

We can also prove the following result, which is related to Lemma 2.1 and will be useful throughout our work.

Lemma 2.3. *It holds that* $n \log u = uH_k(T) + o(u \log \sigma)$ *for any* $k = o(\log_\sigma u)$.

Proof. Note that $n \log u = n \log n + n \log \frac{u}{n}$. As shown in the proof of Lemma 2.2, $n \log \frac{u}{n} = o(u \log \sigma)$. On the other hand, because of Lemma 2.1 we have that $n \log n = uH_k(T) + o(u \log \sigma)$ for any $k = o(\log_\sigma u)$. Overall, we have that $n \log u = uH_k(T) + o(u \log \sigma)$ for any $k = o(\log_\sigma u)$. □

2.4.2 The Burrows-Wheeler Transform, the Backward-Search Concept, and the Ψ Function

The Burrows-Wheeler Transform. The *Burrows-Wheeler Transform* [BW94] of a text T ($bwt(T)$ for short) produces a permutation of the text which is easier to compress than the original text. To compute the transform, after appending the special symbol '\$' to T, we construct the conceptual matrix M whose rows are the lexicographically-sorted cyclic shifts of $T\$$, and finally take the last column of M as the $bwt(T)$. This is a reversible transform, as we can recover the original text from $bwt(T)$.

Example 2.8. In Fig. 2.2 we show the Burrows-Wheeler transform for our running example text.

As it can be noted in Fig. 2.2, there is a close relation between suffix arrays and the Burrows-Wheeler transform: as every row in M corresponds to a text suffix, and since rows are lexicographically sorted[1], if we associate to each row the starting position of the suffix what we get is the suffix array of T. Also, note that the last column of M (i.e., $bwt(T)$) is the symbol preceding each suffix of T, that is, $T[SA_T[i] - 1]$ (assuming $T[0] = T[u]$).

The Backward Search Concept. Given the close relation between suffix arrays and the $bwt(T)$, Ferragina and Manzini [FM05] define the concept of *backward search*, which consists of looking for the suffix array interval containing the pattern occurrences just using the $bwt(T)$. Given a pattern $P = p_1 \dots p_m$, for $p_i \in \Sigma$, the search proceeds in $O(m)$ steps: in the first step, we find the suffix array interval for the occurrences of p_m; in the second step, using the result of the previous step, we find the interval corresponding to $p_{m-1}p_m$ and so on, to finally find the suffix array interval for the whole pattern $p_1 \dots p_m$.

Example 2.9. For instance, if we search for the pattern '`bar`' in our example of Fig. 2.2, we first find the interval for the occurrences of '`r`', which is [29..33], then the interval [10..13] for the occurrences of '`ar`', to finally find the interval [18..19] corresponding to the occurrences of '`bar`'.

The first realization of the backward-search concept was the *FM-index* [FM05], yet its use is appropriate only for texts on small alphabets (e.g., constant-size alphabets) because of an exponential dependence on σ in the space requirement. A further improvement on this line is the *Alphabet-Friendly FM-index* (AF-FMI) [FMMN07], which avoids these alphabet dependencies and requires $uH_k(T) + o(u \log \sigma)$ bits of space. In this index, each step of the backward search takes $O(1 + \frac{\log \sigma}{\log \log u})$ time, and thus the counting time is $O(m(1 + \frac{\log \sigma}{\log \log u}))$, which is $O(m)$ time whenever $\sigma = O(\text{polylog}(u))$ holds. It supports the location of the pattern occurrences in $O(\log^{1+\epsilon} u)$ time per occurrence, provided it stores

[1] Because of the unique terminator '\$', sorting the rows is the same as sorting the suffixes.

a sampling of the suffix array requiring $o(u \log \sigma)$ extra bits of space. Although requiring "sublinear" space, this extra information stored is not compressible at all, as it consists of a sampling of raw suffix-array positions. For instance, to achieve the previous locate time a sampling step of $\frac{\log^{1+\epsilon} u}{\log \sigma}$ must be used [FGNV08]. By using the same samplig step, the time to extract any string of length ℓ is $O(\log^{1+\epsilon} u + \ell(1 + \frac{\log \sigma}{\log \log u}))$.

The Ψ Function and Compressed Suffix Arrays. Another concept related to suffix arrays is that of function Ψ [GV05], which also allows us to compress the suffix array. This function is defined as $\Psi[i] = SA_T^{-1}[SA_T[i] + 1]$, for $i = 2, \ldots, u$, and $\Psi[1] = SA_T^{-1}[1]$. In other words, $\Psi[i]$ stores the suffix array position (i.e., lexicographic rank) of the longest proper suffix of $SA_T[i]$, and hence it can be seen as a *suffix link* for $SA_T[i]$ [Ukk95].

A naive way to represent Ψ requires $u \log u$ bits of space, the same as the suffix array itself. However, it can be shown that Ψ can be divided into σ strictly increasing sequences: for every $i < j$, if $T[SA_T[i]] = T[SA_T[j]]$ holds, then $\Psi[i] < \Psi[j]$.

Thus, Ψ can be represented in the following way in order to require $uH_0(T)+o(u \log \sigma)$ bits of space [Sad03]: rather than storing $\Psi[i]$, store the δ-code [Eli75] of the differences $\Psi[i] - \Psi[i-1]$ if $T[SA_T[i]] = T[SA_T[j]]$. Otherwise store the δ-code of $\Psi[i]$.

In order to access $\Psi[i]$ in constant time, absolute Ψ values are inserted every $\Theta(\log u)$ bits, which adds $O(u)$ bits. To extract an arbitrary position of Ψ, we go to the nearest absolute sample before that position and then sequentially advance summing up differences until reaching the desired position. By maintaining a precomputed table with the total number of differences encoded inside every possible chunk of $\frac{\log u}{2}$ bits, we can process each such chunk in constant time, so the $\Theta(\log u)$ bits of differences can be processed in constant time. The size of that table is only $O(\sqrt{u} \log^2 u) = o(u)$ bits. See [Sad03] for further details.

Example 2.10. In Fig. 2.2 we show Ψ for the running example. We divide the rows of the suffix array to separate suffixes starting with the same symbol. Notice the strictly increasing runs of Ψ within these intervals.

2.5 Succinct Data Structures

A *succinct data structure* requires space close to the information-theoretic lower bound, while supporting the corresponding operations efficiently. In this section and the next, we review some results on succinct data structures, which are necessary to understand our work.

2.5.1 Data Structures for Sequences, Permutations, and Range Searching

Data Structures for Rank and Select. Given a bit vector $\mathcal{B}[1..n]$, we define the operation $rank_0(\mathcal{B}, i)$ (similarly $rank_1$) as the number of 0s (1s) occurring up to the i-th position of $\mathcal{B}$. The operation $select_0(\mathcal{B}, i)$ (similarly $select_1$) is defined as the position of the i-th 0 (i-th 1) in $\mathcal{B}$. We assume that $select_0(\mathcal{B}, 0)$ always equals 0 (similarly for $select_1$). We also define operation $access(\mathcal{B}, i)$, which yields the value $\mathcal{B}[i]$. These operations can be supported in constant time and requiring $n + o(n)$ bits [Mun96], or even $nH_0(\mathcal{B}) + o(n)$ bits [RRR02].

Lemma 2.4. *Let $\mathcal{B}[1..n]$ denote a binary sequence, then:*

(1) There exists a representation [Mun96] that requires $n + o(n)$ bits of space and supports operations rank, select, and access on $\mathcal{B}$ in constant time.

(2) There exists a representation [RRR02] that requires $nH_0(\mathcal{B}) + o(n)$ bits of space and supports operations rank, select, and access on $\mathcal{B}$ in constant time.

There exist a number of practical data structures supporting *rank* and *select*, like the one by González et al. [GGMN05], Kim et al. [KNKP05], Okanohara and Sadakane [OS07], Claude and Navarro [CN08b], etc. Among these, the index of González et al. [GGMN05] is very (perhaps the most) efficient in practice to compute *rank*, requiring little space on top of the sequence itself. Operation *select* is implemented by binary searching the directory built for operation *rank*, and thus without requiring any extra space for that operation (yet, the time for *select* becomes $O(\log n)$).

Given a sequence $S[1..n]$ over an alphabet $\Sigma = \{1, \ldots, \sigma\}$, we generalize the above definition for $rank_c(S, i)$ and $select_c(S, i)$ for any $c \in \Sigma$. If $\sigma = O(\text{polylog}(n))$, the solution of Ferragina at al. [FMMN07] allows one to compute both $rank_c$ and $select_c$, as well as accessing to $S[i]$ for any i, in constant time and requiring $nH_0(S) + o(n)$ bits of space. Otherwise the time is $O(\frac{\log \sigma}{\log \log n})$ and the space is $nH_0(S) + o(n \log \sigma)$ bits. The representation of Golynski et al. [GMR06] requires $n(\log \sigma + o(\log \sigma)) = O(n \log \sigma)$ bits of space [BHMR07], allowing us to compute $select_c$ in $O(1)$ time, and $rank_c$ and access to $S[i]$ in $O(\log \log \sigma)$ time.

Lemma 2.5. *Let $S[1..n]$ denote a sequence over an alphabet Σ of size σ, then*

(1) There exists a representation [FMMN07] that requires $nH_0(S) + o(n \log \sigma)$ bits of space and supports operations $rank_c$ and $select_c$ on S, for any $c \in \Sigma$, as well as access to any $S[i]$, in $O(\frac{\log \sigma}{\log \log n})$ time.

(2) There exists a representation [GMR06] that requires $O(n \log \sigma)$ bits of space and supports operations $rank_c$ and access to any $S[i]$ in $O(\log \log \sigma)$ time, and $select_c$ on S in constant time, for any $c \in \Sigma$.

Data Structures for Searchable Partial Sums. Given an array $A[1..n]$ of n integers of k' bits each, a data structure for searchable partial sums allows one to retrieve $A[i]$ and supports operations:

— $Sum(A, i)$, which computes $\sum_{j=1}^{i} A[j]$;

— $Search(A, i)$, which finds the smallest j' such that $Sum(A, j') \geqslant i$;

— $Update(A, i, \delta)$, which sets $A[i] \leftarrow A[i] + \delta$;

— $Insert(A, i, e)$, which adds a new element e to the set between elements $A[i-1]$ and $A[i]$; and

— $Delete(A, j)$, which deletes element $A[j]$.

The data structure of [MN08b] supports all these operations in $O(\log n)$ worst-case time, and requires $nk' + o(nk')$ bits of space. It is interesting to note that the space can be made $nk' + O(n)$ bits, see [MN08b] for details.

Lemma 2.6 ([MN08b])**.** *There exists a representation for a sequence of n integers of k' bits each, requiring $nk' + O(n)$ bits of space and supporting operations Sum, Search, Update, Insert, and Delete, all of them in $O(\log n)$ worst-case time.*

Succinct Representation of Permutations. The problem here is to represent a permutation π of $[n] = \{1, \ldots, n\}$, such that we can compute both $\pi(i)$ and its inverse $\pi^{-1}(j)$ in constant time and using as little space as possible. A natural representation for π is to store the values $\pi(i)$, $i = 1, \ldots, n$, in an array of $n \log n$ bits. The brute-force solution to the problem computes $\pi^{-1}(j)$ by looking for j sequentially in the array representing π. If j is stored at position i, that is, $\pi(i) = j$, then $\pi^{-1}(j) = i$. Although this solution does not require any extra space to compute π^{-1}, it takes $O(n)$ time in the worst case.

A much more efficient solution is based on the cycle notation of a permutation. The cycle for the i-th element of π is formed by elements i, $\pi(i)$, $\pi(\pi(i))$, and so on until the value i is found again. It is important to note that every element occurs in one and only one cycle of π.

Example 2.11. For example, the cycle notation for permutation *ids* of Fig. 2.4(a) is shown in Fig. 2.4(b).

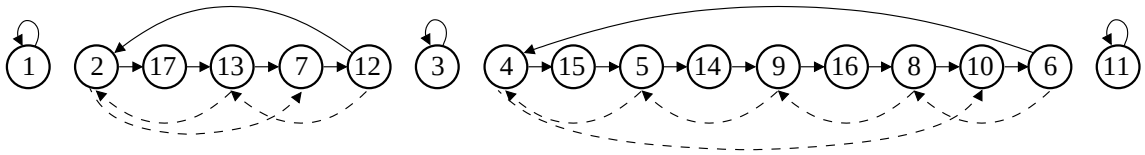
So, to compute $\pi^{-1}(j)$, instead of looking sequentially for j in π, we only need to look for j in its cycle: $\pi^{-1}(j)$ is just the value "pointing" to j in the diagram of Fig. 2.4(b).

Example 2.12. To compute $ids^{-1}(13)$, we start at position 13, then move to position $ids(13) = 7$, then to position $ids(7) = 12$, then to $ids(12) = 2$, then to $ids(2) = 17$, and as $ids(17) = 13$ we conclude that $ids^{-1}(13) = 17$.

The only problem here is that there are no bounds for the size of a cycle, hence this algorithm takes also $O(n)$ in the worst case. However, it can be improved for a more efficient computation of $\pi^{-1}(j)$.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$ids[i]$	1	17	3	15	14	4	12	10	16	6	11	2	7	9	5	8	13

(a) An example of permutation ids.



(b) Cycle notation of permutation ids.

Figure 2.4: Cycle representation of a permutation. Each solid arrow $i \rightarrow j$ in the diagram means $ids(i) = j$. Dashed arrows represent backward pointers.

Given $0 < \epsilon < 1$, we create subcycles of size $O(1/\epsilon)$ by adding a *backward pointer* out of $O(1/\epsilon)$ elements in each cycle of π. Dashed arrows in Fig. 2.4(b) show backward pointers for $1/\epsilon = 2$.

Example 2.13. Now, to compute $ids^{-1}(17)$, we first move to $ids(17) = 13$; as 13 has a backward pointer we follow it and hence we move to position 2. Then, as $ids(2) = 17$ we conclude that $ids^{-1}(17) = 2$, in $O(1/\epsilon)$ worst-case time.

We store the backward pointers compactly in an array of $\epsilon n \log n$ bits. We mark the elements having a backward pointer by using a bit vector supporting *rank* queries, which also help us to find the backward pointer corresponding to a given element (see [MRRR03] for details). Overall, this solution requires $(1+\epsilon)n \log n + n + o(n)$ bits of storage if we use the data structure of Lemma 2.4 (1) to support operation *rank*.

Lemma 2.7 ([MRRR03])**.** *There exists a representation for a permutation π of $[n]$ that requires $(1+\epsilon)n \log n + n + o(n)$ bits of space, for any $0 < \epsilon < 1$, and supports the computation of $\pi(i)$ in $O(1)$ time, as well as the inverse permutation $\pi^{-1}(j)$ in $O(1/\epsilon)$ time.*

Next we present a result which shall be useful later for our purposes of constructing the LZ-index for a text T. This states that any permutation π can be inverted in-place in linear time and using only n extra bits of space. This can be seen as a particular case of *rearranging a permutation* [FMP95], where we are given an array and a permutation, and want to rearrange the array according to the permutation.

Lemma 2.8. *Given a permutation π of $\{1, \ldots, n\}$ represented by an array using $n \log n$ bits of space, we can compute on the same array the inverse permutation π^{-1} in $O(n)$ time and requiring n bits of extra space.*

Proof. Let $A_\pi[1..n]$ be an auxiliary bit vector requiring n bits of storage, which is initialized with all zeros (this is just the raw bit vector, no additional data structure for *rank* and *select* is added). Let π be the array representing the permutation, using $n \log n$ bits of space. The idea to construct π^{-1} is to use the cycle structure of π to reverse the "arrows" that form the cycles (i.e., "$i \to j$" in a cycle of π, which means $\pi[i] = j$, now becomes "$i \leftarrow j$", which means $\pi^{-1}[j] = i$). So, the main idea is to regard the cycles of π as "linked lists". Thus, constructing π^{-1} is a matter of reversing the pointers in the lists, and therefore we shall need three auxiliary pointers to do that job. We follow the cycles of π, using A_π to mark with a `1` those positions which have been already visited during this process.

We start with the cycle at position $a \leftarrow 1$, and traverse it from position $p \leftarrow \pi[a]$. We then set $b \leftarrow \pi[p]$, $\pi[p] \leftarrow a$ (i.e., we store the position a which brings us to the current one), and $A_\pi[p] \leftarrow$ `1`. Then we move to position $a \leftarrow p$, set $p \leftarrow b$, and repeat the process again, stopping as soon as we find a `1` in A_π, where p has value 1. Then we try with the cycle starting at position $p + 1$, which is the next one after the position that started the previous cycle, and follow it if the corresponding bit in A_π is `0`. Otherwise we try with $p + 2$, $p + 3$, until we traverse the whole array.

Thus, each element in the permutation is visited twice: elements starting a cycle are visited at the beginning and at the end of the cycle, while elements in the middle of a cycle are visited when traversing the cycle to which they belong, and when trying to start a cycle from them. Thus, the overall time is $O(n)$, we use n extra bits on top of the space of π, and the lemma follows. □

Succinct Representation for Two-Dimensional Range Searching. *Orthogonal Range Searching* is a classical problem in *Computational Geometry* [Mat94, AE99], with a vast number of applications [dBCvKO08]. In the two-dimensional case, we are given a set U of n points in the plane, and want to find those points which lie within a given query range $[i_1..i_2] \times [j_1..j_2]$. We are particularly interested in the case where the search space is a two-dimensional grid $[1..n] \times [1..n]$, and the points in U have integer coordinates within this grid [Ove88].

Kärkkäinen [Kär99] showed the relation between range searching and compressed text searching, which is intensively used by Lempel-Ziv compressed indexes [KU96a, FM05, Nav04, RO07], among many others [HMR05, CHSV08].

We describe here a data structure by Chazelle [Cha88, Kär99], since it can be represented succinctly [MN07]. We assume first that the points represent a permutation

of $\{1, \ldots, n\}$, i.e., no two points share the same coordinate. Moreover, there is a point with first coordinate i for any $1 \leqslant i \leqslant n$, and a point with second coordinate j for any $1 \leqslant j \leqslant n$.

To construct the data structure, we first sort the set by the second coordinate j. Then, we divide the set according to the first coordinate i, to form a perfect binary tree where each node handles an interval of the first coordinate i, and thus knows only the points whose first coordinate falls in that interval. The root handles those points whose first coordinate i lies within the interval $[1..n]$, and the children of a node handling the interval $[i..i']$ are associated to $[i..\lfloor (i+i')/2 \rfloor]$ and $[\lfloor (i+i')/2 \rfloor + 1..i']$. The leaves handle intervals of the form $[i..i]$.

Every tree node v is then represented with a bit vector B_v indicating, for each point handled by v, whether the point belongs to the left or right child. In other words, $B_v[r] = 0$ iff the r-th point handled by node v (in the order given by the second coordinate j) belongs to the left child. Every level of the tree is represented as a single bit vector of n bits, using the data structure for constant-time *rank* and *select* of Lemma 2.4 (1), which are needed to support the search (as well as, given a node, finding the corresponding starting position within the level, see [MN07] for more details). Thus, we only need $O(\log n)$ pointers to represent the levels of the tree, avoiding in this way to store the pointers that represent the balanced tree.

This data structure supports counting the number of points that lie within a two-dimensional range (i.e., a rectangle) in $O(\log n)$ time. After counting the points, we can get the coordinates of the points in $O(\log n)$ time per point. Thus, the overall time to report the *occ* points inside the search range is $O((1 + occ) \log n)$ (see [MN07]).

Lemma 2.9 ([Cha88, MN07])**.** *There exists a representation for a set of n two-dimensional points requiring $n \log n + O(n \log \log n)$ bits of space, and supporting the counting of the number of points lying inside a two-dimensional search range in $O(\log n)$ time, and reporting the occ points lying inside the search range in $O((1 + occ) \log n)$ time.*

2.5.2 Succinct Representation of Trees

Given a tree with n nodes, there exist a number of succinct representations requiring $2n + o(n)$ bits of space [Jac89, MR01, BDM$^+$05, GRR06, JSS07b], which is close to the information-theoretic lower bound of $2n - \Theta(\log n)$ bits. We explain the representations that we shall need in our work.

Balanced Parentheses. The problem of representing a sequence of balanced parentheses is highly related to the succinct representation of trees [MR01]. Given a sequence *par* of $2n$ balanced parentheses, we want to support the following operations on *par*:

— $findclose(par, i)$, which given an opening parenthesis at position i, finds the position of the matching closing parenthesis;

— $findopen(par, j)$, which given a closing parenthesis at position j, finds the position of the matching opening parenthesis;

— $excess(par, i)$, which yields the difference between the number of opening and closing parentheses up to position i in the parentheses sequence; and

— $enclose(par, i)$, which given a parentheses pair whose opening parenthesis is at position i, yields the position of the opening parenthesis corresponding to the closest matching parentheses pair enclosing the one at position i.

Munro and Raman [MR01] show how to compute all these operations in constant time and requiring $2n + o(n)$ bits of space. They also show one of the main applications of maintaining a sequence of balanced parentheses: the succinct representation of general trees. Among the practical alternatives, we have the representation of Geary et al. [GRRR06], Sadakane [Sad08], and the one by Navarro [Nav08, Section 6]. The latter has shown to be very effective for representing LZ-indexes, and therefore we briefly review it in what follows.

Navarro's Practical Representation of Balanced Parentheses [Nav08]. To support the operations we could simply precompute and store all the possible answers, requiring $O(n \log n)$ bits overall. However, in many applications (e.g., the representation of trees) most matching opening and closing parentheses are close to each other. Profiting from this property, and for instance to support $findclose$, Navarro uses a brute-force approach for these parentheses, sequentially looking for the closing parenthesis within the next few, say 32, parentheses. Actually, this search is performed by using precomputed tables to avoid a bit-by-bit scan.

If the answer cannot be found in this way, Navarro searches a hash table storing the answers for parentheses that are not so close, though not so far away from each other. Say, for example, matching parentheses with a difference of up to 256 positions (parentheses). Instead of storing absolute positions, the difference between positions is stored, and thus we can use 8 bits to code these numbers, which saves space. Finally, if the answer cannot be found in the previous hash table, another table is searched for matching parentheses that are far away from each other (here full numbers are stored, but there are hopefully few entries). A similar approach is used to compute *enclose* and $findopen$ operations.

The parentheses operations are supported in $O(\log \log n)$ average time [Nav08]. However, this representation does not provide theoretical worst-case guarantees in the space requirement, since in the worst case almost every opening parenthesis has its

matching parenthesis far away, so we have to store its information in the tables. Fortunately these cases are not common in practice.

Operation $excess(i)$ can be supported through operation $rank$ over the binary sequence of parentheses, since $excess(i) \equiv rank_{(}(par, i) - rank_{)}(par, i)$. We use the representation of González et al. [GGMN05] to efficiently support $rank$ and $select$ (which will be needed later) on par, while requiring little space on top of the sequence.

BP **Representation of Trees.** The *balanced parentheses* (BP) representation of a tree defined by Munro and Raman [MR01] is built from a depth-first preorder traversal of the tree, writing an *opening* parenthesis when arriving to a node for the first time, and a *closing* parenthesis when going up (after traversing the subtree of the node). In this way, we get a sequence of balanced parentheses, where each node is represented by a pair of opening and closing parentheses. We identify a tree node x with its opening parenthesis in the representation. The subtree of x contains those nodes (parentheses) enclosed between the opening parenthesis representing x and its matching closing parenthesis. In this way, a leaf is represented by '`()`' in BP, because it encloses an empty tree.

This representation requires $2n + o(n)$ bits and supports, in constant time, the following operations:

— $parent(x)$, which gets the parent of node x;

— $subtreesize(x)$, which gets the size of the subtree of node x, including x itself;

— $depth(x)$, which gets the depth of node x in the tree;

— $firstchild(x)$, which gets the first child of node x;

— $nextsibling(x)$, which gets the next sibling of node x; and

— $ancestor(x, y)$, which tell us whether node x is an ancestor of node y

If we assume that par represents the sequence of balanced parentheses representing the tree, these operations are supported by:

$$\begin{aligned} parent(x) &\equiv enclose(par, x) \\ subtreesize(x) &\equiv (findclose(par, x) - x + 1)/2 \\ depth(x) &\equiv excess(par, x) \\ firstchild(x) &\equiv x + 1 \\ nextsibling(x) &\equiv findclose(par, x) + 1 \\ ancestor(x, y) &\equiv x \leqslant y \leqslant findclose(par, x) \end{aligned}$$

Operation $child(x, i)$ (which gets the i-th child of node x) can be computed in $O(i)$ time by repeatedly applying operation $nextsibling$. This takes, in the worst case, linear time on the maximum arity of the tree. There exist some recent representations for BP [LY08, Sad08] that support operation $child(x, i)$ in constant time. However, we disregard these in our work and use a different approach to support constant-time $child(x, i)$ operations, as we shall see later.

The *preorder position* of a node can be computed in this representation as the number of opening parentheses before the one representing the node. That is,

$$preorder(x) \equiv rank_{(}(par, x) - 1$$

Notice that in this way we assume that the preorder of the tree root is always 0. Given a preorder position p, the corresponding node is computed by

$$selectnode(p) \equiv select_{(}(par, p + 1)$$

This is a very important aspect of this (and many other) succinct representations of trees, since we can map between tree nodes and preorder positions without requiring extra space for the mapping (e.g., extra pointers in case of using a pointer-based representation).

Lemma 2.10 ([MR01])**.** *There exists a representation for a general tree of n nodes requiring $2n + o(n)$ bits of space and supporting operations parent, subtreesize, depth, $firstchild$, nextsibling, and ancestor in $O(1)$ time. Operation $child(x, i)$ is supported in $O(i)$ time.*

Example 2.14. In Fig. 2.5(a) we show the balanced parentheses representation for the *LZTrie* of Fig. 2.3(c), along with the sequence of phrase identifiers sequence (*ids*) in preorder, and the sequence of symbols labeling the edges of the trie (*letts*), also in preorder. As the identifier corresponding to the *LZTrie* root is always 0, we do not store it in *ids*. The data associated with node x is stored at position $preorder(x)$ both in *ids* and *letts* sequences. Note this information is sufficient to reconstruct *LZTrie*.

DFUDS **Representation of Trees.** To get this representation [BDM+05] we perform a preorder traversal on the tree, and for every node reached we write its degree in unary using parentheses. For example, a node of degree 3 reads '`(((`' under this representation. Notice that a leaf is represented by '`)`' in DFUDS. What we get is almost a balanced parentheses representation: we only need to add a fictitious '`(`' at the beginning of the sequence. A node of degree d is identified by the position of the first of the $(d + 1)$ parentheses representing the node.

This representation requires also $2n + o(n)$ bits, and supports operations $parent(x)$, $subtreesize(x)$, $ancestor(x, y)$, $child(x, i)$, as well as

	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	25	26	27	28	29	30	31	32	33	34	35
par:	(	(	(	)	(	(	)	)	(	)	(	(	)	(	)	(	)	)	(	(	)	)	)	(	(	(	)	)	)	(	(	(	)	)	)	)
ids:		1	17		3	15			14		4	12		10		16			6	11				2	7	9				5	8	13				
letts:		a	$		b	r			l		r	a		d		l			␣	p				l	a	b				␣	a	p				

(a) Balanced parentheses representation of *LZTrie* for the running example.

	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	25	26	27	28	29	30	31	32	33	34	35
par:	(	(	(	(	)	(	(	(	(	(	)	)	(	)	)	)	(	(	(	)	)	)	)	(	)	)	(	)	(	)	)	(	)	(	)	)
ids:				1						17	3		15	14	4				12	10	16	6		11	2		7		9	5		8		13		
letts:	a	l	␣		$	b	l	r	␣		r				a	d	l					p		a		b		a		p					

(b) DFUDS representation of *LZTrie* for the running example. The phrase identifiers are stored in preorder, and the symbols labeling the arcs of the trie are stored according to DFUDS.

Figure 2.5: Succinct representations of *LZTrie* for the running example.

— $degree(x)$, which gets the degree, i.e., the number of children, of node x; and

— $childrank(x)$, which gets the rank of node x among its siblings [JSS07b];

all in $O(1)$ time in the following way, assuming that *par* represents now the DFUDS sequence of the tree:

$$\begin{aligned}
parent(x) &\equiv select_)(par, rank_)(par, findopen(par, x-1))) + 1 \\
child(x,i) &\equiv findclose(par, select_)(par, rank_)(par, x) + 1) - i) + 1 \\
subtreesize(x) &\equiv (findclose(par, enclose(par, x)) - x)/2 + 1 \\
degree(x) &\equiv select_)(par, rank_)(par, x) + 1) - x \\
childrank(x) &\equiv select_)(par, rank_)(par, findopen(par, x-1)) + 1) \\
&\quad - findopen(par, x-1) \\
ancestor(x,y) &\equiv x \leqslant y \leqslant findclose(par, enclose(par, x))
\end{aligned}$$

Operation $depth(x)$ can be also computed in constant time on DFUDS by using the approach of Jansson et al. [JSS07b], requiring $o(n)$ extra bits. It is important to note that, unlike the BP representation, DFUDS needs operation $findopen$ on the parentheses in order to compute operation *parent* on the tree. In practice, if we build on Navarro's parenthesis data structure, this implies that DFUDS needs more space than BP since we need additional hash tables to support $findopen$.

Given a node in this representation, say at position i, its preorder position can be computed by counting the number of closing parentheses before position i; in other words,

$$preorder(x) \equiv rank_)(par, x-1)$$

Given a preorder position p, the corresponding node is computed by

$$selectnode(p) \equiv select_)(par, p) + 1$$

Representing σ-ary trees with DFUDS. For cardinal trees, σ-ary trees, or simply tries (i.e., trees where each node has at most σ children, each child labeled by a symbol in the set $\{1, \ldots, \sigma\}$) we use the DFUDS sequence par plus an array $letts[1..n]$ storing the edge labels according to a DFUDS traversal of the tree: we traverse the tree in depth-first preorder, and every time we reach a node x we write the symbols labeling the children of x. In this way, the labels of the children of a given node are all stored contiguously in $letts$, which will allow us to efficiently compute operation

— $child(x, \alpha)$, which gets the child of node x with label $\alpha \in \{1, \ldots, \sigma\}$.

Example 2.15. In Fig. 2.5(b) we show the DFUDS representation of *LZTrie* for our running example.

Notice the inverse relation between the d opening parentheses defining x and the symbols of the children of x. The label of the i-th child is at position i within the symbols of the children of x, while the corresponding opening parenthesis is at position $(d - i + 1)$ within the definition of x. This shall mean extra work when retrieving the symbol by which a given node descends from its parent.

We support operation $child(x, \alpha)$ as follows. Suppose that node x has position p within the DFUDS sequence par, and let $p' = rank_(par, p) - 1$ be the position in $letts$ for the symbol of the first child of x. Let $n_\alpha = rank_\alpha(letts, p' - 1)$ be the number of αs up to position $p' - 1$ in $letts$, and let $i = select_\alpha(letts, n_\alpha + 1)$ be the position of the $(n_\alpha + 1)$-th α in $letts$. If i lies within positions p' and $p' + degree(x) - 1$, the child we are looking for is $child(x, i - p' + 1)$, which, as we said before, is computed in constant time over par; otherwise x has not a child labeled α. We can also retrieve the symbol by which x descends from its parent with $letts[rank_(par, parent(x)) - 1 + childrank(x) - 1]$, where the first term stands for the position in $letts$ corresponding to the first symbol of the parent of node x. The second term, $childrank(x)$, comes from the inverse relation between symbols and opening parentheses representing a node.

Thus, the time for operation $child(x, \alpha)$ depends on the representation we use for $rank_\alpha$ and $select_\alpha$ queries (see Lemma 2.5). Notice that $child(x, \alpha)$ could be solved in a straightforward way by binary searching the labels of the children of x, in $O(\log \sigma)$ worst-case time and not needing any extra space on top of array $letts$. The access to $letts[\cdot]$ takes constant time.

We can represent $letts$ with the data structure of Ferragina et al. (see Lemma 2.5 (1)) which requires $n \log \sigma + o(n \log \sigma)$ bits of space, and allows us to compute $child(x, \alpha)$ in

$O(\frac{\log \sigma}{\log \log u})$ time. The access to $letts[\cdot]$ also takes $O(\frac{\log \sigma}{\log \log u})$ time. These times are $O(1)$ whenever $\sigma = O(\text{polylog}(u))$ holds. On the other hand, we can use the data structure of Golynski et al. (see Lemma 2.5 (2)), requiring $O(n \log \sigma)$ bits of space, yet allowing us to compute $child(x, \alpha)$ in $O(\log \log \sigma)$ time, and access to $letts[\cdot]$ also in $O(\log \log \sigma)$ time. In most of this thesis we will use the data structure of Lemma 2.5 (1) to represent the symbols, since it is able to improve its time complexity to $O(1)$ for moderate alphabets.

The scheme we have defined to represent *letts* is slightly different than the original one [BDM$^+$05], which achieves $O(1)$ time for $child(x, \alpha)$ for any σ. However, ours is simpler and allows us to efficiently access $letts[\cdot]$, which will be very important in our indexes to extract text substrings. Also, our method is simpler to build, since the original one is based on perfect hashing, which is expensive to construct. We need to store the array of symbols explicitly if we use the original approach and we need to access them (fortunately, this will not asymptotically affect the space requirement of our results).

Lemma 2.11 ([BDM$^+$05, JSS07b])**.** *There exists a representation for a σ-ary tree of n nodes requiring $2n + n \log \sigma + o(n)$ bits of space and supporting operations parent, $child(x, i)$, $child(x, \alpha)$, subtreesize, depth, degree, childrank, and ancestor in $O(1)$ time.*

***xbw* Representation.** The *xbw transform* of Ferragina et al. [FLMM05] is a succinct representation for *labeled trees*: given a labeled tree $\mathcal{T}$, with n nodes and labels taken from an alphabet Σ of size σ, the *xbw* transform of $\mathcal{T}$ is computed by first traversing the tree in preorder, and for each node writing a triplet in a table $S_{\mathcal{T}}$. The first component of each triplet indicates whether the node is the last child of its parent in the tree, the second component is the symbol labeling the edge by which we reach the node, and the third component is the string labeling the path from the parent of the node to the root of $\mathcal{T}$. In this way each node is represented by a row in $S_{\mathcal{T}}$. As in the original work [FLMM05], we call S_{last}, S_α, and S_π the columns of table $S_{\mathcal{T}}$, storing respectively the first, second and third components of each triplet. As a last step we perform an upward-path-sorting of the table by stably sorting the rows of $S_{\mathcal{T}}$ lexicographically according to the strings in S_π.

Example 2.16. In Table 2.1 we show the *xbw* transform for the *LZTrie* of Fig. 3.1(a).

We have to add a dummy child to each leaf, labeling the dummy edge with a special symbol Δ not in Σ, so that the paths leading to the leaves appear in column S_π, and hence later we will be able of searching for them (notice that, in the worst case, this duplicates the number of nodes in the tree). As we said before, each node in the *LZTrie* is represented by a row in the table, being the row number what we call the *xbw position* of the node.

The *xbw* representation supports operations $parent(x)$, $child(x, i)$, and $child(x, \alpha)$, all of them in $O(1)$ time if $\sigma = O(\text{polylog}(u))$, and using $2n \log \sigma + O(n)$ bits of space, because

the column S_π of the table is not stored. The representation also allows *subpath queries*, a very powerful operation which, given a string s, returns all the nodes x such that s is a prefix of the string labeling the path from the parent of x to the root. If $\sigma = O(\text{polylog}(n))$, subpath queries can be computed in $O(|s|)$ time [FLMM05]. For general σ, the time for all these operations depends on the representation used for S_α (since we need to support *rank* and *select* operations on it), which is $O(1 + \frac{\log \sigma}{\log \log u})$ time if we use the representation of Lemma 2.5 (1), and $O(\log \log \sigma)$ time if we use the data structure of Lemma 2.5 (2), in which case the space requirement is $O(n \log \sigma)$. Also [FLMM05] show how to achieve constant time for the operations (and $O(|s|)$ time for subpath queries), for any alphabet, though duplicating the space, since the representation given for *rank* and *select* does not provide operation *access*.

Because of the upward-path sorting in table $S_\mathcal{T}$, the result of a subpath query is a contiguous interval in such table, containing the answers to the query. For example, a subpath query for string '`r`' yields the interval [21..24] in Table 2.1, corresponding respectively to the nodes with preorders 7, 8, and 9 in Fig. 3.1(a), plus a fictitious leaf which is a child of node with preorder 4. As another example, a subpath query for string '`ba`' yields the *xbw* interval [13..14], for node with preorder 4 plus a fictitious leaf which is a child of node with preorder 14. In all cases, note that the string s we are looking for is a prefix of the corresponding string in S_π.

Lemma 2.12. *[FLMM05] There exists a representation for a labeled tree of n nodes that requires $2n \log \sigma + O(n) + o(n \log \sigma)$ bits of space, where σ is the size of the alphabet of the symbols labeling the tree, and supports operations parent and child(x, i) in $O(1)$ time, operation child(x, α) in $O(1 + \frac{\log \sigma}{\log \log n})$ time, and subpath queries for a string s in $O(|s|(1 + \frac{\log \sigma}{\log \log n}))$ time.*

Notice that the *xbw* representation does not support operations *preorder*, *selectnode*, *subtreesize*, etc., many of which are useful for our purposes. Thus, in such cases we shall enrich the *xbw* representation to support these operations, basically by succinctly combining this representation with others.

Table 2.1: *xbw* representation for the *LZTrie* of Fig. 3.1(a).

i	S_{last}	S_α	S_π
1	0	a	empty string
2	0	l	empty string
3	1	_	empty string
4	1	Δ	$a
5	0	$	a
6	0	b	a
7	0	l	a
8	0	r	a
9	1	_	a
10	1	b	al
11	1	Δ	ara
12	1	p	a_
13	1	r	ba
14	1	Δ	bal
15	1	Δ	dra
16	1	a	l
17	1	Δ	la
18	1	Δ	lra
19	1	Δ	pa_
20	1	Δ	p_a
21	0	a	ra
22	0	d	ra
23	1	l	ra
24	1	Δ	rba
25	1	a	_
26	1	p	_a

Chapter 3

A Survey of Lempel-Ziv Indexes

The general idea of Lempel-Ziv (LZ) compression algorithms [LZ76, ZL77, ZL78] is to parse the text $T[1..u]$ into a set of n phrases $T = B_1 \dots B_n$. Every phrase B_i is somehow formed by a substring of T that already appeared in a linear scan of the text. Different variants of LZ compression differ in the way they define the phrases. See Section 2.4.1 for a review of the LZ78 compression algorithm [ZL78].

The search of a pattern $P[1..m] = p_1 \dots p_m$ in an LZ-compressed text has the additional problem that, as the text has been parsed into phrases, a pattern occurrence could span several (two or more) consecutive phrases. We call *occurrences of type 1* those occurrences contained in a single phrase (say there are occ_1 occurrences of type 1); *occurrences of type 2* are those occurrences spanning two consecutive phrases (there are occ_2 occurrences of type 2); and *occurrences of type 3* are those spanning more than two consecutive phrases (there are occ_3 occurrences of type 3).

Example 3.1. As an example of occurrences of type 1 on our running example text of Fig. 2.3(a), let us consider the pattern "`ab`", occurring in phrases 3, 15, and 9. As an example of occurrences of type 2, consider the pattern "`ala`", spanning phrases 8 and 9 (partitioned as `a` · `la`) and spanning phrases 14 and 15 (partitioned as `al` · `a`). As an example of occurrences of type 3, consider the pattern "alabar", spanning phrases from 1 to 4 (partitioned as `a` · `l` · `ab` · `ar`) and spanning phrases from 8 to 10 (partitioned as `a` · `lab` · `ar`).

In this chapter we review the existing Lempel-Ziv compressed indexes (LZ-indexes for short), showing the results and main concepts introduced by these works. We show the state of the art previous to our thesis, and also show some works that appeared in parallel to our research.

3.1 Kärkkäinen and Ukkonen's LZ-index (KU-LZI)

Historically, the LZ-index of Kärkkäinen and Ukkonen [KU96a, Kär99] (the KU-LZI, for short) was the first LZ-index. Indeed, up to the best of our knowledge, this was the first compressed index. The KU-LZI is based on a variant of the original Lempel-Ziv parsing of 1976 (LZ76 for short) [LZ76].

In the LZ76 parsing, the text $T[1..u]$ is represented as a sequence of phrases $T = B_1, \ldots, B_n$, which are built as follows. Assume that a prefix $T[1..j]$ of T has been compressed into a sequence of phrases $B_1, \ldots, B_r$. Then, we look for the longest prefix $T[j+1..j']$ of the rest of the text $T[j+1..u]$ which equals a substring $T[s..s+j'-j+1]$ of $T[1..j]$ [1].

If $j' > j$, then $B_{r+1} = T[j+1..j']$, and we go on to process $T[j'+1..u]$. We call the substring $T[s..s+j'-j+1]$ the *source* for phrase B_{r+1}. Otherwise, the symbol $T[j+1]$ has not appeared before and then $B_{r+1} = T[j+1]$, and we go on to process $T[j+2..u]$. The process finishes once we obtain $B_n =$"\$".

The output is essentially the starting position of the source of every phrase and its length (new symbols in Σ that appear are exceptions in this encoding). It is worth to note that Property 2.7 (on page 28) and Lemma 2.1 (on page 29) are also valid if n is the number of phrases generated by the LZ76 parsing of the text [Kär99, NM07]. An important property of this parsing is that every phrase has appeared before, unless it is a new symbol of Σ. Because of this property, the first occurrence of a pattern $P[1..m]$ in T (i.e., the leftmost occurrence of P in T) cannot be completely contained inside a phrase, otherwise it would have appeared before in T. Thus, in this parsing occurrences of type 1 are repetitions of other occurrences either of type 2 or type 1 (henceforth, Kärkkäinen and Ukkonen call *primary occurrences* those of type 2, and *secondary occurrences* those of type 1). Occurrences of type 3 are treated as a particular case of occurrences of type 2 in this index.

Occurrences of type 2 are found as follows. Assume that for some $1 \leqslant i < m$, $P[1..i]$ is the suffix of a phrase B_t and that $P[i+1..m]$ is aligned with the next phrase B_{t+1}. Thus, occurrences of $P[i+1..m]$ are found using a *sparse suffix tree* [KU96b] that only indexes the text suffixes which are aligned with the phrase beginnings. Notice that we are interested only in the phrase-aligned occurrences of $P[i+1..m]$, so we only must descend in the suffix tree by using the symbols of $P[i+1..m]$, reaching node v_2 corresponding to the occurrences. Thus, we avoid the more costly process of looking for the non-aligned occurrences [KU96b]. Let $[l_2..r_2]$ be the preorder interval corresponding to node v_2.

[1] Notice the difference with the LZ78 parsing, where $T[j+1..j']$ must exists as a whole phrase in $T[1..j]$, see Section 2.4.1

To find the phrases ending with $P[1..i]$, the text prefixes ending where the LZ phrases end are reversed and stored in a dual sparse suffix tree. Thus, by searching for $P^r[1..i]$ in the reverse suffix tree (reaching node v_1) we find the set of phrases ending with $P[1..i]$. Let $[l_1..r_1]$ be the preorder interval corresponding to node v_1 in the reverse suffix tree.

Thus, for every partition $P[1..i]$ and $P[i+1..m]$ of P we have a subtree containing the phrases ending with $P[1..i]$, and a subtree containing the phrases aligned with the occurrences of $P[i+1..m]$. The search for these subtrees (or preorder intervals) takes overall $O(m^2)$ time. We want to find those phrases B_t in the former subtree such that phrase B_{t+1} is in the latter subtree.

Kärkkäinen and Ukkonen relate this problem to a two-dimensional range search problem: for every phrase B_t, for $1 \leqslant t < n$, let i' be the preorder of the node representing B_t^r in the reverse sparse suffix tree, and let j' be the preorder of the node for the next phrase B_{t+1} in the sparse suffix tree. Then, we store the point (i', j') in a two-dimensional range search data structure. Thus, the points corresponding to phrases ending with $P[1..i]$ can be found if we search for the range $[l_1..r_1] \times [-\infty, +\infty]$ in the range search data structure. The phrases that are aligned with occurrences of $P[i+1..m]$ can be found with the range $[-\infty, +\infty] \times [l_2..r_2]$. Finally, occurrences of type 2 correspond to those points lying in the range $[l_1..r_1] \times [l_2..r_2]$. A more detailed example of using a range search to find occurrences of type 2 is given in Section 3.3. We can use the data structure of Lemma 2.9 to report all these points in $O((m + occ_2) \log u)$ time, and requiring $n \log n + o(n \log n)$ bits of space.

For occurrences of type 1, given an occurrence of type 2 $T[j..j+m-1]$, we wish to find all phrases B_t whose source contains $[j..j+m-1]$. Those phrases contain occurrences $T[j'..j'+m-1]$ of type 1, which are again tracked for new occurrences. With some slight changes to the original LZ76 parsing [KU96a, Kär99], they ensure that no source contains another source, and thus source intervals can be linearly ordered.

Let S be an array of phrase numbers sorted by their source interval position in the text, plus a bit vector $B[1..u]$ indicating the starting text positions for the sources. Array B is represented with the data structure for constant-time *rank* and *select* of Lemma 2.4 (2), so this requires $uH_0(B) + o(u) \leqslant n \log \frac{u}{n} + o(u) \leqslant u \frac{\log \log u}{\log_\sigma u} + o(u) = o(u \log \sigma)$ bits of space (recall Property 2.7 and Lemma 2.2).

Therefore, all phrases whose source contains $[j..j+m-1]$ are found in constant time by using these data structures: $S[rank_1(B, j)]$ is the last phrase in S whose source starts in $T[1..j]$. We traverse S backwards from that position until the source intervals finish before $T[j+m-1]$. Proceed recursively on the phrases found. We report a new occurrence per unit of work. Thus, occurrences of type 1 are reported in $O(occ_1)$ overall time.

Lemma 3.1 ([KU96a, Kär99])**.** *Given a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, the LZ-index of Kärkkäinen and Ukkonen (KU-LZI)*

requires $O(uH_k(T)) + u\log\sigma + o(u\log\sigma)$ *bits of space, for any* $k = o(\log_\sigma u)$. *Given a search pattern* $P[1..m]$, *this index is able to:*

(*1*) *locate the occ occurrences of pattern* P *in text* T *in* $O(m^2 + (m + occ)\log u)$ *time;*

(*2*) *count the number of pattern occurrences in* $O(m^2 + m\log u + occ)$ *time;*

(*3*) *determine whether pattern* P *exists in* T *in* $O(m^2 + m\log u)$ *time; and*

(*4*) *extract any text substring of length* ℓ *in optimal* $O(\ell/\log_\sigma u)$ *time (the text is available).*

Many other trade-offs for the KU-LZI are presented in [Kär99], yet we disregard them here. It is important to note that the extracting time is reported as $O(\ell)$ in the survey [NM07], which is a mistake since the text is available and therefore $O(\log_\sigma u)$ text symbols can be accessed per memory access in a RAM.

3.2 Ferragina and Manzini's LZ-index (FM-LZI)

The LZ-index of Ferragina and Manzini [FM05] (the FM-LZI for short) is a compressed full-text self-index based on the combination of the LZ78 parsing of text T [ZL78] and the Burrows-Wheeler transform of T [BW94]. The result is a very efficient full-text self-index, as we shall see below

The following data structures compose the FM-LZI:

(1) *LZTrie*: the trie formed by the LZ78 phrases $B_0, \ldots, B_n$ of text T. Every trie node stores the starting text position of the corresponding LZ78 phrase. Since the number of *LZTrie* nodes is n, and since a pointer representation is originally used, the space requirement is $O(n\log n) = O(uH_k(T)) + o(u\log\sigma)$ bits of space.

(2) $FMI(T)$: the FM-index [FM00, FM05] of text T, which is based on the Burrows-Wheeler transform of T and hence requires $O(uH_k(T)) + o(u\log\sigma)$ bits of space. By itself, this index is able to search for the pattern occurrences. However, it is slower than the LZ-indexes to locate the pattern occurrences.

(3) $FMI(T^r_\#)$: the FM-index of text $T^r_\#$, where $T_\# = B_1\# \ldots \#B_n\#$ is the string formed by adding an special symbol '#' $\notin \Sigma$ after every LZ78 phrase of T. Thus, the length of $T_\#$ is $n + u$ symbols. By using this data structure we shall be able to find all phrases ending with a given substring, as we will see later. Ferragina and Manzini [FM05] proved that $(u+n)H_k(T^r_\#) \leqslant uH_k(T) + o(u\log\sigma)$ always holds, for any fixed $k \geqslant 0$. Thus, $FMI(T^r_\#)$ requires $O(uH_k(T)) + o(u\log\sigma)$ bits of space.

(4) $N[1..n]$: an array that, for each position of $FMI(T^r_\#)$ indexing a suffix of $T^r_\#$ starting with $\#B^r_t$, stores the corresponding $LZTrie$ node for phrase B_t. This array requires $n \log n = uH_k(T) + o(u \log \sigma)$ bits of space.

(5) *Range*: a data structure for two-dimensional range searching in the grid $[1..u] \times [1..u]$. If the position of $FMI(T^r_\#)$ corresponding to phrase B_t (i.e., the suffix array position indexing the suffix of $T^r_\#$ starting with $\#B^r_t$) is i and the position for B_{t+1} in $FMI(T)$ (i.e., the suffix array position indexing the suffix of T starting with B_{t+1}) is j, then we store the point (i, j) corresponding to phrase t. Hence, this data structure stores n points, so it can be stored as an $[1..n] \times [1..n]$ grid by adding two bit vectors of u bits each, indicating in each case which positions of the corresponding *FMI* define the coordinates of the points in *Range*. These bit vectors can be represented with the data structure of Lemma 2.4 (2), supporting *rank* and *select* queries and requiring $o(u \log \sigma)$ bits of space. Ferragina and Manzini use the data structure of Alstrup et al. [ABR00] to support range queries, which is modified to support finding the *occ* occurrences inside a query range in $O(m + occ)$ time, while requiring $O(n \log^{1+\gamma} n)$ bits of space. In our case this is $O(uH_k(T) \log^\gamma u) + o(u \log \sigma \log^\gamma u)$ bits, for any $\gamma > 0$.

To search for occurrences of type 1 using these data structures, assume that phrase B_t contains P. If B_t does not end with P and if $B_t = B_\ell \cdot c$, for $\ell < t$ and $c \in \Sigma$, then B_ℓ contains P as well. Therefore, we must search for the minimal phrases containing P. By properties of LZ78 compression, all those phrases end with P. The rest of the phrases containing P are formed from those phrases, and can be found using the $LZTrie$.

To find the phrases ending with P, we search for $\#P^r = \#p_m \dots p_1$ in $FMI(T^r_\#)$, which give us the corresponding suffix array interval $[first, last]$ in $FMI(T^r_\#)$. Then, for each $i \in [first, last]$ we use $N[i]$ to get the $LZTrie$ node corresponding to a phrase ending with P. We then traverse the subtree of node $N[i]$ to report occurrences of type 1. As the search in $FMI(T^r_\#)$ takes $O(m)$ time and the work in $LZTrie$ is proportional to occ_1, occurrences of type 1 are found in $O(m + occ_1)$ time.

For occurrences of type 2 and type 3, we proceed in a similar way as for the KU-LZI: for every possible partition $P[1..i]$ and $P[i+1..m]$ of P, we search for phrases ending with $P[1..i]$ and for phrases aligned with an occurrence of $P[i+1..m]$. However, now we have different data structures, which help us to solve this problem very efficiently.

We first search for P in $FMI(T)$, using the backward-search algorithm (recall Section 2.4.2). Thus, in $O(m)$ steps we find the suffix-array intervals $[l_{i+1}..r_{i+1}]$ in $FMI(T)$, corresponding to the occurrences of $P[i+1..m]$, for $i = m-1, \dots, 1$.

We then search for $\#P^r$ in $FMI(T^r_\#)$. At every backward-search step we find the suffix array interval $[l_i..r_i]$ containing the occurrences of $P^r[1..i]$ in $FMI(T^r_\#)$, for $i = m-1, \dots, 1$.

After finding every such interval in $FMI(T^r_{\#})$, we add the symbol '#' to obtain the interval $[l'_i..r'_i]$ in $FMI(T^r_{\#})$ containing the occurrences of $\#P^r[1..i]$.

Note that the search in $FMI(T^r_{\#})$ ensures that $P[1..i]$ occurs at the end of a phrase, and the search in $FMI(T)$ permits $P[i+1..m]$ to span as many phrases as necessary (in other words, there is no difference between occurrences of type 2 and those of type 3). All this process takes $O(m)$ time, because of the backward-search process (compare it against the $O(m^2)$ time needed to find the preorder intervals in the KU-LZI).

Then, we search for the two-dimensional range $[l'_i..r'_i] \times [l_{i+1}..r_{i+1}]$ in the *Range* data structure, in order to find the occurrences of type 2 for the partition $P[1..i]$ and $P[i+1..m]$ of P, for $1 \leqslant i < m$ (before performing the query, both intervals must be mapped to the $[1..n] \times [1..n]$ grid by using $rank$ over the two bit vectors that we mentioned previously).

Lemma 3.2 ([FM05])**.** *Given a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, the LZ-index of Ferragina and Manzini (FM-LZI) requires $O(uH_k(T)\log^\gamma u) + o(u \log\sigma \log^\gamma u)$ bits of space, for any $\gamma > 0$ and any $k = o(\log_\sigma u)$. Given a search pattern $P[1..m]$, this index is able to:*

(*1*) *locate the occ occurrences of pattern P in text T in $O(m + occ)$ time;*

(*2*) *count the number of pattern occurrences in $O(m)$ time;*

(*3*) *determine whether pattern P exists in T in $O(m)$ time; and*

(*4*) *extract any text substring of length ℓ in $O(\ell)$ time.*

Later in this chapter we shall explain the procedure used to extract text substrings.

3.3 Navarro's LZ-index (NAV-LZI)

Navarro [Nav04, Nav08] uses a somehow different approach to define an LZ-index (the NAV-LZI, or simply the LZ-index throughout this thesis): Navarro bases the search process mainly on the *LZTrie*, and exploits its relation with the trie of reversed LZ78 phrases, the so-called *RevTrie*.

As a result, this is the only existing LZ-index not using the concept of suffix tree or array at all. This introduces the extra problem that, as we do not index whole text suffixes but just the phrases, using an implicit representation of the LZ78 phrases (the *LZTrie*), we cannot use range search to find occurrences of type 3 as done in the KU-LZI and the FM-LZI indexes. Thus, in this index range searching only works for occurrences of type 2, while occurrences of type 3 must be searched by hand, checking each of the $O(m^2)$ possible candidates, and hence introducing an extra quadratic term to the time complexity of the

index. Despite of this, the NAV-LZI has shown to be effective in practice [Nav08], being the first LZ-index to be implemented.

At the time of its introduction, this LZ-index was smaller than all the others in exchange for weaker search structures. We will show in this thesis that the original search performance of the NAV-LZI can be improved to achieve that of competing schemes, yet with a smaller index.

As we base our research on the NAV-LZI (yet many of our results can be extended to other LZ-indexes), we review it in more detail.

3.3.1 Original NAV-LZI Components

The following data structures compose the original LZ-index [Nav04, Nav08]:

(1) *LZTrie*: is the trie formed by all the phrases $B_0 \dots B_n$. Given the properties of LZ78 compression (Property 2.5), this trie has exactly $n+1$ nodes, each one corresponding to a string.

(2) *RevTrie*: is the trie formed by all the reverse strings $B_0^r \dots B_n^r$. In this trie there could be internal nodes not representing any phrase. We call these nodes "*empty*".

(3) *Node*: is a mapping from phrase identifiers to their node in *LZTrie*.

(4) *Range*: is a data structure for two-dimensional range searching in the space $[0..n] \times [0..n]$. We store the points

$$\{(preorder_r(t), preorder_{lz}(t+1)), \quad t \in 0 \dots n-1\}$$

in this structure, where $preorder_r(t)$ is the *RevTrie* preorder of the node for phrase t (considering only the non-empty nodes in the preorder enumeration), and $preorder_{lz}(t+1)$ is the *LZTrie* preorder of node for phrase $t+1$. For each such point, the corresponding t value is stored.

Example 3.2. Fig. 3.1 shows the *LZTrie* and *RevTrie* data structures, and Fig. 3.2 shows the *Range* and *Node* data structures, all of them corresponding to our running example. We show preorder numbers, both in *LZTrie* and *RevTrie* (in the latter case only counting non-empty nodes), outside each trie node. In the case of *RevTrie*, empty nodes are shown in black.

The next example gives a hint on the usage of those structures for searching, which will be detailed in Section 3.3.5.

Example 3.3. To find all phrases ending with substring '`ab`' in the running example, we search for the reversed string '`ba`' in *RevTrie*, reaching the node with preorder 6. The subtree of this *RevTrie* node contains the phrases we are looking for: phrases 3 and 9 (see Fig. 2.3(a)). As the preorder interval in *RevTrie* defined by this subtree is [6..7], this means that the horizontal semi-infinite range $[-\infty..\infty] \times [6..7]$ in *Range* also contains those phrases. To find all phrases starting with '`ar`', note that the *LZTrie* subtree for node with preorder (incidentally also) 6 (which corresponds to string '`ar`') contains the phrases starting with '`ar`': phrases 4, 12, 10, and 16. The *LZTrie* preorder interval for this subtree is [6..9]. This means that the vertical semi-infinite range $[6..9] \times [-\infty..\infty]$ contains phrases i such that phrase $i+1$ starts with '`ar`': phrases 3, 11, 9, and 15. Finally, the range $[6..9] \times [6..7]$ contains the phrase numbers i such that phrase i ends with '`ab`' followed by phrase $i+1$ starting with '`ar`': phrases 3 and 9, see Fig. 3.2(a).

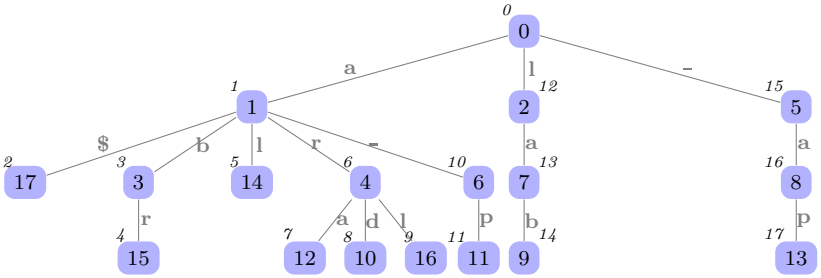
3.3.2 Succinct Representation of the NAV-LZI Components

In the original work [Nav04], each of the four structures described requires $n \log n + o(u \log \sigma)$ bits of space if they are represented succinctly.

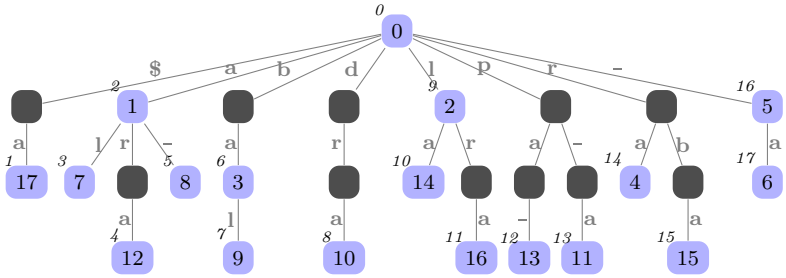
— *LZTrie* is represented using the balanced parentheses representation of Lemma 2.10, requiring $2n + o(n)$ bits; plus the sequence *letts* of symbols labeling each trie edge, requiring $n \log \sigma$ bits; and the sequence *ids* of $n \log n$ bits storing the LZ78 phrase identifiers. Both *letts* and *ids* are stored in preorder, so we use $preorder(x)$ to index them. See Fig. 2.5(a) for an illustration.

— For *RevTrie*, balanced parentheses are also used to represent the *Patricia* tree [Mor68] structure of the trie, compressing empty unary nodes and so ensuring $n' \leqslant 2n$ nodes. This requires at most $4n + o(n)$ bits. The *RevTrie*-preorder sequence of identifiers (*rids*) is stored in $n \log n$ bits (i.e., we only store the identifiers for non-empty nodes). Non-empty nodes are marked with bit vector $B[1..n']$, such that $B[j] = 0$ iff node x with preorder j is empty. Thus, the phrase identifier for node x is $rids[rank_1(B, j)]$. The symbols labeling the arcs of the trie and the Patricia-tree skips are not stored in this representation, since they can be retrieved by using the connection with *LZTrie*. Therefore, the navigation on *RevTrie* is more expensive than that on *LZTrie*. See Fig. 3.1(c) for an illustration.

— For *Range*, the data structure of Lemma 2.9 permits two-dimensional range searching in a grid of n pairs of integers in the range $[0..n] \times [0..n]$, answering queries in $O((occ+1) \log n)$ time, where *occ* is the number of occurrences reported, and requiring $n \log n + O(n \log \log n)$ bits of space [MN07]; note that since n is the number of LZ78 phrases of T, the latter term $O(n \log \log n)$ is just $o(u \log \sigma)$ bits. This data structure



(a) Lempel-Ziv Trie (*LZTrie*) for the running-example text. Phrase identifiers are shown inside each node. Preorder numbers are shown outside each node.

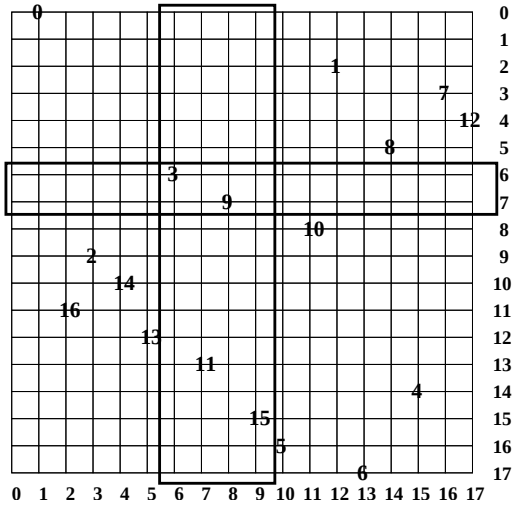


(b) *RevTrie* data structure. Empty nodes are shown in light gray.

	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	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
par:	(	(	)	(	(	)	(	)	(	)	)	(	(	)	)	(	)	(	(	)	(	)	)	(	(	)	(	)	)	(	(	)	(	)	)	(	(	)	)	)
B:	1	1		1	1		1		1			1	1			1		1	1		1			0	1		1			0	1		1			1	1			
rletts:		$		a	l		r		_			b	l			d		l	a		r			p	a		_			r	a		b			_	a			
rids:	0	17		1	7		12		8			3	9			10		2	14		16				13		11				4		15			5	6			

(c) Balanced parentheses representation of *RevTrie*, compressing empty unary paths. The bitmap B marks with a 0 the empty non-unary nodes. Array *rletts* stores only the first symbol of every edge label, after compressing empty-unary paths. Array *rids* is indexed by means of *preorder* and *rank* over B.

Figure 3.1: Tries composing the LZ-index for the running example.



(a) *Range* data structure. Horizontal coordinates are for *LZTrie* preorders, and vertical coordinates are for *RevTrie* preorders.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$Node[i]$	0	1	23	4	10	29	18	24	30	25	13	19	11	31	8	5	15	2

(b) *Node* data structure, assuming that the parentheses sequence starts from position zero, cf. Fig. 2.5(a).

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$RNode[i]$	0	3	17	11	30	35	36	4	8	12	15	26	6	24	18	32	20	1

(c) *RNode* data structure, assuming that the parentheses sequence for *RevTrie* starts from position zero.

Figure 3.2: Remaining LZ-index components, $Range$ and $Node$ for the running example. The $RNode$ data structure is used instead of $Range$ in the practical implementation of the LZ-index.

supports counting the number of points in a given range in $O(\log n)$ time, while requiring $uH_k(T) + o(u \log \sigma)$ bits of space.

— Finally, *Node* is just a sequence of n pointers to *LZTrie* nodes. As *LZTrie* is implemented using balanced parentheses, $Node[i]$ stores the position within the sequence for the opening parenthesis representing the node corresponding to phrase i. As there are $2n$ such positions, we need $n \log 2n = n \log n + n$ bits of storage. See Fig. 3.2(b) for an illustration.

According to Lemma 2.1, the final size of the LZ-index is $4uH_k(T) + o(u \log \sigma)$ bits for $k = o(\log_\sigma u)$ (and hence $\log \sigma = o(\log u)$ if we want to allow for $k > 0$).

In theory, the succinct trie representations used [Nav04] implement (among others) operations $parent(x)$ and $child(x, \alpha)$, both in $O(\log \sigma)$ time for *LZTrie* (the $O(\log \sigma)$ factor comes from the fact that the tries are represented using a binary encoding, which is defined in [Nav04]), and $O(\log \sigma)$ and $O(h \log \sigma)$ time respectively for *RevTrie*, where h is the depth of node x in *RevTrie* (the h in the cost comes from the fact that we must access *LZTrie* to get the label of a *RevTrie* edge). The operation $ancestor(x, y)$ is implemented in $O(1)$ time in both *LZTrie* and *RevTrie*.

In practice, however, Navarro's practical implementation is used (recall Section 2.5.2). Despite that under this representation operation $child(x, \alpha)$ is implemented by using operation $child(x, i)$ in $O(\sigma \log \log n)$ worst-case time, this has shown to be very effective in practice [Nav08]. Operation *parent* is supported in $O(\log \log n)$ time under this representation.

3.3.3 Constructing the NAV-LZI

The data structures that compose the LZ-index are built and represented as follows.

LZTrie. For the construction of *LZTrie* we traverse the text and at the same time build a trie representing the Lempel-Ziv phrases, spending (as usual) one pointer per parent-child relation. At step t (assume $B_t = B_\ell \cdot c$), we read the text that follows and step down the trie until we cannot continue. At this point we create a new trie leaf (child of the trie node of phrase ℓ, by symbol c, and assigning the leaf phrase number t), go to the root again, and go on with step $t+1$ to read the rest of the text. The process completes when the last phrase finishes with the text terminator "$". After we build the trie, we free the text as it is not anymore necessary, since we have now enough information to build the remaining index components.

Then we build the final succinct representation of *LZTrie*, essentially using the parentheses representation of Munro and Raman [MR01]. Arrays *ids* and *letts* are also created at this stage.

Node. Once the *LZTrie* is built, we free the space of the pointer-based trie and build *Node*. This is just an array with the n nodes of *LZTrie*. If the i-th position of the *ids* array corresponds to the j-th phrase identifier (i.e., $ids[i] = j$), then the j-th position of *Node* stores the position of the i-th node within the balanced parentheses. As there are $2n$ parentheses, *Node* requires $n \log 2n$ bits.

RevTrie. To construct *RevTrie* we traverse *LZTrie* in preorder, generating each LZ78 phrase B_i stored in *LZTrie* in constant time, and then inserting it into a *trie of reversed strings* (represented with pointers). For simplicity, empty unary paths are not compressed in the pointer-based trie. When we finish, we traverse the trie and represent the trie topology of *RevTrie* and the phrase identifiers in array *rids*. Empty unary nodes are removed only at this step, and so the final number of nodes in *RevTrie* is $n \leqslant n' \leqslant 2n$.

Range. The *Range* data structure is built just as explained in Section 2.5.1.

3.3.4 Experimental Indexing Space

Although the LZ-index is a compressed full-text self-index, and its final representation requires little space, a large amount of storage is needed to construct the index, mainly because of the pointer representation of the tries used originally at construction time. In the experimental results obtained with the original LZ-index [Nav08], over an English text and DNA sequences, the largest extra space needed to build *LZTrie* is that of the pointer-based trie, which is 1.7–2.0 times the text size [Nav08].

On the other hand, the indexing space for the pointer-based *reverse* trie is, in some cases, 4 times the text size. This is, mainly, because of the empty unary nodes. This space dictates the maximum indexing space of the algorithm. The overall indexing space was 4.8–5.8 times the text size for the case of English text, and 3.4–3.7 times the text size for the DNA sequences. As a comparison, the construction of a plain suffix array without any extra data structure requires 5 times the text size [MF04].

3.3.5 The Nav-LZI Search Algorithm

Let us consider now the search algorithm for a pattern $P[1..m]$ [Nav04]. For `locate` queries, pattern occurrences are reported in the format $[\![t, o]\!]$, where t is the phrase where the occurrence starts, and o is the distance between the beginning of the occurrence and the end of the phrase. Later, in Section 5.4, we will show how to map these two values into a single *text position*. As we deal with an implicit representation of the text (the *LZTrie*), and not the text itself, we distinguish three types of occurrences of P in T, depending on the phrase layout.

Occurrences of Type 1. These are found by using the same properties as in the FM-LZI, yet in a different way (because the data structures are different). Given the properties of LZ78 compression, every phrase B_t containing P is formed by a shorter phrase B_ℓ concatenated to a symbol c (Property 2.5). If P does not occur at the end of B_t, then B_ℓ contains P as well. We want to find the shortest possible phrase B_i in the LZ78 referencing chain for B_t that contains the occurrence of P.

Since we can only perform prefix searching with *LZTrie*, and since phrase B_i has the string P as a suffix, we cannot use the *LZTrie* to find B_i. But note that P^r is a prefix of B_i^r, therefore it can be easily found by searching for P^r in *RevTrie* in $O(m^2 \log \sigma)$ time. Say we arrive at node v_r. Any node v'_r descending from v_r in *RevTrie* (including v_r itself) corresponds to a phrase terminated with P. Notice the relation with subpath queries (see Section 2.5.2). For each such v'_r, we traverse and report the subtree of the corresponding *LZTrie* node v_{lz} (found using $rids$ and $Node$). For any node v'_{lz} in the subtree of v_{lz}, we report an occurrence $[\![t, m + (depth(v'_{lz}) - depth(v_{lz}))]\!]$, where t is the phrase identifier (ids) of node v'_{lz}.

Occurrences of type 1 are located in $O(m^2 \log \sigma + occ_1)$ time, since each occurrence takes constant time in *LZTrie*. For cardinality queries we just need to compute the subtree size of each v_{lz} in *LZTrie*, as every v'_{lz} in that subtree corresponds to an occurrence of type 1.

Occurrences of Type 2. The occurrence spans two consecutive phrases, B_t and B_{t+1}, such that a prefix $P[1..i]$ matches a suffix of B_t and the suffix $P[i+1..m]$ matches a prefix of B_{t+1}. P can be split at any position, so we have to try them all. The idea is that, for every possible split, we search for the reverse pattern prefix $P^r[1..i]$ in *RevTrie* (getting node v_r) and for the pattern suffix $P[i+1 \ldots m]$ in *LZTrie* (getting node v_{lz}). The *RevTrie* node v_r for $P^r[1..i]$ is stored in array $C_r[i]$, since it shall be needed later.

As in a trie all the strings represented in a subtree form a preorder interval, we have two preorder intervals: one in the space of reversed phrases (phrases finishing with $P[1..i]$) and one in that of the normal phrases (phrases starting with $P[i+1..m]$), and need to find the phrase pairs $(t, t+1)$ such that t is in the *RevTrie* preorder interval and $t+1$ is in the *LZTrie* preorder interval. As we have seen in Example 3.3, this is what the range searching data structure (*Range*) is for. If we denote $p_{lz} = preorder(v_{lz})$ and $p_r = preorder(v_r)$, we must search *Range* for:

$$[p_{lz}..p_{lz} + subtreesize(v_{lz}) - 1] \times [p_r..p_r + subtreesize(v_r) - 1]$$

For every point (p, p') found, we use either p or p' to map to the corresponding *RevTrie* or *LZTrie* node, respectively (recall that the coordinates p and p' are preorders in these

tries). Depending on the representation used for the data structures, throughout this thesis sometimes we will choose to map to *RevTrie* (obtaining the corresponding phrase identifier t) or we will choose to map to *LZTrie* (obtaining the corresponding phrase identifier $t+1$). For every such point we report an occurrence $[\![t, i]\!]$. Occurrences of type 2 are then located in $O(m^3 \log \sigma + (m + occ_2) \log n)$ time, where the first term comes from searching the tries (in particular, searching for the $O(m)$ partitions of P in the *RevTrie*), and the second one is for the $m - 1$ range searches on *Range*.

Occurrences of Type 3. The occurrence spans three or more phrases, $B_{t-1}, \ldots, B_{\ell+1}$, such that $P[i..j] = B_t \ldots B_\ell$, $P[1..i-1]$ matches a suffix of B_{t-1} and $P[j+1..m]$ matches a prefix of $B_{\ell+1}$. We need one more observation for this part: Since the LZ78 algorithm guarantees that every phrase represents a different string (Property 2.6), there is at most one phrase matching $P[i..j]$ for each choice of i and j. Therefore, if we partition P into more than two consecutive substrings, there is at most one pattern occurrence for such partition, which severely limits occ_3 to $O(m^2)$, the number of different partitions of P.

Let us define matrix $C_{lz}[1..m, 1..m]$ and arrays A_i, for $1 \leqslant i \leqslant m$, which store information about the search. We first identify the only possible phrase matching each substring $P[i..j]$. This is done by searching for every pattern substring $P[i..j]$ in *LZTrie*, for increasing i and for each i value we increase j. Thus, we perform a single search in the trie for each i. We record in $C_{lz}[i, j]$ the *LZTrie* node corresponding to $P[i..j]$, and store the pair (id, j) at the end of A_i, such that id is the phrase identifier of the node corresponding to $P[i..j]$. Note that since we search for $P[i..j]$ for increasing j, we get the values of id in increasing order, as the phrase identifier of a node is always larger than that of the parent node. Therefore, the corresponding pairs in A_i are stored by increasing value of id. This process takes $O(m^2 \log \sigma)$ time.

Then we find the $O(m^2)$ maximal concatenations of successive phrases that match contiguous pattern substrings. For $1 \leqslant i \leqslant j \leqslant m$, for increasing j, we try to extend the match of $P[i..j]$ to the right. If id is the phrase identifier for node $C_{lz}[i, j]$, then we have to search for $(id + 1, r)$ in array A_{j+1}, for some r. Array A_{j+1} can be binary searched because it is sorted. If we find $(id + 1, r)$ in A_{j+1}, this means that $B_{id} = P[i..j]$ and $B_{id+1} = P[j + 1..r]$, which also means that the concatenation of phrases $B_{id}B_{id+1}$ equals $P[i..r]$. We repeat the process from $j = r$, and stop when the pair $(id+1, r)$ is not found in the corresponding array (this means that a concatenation of phrases cannot be extended further, so the current concatenation is maximal). See [Nav04] for further details.

As we have to perform $O(m^2)$ binary searches in arrays of size $O(m)$, this procedure takes $O(m^2 \log m)$ worst-case time. In practice, the binary search is replaced by hashing schemes, taking $O(m^2)$ time on average [Nav04, Section 6.5].

If $P[i..j] = B_t \ldots B_\ell$ is a maximal concatenation, then we check whether phrase $B_{\ell+1}$

starts with $P[j+1..m]$, that is, we check whether $Node[\ell+1]$ is a descendant of node $C_{lz}[j+1, m]$, in constant time per maximal concatenation. Finally we check whether phrase B_{t-1} ends with $P[1..i-1]$, by starting from $Node[i-1]$ in *LZTrie* and successively going to the parent to check whether the last $i-1$ symbols, read upwards, equal $P^r[1..i-1]$, in $O(m \log \sigma)$ time per maximal concatenation. If all these conditions hold, we report an occurrence $[\![t-1, i-1]\!]$. Overall, occurrences of type 3 are located in $O(m^3 \log \sigma)$ time.

Overall Query Time. Note that each of the $occ = occ_1 + occ_2 + occ_3$ possible occurrences of P lies exactly in one of the three cases above. Overall, the total search time to report the *occ* occurrences of P in T is $O(m^3 \log \sigma + (m + occ) \log u)$.

Extracting Text Substrings. The original LZ-index is able to extract text substrings, yet not in the way we have defined before: we have to provide an LZ78 phrase number from where to start the extraction. We assume also that the ℓ symbols we want to extract correspond to whole phrases (in Section 5.4 we shall avoid all these restrictions). Given phrase i, we follow the upward path from $Node[i]$ up to the *LZTrie* root, outputting the symbols labeling the upward path. Then we perform the same procedure but now starting from $Node[i+1]$ in *LZTrie*, and so on until we extract the ℓ desired symbols, taking overall $O(\ell \log \sigma)$ time, because operation *parent* is supported in $O(\log \sigma)$ time in theory [Nav04]. Finally, we can uncompress the whole text T in $O(u \log \sigma)$ time using the same idea, starting the procedure from the first LZ78 phrase.

Lemma 3.3 ([Nav04, Nav08])**.** *Given a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, the LZ-index of Navarro (*Nav-LZI*) requires $4uH_k(T) + o(u \log \sigma)$ bits of space, for any $k = o(\log_\sigma u)$. Given a search pattern $P[1..m]$, this index is able to:*

(*1*) *locate the occ occurrences of pattern P in text T in $O(m^3 \log \sigma + (m + occ) \log u)$ worst-case time;*

(*2*) *count the pattern occurrences in $O(m^3 \log \sigma + m \log u + occ)$ time;*

(*3*) *determine whether pattern P exists in T in $O(m^3 \log \sigma + m \log u)$ time; and*

(*4*) *extract any text substring of length ℓ in $O(\ell)$ time.*

3.3.6 Improving the Algorithm for Finding Maximal Concatenations

We now improve the algorithm for finding maximal concatenation of phrases, which is needed to find occurrences of type 3. We replace the binary searches on arrays A_i by an access to the correct position in matrix C_{lz}.

When computing maximal concatenations of phrases, for each $1 \leqslant i \leqslant j \leqslant m$, for increasing j, we try to extend the match of $P[i..j]$ to the right. Let id be the phrase identifier for node $C_{lz}[i,j]$, then $P[i..j] = B_{id}$ holds. Then, to check whether $P[j+1..r] = B_{id+1}$ holds, instead of searching for $(id+1, r)$ in A_{j+1} as before, we note that $r = j + l$, where l is the length of phrase B_{id+1}, which in turn is computed as $l = \mathit{depth}(\mathit{Node}[id+1])$ in *LZTrie*. Then we check, in constant time, whether the node $C_{lz}[j+1, j+l]$ corresponds to identifier $id+1$. This means that $P[j+1..r] = B_{id+1}$ holds, for $r = j + l$, and hence we can extend the concatenation of phrases to $P[i..r] = B_{id}B_{id+1}$. We repeat the process for $j = r$ and stop the procedure when the above condition does not hold, or r becomes greater than m.

Lemma 3.4. *Given the LZ78 parsing of text $T\$ = B_0 \dots B_n$, and given a pattern $P[1..m]$, we can compute the maximal concatenation of successive phrases $B_t \dots B_\ell$ that match contiguous pattern substrings $P[i..j]$, for any $0 \leqslant i \leqslant j \leqslant m$, in $O(m^2)$ time overall.*

Note that using this algorithm to find maximal concatenations we do not improve the total performance of the algorithm for finding occurrences of type 3. However, the reduction will be relevant in Chapter 5 for improved versions of the LZ-index.

3.3.7 The NAV-LZI in Practice

In the practical implementation of LZ-index [Nav04, Nav08], the *Range* data structure defined in Section 3.3.1 is replaced by *RNode*, which is a mapping from phrase identifiers to their node in *RevTrie*. The *RNode* data structure requires $n \log n$ bits, and so this practical version of LZ-index also requires $4uH_k(T) + o(u \log \sigma)$ bits, for any $k = o(\log_\sigma u)$.

Now occurrences of type 2 are found as follows: For every possible split $P[1..i]$ and $P[i+1..m]$ of P, assume the search for $P^r[1..i]$ in *RevTrie* yields node v_r, and the search for $P[i+1..m]$ in *LZTrie* yields node v_{lz}. Then, one checks each phrase t in the subtree of v_r and reports it if $\mathit{Node}[t+1]$ descends from v_{lz}. Each such check takes constant time. Yet, if the subtree of v_{lz} has fewer elements, one does the opposite: check phrases from v_{lz} in v_r, using $\mathit{RNode}[t-1]$. Unlike when using *Range*, now the time to solve occurrences of type 2 is proportional to the smallest subtree size among v_r and v_{lz}, which can be arbitrarily larger than the number of occurrences reported. That is, by using *RNode* we have no worst-case guarantees at search time. However, the average search time for occurrences of type 2 is $O(n/\sigma^{m/2})$ [Nav04, Nav08]. This is $O(1)$ for long patterns, $m \geqslant 2 \log_\sigma n$.

For occurrences of type 3, after finding that $P[i..j] = B_t \dots B_\ell$ is a maximal concatenation, one checks whether phrase $B_{\ell+1}$ starts with $P[j+1..m]$ by using operation $\mathit{ancestor}(C_{lz}[j+1, m], \mathit{Node}[\ell+1])$, just as in Section 3.3.5. To check whether phrase B_{t-1} ends with $P[1..i-1]$, instead of performing a symbol-per-symbol checking in *LZTrie*, as

done in the original LZ-index, one simply checks whether $ancestor(C_r[i-1], RNode[t-1])$ holds in *RevTrie*.

Despite this version of LZ-index does not provide worst-case guarantees at search time, it has been shown to be competitive against state-of-the-art indexes [Nav08].

3.4 Russo and Oliveira's LZ-index (ILZI)

Russo and Oliveira [RO07] define the *Inverted Lempel-Ziv Index* (ILZI for short). They discard the original LZ78 parsing of the text, and define a variant parsing instead, which they call the *maximal parsing* of text T.

To construct the ILZI, they first construct the LZ78 parsing of $T^r = Z_1 \dots Z_{n'}$, and then construct, among other data structures, the sparse suffix tree $\mathcal{T}_{78}$ for the set of strings $\{Z_1^r, \dots, Z_{n'}^r\}$. Russo and Oliveira prove that the number of nodes in $\mathcal{T}_{78}$ is at most $2n$, where n is the number of phrases in the LZ78 parsing of T.

At search time, occurrences of type 1 are found in a similar way as for the NAV-LZI, in $O(m + occ_1)$ time. Occurrences of type 2 and type 3 are found in time proportional to $O(m)$ in a very clever way. Instead of searching for every possible pattern substring, as done by the NAV-LZI for occurrences of type 3, Russo and Oliveira divide the pattern into maximal substrings, i.e., substrings of P that are nodes of $\mathcal{T}_{78}$ and are not the prefix of another substring of P that is also a node of $\mathcal{T}_{78}$. This process of dividing P into maximal substrings is carried out by means of a dynamic programming algorithm, and using some properties of the sparse suffix tree $\mathcal{T}_{78}$ to reuse the work done for some substring of P (the so-called descend and suffix walk process [RO07]). An important aspect is that pattern P cannot have more than $O(m)$ maximal substrings.

After partitioning the pattern into maximal substrings, and using these to find the sparse-suffix-tree ranges which allows us to search for occurrences spanning several phrases (as explained for other LZ-index in this chapter), they use these ranges in order to search for occurrences of type 2 and type 3 by means of a two-dimensional range search, in $O((m + occ_2) \log u)$ time.

Lemma 3.5 ([RO07]). *Given a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, the LZ-index of Russo and Oliveira (ILZI) requires $(5 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits of space, for any $0 < \epsilon < 1$. Given a search pattern $P[1..m]$, this index is able to:*

(1) locate the occ occurrences of pattern P in text T in $O((\frac{m}{\epsilon} + occ) \log u)$ time;

(2) count the pattern occurrences in $O(\frac{m}{\epsilon} \log u + occ)$ time;

(3) determine whether pattern P exists in T in $O(\frac{m}{\epsilon} \log u)$ time; and

(*4*) *extract any text substring of length* ℓ *in* $O(\frac{\ell}{\epsilon \log_\sigma u})$ *time*

It is important to note that the optimal bound for `extract` queries is achieved by Russo and Oliveira [RO07] by using the same approach that we shall define later in this thesis.

Chapter 4

Reducing the Space Requirement of LZ-indexes

The Nav-LZI, or simply the LZ-index, has a space requirement which is relatively large compared with competing schemes [Nav08]: 1.2–1.6 times the text size versus 0.6–0.7 and 0.3–0.8 times the text size of *Compressed Suffix Array* (CSA) [Sad03] and *FM-index* [FM05], respectively. In addition, the LZ-index does not offer space/time trade-offs, which limits its applicability. Yet, the LZ-index is faster to locate and to display the context of an occurrence. As we said before, fast displaying of text substrings is very important in self-indexes, as the text is not available otherwise.

Therefore, the challenge in this chapter is to reduce the space requirement of LZ-index (providing space/time trade-offs), while retaining the good features of fast locating and fast displaying of text substrings. We study how to reduce the space requirement of Navarro's LZ-index, using what we call the *navigational scheme* approach. This shall be the first step towards the more principled approach of Chapter 5.

4.1 The LZ-index as a Navigation Scheme

4.1.1 The Original Navigation Scheme

The LZ-index structure defined in Section 3.3.7 can be regarded as a *navigation scheme* that permits us moving back and forth from trie nodes to the corresponding *preorder positions*, both in *LZTrie* and *RevTrie*. The phrase identifiers are common to both tries (arrays *ids* and *rids*) and permit moving from one trie to the other by using *Node* and *RNode* mappings.

Fig. 4.1 shows the navigation scheme, where solid arrows represent the main data structures of the index. Dashed arrows are asymptotically "for free" in terms of

space requirement, since they are followed by applying *preorder* and *selectnode* on the corresponding parentheses structure (see Section 2.5.2). From now on we use the subscript "lz" for the operations on *LZTrie*, and subscript "r" for *RevTrie*. The four solid arrows in the diagram are in fact the four main components in the space usage of the LZ-index: array of phrase identifiers in *LZTrie* (*ids*) and in *RevTrie* (*rids*), and mapping from phrase identifiers to tree nodes in *LZTrie* (*Node*) and in *RevTrie* (*RNode*). The structure is symmetric and we can move from any point to any other.

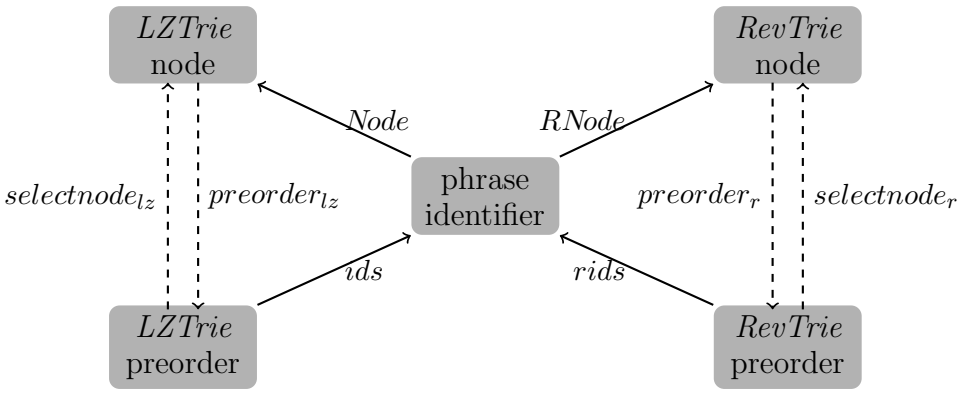


Figure 4.1: The original LZ-index navigation structures over index components.

The structure, however, is redundant in the sense that the number of arrows is not minimal. Given a graph with t nodes (in our case t index components), t arrows are sufficient to connect them in both directions (actually forming a ring structure). Since nodes and preorder positions in the tries are "connected" using operations *preorder* and *selectnode* over the trie representations (see Section 2.5.2), we can think that there are only three main index components to connect: *LZTrie* (either nodes or preorder positions), phrase identifiers, and *RevTrie* (either nodes or preorder positions). Next we define more space-efficient representations for LZ-index, trying to reduce the number of arrows in the scheme. Note that, because of Lemma 2.1, we are interested in reducing the number of index components that require $n \log n = uH_k(T) + o(u \log \sigma)$ bits of storage.

4.1.2 Schemes Requiring $3uH_k + o(u \log \sigma)$ bits

In this section we present schemes requiring only three solid arrows to connect the LZ-index components, thus forming a ring structure that still allows the same navigation as in the original LZ-index. Different choices yield different efficiencies depending on how often each type of navigation is used during the search.

Scheme 1. The following data structures compose this version of LZ-index:

(1) *LZTrie*: The Lempel-Ziv trie, which is implemented with the following data structures

— $par[0..2n-1]$: The tree shape of *LZTrie* represented either with balanced parentheses (Lemma 2.10) or with DFUDS (Lemma 2.11), requiring in any case $2n + o(n)$ bits.

— $letts[1..n]$: The array of symbols labeling the arcs of *LZTrie*, represented as explained in Section 2.5.2, depending on the representation used for *par*. The space requirement is, in any case, $n \log \sigma + o(n \log \sigma) = o(u \log \sigma)$ bits.

— $ids[1..n]$: The array of LZ78 phrase identifiers in preorder. Since $ids[0] = 0$, we do not store this value. Note that *ids* is a permutation of $\{1, \ldots, n\}$. The space requirement is $n \log n$ bits.

(2) *RevTrie*: The *Patricia* tree [Mor68] of the reversed LZ78 phrases, which is implemented with the following data structures

— $rpar[0..2n'-1]$: The *RevTrie* structure, represented either with BP or with DFUDS, compressing empty unary paths and thus ensuring $n' \leqslant 2n$ nodes, because empty non-unary nodes still exist. Thus, the space requirement is $2n' + o(n')$ bits.

— $rletts[1..n']$: The array storing the first symbol of each edge label in *RevTrie*, represented as for *LZTrie* and requiring $n' \log \sigma + o(n')$ bits of space.

— $skips[1..n']$: the *Patricia tree* skips of the nodes in preorder, using $\log \log u$ bits per node and inserting empty unary nodes when the skip exceeds $\log u$. In this way, one out of $\log u$ empty unary nodes could be explicitly represented. In the worst case there are $O(u)$ empty unary nodes, of which $O(u/\log u)$ are explicitly represented. This adds $O(u/\log u)$ nodes to n', which translates into $O((n' + \frac{u}{\log u})(3 + \log \sigma + \log \log u)) = o(u \log \sigma)$ bits overall for the *RevTrie* nodes, symbols, and skips.

— $B[1..n']$: A bit vector supporting *rank* and *select* queries, and requiring $n'(1 + o(1))$ bits of space (see Lemma 2.4 (1)). This bit vector marks the non-empty nodes: The j-th bit of B is 1 iff the node with preorder position j in *rpar* is not empty, otherwise the bit is 0. Given a position i in *rpar* corresponding to a *RevTrie* node, the corresponding bit in B is $B[preorder_r(i)]$. The preorder of a node p counting only non-empty nodes can be computed as $rank_1(B, preorder_r(i))$.

(3) $RNode[1..n]$: The mapping from phrase identifiers to the corresponding *RevTrie* node. Since we represent nodes as the positions of opening parentheses, and since there are $2n' \leqslant 4n$ such positions in *RevTrie*, this mapping needs $n \log 4n = n \log n + 2n$ bits. We only store pointers to non-empty nodes.

(4) $Rev[1..n]$: A mapping from a *RevTrie* preorder position to the corresponding *LZTrie* node, defined as $Rev[i] = Node[rids[i]]$. Given a position i in *rpar* corresponding to a non-empty *RevTrie* node, the corresponding *Rev* value (i.e., *LZTrie* node) is $Rev[rank_1(B, preorder_r(i))]$. The space requirement is $n \log n + n$ bits.

The resulting navigation scheme is shown in Fig. 4.2(a). The search algorithm remains the same since we can map preorder positions to nodes in the tries and vice versa (see Section 2.5.2), and also we can simulate the missing arrays $rids(i) \equiv ids[preorder_{lz}(Rev[i])]$ and $Node(i) \equiv Rev[rank_1(B, preorder_r(RNode[i]))]$, all of which take constant time.

We have reduced the space requirement to $3n \log n + 3n \log \sigma + 2n \log \log u + 11n + o(u) = 3n \log n + o(u \log \sigma)$ bits if $\log \sigma = o(\log u)$, which according to Lemma 2.1 is $3uH_k(T) + o(u \log \sigma)$ bits, for any $k = o(\log_\sigma u)$.

The *child* operation on *RevTrie* can now be supported in $O(1)$ time if we use DFUDS, because we store *rletts* and the skips. This compares well to the $O(h \log \sigma)$ time of the original LZ-index [Nav04]. Now, because *RevTrie* is a Patricia tree and the underlying strings are not readily available, it is not obvious how to traverse it. The next lemma addresses this issue.

Lemma 4.1. *Given a string $s \in \Sigma^*$, we can determine whether it is represented in RevTrie or not (finding the corresponding node in the affirmative case) in $O(|s|)$ time.*

Proof. To find the node corresponding to string s we descend from the *RevTrie* root, using operation $child(x, \alpha)$ on the first symbol of each edge label, which is stored in *rletts*, and using the skips to compute the next symbol of s to use in the descent. If s cannot be consumed while descending, then we determine that it is not represented in *RevTrie* in $O(|s|)$ time. Otherwise, assume that after consuming string s in this way we arrive at node v_r with preorder j in *RevTrie* (counting only non-empty nodes). The string labeling the root-to-v_r path in *RevTrie* can be computed by accessing the node $v_{lz} = Rev[j]$ in *LZTrie*, and then extracting the string labeling the v_{lz}-to-root path in *LZTrie*. Then we compare that string against s to verify that the node we arrived at corresponds to s, or otherwise that s does not occur in *RevTrie*.

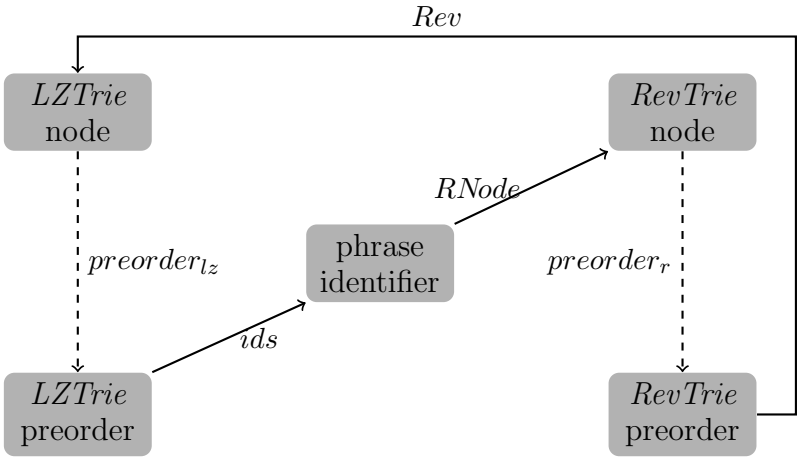
In case node v_r in *RevTrie* is empty, $Rev[j]$ is undefined. Notice, however, that there must be at least one non-empty node descending from this empty node, since leaves in *RevTrie* cannot be empty as they always correspond to an LZ78 phrase. Given that the string represented by every non-empty node in the subtree of node v_r has the string s as a prefix, the corresponding strings in *LZTrie* have s^r as a suffix. So we can use any *Rev* value within the subtree of node v_r in order to map to the *LZTrie* and then extract the string it represents. We can use, for example, the value $Rev[rank_1(B, j) + 1]$, which corresponds to the next non-empty node within the subtree of node v_r. We know when to stop extracting in *LZTrie*, since we know the length of the string we are looking for.

The overall cost for the descending process is therefore $O(|s|)$. □

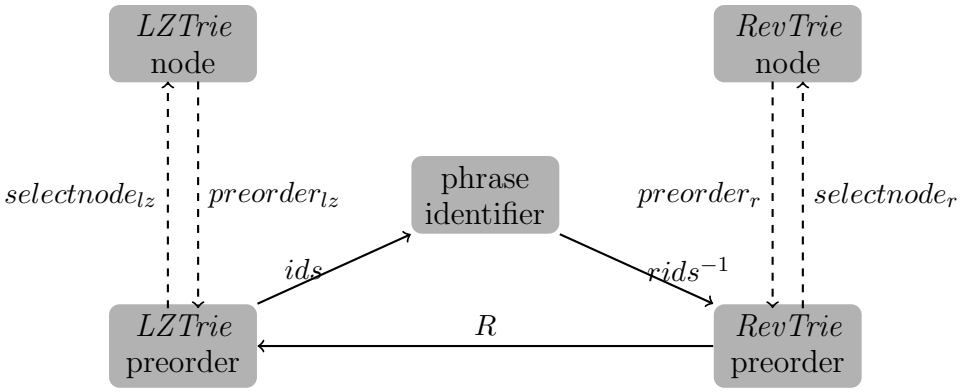
Operations *child* and *parent* on *LZTrie* can be also computed in $O(1)$ time if we use DFUDS on this trie, versus the $O(\log \sigma)$ time (in theory) of the original LZ-index. Hence, occurrences of type 1 are located in $O(m+occ_1)$ time; occurrences of type 2 are located now in $O(\frac{n}{\sigma^{m/2}})$ average time; occurrences of type 3 are located as in the original LZ-index, in $O(m^2)$ average time, by using hashing to find the maximal concatenations of phrases (for practical reasons, we decide to use this approach in practice, instead of the one defined by Lemma 3.4). Therefore, the *occ* occurrences of P can be located in $O(m^2 + \frac{n}{\sigma^{m/2}})$ average time. This time is $O(m^2)$ on average for $m \geqslant 2\log_\sigma u$.

Practical Issues. On the practical side, the access from *RevTrie* nodes to the corresponding *LZTrie* node is faster under this scheme, since the direct link *Rev* is faster than the composition of *rids* and *Node* of the original scheme. This is good, for example, for finding occurrences of type 1, which can be dominant for short patterns, as there is a high probability that an occurrence is contained in a single phrase. However, sometimes we must follow longer navigation paths in the search process: for example, when finding occurrences of type 2, we can choose to traverse the subtree in *RevTrie*, and for each phrase identifier id in such subtree apply $Node(id+1)$ to check whether it descends from the appropriate subtree in *LZTrie*. As now we have to simulate *Node*, this is more expensive (in practice, even if not asymptotically) than in the original scheme. Even worse, since array *rids* is not stored in *RevTrie*, we must simulate $rids(i)$ to get the phrase identifier id. Therefore, the search time could be increased in practice, depending on the number of occurrences of each type.

Let us study occurrences of type 2 in more detail, since they seem to be critical under this scheme. Suppose that for a given partition $P[1..i]$ and $P[i+1..m]$ of P we get nodes v_{lz} and v_r in *LZTrie* and *RevTrie* respectively. If we choose to traverse the subtree of v_r in *RevTrie*, then for each node v'_r in this subtree we get the corresponding phrase identifier $id = ids[preorder_{lz}(Rev[p_r])]$, where p_r is set initially to $p_r = rank_1(B, preorder_r(v_r))$, and it is incremented by one with each node in a preorder traversal of the subtree. We then check whether the node $Rev[rank_1(B, preorder_r(RNode[id+1]))]$ descends from v_{lz} in *LZTrie*. If, on the other hand, we choose to traverse the subtree of v_{lz} in *LZTrie*, then for each node v'_{lz} in this subtree we get the phrase identifier as $id = ids[p_{lz}]$, where p_{lz} is set initially as $p_{lz} = preorder_{lz}(v_{lz})$, and it is incremented by one with each node in a preorder traversal. We then check whether the node $RNode[id-1]$ descends from v_r in *RevTrie*. Empirically, a check from *RevTrie* to *LZTrie* is about 3 times as expensive as in the opposite direction, and thus we choose to traverse the subtree of v_r whenever its size is less than 1/3 the size of the subtree of v_{lz}.



(a) Scheme 1.



(b) Scheme 2.

Figure 4.2: Reduced navigation schemes over LZ-index components, requiring $3uH_k + o(u \log \sigma)$ bits.

Scheme 2. This scheme tries to reduce the space requirement while also reducing the average path length in the navigation scheme:

(1) *LZTrie*: The Lempel-Ziv trie, defined just as for Scheme 1.

(2) *RevTrie*: The *Patricia* tree of the reversed LZ78 phrases, defined just as for Scheme 1, but now we add

— $rids^{-1}[1..n]$: The explicit representation of the inverse of permutation *rids* of the original LZ-index definition, requiring $n \log n$ bits.

(3) $R[1..n]$: A mapping from *RevTrie* preorder positions to *LZTrie* preorder positions defined as $R[i] = ids^{-1}(rids[i])$ and requiring $n \log n$ bits. Given a position i in *rpar* corresponding to a non-empty *RevTrie* node, the corresponding R value (i.e., preorder in *LZTrie*) can be computed as $R[rank_1(B, preorder_r(i))]$.

The resulting navigation scheme is shown in Fig. 4.2(b). We can compute $rids(i) \equiv ids[R[i]]$, $RNode(i) \equiv selectnode_r(rids^{-1}[i])$, and $Node(i) \equiv selectnode_{lz}(R[rids^{-1}[i]])$, all in constant time.

The space requirement is $3n \log n + 3n \log \sigma + 2n \log \log u + 8n + o(u) = 3n \log n + o(u \log \sigma)$ bits if $\log \sigma = o(\log u)$, which according to Lemma 2.1 is $3uH_k(T) + o(u \log \sigma)$ bits, for any $k = o(\log_\sigma u)$.

If we use DFUDS to represent both *LZTrie* and *RevTrie*, then we can locate the *occ* occurrences of pattern P in $O(m^2 + \frac{n}{\sigma^{m/2}})$ average time, which is $O(m^2)$ for $m \geqslant 2 \log_\sigma u$.

Practical Issues. It is interesting to note that the average path length of this scheme is shorter than that of Scheme 1, which can translate into a more efficient navigation among index components. In this scheme, for occurrences of type 1 we have direct access to a *LZTrie* preorder by using R, and then we have to apply *selectnode* to get the node whose subtree contains the pattern occurrences. This can be slightly slower than for Scheme 1, where we have direct access to the corresponding node.

However, for occurrences of type 2 we have to follow shorter paths than for Scheme 1. Suppose that for a given partition $P[1..i]$ and $P[i+1..m]$ of P we get nodes v_{lz} and v_r in *LZTrie* and *RevTrie* respectively. If we choose to traverse the subtree of v_r in *RevTrie*, for each node in this subtree we get the corresponding phrase identifier *id* by using both the R mapping and then *ids*. Then we use $R[rids^{-1}[id+1]]$ to get the corresponding *LZTrie* preorder, and then we check whether this preorder lies within the subtree of v_{lz}. If, on the other hand, we choose to traverse the subtree of v_{lz} in *LZTrie*, then for every node in this subtree we get the corresponding phrase identifier *id* using *ids*, and then we check whether the preorder $rids^{-1}[id-1]$ lies within the subtree of v_r in *RevTrie*. Thus, a check from *RevTrie* to *LZTrie* is twice as expensive as in the opposite direction, and thus we choose

to traverse the subtree of v_r whenever its size is less than half the size of the subtree of v_{lz}.

Fortunately, the checks of type 2 can be carried out directly on the preorders of both tries, avoiding the use of (the usually expensive) *selectnode* to get the corresponding trie node: if we choose to traverse the subtree of v_r, for example, we compute the preorder interval for the subtree of v_{lz} as $[preorder_{lz}(v_{lz})..preorder_{lz}(v_{lz}) + subtreesize_{lz}(v_{lz}) - 1]$ (recall that *preorder* is computed by means of *rank*), and then we check whether the *LZTrie* preorders we get from the nodes in the subtree of v_r lie within the preorder interval of v_{lz}. In this way, we compute just one *rank* per partition to get the interval, and then we check the *LZTrie* preorder of the candidates by using just this interval, rather than computing *selectnode* for every possible candidate in that partition. This introduces very important savings in the practical search time.

There are many other possible schemes that achieve $3uH_k(T)+o(u \log \sigma)$ bits of space. We have focused on the two most promising ones. For example, consider a scheme where we only replace the R mapping of Scheme 2 by the *Rev* mapping of Scheme 1. We have again direct access for occurrences of type 1, but occurrences of type 2 now introduce the computation of *rank* in *LZTrie* for every possible candidate, which is expensive.

4.1.3 Schemes Requiring $(2 + \epsilon)uH_k + o(u \log \sigma)$ bits

In Section 4.1.2 we have used the minimal number of arrows to connect the three main components of LZ-index, forming a ring structure. It seems that we cannot reduce further the space requirement of the index by using our navigation-scheme approach. However, many of the data structures of the LZ-index are just permutations, and so the corresponding arrows can be made bidirectional by means of the data structure for permutations described in Lemma 2.7, using just $(1 + \epsilon)n \log n + n + o(n)$ bits for both arrows. This opens several new possibilities.

Scheme 3. This scheme represents the following data:

(1) *LZTrie*: The Lempel-Ziv trie, defined as for Scheme 1, except that now we use the representation of Lemma 2.7 for *ids* such that the inverse permutation ids^{-1} can be computed in $O(1/\epsilon)$ time, requiring $(1 + \epsilon)n \log n + n + o(n)$ bits for any $0 < \epsilon < 1$.

(2) *RevTrie*: The *Patricia* tree of the reversed LZ78 phrases, defined as in Scheme 1, but now we add the array *rids* represented using the data structure of Lemma 2.7, so as to be able to compute $rids^{-1}$ efficiently.

The resulting navigation scheme is shown in Fig. 4.3(a). We can simulate the missing arrays $Node(i) \equiv selectnode_{lz}(ids^{-1}(i))$ and $RNode(i) \equiv selectnode_r(rids^{-1}(i))$, all in $O(1/\epsilon)$ time.

The space requirement is $(2+\epsilon)n\log n + 3n\log\sigma + 2n\log\log u + 10n + o(u) = (2+\epsilon)n\log n + o(u\log\sigma)$ bits, which according to Lemma 2.1 is $(2+\epsilon)uH_k(T) + o(u\log\sigma)$ bits, for any $k = o(\log_\sigma u)$.

Once again, if we use DFUDS to represent both tries of the LZ-index, then occurrences of type 1 can be located in $O(m+\frac{occ_1}{\epsilon})$ time, because we must use ids^{-1} to access to *LZTrie*; occurrences of type 2 are located in $O(\frac{n}{\epsilon\sigma^{m/2}})$ because we must use inverse permutations to move between tries; occurrences of type 3 are located in $O(\frac{m^2}{\epsilon})$ time, since we need to use *Node* and *RNode* to check every possible candidate, in $O(\frac{m^2}{\epsilon})$ time overall. Thus, the *occ* occurrences of P can be located in $O(\frac{m^2}{\epsilon} + \frac{n}{\epsilon\sigma^{m/2}})$ average time, for $0 < \epsilon < 1$. This is $O(\frac{m^2}{\epsilon})$ for $m \geqslant 2\log_\sigma u$.

Practical Issues. This scheme stores the phrase identifiers for both tries, which, as we have seen for the previous schemes, is very convenient for occurrences of type 2: recall that when traversing the *RevTrie* subtree we have to get the phrase identifier of each node in the subtree; if we do not store the *RevTrie* identifiers, we have to access *LZTrie* to get them (as is the case of Schemes 1 and 2) and then we have to access *LZTrie* again to perform the check. This is not the case for Scheme 3. However, now we have paths including inverse permutations, which introduce an extra time overhead in practice.

Notice also that this scheme is symmetric in the sense that the checks for occurrences of type 2 cost the same in any direction we choose.

Scheme 4. This scheme represents the following data:

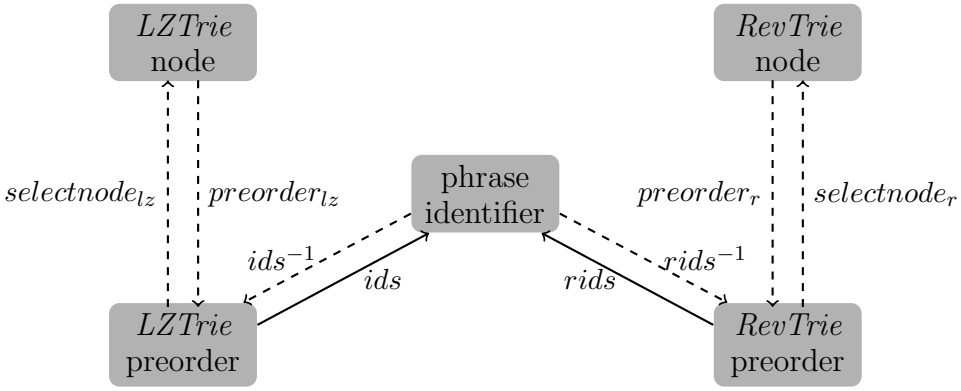
(1) *LZTrie*: The Lempel-Ziv trie, defined just as in Scheme 3.

(2) *RevTrie*: The *Patricia* tree of the reversed LZ78 phrases, defined just as in Scheme 1.

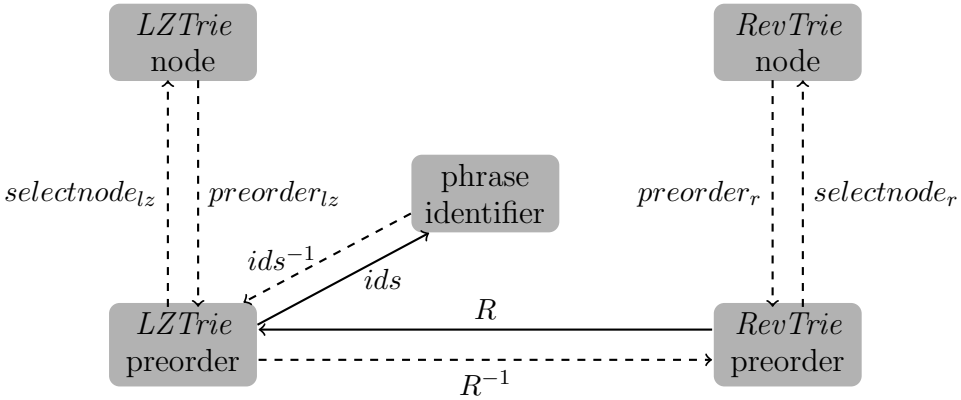
(3) $R[1..n]$: The mapping from *RevTrie* preorder positions to *LZTrie* preorder positions, as defined in Scheme 2. This time R is implemented using the succinct data structure for permutations of Lemma 2.7, requiring $(1+\epsilon)n\log n + n + o(n)$ bits to represent R and compute R^{-1} in $O(1/\epsilon)$ worst-case time.

In Fig. 4.3(b) we draw the navigation scheme. We can simulate the missing arrays $rids(i) \equiv ids[R[i]]$, $RNode(i) \equiv selectnode_r(R^{-1}(ids^{-1}(i)))$, and $Node(i) \equiv selectnode_{lz}(ids^{-1}(i))$, all of which take $O(1/\epsilon)$ time.

The space requirement is $(2+\epsilon)n\log n + 3n\log\sigma + 2n\log\log u + 10n + o(u) = (2+\epsilon)n\log n + o(u\log\sigma)$ bits, which according to Lemma 2.1 is $(2+\epsilon)uH_k(T) + o(u\log\sigma)$ bits, for any $k = o(\log_\sigma u)$. Just as for the previous scheme, the *occ* occurrences of P can be located in $O(\frac{m^2}{\epsilon} + \frac{n}{\epsilon\sigma^{m/2}})$ average time, for $0 < \epsilon < 1$.



(a) Scheme 3.



(b) Scheme 4.

Figure 4.3: Reduced navigation schemes over LZ-index components, requiring $(2+\epsilon)uH_k + o(u \log \sigma)$ bits.

Practical Issues. This scheme has more efficient access between tries than Scheme 3, as we have to use R in the *RevTrie*-to-*LZTrie* direction, and R^{-1} in the opposite way. However, since we store the phrase identifiers only in *LZTrie*, retrieving the identifier of a *RevTrie* node requires to access two arrays. For occurrences of type 2, the checks from *RevTrie* to *LZTrie* require to access R, then ids, and finally ids^{-1}, while in the opposite way we need to use ids, then ids^{-1}, and finally R^{-1}. The latter case can be more expensive since we have to compute two inverse permutations. Note also that ids^{-1} is used in both directions for occurrences of type 2, which means that this inverse permutation is the most used at search time. Hence, given an amount of space we are able to use, we should use a denser sampling for ids^{-1} than for R^{-1}.

Again, we have focused on the most promising schemes requiring $(2+\epsilon)uH_k(T) + o(u \log \sigma)$ bits, although there are many other choices.

4.2 Some Implementation Details

We describe in this section the most important details in the implementation of our indexes. We followed the API interface specification provided in the *Pizza&Chili* Corpus [FN05], and made the source code available at `http://pizzachili.dcc.uchile.cl/indexes/LZ-index/`.

4.2.1 Representing the Tries

We defined our indexes in Section 4.1 in such a way that we can use almost any suitable representation for the tries that compose the indexes. We just need to define accordingly the *preorder* and *selectnode* operations for the chosen representation. Taking this into account, we implement our reduced LZ-indexes in two different ways:

(1) Using the BP representation (Lemma 2.10) for both *LZTrie* and *RevTrie*;

(2) using the DFUDS representation (Lemma 2.11) for *LZTrie*, and the BP representation for *RevTrie*.

Note that we do not use DFUDS for *RevTrie*, as it requires more space in practice. Moreover, just as for the original LZ-index, we do not store the symbols labeling the *RevTrie* edges (i.e., the first symbol of each string labeling an edge) nor the Patricia-tree skips. This is in order to save space in practice, since these can be computed by using the connection with the *LZTrie*: previous experiments [Nav04, Nav08] showed that this is sufficient for *RevTrie* as most navigations on it are supported by using the *LZTrie* (and those that are not are usually deep in the trie, where the arity is very low and the attractive of DFUDS vanishes). We needed to use these arrays in theory in order to guarantee the average-time complexity of our data structures.

We describe these implementations in what follows.

Using BP Representation. We implemented the trie operations on top of the data structure for balanced parentheses of Navarro [Nav04, Section 6.1]. The trie operations are supported as explained in Section 2.5.2. Recall that in order to support the operations on BP we must support operations $findclose$, $excess$, and $enclose$ on the sequence of balanced parentheses. Therefore, we do not store information to support $findopen$, thus saving space. Moreover, we do not need to support operation $parent$ on *RevTrie*, and thus we do not store information to support $enclose$ on the parentheses representing this trie.

Operation $child(x, \alpha)$ is supported by using $child(x, i)$, for $i = 1, 2, \ldots$, until finding the child labeled α. This is because the symbols labeling the children of x are scattered throughout array $letts$ and must be found one by one using the operations on the parentheses. Whenever we need to support $rank$ and $select$ queries (this is, on top of the parenthesis sequences, to represent bitmap B in *RevTrie*, and for the permutation data structures), we use the data structure of González et al. [GGMN05].

Using DFUDS Representation. The main idea of using the DFUDS representation for *LZTrie* is to reduce the time overhead for computing operation $child(x, \alpha)$ incurred by BP.

As done for BP, we represent the DFUDS sequence of *LZTrie* on top of the data structure for balanced parentheses of Navarro [Nav04]. Note that the DFUDS representation of a trie tends to have far matching parentheses, since every node is formed by a number of opening parentheses (indicating the degree of the node), and only a few of these parentheses have the corresponding closing parentheses close enough so as not to be stored in the hash tables. Thus the hash tables tend to require more space under DFUDS. We also use the data structure of González et al. [GGMN05] to support $rank$ and $select$ queries.

We study the way in which the trie operations are used by the LZ-index search algorithm in order to make this representation more efficient. For example, DFUDS introduces a heavier use of $rank$ and $select$ in its operations (see Section 2.5.2), according to the original definitions [BDM+05]. However, many of these are redundant in the LZ-index (sometimes they are repeated twice in a given sequence of operations, as we will see below), and many others can be replaced by sequential scans over par, since the DFUDS position we are looking for should be not so far away from the current position. A list with the most important implementation details follows:

Supporting Operation $findopen$. Unlike the representation of LZ-index based exclusively on BP, with DFUDS we need to provide operation $findopen$ over the parentheses sequence. This is necessary, for example, to compute operation $parent$ [BDM+05] (recall also Section 2.5.2). Therefore, we need to add a second data structure, like the one used by Navarro [Nav04] (i.e., hash tables) to support $findopen$ and $enclose$ operations. This adds extra space to DFUDS.

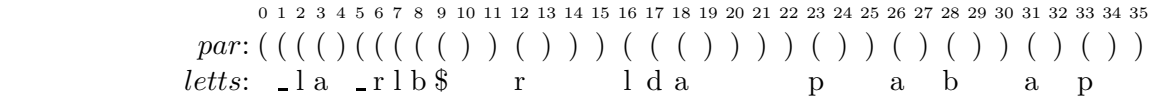


Figure 4.4: Illustration of our practical representation of array *letts* for the DFUDS representation. Notice that the labels of the children of a given node are stored in reverse order.

A Practical Representation of letts. Recall from Section 2.5.2 that the symbol by which a node x descends from its parent can be computed as $letts[rank_{(}(par, parent(x)) - 1 + childrank(x) - 1]$, which involves computing several *rank*, *select*, and parentheses operations. Although these can be computed in constant time in theory, we look for a more practical variant in practice.

This problem comes from the fact that there is an inverse relationship between the symbol labeling the i-th child of node x' of degree d (this symbol is originally stored at position i within the labels of children of x') and the opening parenthesis used to compute the DFUDS position of the i-th child of x' (recall that we use the $(d - i + 1)$-th opening parenthesis within the representation of x').

Therefore, we propose to represent *letts* by traversing the *LZTrie* in preorder and, for every node x reached, writing contiguously the symbols labeling the children of x, this time in *reverse order*. This means that the label of the last child appears in the first place, the label of the first child appears in the last place, and so on. See Fig. 4.4 for an illustration.

In this way there is a direct relationship between each of the opening parentheses defining a given node x' (which are used to find the children of x') and the labels of the children of x'. Thus, the label by which a given node x descends from its parent can be computed as $letts[rank_{(}(par, findopen(par, x - 1))]$, avoiding the use of operation $childrank(x)$ as in Section 2.5.2, which involves more *rank* and *select* operations [JSS07b].

Moreover, in many cases we can reuse the operation $findopen$ computed to get the symbol. For example, in the LZ-index, most of the times that we need to know the symbol of a given node, we also need to go to the parent node, as for instance when computing `extract` and `display` operations, and when we use the *LZTrie* to descend in *RevTrie*. Then, after computing the corresponding symbol, we retain the position given by operation $findopen$ in order to search for the parent of the current node, avoiding to repeat the same operation $findopen$ when computing operation *parent*.

Since the labels of the children of node x are stored contiguously and sorted (yet in reversed order, according to our representation), array *letts* is stored in a plain way, without using any *rank* and *select* data structure, thus saving space. Operation $child(x, \alpha)$ is implemented by binary searching the list of labels of x. When the list is small (say, less

than 10 elements) we perform a sequential scan on the symbols. This saves time compared with the BP representation, where we have to repeatedly use operation $findclose$ on the parentheses in order to find the child we are looking for. With DFUDS, on the other hand, this work is done on the symbols, and then we map again to the DFUDS sequence to find the corresponding child (this involves only one $findclose$ operation).

Avoiding Operation depth on DFUDS. The original DFUDS representation [BDM+05] does not provide operation *depth*, later supported by Jansson et al. [JSS07b] in $O(1)$ time and requiring $o(n)$ extra bits of space. However, in the LZ-index we need to use *depth* in a very limited way, which helps us implement this operation simply and efficiently. We choose not to store any extra depth information, and thus we save space. Instead we completely avoid the computation of *depth* at search time.

In the LZ-index search algorithm, we only need to use operation *depth* when locating occurrences of type 1 (see Section 3.3.5). Recall that to find occurrences of type 1 we must first search for P^r in *RevTrie*, getting node v_r. Then, for each node in the subtree of v_r we map to the corresponding node v_{lz} in *LZTrie*, and then we traverse the subtree of v_{lz} to report occurrences of type 1. The problem here is how to compute the offset of every occurrence within the corresponding phrase (in Section 3.3.5 we use operation *depth* to do that). However, note that the offset for node v_{lz} is m, since the phrase ends with P. Note also that the offset for the children of v_{lz} is $m+1$, and in general the offset of node v'_{lz} within the subtree of v_{lz} is $m+d$, where d is the difference of depths between v_{lz} and v'_{lz}.

So, instead of computing the depth of the nodes within the subtree of v_{lz}, which is expensive in our representation, we compute the offset of the nodes. This problem can be solved in a straightforward way in BP, since we perform a preorder traversal from v_{lz}, with initial offset m. We then increment the offset every time we enter a new subtree (which is indicated by '(' in BP), and decrement it when leaving a subtree (which is marked by ')' in BP). The preorder traversal is carried out by sequentially traversing the BP sequence and array *ids*, and not by using operation *child* on *LZTrie*. However, this is not so easy in DFUDS, since, for example, there is no clear marker of the end of a subtree (i.e., in a sequential scan on DFUDS there is no direct way to know whether a node is the last child of its parent).

Therefore, in the DFUDS representation we start a preorder traversal from node v_{lz}, and store the degree (number of children) of v_{lz} and its offset m in an initially empty stack. We then continue with the next node in preorder. At every step, the offset for the current node is computed as the offset of the parent node (which is the one at the top of the stack) plus one. Every time the degree stored in the top is (or becomes) 0 (leaves are a particular case), we pop it, and then decrement the degree of the node in the top, indicating that a new child of this node has been fully processed (this can eventually produce more pops when all of the children of the top node have been processed). As it can be seen, we are

using the stack to know when the subtree of a node has been completely processed. This procedure ends when the stack becomes empty.

As in the case of BP, the preorder traversal is performed by a sequential scan on the DFUDS sequence, and the degree of every node x is computed by counting the number of opening parentheses in the representation of x.

This procedure could be also used to compute the depth of all nodes within the subtree of v_{lz}, by initializing the stack with the depth of v_{lz}, instead of storing the initial offset m. Note also we do not need to explicitly store depths or offsets, as they correspond to the stack height plus a constant.

Implementation of Operation degree. In our implementation we do not use the original definition for operation *degree*, which is based on operation *select* in order to find the next closing parenthesis which finishes the definition of the node. This allows us to count the number of opening parentheses defining the current node. In practice, in most cases this closing parenthesis is not so far away from the current position in DFUDS, except perhaps for the trie root. This is because in practice the tries tend to have high degrees only in the first levels. Thus, we explicitly store the degree of the root node, and for the rest of the nodes we perform a sequential scan on the DFUDS sequence. To avoid looking at every parenthesis in the process, we advance by machine words (in our experiments this shall mean 32 bits), until finding the first word containing a closing parenthesis. We represent opening parentheses with a 0, and closing ones with a 1. Thus, we advance while the corresponding word represents a 0, i.e., it stores only opening parentheses. However, in the case of very large alphabets the original definition of *degree* could be better, or we could replace the sequential search by an exponential search using the rank subdirectory.

4.2.2 Computing Text Positions

We add to our indexes a data structure in order to transform pattern occurrences in the format $[\![t, o]\!]$ into real text positions. We define array *TPos* storing the absolute starting position (in the text) for the LZ78 phrases with identifier $i \cdot b$, for $i \geqslant 0$, for a total of $\frac{n}{b} \log u$ bits. We also define array *Offset*, storing in $Offset[i \cdot b + j]$, for $j \geqslant 0$, the offset (in number of text symbols) of phrase $i \cdot b + j$ with respect to phrase $i \cdot b$, requiring $n \log M$ extra bits of space, where M is the maximum number of text positions between two consecutive sampled phrases. If *LZTrie* has height $h = O(\log n)$ (which is true in practice with high probability [KS00]), then $M = O(b \log n)$, and thus array *Offset* requires $O(n(\log b + \log\log n))$ bits. By choosing $b = \log u$ the total space requirement for both arrays is $n + O(n \log\log u) = o(u \log \sigma)$ bits.

Given an occurrence $[\![t, o]\!]$, the real text position for that occurrence can be computed in constant time as $TPos[\lfloor (t+1)/b \rfloor] + Offset[t+1] - o$. In our experiments, we choose

$b = 32$ (in a 32-bit machine) such that the division by b and taking the floor can be carried out efficiently by using shifts on machine words. For `extract` queries, we are given a text position from where to extract and we want to know the corresponding *LZTrie* node from where to start uncompressing the text. Therefore, given a text position *pos* we can obtain the phrase containing *pos* by first binary searching *TPos*, finding the greatest phrase $i \cdot b$ such that its position $TPos[i]$ is smaller or equal to *pos*. Then, we sequentially look in the corresponding segment of $Offset[i \cdot b..(i+1) \cdot b]$ for the greatest phrase t whose starting position does not exceed *pos*. Thus, the text position *pos* belongs to phrase t. The time for this operation is $O(\log u)$, because of the binary search on *TPos* and the sequential search in the segment of *Offset*.

In Section 4.3.2 we will show the experimental space requirement of this data structure for a set of real-life texts.

4.2.3 Supporting Partial `locate` Queries

In many applications it is quite common that we do not need to find all of the pattern occurrences, but just a few (arbitrary) of them. For this kind of applications, we design an algorithm to answer *partial* `locate` queries, where we are interested in locating just K arbitrary pattern occurrences. Our algorithm, which profits from the properties of the LZ-index in order to support fast searches, is as follows:

(1) Given a search pattern P, notice that a particular occurrence of it can be found by searching for P in *LZTrie*. In other words, P equals an LZ78 phrase, which is a particular case of occurrence of type 1, and can be found very fast in practice since this is better than using the slower *RevTrie* [Nav08]. If P exists as a phrase, say corresponding to node v_{lz} in *LZTrie*, then all of the nodes descending from v_{lz} in the trie also correspond to occurrences of P, and can be used to answer the query. Thus, we traverse the subtree of v_{lz} and report every node found, as done for occurrences of type 1 (see Section 3.3.5). We stop the procedure as soon as we find K pattern occurrences.

(2) If the previous step was not enough to answer the query, and in case that P exists in *LZTrie*, we map to the node corresponding to P^r in *RevTrie* (which exist for sure since P is an LZ78 phrase), and go on to locate the rest of occurrences of type 1 as usual. Otherwise, we delay occurrences of type 1 for a further step and go to the next step.

(3) In case P does not exist as an LZ78 phrase, we proceed with occurrences of type 2, trying to reuse as much as possible the work already done in step (1). We are thus delaying occurrences of type 1 for a further step. Let $P[1..i]$ be the longest proper prefix of the pattern that exists as an LZ78 phrase. Hence, $P^r[1..i]$ exists as a reverse

phrase in *RevTrie*. Because of Property 2.5, every prefix of $P[1..i]$ also exists as a phrase in LZ78 (and the corresponding reverses exist in *RevTrie*). Then, to reuse the work already done when searching for P in *LZTrie*, we map to the *RevTrie* node corresponding to $P^r[1..i]$, which gives us node v_r, and then search for $P[i+1..m]$ in *LZTrie*, to get node v_{lz}. We then search for occurrences of type 2 corresponding to the partition $P[1..i]$ and $P[i+1..m]$, using the nodes v_{lz} and v_r, in the usual way and stopping as soon as we find K occurrences. Note that by choosing the longest prefix $P[1..i]$ that exists in *LZTrie*, we are reducing as much as possible the length of the suffix $P[i+1..m]$ to be searched in *LZTrie*. If this was not enough to answer the query, we repeat the process for the occurrences of $P[1..i-1]$ and $P[i..m]$, in a similar way, and so on.

(4) We search for the remaining occurrences of type 2 (i.e., using those partitions of the pattern that were not tried in the previous step)

(5) Then, we continue with occurrences of type 1, as usual and just if P does not exist in *LZTrie* (i.e., these were not tried before).

(6) Finally we try occurrences of type 3.

We call *level 0* of the search to the step of searching for P in *LZTrie*, *level 1* the search for occurrences of type 1 (either at steps (2) and (5)), *level 2* the search for occurrences of type 2 (either at steps (3) and (4)), and *level 3* the search for occurrences of type 3 (step (6)). As it can be seen, we try to get fast access to the pattern occurrences, avoiding as much as we can the trie navigations, which become more expensive if we want to locate just a few occurrences. We shall use this approach also to support efficient `exists` queries (similar to $K = 1$).

4.3 Experimental Results

We have now a number of practical reduced schemes for LZ-index, each one requiring a different amount of memory space. Hence given an amount of available storage, it is interesting to know which alternative is the best for that space. We hope that larger alternatives are faster in practice, whereas the smaller ones will still be competitive against the best existing indexes.

4.3.1 Experimental Setup

For the experiments of this section we used an Intel(R) Pentium(R) 4 processor at 3 GHz, about 4 gigabytes of main memory and 1024 kilobytes of cache, running version `2.6.13-gentoo` of Linux kernel. We compiled our algorithms with `gcc 3.3.6` using full optimization.

Text Collections. We test our indexes in different practical scenarios, using the texts provided in the *Pizza&Chili* Corpus [FN05]:

— English Texts: Although in many cases natural-language texts are searched for whole words or phrases, there are many other cases where a more powerful full-text search is needed. For the experiments with English text we use the file `http://pizzachili.dcc.uchile.cl/texts/nlang/english.200MB.gz` of 200 megabytes.

— DNA Sequences: Nowadays, one of the main applications of full-text indexing is that of computational biology, in particular indexing DNA sequences. We test with the file `http://pizzachili.dcc.uchile.cl/texts/dna/dna.200MB.gz`, of 200 megabytes.

— MIDI Pitch Sequences: A very interesting application that has appeared in recent years is that of processing MIDI pitch sequences. In this case we test with the file of about 53 megabytes downloadable from `http://pizzachili.dcc.uchile.cl/texts/music/pitches.gz`.

— XML Texts: Since XML is becoming the standard to represent semi-structured text databases as well as in many other applications, there exists the need of managing a huge amount of texts of this kind. In typical applications, XML documents are automatically generated in large amounts. It is interesting therefore to be able to compress the data, while at the same time being able to search and extract any part of the text, since XML data is usually queried and navigated by other applications. We test with the XML file of 200 megabytes downloadable from `http://pizzachili.dcc.uchile.cl/texts/xml/dblp.xml.200MB.gz`.

— Proteins: Another interesting application of text-indexing tools in biology is that of indexing and searching proteins. We use the file `http://pizzachili.dcc.uchile.cl/texts/protein/proteins.200MB.gz`, of 200 megabytes.

— Source Code: To test our indexes in applications like software development, we use the source-code file `http://pizzachili.dcc.uchile.cl/texts/code/sources.200MB.gz`, of 200 megabytes.

Comparison against Other Indices. We compare our indexes against the most efficient indexes we are aware of, most of them available in *Pizza&Chili*:

Sadakane's Compressed Suffix Array (CSA). This index [Sad03] is a representative of the family of *compressed suffix arrays* [GV05, GGV03, Sad03]. It requires $\epsilon^{-1}uH_0(T) + O(u \log\log \sigma)$ bits of space, a counting time of $O(m \log u)$, a locating time of $O(\log^{\epsilon} u)$ per occurrence reported, and an extracting time $O(\ell + \log^{\epsilon} u)$ for

any text substring of length ℓ, where $0 < \epsilon \leqslant 1$ is any constant. We use the code provided at `http://pizzachili.dcc.uchile.cl/indexes/Compressed_Suffix_Array/sada_csa.tgz`. We have two parameters to set up for this index. The first one is the sample rate of suffix array positions (this information is used to speed up the locating and extracting operations), and the second one is the sample rate of the Ψ function. For `exists` and `count` queries we do not store any suffix array position, but only Ψ values. For `locate` queries, we used values of 4, 8, 16, 32, and 64 for suffix array positions, and the value 128 to sample the Ψ function (as this has shown to be the most efficient alternative [FGNV08]).

Alphabet Friendly FM-index (AF-FMI). This index [FMMN07] is based on the *backward-search* concept [FM05]. It has a space requirement of $uH_k(T) + o(u \log \sigma)$ bits, a counting time $O(m)$, a locating time $O(\log^{1+\epsilon} u)$ per occurrence reported, and an extracting time $O(\ell + \log^{1+\epsilon} u)$, for $\sigma = O(\text{polylog}(u))$, any constant $\epsilon > 0$, and any $k \leqslant \alpha \log_\sigma u$, where $0 < \alpha < 1$ is any constant. We use the code provided at `http://pizzachili.dcc.uchile.cl/indexes/Alphabet-Friendly_FM-Index/af-index_v2.tgz`. We have only one parameter to set up in the code, which is the sample rate of suffix array positions. We have used sample rates of one suffix array position stored out of 4, 8, 16, 32, and 64 text positions. For `exists` and `count` queries we do not store any suffix array position.

Succinct Suffix Array (SSA). This index [MN05] is also based on backward search, but uses only one *wavelet tree* [GGV03], achieving $uH_0(T) + o(u \log \sigma)$ bits of space. The time complexities of this index are just as for the AF-FMI. We use the code provided at `http://pizzachili.dcc.uchile.cl/indexes/Succinct_Suffix_Array/SSA_v2.tgz`. We use the same parameters as for the AF-FMI.

*Navarro's LZ-index (*Nav-*LZI).* The original implementation of Navarro's LZ-index [Nav08]. We added to this index our data structures to report text positions, in order to conform the *Pizza&Chili* API. As explained before, this index does not provide space/time trade-offs.

Inverted LZ-index (ILZI). This is the implementation of the index of Lemma 3.5 by Luis Russo, which does not conform the *Pizza&Chili* API as the other indexes. In particular, pattern occurrences are reported in the format $[\![t, o]\!]$, just like for the original LZ-index. The index does not include the data structure to transform those occurrences into real text positions, as our implementations do. To be fair, we sum the space of the described data structure for text positions to the space of this index. We also sum the average time to transform occurrences into real text positions for `locate` queries. According to our experiments, this is 1.3 microseconds per occurrence. As an additional consequence, this index does not provide `extract` queries, but only `display` queries. So, we are not able to extract arbitrary text substrings.

It is important to note also that the current implementation of the ILZI does not return to the invoking application an array with the pattern occurrences, as required by the *Pizza&Chili* API. This implementation just prints the number of phrases containing the starting position of the occurrences. We get rid of the print operation in the code, and thus there is no reporting operation at all (we just find the occurrences). In particular, we are not accounting for the overhead of managing the occurrence array, which grows dynamically as more occurrences are found.

Russo and Oliveira [RO07] define a practical variant of LZ78 parsing, the so-called *LZ78 maximal parsing with quorum l*. The idea is that for every phrase $B_i = B_j \cdot c$, B_j is the longest prefix of the rest of the text that appears at least $l+1$ times in $B_0 \dots B_{i-1}$. Note that by using $l = 0$ we get the original LZ78 parsing. In this way, by using larger quorum values we can reduce the number of phrases in the LZ78 maximal parsing, hence reducing the number of nodes in the trie representing those phrases, and thus reducing the space of the index. We use quorum values $l = 0$, 1, 2, 4, 8, and 16 to get different space/time trade-offs, though this is not actually a trade-off parameter, but an optimization parameter, as we shall see in our experiments. Smaller values of l do not yield a significant reduction in the space requirement.

4.3.2 Comparison of Space Requirement and Construction Time

In Table 4.1 we show the construction time and final space requirement for the indexes we have tested. For Schemes 3 and 4, we test with $1/\epsilon = 1$, 2, 3, 4, 5, 10, and 15. Notice that before we enforced ϵ to be smaller than 1, because this makes no sense for the permutation data structures. In our implementation, however, we allow for $\epsilon = 1$, by simply storing the inverse permutation explicitly. As it can be seen, we have reduced the space of the original LZ-index, and in the cases of Scheme 3 and Scheme 4 we offer space/time trade-offs, which come from the data structures for permutations used in these schemes. A smaller sampling in these structures yield a smaller representation, yet a higher construction time, since these permutations are used to implement the *Node* data structure, which is used to construct the data structure for text positions of Section 4.2.2. Note that the maximum space requirement of both Scheme 3 and Scheme 4 is about the same as that of the original LZ-index, and that the minimum space requirement we achieve is in all cases around 2/3 the space of the original LZ-index.

In many cases, such as for XML documents and DNA data, our indexes are smaller than the original text. This is important by itself since we are able to provide indexing capabilities with a representation which is smaller than the original text.

We also conclude that our indexes are much faster to build than competing schemes. It can be argued that construction time is not so important in indexed text searching, where one constructs the index once and queries it several times, so that construction

time is amortized over a number of queries. However, as we deal with very large texts (because we would use a classical index otherwise), construction time is not irrelevant. As a comparison between LZ-based schemes, the ILZI, which is in most cases the slowest index to build, is built about 9 times slower than our schemes (e.g., in the case of English text). One reason for this is that our indexes are constructed by performing only one pass over the text, while the ILZI needs two passes [RO07]. Recall that the ILZI does not construct any text-position data structure, which would further increase its construction time.

In Table 4.2 we show the experimental size of the data structures described in Section 4.2.2 for reporting text positions in our LZ-indexes. This space is already accounted for in Table 4.1.

4.3.3 Comparison of Search Time

Next we experimentally test whether the trade-offs we provide are competitive for compressed text searching. In our experiments, we call S2 DFUDS the version of Scheme 2 implemented on DFUDS, and S3 DFUDS the DFUDS version of Scheme 3. We do not include Scheme 1 based on DFUDS, since it is outperformed by S2 DFUDS, both of them requiring about the same space. We no dot include Scheme 4 in our plots, since Scheme 3 outperforms it in most cases (though they require almost the same space). However, Scheme 4 is interesting by itself, as we shall see in Chapter 5, since we can reduce its space requirement even more, which we cannot achieve by using other schemes.

`Extract` **Queries.** Since compressed full-text self-indexes replace the text, the fast extraction of arbitrary text substrings is essential in most applications, and thus one of the most relevant problems the indexes face. To test the performance of our indexes, we extract 10,000 random snippets, each of length 100. We measure the time per symbol extracted, so we average over a total of 1 million extracted symbols.

In Fig. 4.5 we show the experimental results. As we can see, our indexes are very competitive in the range of space they require, in most cases outperforming competing schemes, which in some cases cannot compete even if using more memory than ours. In the particular case of DNA text, our indexes are competitive against the SSA, which in the case of small alphabets is a very good alternative. This is because the corresponding *wavelet tree* [GGV03], on which the SSA is based, is shallow, and then each text symbol can be extracted very fast (every symbol is extracted in time proportional to the height of the corresponding wavelet tree).

This shows the superiority of our LZ-indexes in this important aspect. Also, we can see that our approach to reduce the space of the LZ-index is effective in this case, since we are able to reduce the space while still maintaining a good extracting performance.

Table 4.1: Range of space requirement and construction time for the indexes we have tested. The space is shown as a fraction of the text size. Indexing time is shown in MB per second.

Text	Index	Range of space req. (fract. of text size)	Indexing speed (MB per second)
English	CSA	0.43 – 1.50	0.45 – 0.44
	SSA	0.87 – 2.87	0.93 – 0.90
	AF-FMI	0.65 – 2.65	0.26
	ILZI	1.23	0.15
	Original LZ-index	1.69	1.47
	Scheme 1	1.39	1.30
	Scheme 2	1.38	1.27
	Scheme 3	1.13 – 1.69	0.91 – 1.33
	Scheme 4	1.13 – 1.69	0.71 – 1.31
DNA	CSA	0.46 – 1.53	0.51
	SSA	0.50 – 2.50	1.33 – 1.28
	AF-FMI	0.48 – 2.48	0.43
	ILZI	0.95	0.66
	Original LZ-index	1.24	2.35
	Scheme 1	1.03	2.02
	Scheme 2	1.01	1.99
	Scheme 3	0.83 – 1.24	1.36 – 2.12
	Scheme 4	0.83 – 1.24	1.03 – 2.06
MIDI Pitches	CSA	0.62 – 1.68	0.94 – 0.92
	SSA	1.04 – 3.04	1.80 – 1.71
	AF-FMI	0.93 – 2.94	0.36
	ILZI	1.86	0.24
	Original LZ-index	2.58	1.76
	Scheme 1	2.16	1.49
	Scheme 2	2.12	1.47
	Scheme 3	1.76 – 2.58	0.88 – 1.58
	Scheme 4	1.76 – 2.58	0.62 – 1.56
XML	CSA	0.29 – 1.35	0.68
	SSA	0.98 – 2.98	1.34 – 1.29
	AF-FMI	0.54 – 2.54	0.44 – 0.43
	ILZI	0.61	0.34
	Original LZ-index	0.93	2.38
	Scheme 1	0.77	2.15
	Scheme 2	0.76	2.15
	Scheme 3	0.63 – 0.93	1.57 – 2.23
	Scheme 4	0.63 – 0.93	1.24 – 2.17
Proteins	CSA	0.67 – 1.73	0.57 – 0.56
	SSA	0.82 – 2.82	0.97 – 0.95
	AF-FMI	0.82 – 2.82	0.38 – 0.37
	ILZI	1.73	0.24
	Original LZ-index	2.40	1.55
	Scheme 1	1.99	1.29
	Scheme 2	1.96	1.25
	Scheme 3	1.62 – 2.40	0.79 – 1.35
	Scheme 4	1.62 – 2.40	0.58 – 1.30
Sources	CSA	0.38 – 1.44	0.76 – 0.75
	SSA	1.01 – 3.01	1.37 – 1.32
	AF-FMI	0.73 – 2.73	0.30
	ILZI	1.15	0.17
	Original LZ-index	1.67	1.73
	Scheme 1	1.39	1.54
	Scheme 2	1.37	1.51
	Scheme 3	1.13 – 1.67	1.04 – 1.59
	Scheme 4	1.13 – 1.67	0.79 – 1.54

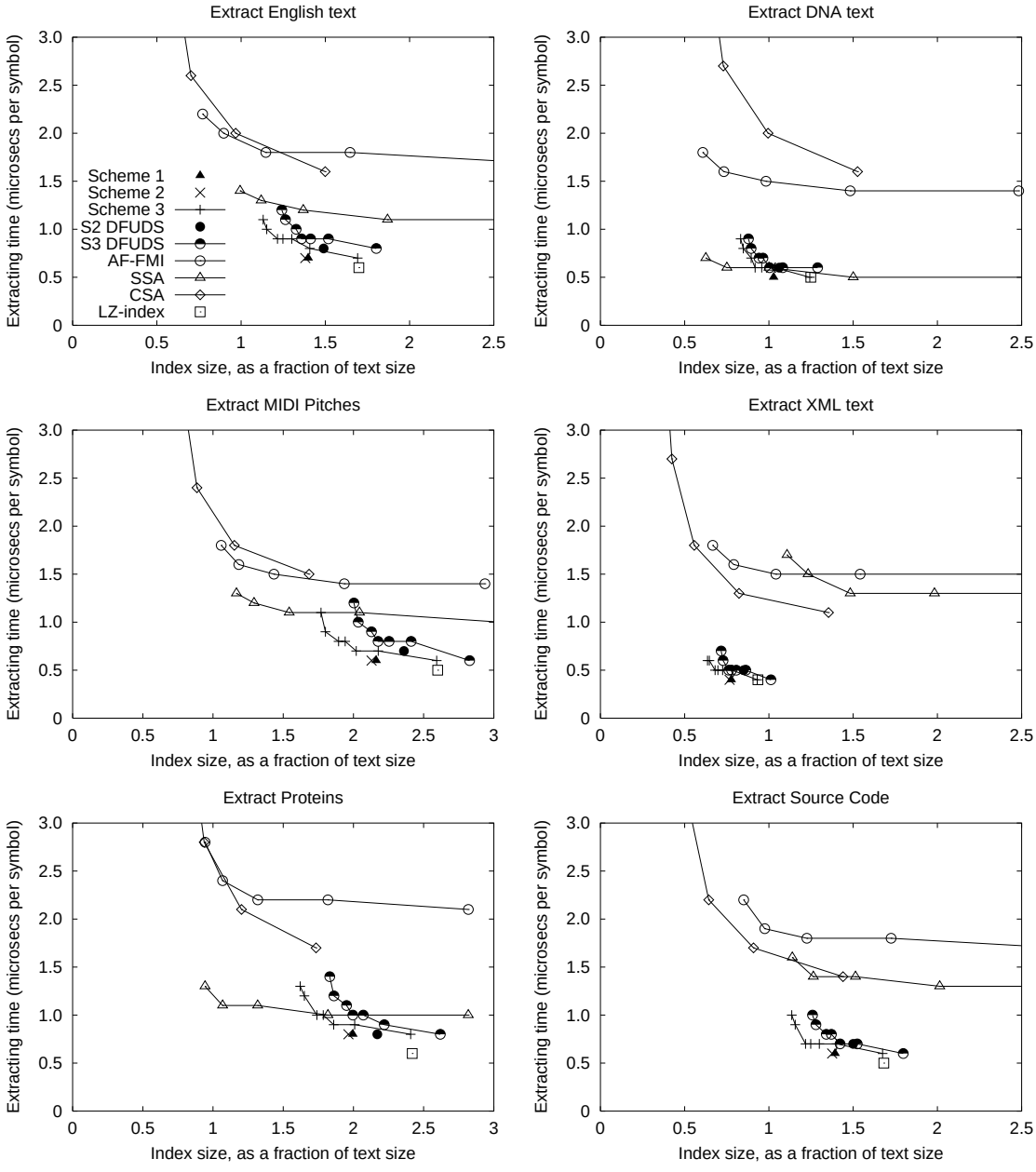


Figure 4.5: Experimental extracting time, for random snippets of length $\ell = 100$. Times are measured in microseconds per symbol extracted.

Table 4.2: Size of the data structure for reporting text positions in our LZ-indexes.

Text	Size of text-position data structure (as a fraction of text size)
English	0.14
DNA	0.11
MIDI Pitches	0.27
XML	0.09
Proteins	0.22
Sources	0.16

`Display` **Queries.** In many applications displaying a context surrounding the occurrences is as important as (and sometimes more important than) the occurrence positions themselves. This is not a problem for classical indexes, since the text is available and hence we can get the contexts from the occurrence positions obtained with `locate`. In the case of self-indexes, on the other hand, one must ask the index to reproduce the occurrence contexts, which is achieved with `display` queries since we do not have the text at hand. Thus, fast displaying the occurrence contexts is also important for a self-index.

To test our indexes, we search for 5 million pattern occurrences and then show a context of 50 symbols surrounding every occurrence, for patterns of length 10 (in other words, we display 110 symbols per occurrence).

In most indexes, `display`(P, ℓ) queries can be thought of as a `locate`(P) query (in order to find the pattern occurrences) followed by an `extract`(i, j) query (where i and j are computed by means of the positions obtained with `locate` and the context length ℓ). In our LZ-indexes, however, we originally get the occurrences in the format $[\![t, o]\!]$, to finally transform t and o into a text position (by means of the data structure described in Section 4.2.2). This text position must be transformed by `extract` again into an LZ78 phrase (recall that this involves binary searching the text-position data structures), from where we start the extraction of text in *LZTrie*. To avoid repeating this work, we do not transform the occurrences into text positions when performing `display` queries. We rather display text with a simplified version of `extract` that works on LZ78 phrases rather than on text positions.

In Fig. 4.6 we show the experimental results. The current implementation of the ILZI shows only a context preceding the pattern occurrences, and not surrounding the occurrences as other schemes do. For our LZ-indexes, showing a context surrounding the occurrences (which is usually required) introduces the use of extra operations which are not needed when showing a context preceding an occurrence. As it can be seen, just like for `extract` queries, our indexes are among the most competitive schemes for displaying

text, in many cases outperforming the competing ILZI.

`Locate` **Queries.** For `locate` queries we search for 10,000 patterns extracted at random positions from the text, with length 5, 10, and 15. For short patterns, we limit the total number of occurrences found to 5 million. We measure the time in microseconds per occurrence found.

As we said before, `locate` queries are important in classical full-text indexing, where one has the text at hand in order to access the occurrences and the contexts surrounding them. This is somehow equivalent to `display` queries in the scenario of compressed full-text self-indexes. Obtaining just text positions (and nothing else) can be interesting in specific cases, but `display` queries have broader applications.

Partial `locate` *Queries.* In many applications we do not need to locate all of the pattern occurrences at once, but just a few of them. This is a very challenging problem for our LZ-indexes since, for example, the indexes based on suffix arrays are very efficient to find the suffix-array interval containing the occurrences, and hence they are rapidly ready to start locating the occurrences. The ILZI has also a very fast $O(m)$-time trie navigation before starting the locating procedure.

We test here our algorithm defined in Section 4.2.3 (recall that we divide the search process into four levels, level 0 up to 3). In Table 4.3 we show the percentage of occurrences that are found in each level of search, for $K = 1$ and for the different text collections. Notice that the search of level 0 (i.e., searching for P as a whole phrase in *LZTrie*) is very effective for patterns of length 5 to 10 (in the case of DNA, for example, almost 100% of the queries can be answered at level 0). Notice also that the percentage found at level 1 is relatively small compared to the corresponding percentage of level 0. Recall that with the search of level 0 we look for a particular case of occurrences of type 1. This leads us to conclude that most of the times a pattern exists as an occurrence of type 1, this can be found at level 0 of the search. For longer patterns, $m = 15$, there is a smaller probability of finding the occurrence contained in a single phrase, and thus the percentage found at level 0 is smaller.

In Figs. 4.7 up to 4.10 we show the experimental result for values $K = 1$ and 5, and for $m = 5$ and 10. We do not show the results for $m = 15$ since the results are poorer, as predicted by the results of Table 4.3.

For $m = 5$ and $K = 1$, the most interesting results are obtained in the cases of English text, DNA, XML, and proteins, though in the latter case our indexes do not obtain good compression. For DNA, S3 DFUDS is unbeatable since, as we have seen in Table 4.3, 100% of the occurrences are found at level 0 of the search. Notice that for MIDI pitches our performance is not so competitive (except perhaps for S2 DFUDS), as it was also predicted

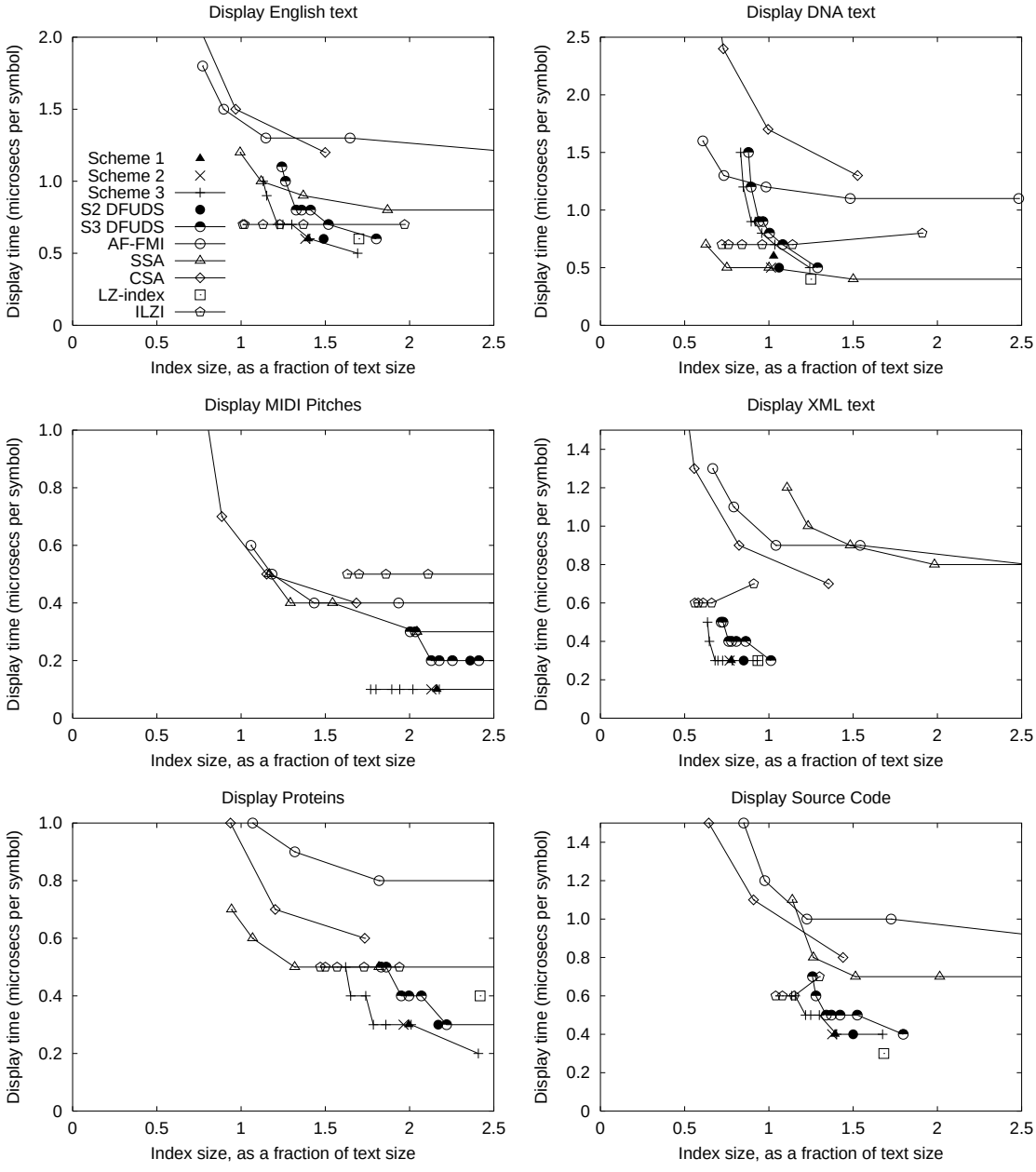


Figure 4.6: Experimental display time, for snippets of length $\ell = 110$ around every pattern occurrence. Times are measured in microseconds per symbol extracted.

Table 4.3: Percentage of patterns found at each level of search, for partial `locate` queries with $K = 1$ and for different pattern lengths. Level 0: P is found as an LZ78 phrase in $LZTrie$; Level 1: P is found as occurrence of type 1, but not as a whole phrase; Level 2: P is found as occurrence of type 2; Level 3: P is found as occurrence of type 3.

Text	Level	Percentage per level		
		$m = 5$	$m = 10$	$m = 15$
English	0	98.64	55.86	7.80
	1	0.54	6.27	1.83
	2	0.82	34.97	63.14
	3	0.0	2.90	27.23
DNA	0	100.0	99.26	11.63
	1	0.0	0.33	1.06
	2	0.0	0.41	78.74
	3	0.0	0.0	8.57
MIDI Pitches	0	72.28	20.01	9.69
	1	3.69	2.10	1.38
	2	23.82	42.66	19.67
	3	0.21	35.23	69.26
XML	0	95.41	65.61	37.56
	1	2.59	11.53	15.53
	2	1.99	19.55	32.13
	3	0.01	3.31	14.78
Proteins	0	98.96	13.29	8.51
	1	0.71	1.41	1.29
	2	0.33	63.87	11.23
	3	0.0	21.43	78.97
Sources	0	94.31	48.40	21.41
	1	2.73	9.34	6.01
	2	2.94	37.15	44.11
	3	0.02	5.11	28.47

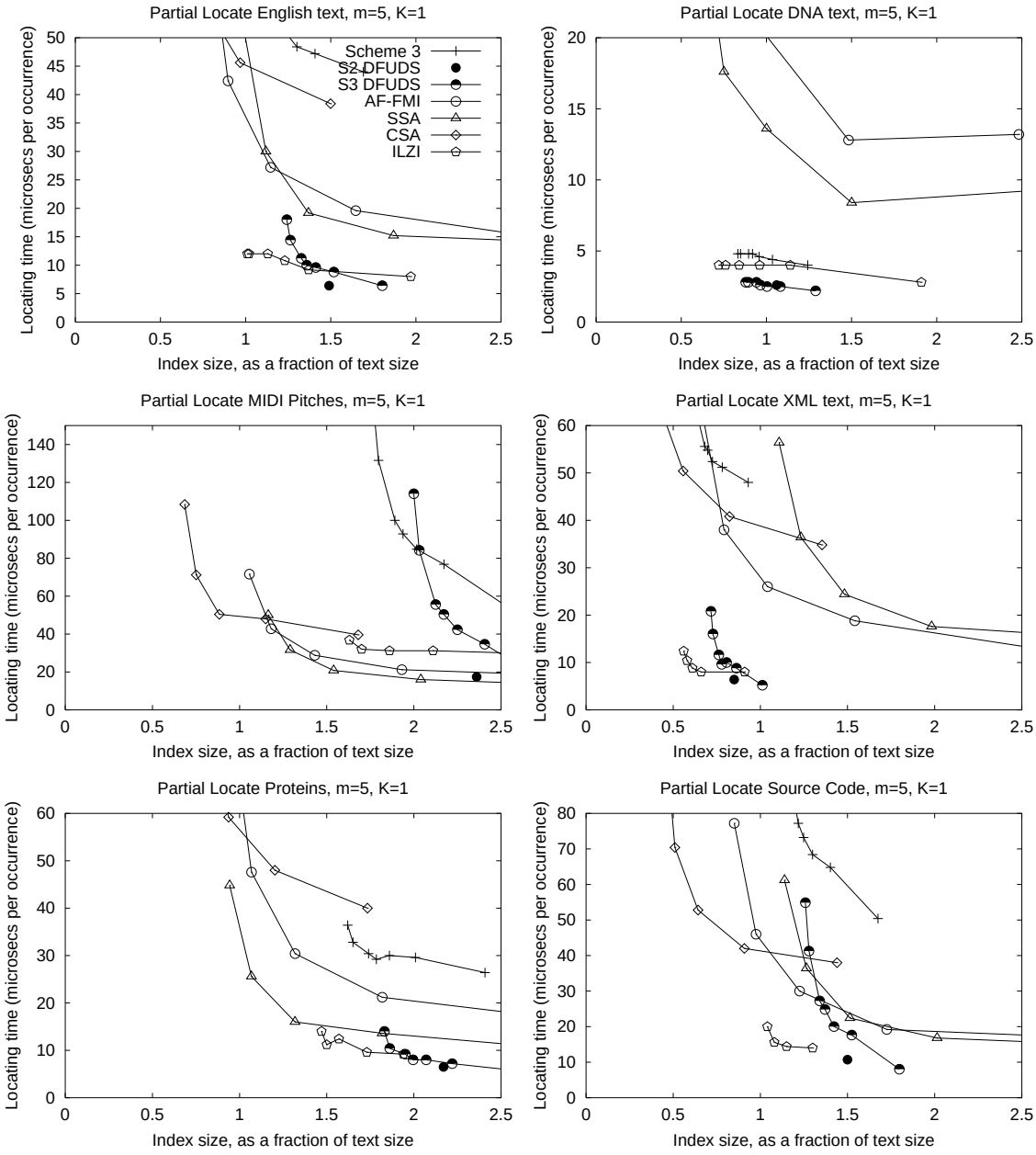


Figure 4.7: Experimental time for partial `locate` queries, for patterns of length $m = 5$ and retrieving just $K = 1$ occurrence. We measure the time in microseconds per occurrence reported.

by Table 4.3: only 73% of the occurrences are found at level 0, and thus we need more trie navigations.

Except for MIDI pitches, the best alternatives are the indexes based on Lempel-Ziv compression: ours and the ILZI (the best in each case depends on the space usage, as it can be seen). This shows that Lempel-Ziv-based indexing is a very competitive choice in general, despite that suffix-array-based indexes basically compute the suffix-array interval containing the occurrences, and then we ask the index to obtain just one of these occurrences. As we shall see later with `count` queries, these indexes are very efficient to find the suffix-array interval; however, asking them to obtain just one occurrence makes them significantly less competitive.

Notice also that our indexes based on DFUDS outperform (in most cases by far) our indexes based on BP. This is because the fast navigation on the tries becomes a fundamental aspect, since we are reporting just a few occurrences. For this reason, and in order to make our plots clearer, we do not show the results for the BP-based Scheme 1, Scheme 2, and original LZ-index.

As we increase the number of occurrences to locate, in principle the trie navigations are amortized by reporting more occurrences. However, our indexes may need to go on more levels of the search, which means more navigations on the tries. Therefore, the total cost depends on the number of occurrences found in each search level. In general, as K grows, it becomes more difficult to compete since occurrences at higher levels are more expensive to obtain. Yet, we still provide some interesting cases for $K = 5$ and $K = 10$ (the latter case is not shown; results are close to those for $K = 5$).

The trie traversals also raise when we increase the pattern length, as it can be seen for $m = 10$.

Thus, we conclude that our technique for solving partial `locate` queries of Section 4.2.3 is relevant, and more efficient when the probability of finding the occurrences at level 0 of the search is high, as for example when we search for short patterns, the alphabet is small, or we look for very few occurrences (e.g., $K = 1$).

Full `locate` *Queries.* We test here `locate` queries without limiting the number of occurrences. In Fig. 4.11 we show the experimental results for patterns of length $m = 5$. As we can see, there is no clear winner in all cases, but the performance depends on the available space. However, we can see some clear performance patterns: in most cases our schemes outperform all competing schemes (including the very competitive ILZI) as soon as we have space available to store, at least, Scheme 2. When we reduce the space usage of the index, however, the ILZI outperforms Scheme 3, yet the latter is still competitive (in most cases outperforming the competitive CSA). In general, for `locate` queries with short patterns the superiority of Lempel-Ziv-based indexes is clear.

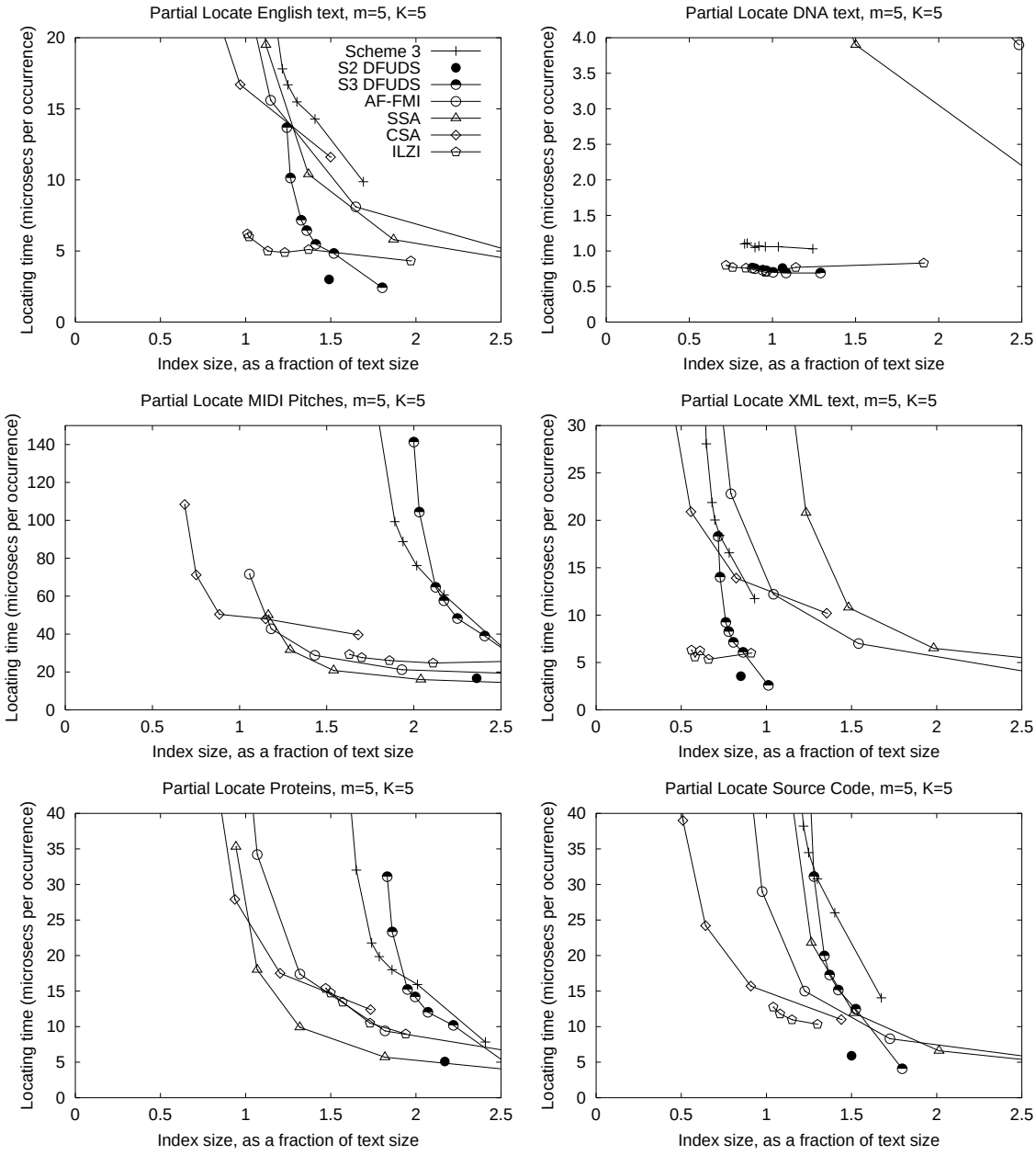


Figure 4.8: Experimental time for partial `locate` queries, for patterns of length $m = 5$ and retrieving just $K = 5$ occurrences. We measure the time in microseconds per occurrence reported.

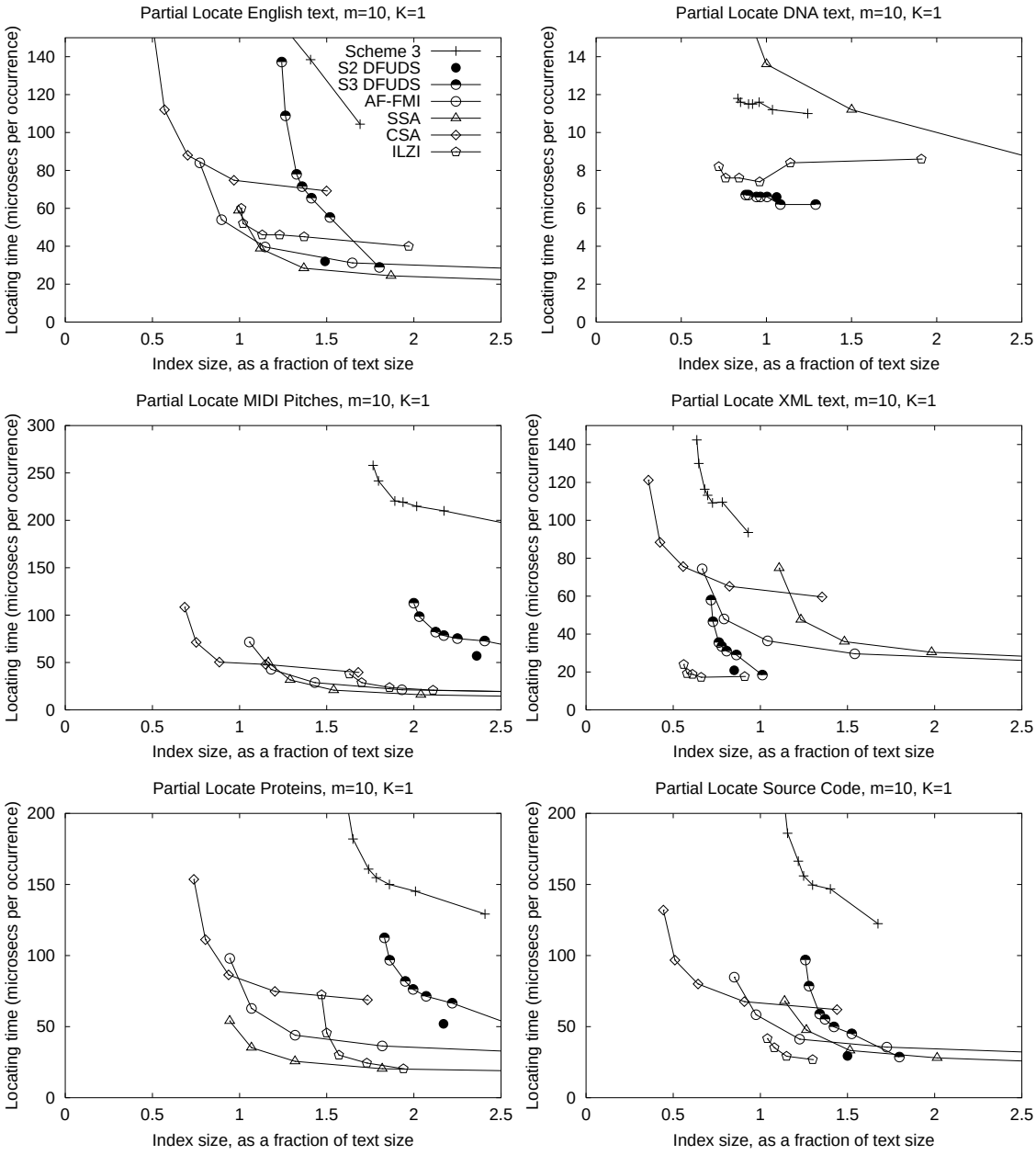


Figure 4.9: Experimental time for partial `locate` queries, for patterns of length $m = 10$ and retrieving just $K = 1$ occurrence. We measure the time in microseconds per occurrence reported.

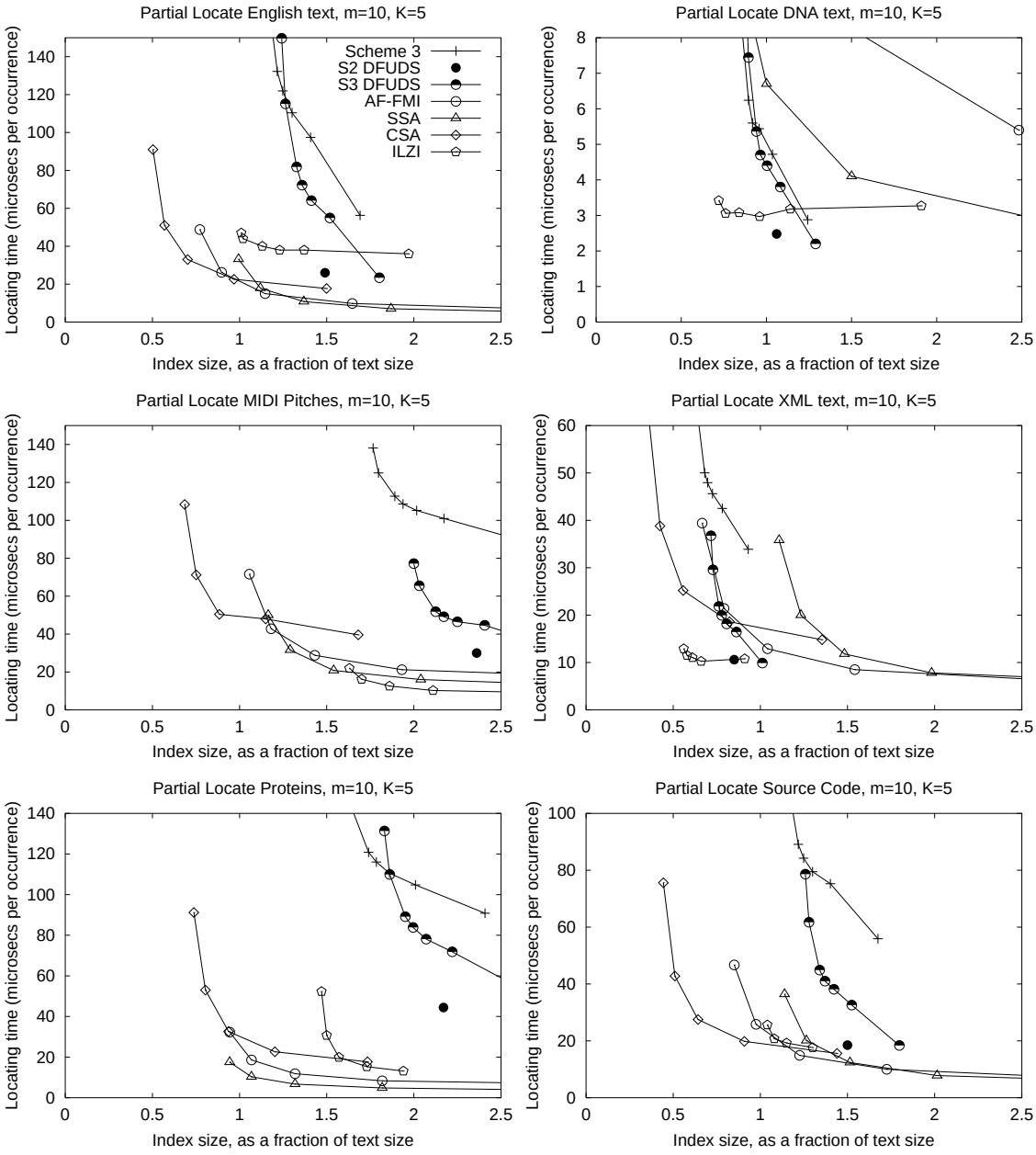


Figure 4.10: Experimental time for partial `locate` queries, for patterns of length $m = 10$ and retrieving just $K = 5$ occurrences. We measure the time in microseconds per occurrence reported.

Notice also that, in most cases, Scheme 2 outperforms Scheme 1, both requiring about the same space. This means that the shorter average path length of Scheme 2 is better than having direct access from *RevTrie* to *LZTrie* nodes as in Scheme 1, which is good only for occurrences of type 1. As we said before, occurrences of type 2 are more costly in Scheme 1. It is also important to note that the DFUDS versions of LZ-index have a performance which is similar to the LZ-index based on BP. Notice, however, that DFUDS requires more space than BP, as it was predicted in Section 4.2.1.

In the particular case of XML text, we can see a very important aspect that differentiate LZ-indexes from indexes based on suffix arrays. The latter need to store extra non-compressible information (the sampled suffix array positions) to efficiently carry out `locate` and `extract` queries. The extra data stored by LZ-indexes, on the other hand, is compressible, for example the size of the arrays for which we sample the inverse-permutation information depends on n, the number of LZ78 phrases of T. Therefore, when the texts are highly compressible, the LZ-indexes can be smaller and faster than alternative indexes.

For proteins, as an opposite case, our LZ-indexes are larger and slower. They are large since the text is not as compressible as others, which can be also noted in the size of competing schemes. Our LZ-indexes are in addition slower in this case, since each search retrieves on average only a few patterns, and therefore the work on the tries is not amortized by reporting many occurrences.

We show the results for patterns of length $m = 10$ in Fig. 4.12. As we can see, our indexes are still competitive, yet not as significantly as in the previous case where $m = 5$. The DFUDS implementation of LZ-index outperforms BP in all cases, except for MIDI pitches, XML text, and source code, where these have about the same performance. In particular for proteins and English text, where the number of occurrences per pattern is relatively small, the difference is greater for DFUDS, since the cost of navigating the tries becomes predominant. It is important to note also that Scheme 2 (both for BP and DFUDS implementations) is interesting (in some cases the best) alternative, for the memory space it requires. For patterns of length $m = 15$ (figure omitted), the behavior of the indexes is similar to that of length $m = 10$.

Our results indicate that our LZ-indexes, despite not being the best in many cases, are an attractive alternative for `locate` queries. In most cases we need at least the space required by Scheme 2 in order to outperform competing schemes. In particular, our LZ-indexes are an interesting alternative for full `locate` when the total number of occurrences to report is considerable. For long patterns, the number of pattern occurrences becomes smaller, and therefore the time of searching for the $O(m^2)$ pattern substrings in the tries dominates. This problem can be somehow alleviated by using the DFUDS implementation for the tries. We can conclude that the LZ-indexes based on BP and on DFUDS behave very

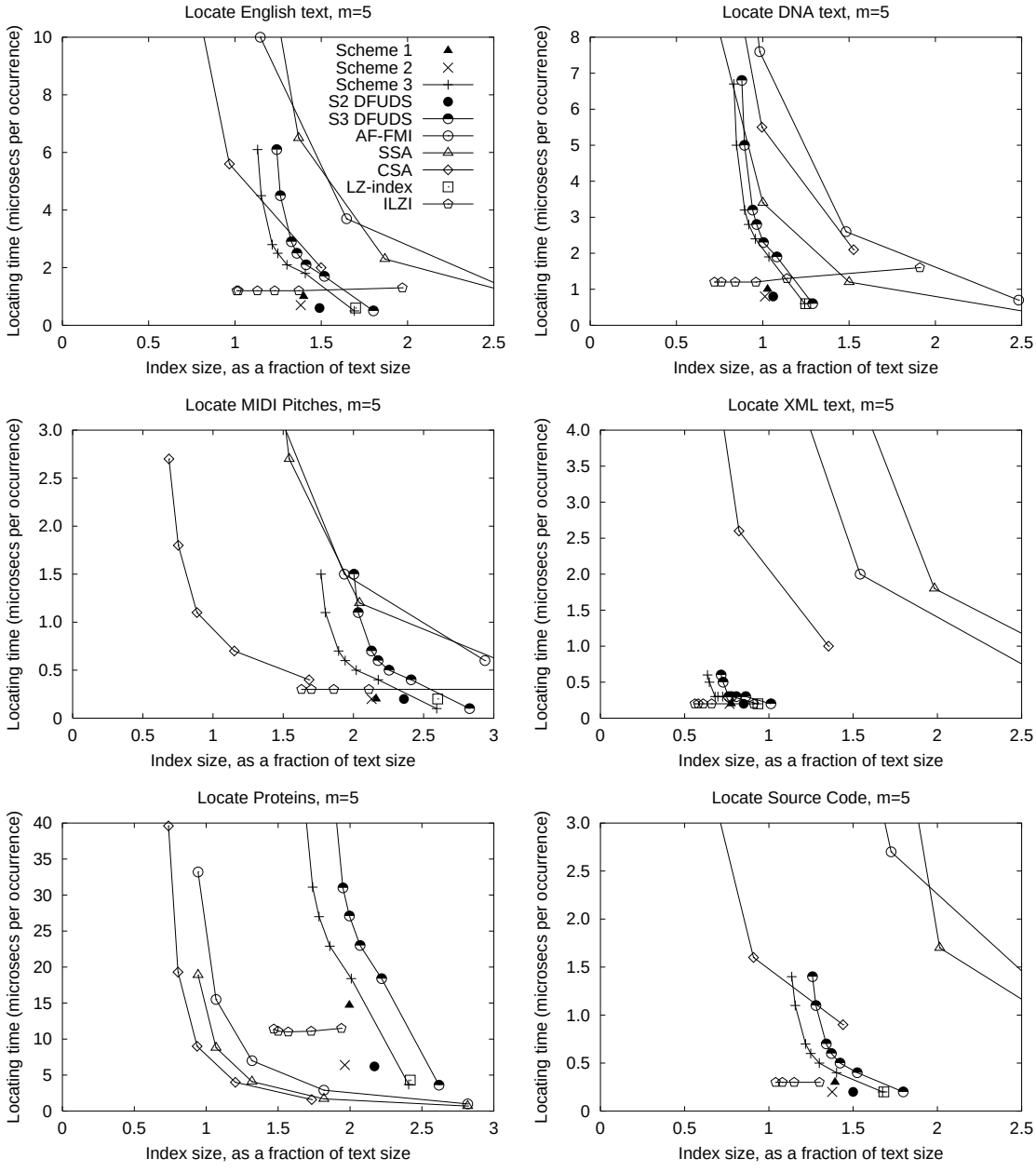


Figure 4.11: Experimental locating time, for patterns of length $m = 5$. We measure the time in microseconds per occurrence reported.

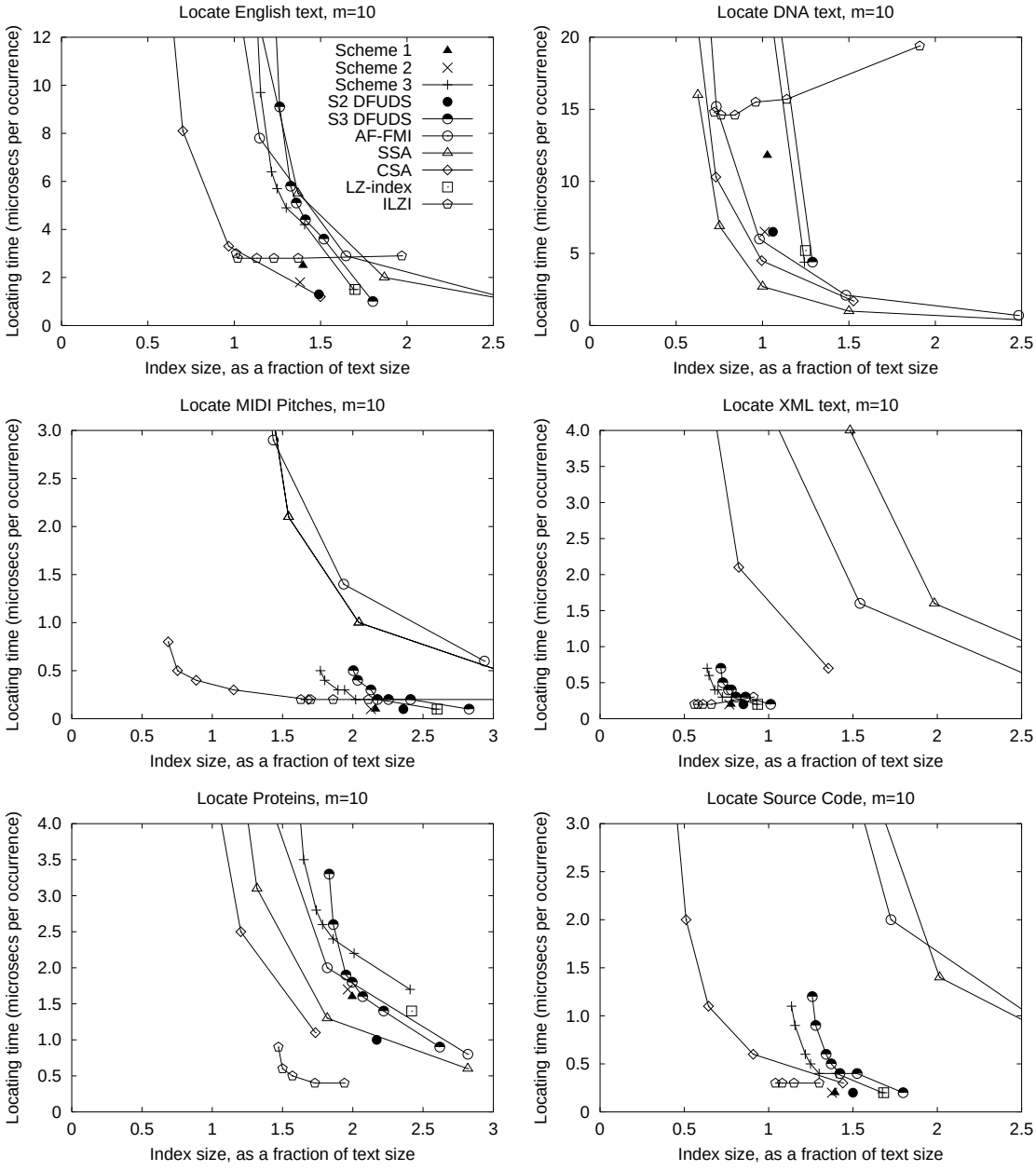


Figure 4.12: Experimental locating time, for patterns of length $m = 10$. We measure the time in microseconds per occurrence reported.

similarly when reporting a great number of occurrences fast is an important issue (e.g., when $m = 5$). Yet DFUDS requires more space. On the other hand, when the navigation in the tries becomes more important (e.g., for larger m), the DFUDS implementation offers a more attractive alternative, which can replace the BP implementation at the cost of a little extra space requirement.

`Count` Queries. This kind of queries has applications in much more specific and limited cases, generally composing the internal machinery of more complex tasks.

In our experiments we search for patterns of length 20 extracted at random positions from the text. We measure the times per symbol of the pattern. As competing schemes based on suffix arrays do not store suffix-array sampling information in order to count (and thus are able to support efficiently only `count` queries), to be fair in this case we do not store the data structure for text positions in our LZ-indexes. Yet, note that, within this space, we are still able to support fast `locate` queries (though without reporting text positions), and `display` queries.

The experimental results for `count` queries are shown in Fig. 4.13. As we can see, our schemes can implement this query, yet they cannot compete against the indexes based on suffix arrays, since the counting complexity of these indexes is related to the pattern length, and not to the number of pattern occurrences. Our schemes basically need to locate the pattern occurrences in order to count them. We can also see that the DFUDS implementation of LZ-index outperforms in all cases the BP implementation, yet the former requires slightly more space. This is mainly because we are searching for long patterns, and DFUDS provides more efficient descent in the *LZTrie*.

`Exists` Queries. In some specialized applications we just need to know whether pattern P exists or not in the text. Indices based on suffix arrays basically need to count the number of occurrences, since they first search for the pattern, and then check whether the suffix-array interval they get is empty or not. Despite that our LZ-indexes are not competitive for `count` queries, we show here that they are much more efficient for finding the first occurrence of a pattern, which is useful to support `exists` queries (indeed, this has been already shown in the experiment with partial `locate` queries). The key is that we do not necessarily need to count the occurrences, but just to find the first pattern occurrence as fast as we can. We test with patterns of length 5, 10, and 15, and search for 10,000 patterns that exist in the text. We implement existential queries in our indexes using the idea of partial `locate` queries, explained in Section 4.2.3.

In Fig. 4.14 we show the experimental results for `exists` queries, for patterns of length 5. Excluded plots for patterns of length $m = 10$ produced similar results, while those for patterns of length $m = 15$ gave worse results (this is because, as predicted in

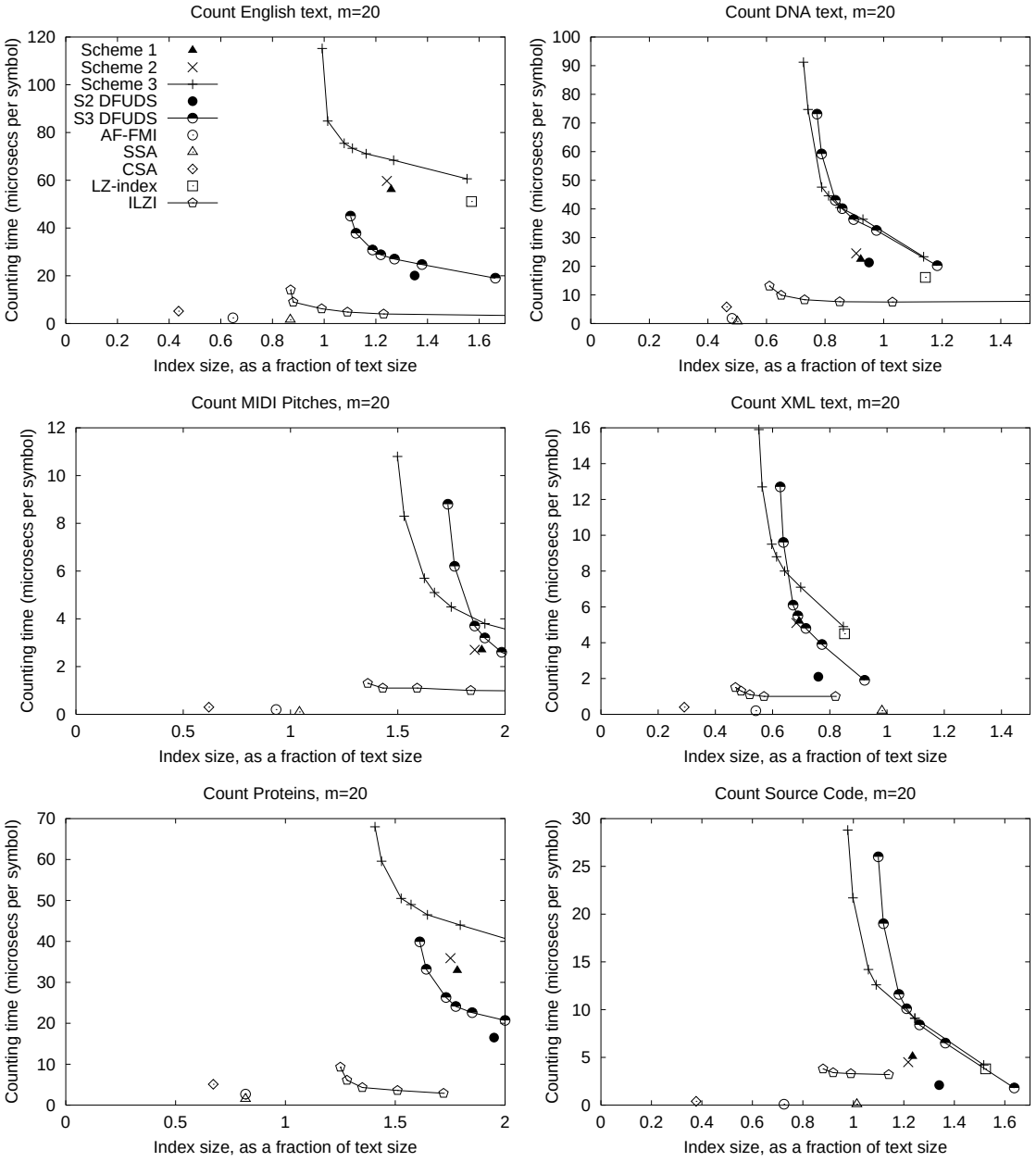


Figure 4.13: Experimental counting time, for patterns of length $m = 20$. We measure the time in microseconds per symbol of the pattern.

Table 4.3, the heuristic of level 0 is not so effective for longer patterns). As it can be seen, our indexes are much more competitive than for `count` queries, showing that our approach of first looking for P in *LZTrie* is effective in practice. Our indexes achieve the same times (though not the same space) in many cases. As in the case of `count` queries, our indexes are larger than competing schemes, yet ours are able to support more than just `count` and `exists` queries within this space.

4.4 Final Comments

In this chapter we have reduced the space requirement of LZ-indexes, in particular, Navarro's LZ-index. Given a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$ [Man01], the original LZ-index of Navarro [Nav04] requires $4uH_k(T) + o(u \log \sigma)$ bits of space. In this chapter we define several new versions of the LZ-index, requiring $3uH_k(T) + o(u \log \sigma)$ and $(2+\epsilon)uH_k(T) + o(u \log \sigma)$ bits of space. The latter ones allow us to partially overcome one of the drawbacks of the original LZ-index: the lack of space/time tuning parameters. Although our schemes do not provide worst-case guarantees at search time, they ensure $O(\frac{m^2}{\epsilon} + \frac{n}{\epsilon\sigma^{m/2}})$ average-case time for locating the occurrences of pattern $P[1..m]$ in T. This is $O(\frac{m^2}{\epsilon})$ for $m \geqslant 2\log_\sigma u$. In Chapter 5 we shall add worst-case guarantees to some of our reduced schemes.

Our experimental results show that the space of the index can be dropped up to 2/3 the size of the original LZ-index. When comparing the search performance, we conclude that our indexes are able to reduce the space of the original LZ-index while retaining much of the good features of the original LZ-index. In particular, we can conclude that:

— Our indexes are very competitive (in most cases the best) alternatives for `extract` and `display` queries, which we argue are the most basic queries in the scenario of compressed full-text self-indexes, where the text is not available otherwise[1]. In some cases we are able to extract about 1 to 1.5 million symbols per second, being about twice as fast as the most competitive alternatives.

— For *partial* `locate` queries (where a fixed number of occurrences is located), our algorithm is efficient in cases of searching for short patterns, of texts with small alphabets, or when we want to locate very few occurrences (e.g., only one occurrence).

— For *full* `locate` queries, we showed that our indexes are more effective when the search pattern is not too long, or there are many occurrences to report. In other cases, the $O(m^2)$-time navigation on the tries is predominant, and thus our performance degrades. For example, for short patterns of length 5, in most scenarios our schemes

[1] In the literature [NM07] the `count` operation is taken as the most basic one, but this is probably biased towards suffix-array-based indexes, more than to considering typical applications.

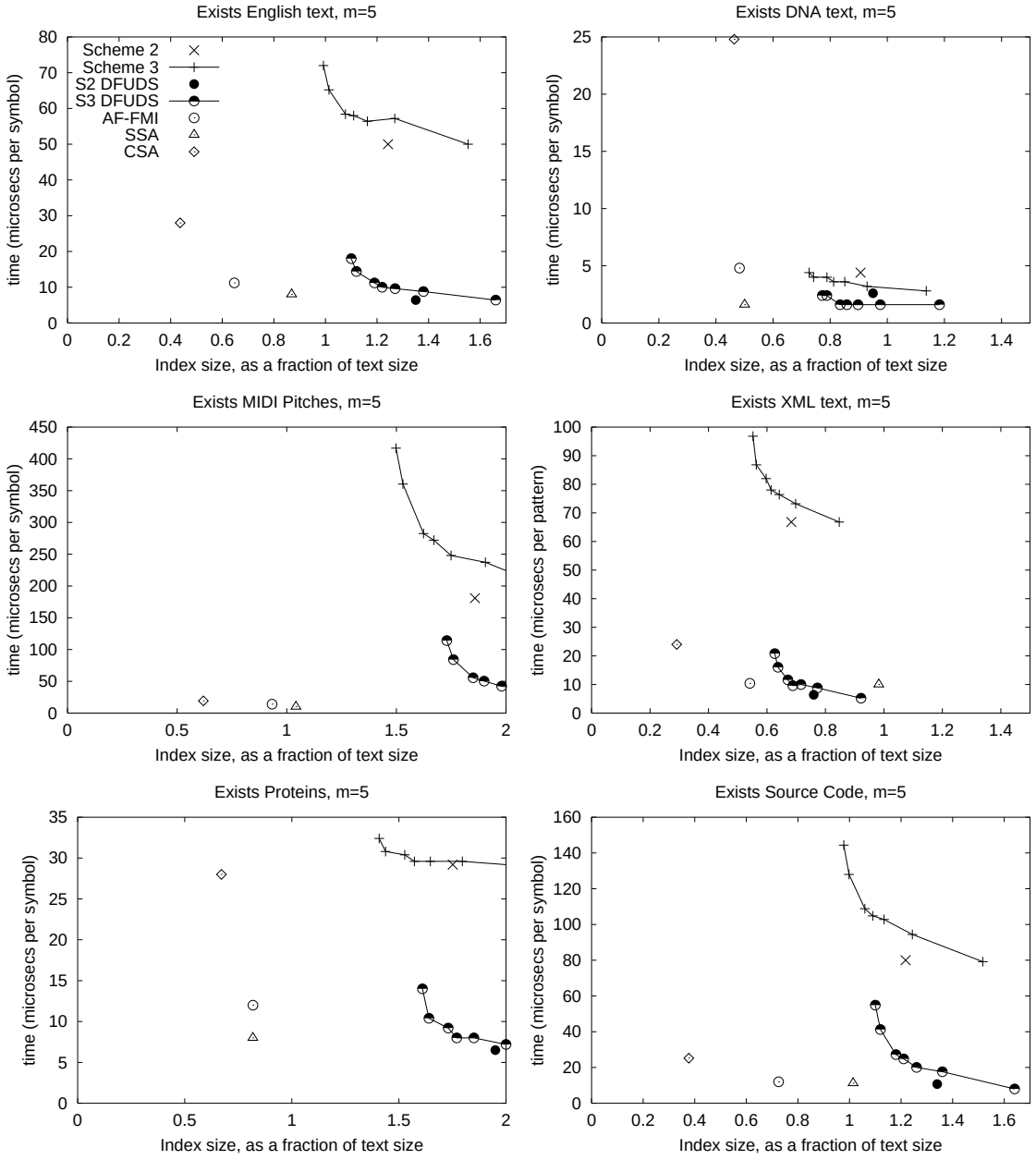


Figure 4.14: Experimental time for `exists` queries, for patterns of length $m = 5$. We measure the time in microseconds per pattern.

are the best alternative if we can spend at least 80% the size of the original LZ-index. When less memory space is available, our indexes are outperformed by the very competitive Inverted LZ-index (ILZI) [RO07], which is yet another variant of the Lempel-Ziv-based index family. In this case, however, our indexes are still competitive with all suffix-array-based schemes (e.g., the competitive Compressed Suffix Array of Sadakane [Sad03]).

It is well known that the original LZ-index is more adequate for short patterns [Nav08]. We somehow alleviated the problem of long patterns, by using a more efficient representation, the so-called DFUDS representation of Lemma 2.11, for the tries that compose the LZ-index. From our experiments we can conclude that our DFUDS implementation allows us for very efficient search, in many cases outperforming the traditional balanced-parentheses representation (Lemma 2.10) of the LZ-index. However, more work is needed to compete in this respect against suffix-array-based indexes. We hope to get further improvements on this line when working on alphabets larger than those tried in this chapter, for example, if defining an LZ-index working on words, for applications of natural-language processing.

We also exhibit an important difference between LZ-indexes and those based on suffix arrays: the latter need to store extra non-compressible information (the suffix-array samples) in order to carry out `extract`, `display`, and `locate` queries, whereas the information stored by our LZ-indexes is compressible. Thus, when the text is highly compressible, we get very small LZ-indexes, which are still fast. Indices based on suffix arrays, on the other hand, cannot use a denser suffix-array sampling (because otherwise they become larger), and henceforth their performance is poor. Thus, our indexes are also a very good option for highly compressible texts.

Overall, we believe that our indexes offer an extremely relevant alternative considering their overall performance across the multiple tasks of interest in many real text-search applications. We made the code of our indexes available in the *Pizza&Chili* corpus, at the site `http://pizzachili.dcc.uchile.cl/indexes/LZ-index/`.

We do not consider in this chapter the space-efficient construction of our indexes, which is a very important issue in practice, since many times a small index requires a large amount of memory space to be build. Currently, our indexes are constructed in an uncompressed way, needing about the same space used to build suffix-array-based compressed self-indexes. The space-efficient construction of the latter is still at a theoretical stage [MN08b], or still does not achieve higher-order entropy-bound space [HLS$^+$07]. In Chapter 6 we will show how to space-efficiently construct our LZ-index schemes.

Chapter 5

Stronger Lempel-Ziv Text Indexes

In Chapter 4 we have shown that the space of the NAV-LZI (or simply the LZ-index) can be reduced (in practice to about 2/3 the space of the original LZ-index) while retaining much of the good features of the original LZ-index. However, that is a practical approach since we cannot provide worst-case guarantees at search time; moreover, we are not able to reduce even more the space requirement of LZ-index by those means.

As we have seen in Chapter 4, for the LZ-index we can achieve $(2+\epsilon)uH_k(T) + o(u\log\sigma)$ bits of space and $O(\frac{m^2}{\epsilon})$ average search time for patterns of length $m \geqslant 2\log_\sigma u$. Hence, two questions may arise:

Question 5.1. Can we reduce the space requirement of LZ-index to $(1+\epsilon)uH_k(T) + o(u\log\sigma)$ bits?, that is, to almost optimal space (in terms of H_k).

Question 5.2. Can we retain worst-case guarantees at search time (as for the original LZ-index), yet requiring $(2+\epsilon)uH_k(T) + o(u\log\sigma)$ bits of storage (as for the schemes of Fig. 4.3)?

In this chapter we will find affirmative answers to these questions. We shall study how to reduce by about a half the space requirement of the original LZ-index, while at the same time improving its time complexities. The result are attractive, much stronger, alternatives to current state of the art in compressed self-indexing.

In a first approach (Section 5.1), we compress one of the data structures composing the original LZ-index by using an idea which is, in some sense, related to the compression of suffix arrays [GV05, Sad03]. Theorem 5.1 shall answer Question 5.1 and Theorem 5.2 shall answer Question 5.2. In a second approach (Section 5.2) we combine the balanced parentheses representation of Munro and Raman [MR01] (see Lemma 2.10) of the LZ78 trie

This chapter is based on joint work [ANS06, ANS08] with Kunihiko Sadakane, from Kyushu University, Japan.

with the *xbw transform* of Ferragina et al. [FLMM05] (see Lemma 2.12), whose powerful operations are useful for the LZ-index search algorithm. In Section 5.3 we will show how to achieve the same results as for the ILZI [RO07] (see Lemma 3.5), yet with a smaller index. Finally, in Section 5.4 we show how to achieve optimal extracting time with all of our indexes.

5.1 Suffix Links in *RevTrie*

In this section we will build on the reduced Scheme 4 described in Section 4.1.1 and illustrated in Fig. 4.3(b) (on page 72). We assume that the tries composing the LZ-index are represented with DFUDS (see Lemma 2.11). Given an *LZTrie* node with preorder position i, let us define

$$parent_{lz}(i) \equiv preorder(parent(selectnode(i)))$$

That is, $parent_{lz}$ is the *parent* operation working on preorders rather than on the corresponding nodes. Let $child_{lz}(i, \alpha)$ be defined similarly as

$$child_{lz}(i, \alpha) \equiv preorder(child(selectnode(i), \alpha))$$

Also, let us define

$$letts_{lz}(i) \equiv letts[rank_{\text{'}}(par, parent(selectnode(i))) - 1 + childrank(selectnode(i)) - 1]$$

which yields the symbol by which the node with preorder i descends from its parent (recall from Section 2.5.2 how *letts* is represented in the case of DFUDS). We denote $str_{lz}(i)$ the string represented by node with preorder i in *LZTrie*. In the same way we define $str_r(j)$ for node with preorder j in *RevTrie*.

The idea is that we are going to compress the R mapping defined for Scheme 4 of LZ-index. Let us see this array as a kind of *suffix array* which, instead of storing text positions, stores *LZTrie* preorder positions (see Section 2.3.2). Array R is a lexicographically sorted array of the reversed LZ78 phases (because it is sorted according to *RevTrie* preorders). Given a reversed phrase with preorder i in *RevTrie* (and preorder $R[i]$ in *LZTrie*), its longest proper suffix has position $parent_{lz}(R[i])$ in *LZTrie* (as this corresponds to the longest proper prefix in *LZTrie*). Compare it with Property 2.1 for suffix arrays. Given a reverse phrase with position j in *LZTrie*, its lexicographic rank is $R^{-1}[j]$. Compare it with Property 2.2 for suffix arrays.

Given this analogy, the question is: can we compress the R mapping just as we can compress a suffix array [GV05, Sad03]? We define now the analogue in LZ-index to function Ψ of Compressed Suffix Arrays (CSA), recall Section 2.4.2.

5.1.1 Defining Suffix Links in *RevTrie*

Definition 5.1. For every *RevTrie* preorder $1 \leqslant i \leqslant n$ we define function φ such that $\varphi(i) = R^{-1}(parent_{lz}(R[i]))$, and $\varphi(0) = 0$.

We have the following properties for function φ.

Property 5.1. Given a non-empty node with preorder i in *RevTrie*, such that $str_r(i) = ax$, for some $a \in \Sigma$, $x \in \Sigma^*$, then

(1) $str_r(\varphi(i)) = x$,

(2) $R[\varphi(i)] = parent_{lz}(R[i])$, and

(3) $letts_{lz}(R[i]) = a$.

Point (1) means that φ acts as a *suffix link* in *RevTrie*, and it follows from the fact that since $str_r(i) = ax$, then $str_{lz}(R[i]) = x^r a$, because node i in *RevTrie* corresponds to node $R[i]$ in *LZTrie*. Therefore, $str_{lz}(parent_{lz}(R[i])) = x^r$, which finally means $str_r(R^{-1}(parent_{lz}(R[i]))) = str_r(\varphi(i)) = x$. Point (2) implies that by following a suffix link in *RevTrie*, we are "going to the parent" in *LZTrie*, and it follows from applying R to both sides of the equation in Definition 5.1. Fig. 5.1 illustrates. Note that the edge connecting the *LZTrie* nodes with preorders $R[\varphi(i)]$ and $R[i]$ is labelled a (as stated by point (3)) which is the same symbol we are missing when following the suffix link $\varphi(i)$ in *RevTrie*.

We can prove that *RevTrie* is suffix closed since *LZTrie* is prefix closed, hence suffix links are well defined.

Lemma 5.1. *Every non-empty node in RevTrie has a suffix link.*

Proof. Let us consider any non-empty node in *RevTrie* with preorder i, such that $str_r(i) = ax$, for $a \in \Sigma$ and $x \in \Sigma^*$. As ax is a *RevTrie* phrase (with preorder i), then $x^r a$ must be a *LZTrie* phrase (with preorder $R[i]$). By Property 2.5 of the LZ78 parsing it follows that x^r is also a *LZTrie* phrase and thus x must be a *RevTrie* phrase. Hence, every non-empty node in *RevTrie* (i.e., every *RevTrie* node belonging to a reverse LZ78 phrase) has a suffix link. □

We will use Property 5.1 to reduce the space requirement of the R mapping: suppose that we do not store $R[i]$ for the *RevTrie* node with preorder i in Fig. 5.1, but we store $R[\varphi(i)]$; then note that $R[i]$ can be computed as $child_{lz}(R[\varphi(i)], a)$. Russo and Oliveira [RO07] also use properties of suffix links in their index; however, their main objective is to reduce the locating complexity of their LZ-index to $O((m + occ) \log u)$, yet not being able of reducing the space requirement as we do. In Section 5.3, however, we will show how to use suffix links to reduce the time complexity of our indexes, achieving their same locating complexity while requiring less space.

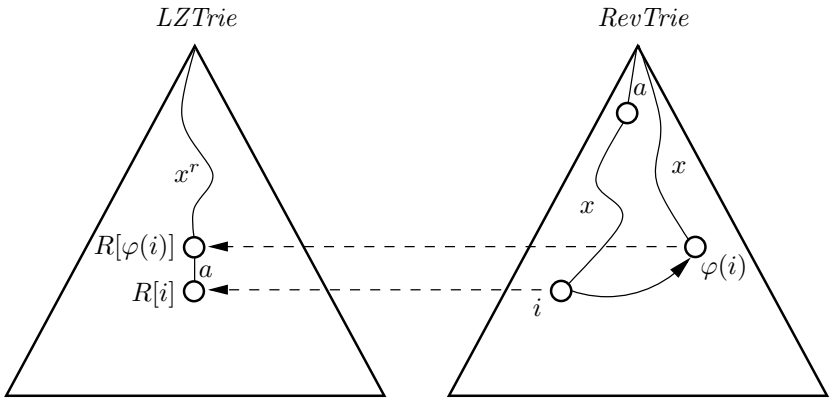


Figure 5.1: Illustration of Property 5.1. Preorder numbers, both in $LZTrie$ and $RevTrie$, are shown outside each node. Dashed arrows associate a $RevTrie$ node with its corresponding node in $LZTrie$. This association is given by the R mapping.

Example 5.1. In Table 5.1 we show some arrays composing the reduced version of LZ-index for our running example, including the φ function and the set of reversed LZ78 phrases in *RevTrie* (in preorder, i.e., lexicographically sorted).

5.1.2 Using Suffix Links to Compute R

Let us show how to compute $R[i]$ using function φ. We define array $L[1..n]$, which for each non-empty node with preorder i in *RevTrie* stores the first symbol of the string $str_r(i)$.

Example 5.2. In the *RevTrie* of Fig. 3.1(b), it holds that $L[i] =$ '`a`' for every i in the preorder interval $[2, 5]$. In the same example, note that if we follow the suffix link $\varphi(i)$, for $2 \leqslant i \leqslant 5$, we discard the symbol '`a`'.

In the example of Fig. 5.1 we have $L[i] = a$; as we said before, this also means that $L[i]$ is the label of the edge connecting *LZTrie* nodes with preorders $R[\varphi(i)]$ and $R[i]$. In other words, $L[i] = letts_{lz}(R[i])$. In Table 5.1 we show the values of L for our example.

It is not hard to prove that $L[i] \leqslant L[j]$ whenever $i \leqslant j$: let i and j be two preorders in *RevTrie*, such that $i \leqslant j$. Therefore, for the strings corresponding to these preorders it holds that $str_r(i) \leqslant str_r(j)$. As $L[i]$ and $L[j]$ store the first symbol of $str_r(i)$ and $str_r(j)$ respectively, then it holds that $L[i] \leqslant L[j]$. Thus, L can be divided into σ runs of equal symbols. In this way L can be represented by an array L' of at most $\sigma \log \sigma$ bits and a bit vector L_B of $n + o(n)$ bits, such that $L_B[i] = 1$ iff $L[i] \neq L[i-1]$, for $i = 2 \ldots n$, and $L_B[1] = 0$ (this position belongs to the text terminator "\$", which is not in the alphabet).

Table 5.1: Illustration of the different components of our index for the running example. In the case of $RevTrie$, array $rids$ is shown just for simplicity, yet this is not explicitly stored. In each case, i indicates the preorders in each trie.

LZTrie components

i	$ids[i]$	$letts_{lz}(i)$	$R^{-1}(i)$
0	0		0
1	1	a	2
2	17	$	1
3	3	b	6
4	15	r	15
5	14	l	10
6	4	r	14
7	12	a	4
8	10	d	8
9	16	l	11
10	6	_	17
11	11	p	13
12	2	l	9
13	7	a	3
14	9	b	7
15	5	_	16
16	8	a	5
17	13	p	12

RevTrie components

i	$rids[i]$	$R[i]$	$\varphi(i)$	$L[i]$	$L_B[i]$	string in *RevTrie*
0	0	0	0			(empty string)
1	17	2	2	$	0	$a
2	1	1	0	a	1	a
3	7	13	9	a	0	al
4	12	7	14	a	0	ara
5	8	16	16	a	0	a_
6	3	3	2	b	1	ba
7	9	14	3	b	0	bal
8	10	8	14	d	1	dra
9	2	12	0	l	1	l
10	14	5	2	l	0	la
11	16	9	14	l	0	lra
12	13	17	5	p	1	pa_
13	11	11	17	p	0	p_a
14	4	6	2	r	1	ra
15	15	4	6	r	0	rba
16	5	15	0	_	1	_
17	6	10	2	_	0	_a

For every i such that $L_B[i] = 1$, we store $L'[rank_1(L_B, i)] = L[i]$. Hence, $L[i]$ can be computed as $L'[rank_1(L_B, i)]$ in $O(1)$ time.

Given a *RevTrie* preorder position i, in order to compute $R[i]$ we could follow suffix links in *RevTrie* starting from node with preorder i, until we reach the *RevTrie* root. At this point we could apply, starting from the root of *LZTrie*, *child* operations using the first symbol of each *RevTrie* string we got while following suffix links, in reverse order. This procedure is formalized in the following lemma.

Lemma 5.2. *Given a RevTrie preorder position $0 \leqslant i \leqslant n$, the corresponding LZTrie preorder position $R[i]$ can be computed by the following recurrence:*

$$R[i] = \begin{cases} child_{lz}(R[\varphi(i)], L[i]) & \text{if } i \neq 0 \\ 0 & \text{if } i = 0 \end{cases}$$

Proof. $R[0] = 0$ holds from the fact that the preorder position corresponding to the empty string, both in *LZTrie* and *RevTrie*, is 0. To prove the other part we note that if x is the parent in *LZTrie* of node y with preorder position $R[i]$, then the symbol labeling the edge connecting x to y is stored in $L[i] = letts_{lz}(R[i])$. That is, $child_{lz}(parent_{lz}(R[i]), L[i]) = R[i]$. The lemma follows from this fact and replacing $\varphi(i)$ by Definition 5.1 in the recurrence. □

Example 5.3. To compute $R[13]$ for the example of Table 5.1, which corresponds to string '`p_a`' in *RevTrie*, we need to compute $child_{lz}(R[17], L[13])$, where $L[13] =$ '`p`' and $\varphi(13) = 17$ corresponds to the *RevTrie* preorder position of string '`_a`'. Now, to compute $R[17]$ we need to compute $child_{lz}(R[2], L[17])$, where $L[17] =$ '`_`'. Then, $R[2]$ is computed as $child_{lz}(R[0], L[2])$, which is just $child_{lz}(0, \text{'a'})$. At this point we must perform the *child* operations from the *LZTrie* root. Recall that we assume the $child_{lz}$ operation works on preorder positions, so we must compute $child_{lz}(child_{lz}(child_{lz}(0, \text{'a'}), \text{'\_'}), \text{'p'})$ in *LZTrie*, which is the same as $child_{lz}(child_{lz}(1, \text{'\_'}), \text{'p'})$, which in turn is $child_{lz}(10, \text{'p'})$, which finally yields the node with preorder position 11. Hence, we conclude that $R[13] = 11$.

5.1.3 Compressing the R Mapping

As in the case of the Ψ function of CSA [GV05, Sad03], we can prove the following lemma for the φ function, which is the key to compress the R mapping.

Lemma 5.3. *For every $i < j$, if $L[i] = L[j]$, then $\varphi(i) < \varphi(j)$.*

Proof. Let $str_r(i)$ denote the i-th string in the lexicographically sorted set of reversed strings. Note that $str_r(i) < str_r(j)$ iff $i < j$. If $i < j$ and $L[i] = L[j]$ (i.e., $str_r(i)$ and $str_r(j)$ start with the same symbol), then $str_r(\varphi(i)) < str_r(\varphi(j))$ (as $str_r(\varphi(i))$ is $str_r(i)$ without its first symbol, recall Property 5.1, point (1)), and thus $\varphi(i) < \varphi(j)$. □

Corollary 5.1. *Array φ can be partitioned into at most σ strictly increasing sequences.*

This fact is illustrated in Table 5.1, where the increasing runs of φ, corresponding to runs of equal symbols in L, are separated by horizontal lines.

As a result, we replace R by φ, L', and L_B, and use them to compute a given value $R[i]$. According to Corollary 5.1, we can represent φ using the idea of Sadakane [Sad03] to represent Ψ, which was explained in Section 2.4.2. Thus, φ can be encoded with $nH_0(letts) + O(n \log\log\sigma)$ bits, and hence we replace the $n \log n$-bits representation of R by the $nH_0(letts) + O(n \log\log\sigma) + n + o(n) = O(n \log\sigma) = o(u \log\sigma)$ bits of the representation of φ, L', and L_B. The absolute values of φ now add $O(n)$ bits, while the size of the table used to sum the differences is $o(n)$ bits.

5.1.4 Computing R and R^{-1} in $O(1/\epsilon)$ Time

The time to compute $R[i]$ according to Lemma 5.2 is $O(|str_r(i)|)$, which actually corresponds to traversing *LZTrie* from the root with the symbols of $str_r(i)$ in reverse order. However, the procedure of Lemma 5.2 can be adapted to allow constant-time computation of $R[i]$. We store ϵn values of R in an array R', plus a bit vector R_B of $n + o(n)$ bits indicating which values of R have been stored, ensuring that $R[i]$ can be computed in $O(1/\epsilon)$ time while requiring $\epsilon n \log n$ extra bits.

To determine the R values to be explicitly stored, we fix $l = \Theta(1/\epsilon)$ and carry out a preorder traversal on *LZTrie* to mark the nodes (1) whose *depth* is $j \cdot l$, for some $j > 0$, and such that (2) the corresponding node *height* is greater or equal to l. Since for every such marked node we have at least l non-marked nodes descending from it, we mark $O(\epsilon n)$ nodes overall. We also ensure that, if we start at an arbitrary node in *LZTrie* and go successively to the parent, in the worst case we must apply $O(1/\epsilon)$ *parent* operations to find a marked node. It can be the case that near the leaves of the trie we must follow a longer path to get a marked node, because of condition (2) above. However, notice that this path is never longer than $2l$, which still is $O(1/\epsilon)$. On the *RevTrie* side, this means that in the worst case we must follow $O(1/\epsilon)$ suffix links to find a node whose R value has been stored.

If the node to mark is at preorder position j, then we set $R_B[R^{-1}(j)] = 1$ (note that R_B is indexed by *RevTrie* preorder). After we mark the positions of R to be stored, we scan R_B sequentially from left to right, and for every i such that $R_B[i] = 1$, we set $R'[rank_1(R_B, i)] = R[i]$. Then, we free R since $R[i]$ can be computed in $O(1/\epsilon)$ worst-case time, as stated by the following lemma.

Lemma 5.4. *Given a RevTrie preorder position $0 \leqslant i \leqslant n$ and given any constant $0 < \epsilon < 1$, the corresponding LZTrie preorder position $R(i)$ can be computed in $O(1/\epsilon)$ worst-case*

time by the following recurrence:

$$R(i) = \begin{cases} child_{lz}(R(\varphi(i)), L'[rank_1(L_B, i)]) & \text{if } R_B[i] = 0 \\ R'[rank_1(R_B, i)] & \text{if } R_B[i] = 1 \end{cases}$$

requiring $\epsilon n \log n + O(n \log \sigma) = \epsilon H_k(T) + o(u \log \sigma)$ *bits of space.*

Note that the same structure used to compute R^{-1} using the explicit representation of R (see Section 2.5.1, Lemma 2.7) can be used under this reduced-space representation of R, with cost $O(1/\epsilon^2)$ to compute $R^{-1}(j)$ (as we have to access $O(1/\epsilon)$ positions in R). However, we shall show how to compute $R^{-1}(j)$ in $O(1/\epsilon)$ time, using a novel approach which basically consists in reverting the process used to compute R.

Definition 5.2. For every *RevTrie* preorder $0 \leqslant i \leqslant n$ and every symbol $a \in \Sigma$, we define function φ' such that $\varphi'(i, a) = R^{-1}(child_{lz}(R[i], a))$.

We have the following properties for function φ'.

Property 5.2. Given a non-empty node with preorder i in *RevTrie*, such that $str_r(i) = x$, for $x \in \Sigma^*$, then for $a \in \Sigma$ it holds that

(1) $str_r(\varphi'(i, a)) = ax$,

(2) $R[\varphi'(i, a)] = child_{lz}(R[i], a)$.

Point (1) means that φ' acts as a *Weiner link* [Wei73] in *RevTrie*. Point (2) means that by following a Weiner link by symbol a from node with preorder i, we are "going to a child by symbol a" in *LZTrie*. Fig. 5.2 illustrates.

Next we show how to efficiently compute φ' while requiring little space (since the obvious way to represent it requires basically $n \log n$ bits). Let $S_W[1..n]$ be an array of $n \log \sigma$ bits storing, for every *RevTrie* node, in preorder, the symbols by which the node has Weiner links defined, and let V_W be a bit vector. Because of Property 5.2 (2) (i.e., following a Weiner link by a symbol in *RevTrie* means going to the child by the same symbol in *LZTrie*), we can use the *LZTrie* as an aid to construct S_W: We perform a preorder traversal on *RevTrie*, and for every non-empty node with preorder i, let d be the degree of the corresponding *LZTrie* node $R[i]$. Then, we write the degree d in unary in V_W, in the format $\texttt{10}^d$. Thus, the `1`s in V_W will be used to locate the position of a node within the data structure (via operation $select_{\texttt{1}}$), while the `0`s in V_W shall be used to locate the position for the symbols of the links of a given node, as we will see soon. In the same traversal we also store in S_W the symbols labeling the children of node $R[i]$ in *LZTrie*. We represent arrays V_W and S_W with data structures for *rank* and *select* queries, requiring $o(u \log \sigma)$ bits overall.

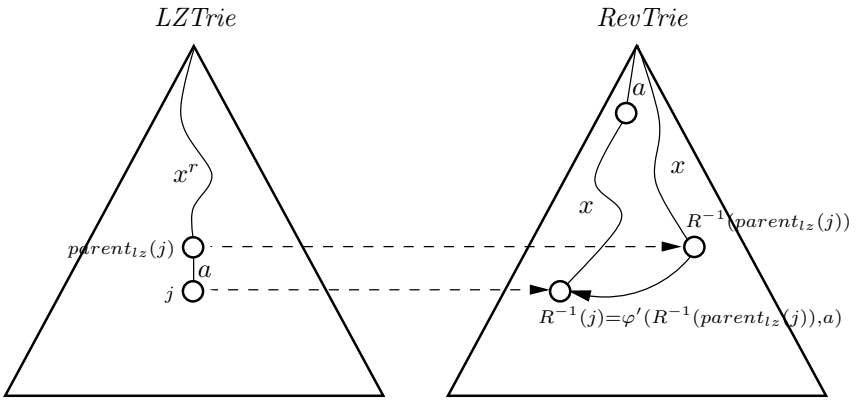


Figure 5.2: Illustration of Property 5.2 for function φ'. Preorder numbers, both in $LZTrie$ and $RevTrie$, are shown outside each node. Dashed arrows associate an $LZTrie$ node with its corresponding node in $RevTrie$. This association is given by R.

In order to understand how Weiner links can be represented in a compact way and computed efficiently, we shall store them (*conceptually*) in such a way that we can divide the resulting array into at most σ strictly increasing subsequences (note that this cannot be ensured if we simply store the links in preorder). Let $W[1..n]$ be the (conceptual) array storing the sequence of Weiner links. We will have an increasing subsequence in W for every symbol in the alphabet; every such subsequence stores the links going out by that symbol. Let $C_W[1..\sigma]$ be an array storing the starting position for the subsequence corresponding to every alphabet symbol. We then go on a new preorder traversal on *RevTrie*. For every non-empty node with preorder i (counting just non-empty nodes), let $R[i]$ be the corresponding *LZTrie* node, of degree d. Let $i_1 \leftarrow select_1(V_W, i+1)$ be the position in V_W corresponding to the current *RevTrie* node. Let $i_2 \leftarrow rank_0(V_W, i_1) + 1$ be the starting position in S_W corresponding to the current node. Then, for every child $j = 1, \ldots, d$ of node $R[i]$, which is labeled by symbol $s \leftarrow letts_{lz}(child_{lz}(R[i], j))$ in *LZTrie*, we store $W[C_W[s] + rank_s(S_W, i_2 - 1)] \leftarrow R^{-1}(child_{lz}(R[i], j))$.

Given this representation, we can compute, for any non-empty node with preorder i in *RevTrie* and a symbol $a \in \Sigma$:

$$\varphi'(i, a) \equiv W[C_W[a] + rank_a(S_W, rank_0(V_W, select_1(V_W, i+1)) + 1)]$$

Now it remains to show that this representation can be compressed.

Lemma 5.5. *Array W can be partitioned into at most σ strictly increasing sequences.*

Proof. Let positions i and j in W, for $i < j$, correspond to Weiner links going out by the same symbol $a \in \Sigma$. Assume that position i corresponds to node for string $x \in \Sigma^*$ in

RevTrie, and position j corresponds to string $y \in \Sigma^*$. Since $i < j$ and given the way in which W is constructed, it follows that the preorder of the node for x is smaller than the preorder of node representing y. This also means that $x < y$. Then, $ax < ay$ also holds. Therefore the preorder stored at $W[i]$ (i.e., the one pointing to the node for string ax) is smaller than the preorder stored at $W[j]$ (which points to the node for string ay). □

Thus, we could represent W in the same way as array φ, requiring overall $O(n \log \sigma) = o(u \log \sigma)$ bits of space. However, we can do better. Let $j \leftarrow child_r(0, a)$ be the preorder of the child of the *RevTrie* root by symbol a. Notice that all Weiner links going out by a given symbol, let us say symbol a, point to a node within the subtree of node with preorder j. Since Lemma 5.5 states that the Weiner links for symbol a appear in increasing order within the corresponding subsequence of W, this means that the first link for a points to the first node in preorder within the subtree of node with preorder j, the second link points to the second node in preorder within the subtree of node with preorder j, and so on. This means that just performing a *rank* on S_W allows us to compute the corresponding link, so we do not need to store array W. Formally, we have:

$$\varphi'(i, a) \equiv child_r(0, a) + rank_a(S_W, rank_0(V_W, select_1(V_W, i + 1)) + 1) - 1$$

We will use function φ' and its properties in order to compute R': suppose that we do not store $R^{-1}(j)$ for the *LZTrie* node with preorder j in Fig. 5.2, but we store $R^{-1}(parent_{lz}(j))$. Then notice that $R^{-1}(j)$ can be computed as $\varphi'(R^{-1}(parent_{lz}(j)), a)$. For every *LZTrie* node that has been marked to store an R value, as explained above, we also store the corresponding value of R^{-1} in array R''. We mark in a bit vector R_B^{-1} (according to preorder in *LZTrie*) the nodes whose R^{-1} value has been stored. This ensures that, starting at an arbitrary node in *LZTrie*, we shall find a sampled node after performing at most $O(1/\epsilon)$ *parent* operations. Then we can conclude:

Lemma 5.6. *Given an LZTrie preorder position $0 \leqslant j \leqslant n$ and given any constant $0 < \epsilon < 1$, the corresponding RevTrie preorder position $R^{-1}(j)$ can be computed in $O(1/\epsilon)$ worst-case time by the following recurrence:*

$$R^{-1}(j) = \begin{cases} \varphi'(R^{-1}(parent_{lz}(j)), letts_{lz}(j)) & \text{if } R_B^{-1}[j] = 0 \\ R''[rank_1(R_B^{-1}, j)] & \text{if } R_B^{-1}[j] = 1 \end{cases}$$

requiring $\epsilon n \log n + O(n \log \sigma) = \epsilon H_k(T) + o(u \log \sigma)$ bits of space.

5.1.5 Space and Time Analysis

As now we store *ids* in $n \log n$ bits, ids^{-1}, R' and R'' in $\epsilon n \log n$ bits each, and φ, φ', *letts*, and *rletts* in $O(n \log \sigma) = o(u \log \sigma)$ bits, the total space requirement is $(1 + \epsilon)n \log n + o(u \log \sigma)$ bits (renaming $4\epsilon = \epsilon$), and we provide the same navigation

scheme as in Fig. 4.3(b) (on page 72). Occurrences of type 1 are found as usual, in $O(m + \frac{occ_1}{\epsilon})$ time, where the extra $O(\frac{occ_1}{\epsilon})$ term appears because we have to use R to map from *RevTrie* to *LZTrie*, which takes $O(1/\epsilon)$ each time. Occurrences of type 2 are solved as explained in Section 3.3.7, in $O(\frac{n}{\epsilon\sigma^{m/2}})$ average time since now the access between tries is provided by R and R^{-1}. Finally, for solving occurrences of type 3, we first search for all the pattern substrings in *LZTrie* in $O(m^2)$ time, then compute the maximal concatenations of phrases, in $O(\frac{m^2}{\epsilon})$ time by using the improved algorithm of Lemma 3.4 (the $O(1/\epsilon)$ factor comes from the fact that we use ids^{-1} to simulate *Node*), and finally for each of the $O(m^2)$ maximal concatenations found we carry out the tests as explained in Section 3.3.7, with cost $O(\frac{m^2}{\epsilon})$ because *RNode* is implemented by using R^{-1}. We have proved:

Theorem 5.1. *Given a text $T[1..u]$ over an alphabet of size σ and with k-th order empirical entropy $H_k(T)$, and let n be the number of phrases in the LZ78 parsing of T, there exists a compressed full-text self-index requiring $(1+\epsilon)uH_k(T) + o(u\log\sigma)$ bits of space, for $\sigma = O(\text{polylog}(u))$, any $k = o(\log_\sigma u)$ and any $0 < \epsilon < 1$. Given a pattern $P[1..m]$, this index is able to locate (and count) the occ occurrences of pattern P in text T in $O(\frac{m^2}{\epsilon} + \frac{n}{\epsilon\sigma^{m/2}})$ average time, which is $O(\frac{m^2}{\epsilon})$ if $m \geqslant 2\log_\sigma n$.*

Now we can get worst-case guarantees in the search process by adding *Range*, the two-dimensional range search data structure defined in Section 3.3 for the original LZ-index, requiring $n\log n + o(u\log\sigma)$ extra bits [MN07]. Occurrences of type 2 can now be solved in $O((m + occ_2)\log n)$ worst-case time by using *Range*, and then we use the *LZTrie* coordinate of the point to map to the corresponding *LZTrie* node, in order to get the phrase identifier for that occurrence (we do this because *RevTrie* does not store the phrase identifiers in our representation, and we must use R in order to get them, which would take $O(1/\epsilon)$ per occurrence). Occurrences of type 1 and type 3 are found as for the index of Theorem 5.1. Existential queries, on the other hand, can be solved by first looking whether there is any occurrence of type 1 (i.e., by looking for P^r in *RevTrie* and then checking whether the corresponding subtree is empty or not) in $O(m)$ time. If there are no occurrences of type 1, we check whether there is any occurrence of type 2 by partitioning the pattern and using *Range* to count the number of occurrences for each partition. This takes $O(m\log n)$ time overall, since we use *Range* just to count. Finally, if there are no occurrences of type 2, we look for occurrences of type 3 in $O(m^2/\epsilon)$ time. Hence, we have the following theorem.

Theorem 5.2. *Given a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, there exists a compressed full-text self-index requiring $(2+\epsilon)$ $uH_k(T) + o(u\log\sigma)$ bits of space, for $\sigma = O(\text{polylog}(u))$, any $k = o(\log_\sigma u)$ and any $0 < \epsilon < 1$. Given a search pattern $P[1..m]$, this index is able to:*

(1) locate the occ occurrences of pattern P in text T in $O(\frac{m^2}{\epsilon} + (m + occ)\log u + \frac{occ}{\epsilon})$ worst-case time;

(*2*) *count the number of pattern occurrences in* $O(\frac{m^2}{\epsilon} + m \log u + \frac{occ}{\epsilon})$ *worst-case time; and*

(*3*) *determine whether pattern* P *exists in* T *in* $O(\frac{m^2}{\epsilon} + m \log u)$ *worst-case time.*

In Section 5.5 we show that the theorem is valid for the more general case $\log \sigma = o(\log u)$. We leave for Section 5.4 the study of `display` and `extract` queries on our indexes.

5.2 Using the *xbw* Transform to Represent *LZTrie*

A different idea to reduce the space requirement of LZ-index is to use the *xbw transform* of Ferragina et al. [FLMM05] to represent the *LZTrie*. We show that subpath queries, which are efficiently solved by the *xbw* transform (see Section 2.5.2), are so powerful that we can carry out the work of both *LZTrie* and *RevTrie* only with the *xbw* representation of *LZTrie*, thus achieving the same result as in Section 5.1 (always assuming $\sigma = O(\text{polylog}(u))$), yet by radically different means. Ferragina et al. [FLMM05] have shown how the *xbw* representation can be compressed in order to take advantage of the tree regularities, which can be very important in practice and adds extra value to this representation.

5.2.1 Index Definition

We represent LZ-index with the following data structures:

— *xbw LZTrie*: the *xbw* representation [FLMM05] of *LZTrie* (see Lemma 2.12), where the nodes are lexicographically sorted according to their upward paths in the trie. We store

- S_α: the array of symbols labeling the edges of the trie. In the worst case *LZTrie* has $2n$ nodes (because of the dummy leaves we add, recall Section 2.5.2, Lemma 2.12). We represent this array by using the data structure for *rank* and *select* of Lemma 2.5 (1), which are needed to compute the operations on *xbw*. The space requirement is $2n \log \sigma + o(n \log \sigma)$ bits.
- S_{last}: a bit array such that $S_{last}[i] = 1$ iff the corresponding node in *LZTrie* is the last child of its parent. We represent this array with the data structure for *rank* and *select* of Lemma 2.4 (1). The space requirement is at most $2n + o(n)$ bits.

Example 5.4. See Table 2.1 on page 44 for an illustration of the *xbw* of *LZTrie* for the running example. See Fig. 5.3 for an illustration of our LZ-index.

— Balanced parentheses *LZTrie*: the trie of the Lempel-Ziv phrases, implemented by

- *par*: the balanced parentheses representation (see Lemma 2.10) of *LZTrie*. In order to index the *LZTrie* leaves with *xbw*, we have to add a dummy child to each, as it was explained in Section 2.5.2, Lemma 2.12. In this way, the trie has $n' \leqslant 2n$ nodes. Non-dummy nodes are marked in a bit vector $B[1..n']$ in the same way as empty nodes are marked in *RevTrie* (see Section 4.1.2). We represent array B with a data structure for *rank* and *select* queries (see Lemma 2.4 (1)). The space requirement is $2n' + n' + o(n)$ bits, which is $6n + o(n)$ bits in the worst case. This sequence *par* is needed to solve some operations which are not supported by the *xbw*, such as $ancestor(x, y)$ and $depth(x)$.
- *ids*: the array of LZ78 phrase identifiers in preorder, only for non-dummy nodes (we find the phrase identifier for a given node by using $rank_1$ on B). This array is represented by the data structure of Lemma 2.7, such that we can compute the inverse permutation ids^{-1} in $O(1/\epsilon)$ time, requiring $(1 + \epsilon)n \log n$ bits.

— *Pos*: a mapping from *xbw* positions to the corresponding *LZTrie* preorder positions (i.e., this is a permutation of *LZTrie* preorders). In the worst case there are $2n$ such positions, and so the space requirement is $2n \log (2n)$ bits. We can reduce this space to $\epsilon n \log (2n)$ bits by storing in an array Pos' one out of $O(1/\epsilon)$ values of Pos, such that $Pos[i]$ can be computed in $O(1/\epsilon)$ time. We need a bit vector Pos_B of $2n + o(n)$ bits indicating which values of Pos have been stored. Assume we need to compute the preorder position $Pos[i]$, for a given *xbw* position i. If $Pos_B[i] = 1$, then such preorder position is stored explicitly at $Pos'[rank_1(Pos_B, i)]$. Otherwise, we simulate a preorder traversal in *xbw* from the node at *xbw* position i, until $Pos_B[j] = 1$, for an *xbw* position j. Each preorder step we perform in *xbw* corresponds to moving to the next opening parenthesis in *par*. Once this j is found, we map to the preorder position $j' = Pos'[rank_1(Pos_B, j)]$. If d is the number of nodes in preorder traversal from *xbw* position i to *xbw* position j, then $j' - d$ is the preorder position corresponding to the node at *xbw* position i.

  We also need to compute Pos^{-1}, which can be done in $O(1/\epsilon^2)$ time under this scheme, requiring $\epsilon n \log (2n)$ extra bits if we use the representation of Lemma 2.7 for inverse permutations. However, we can support the computation of Pos^{-1} in $O(1/\epsilon)$ time as follows. For every node such that its Pos value has been stored in Pos', we also store the corresponding value of Pos^{-1} in array Pos''. If we want to compute $Pos^{-1}[i]$, we first compute the preorder of the previous node that has been sampled in Pos'' by $j = l\lfloor \frac{i}{l} \rfloor$, where $l = \Theta(1/\epsilon)$. Then, we use the sampled value stored at $Pos''[\lfloor \frac{i}{l} \rfloor]$ to map to the *xbw*, and then we perform $i - j$ preorder steps in *xbw*, to find the node corresponding to $Pos^{-1}[i]$. This takes $O(1/\epsilon)$ time in the worst case.

— *Range*: a *range search* data structure in which we store the point k (belonging to phrase identifier k) at coordinate (x, y), where x is the *xbw position* of node for

phrase k and y is the *LZTrie preorder position* of node for phrase $k+1$. We use the data structure of Lemma 2.9, as for the original LZ-index. The space requirement is $n \log n + O(n \log \log n) = n \log n + o(u \log \sigma)$ bits.

	0									10									20									30									40									50
par:	(	(	(	(	)	)	(	(	(	)	)	)	(	(	)	)	(	(	(	)	)	(	(	)	)	)	(	(	(	)	)	)	)	(	(	(	(	)	)	)	)	)	(	(	(	(	)	)	)	)	)
B:	1	1	1	0			1	1	0				1	0			1	1	0			1	0				1	0				1	0				1	1	0				1	1	1	0				1	1	1	0			
ids:	0	1	17				3	15					14				4	12				10					16					6	11				2	7	9				5	8	13									
letts:	a	$	Δ				b	r	Δ				l	Δ			r	a	Δ			d	Δ				l	Δ				-	p	Δ			l	a	b	Δ			-	a	p	Δ								
Pos^{-1}:	1	5	4				6	13	24				7	17			8	21	11			22	15				23	18				9	26	20			2	16	10	14			3	25	12	19								

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
$Pos[i]$	1	19	23	3	2	4	7	9	16	21	11	25	5	22	13	20	8	15	26	18	10	12	14	6	24	17

Figure 5.3: Balanced-parentheses representation of $LZTrie$ for the running example, with dummy leaves added in order to index the (original) leaves with the *xbw* representation. Bit vector B marks with a 0 the dummy nodes. Given node x, the corresponding phrase identifier can be computed as $ids[rank_1(B, preorder(x))]$. We also show the Pos mapping (from *xbw* positions, see Table 2.1 on page 44, to $LZTrie$ preorders), and the Pos^{-1} mapping (from $LZTrie$ preorders to *xbw* positions).

In Fig. 5.4 we show the basic resulting navigation scheme following the notation of Chapter 4. The total space requirement is $(2+\epsilon)n \log n + 2n \log \sigma + 10n + O(n \log \log n) + o(n)$ bits, which is $(2+\epsilon)uH_k(T) + o(u \log \sigma)$ bits for $k = o(\log_\sigma u)$.

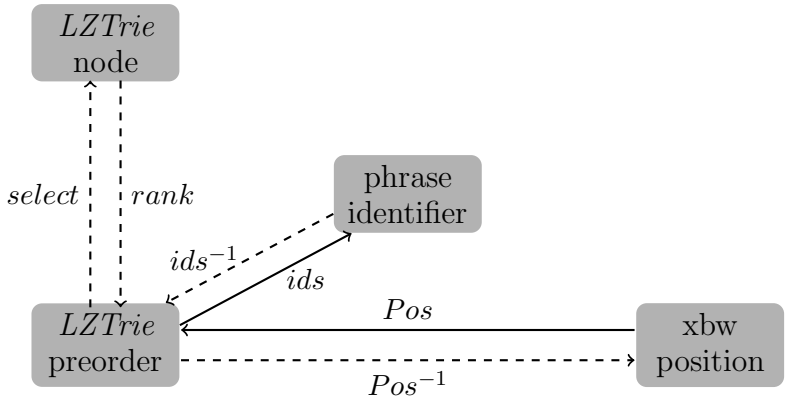


Figure 5.4: Basic navigation for the scheme using the *xbw* representation of $LZTrie$.

5.2.2 Search Algorithm

We depict now the search algorithm for a pattern P of length m.

Occurrences of Type 1. Recall from Section 3.3.5 that first we need to find all phrases having P as a suffix. To do this we perform a subpath query with P^r on the *xbw* representation of *LZTrie*, simulating in this way the work done on *RevTrie* in the original scheme, in $O(m)$ time. Suppose that we obtain the interval $[x_1..x_2]$ in the *xbw* of *LZTrie*, corresponding to all nodes whose phrase ends with P. In other words, the interval $[x_1..x_2]$ contains the roots of the subtrees containing the nodes we are looking for to solve occurrences of type 1. For each position $i \in [x_1..x_2]$, we can get the corresponding preorder in the parentheses representation using $Pos(i)$, which takes $O(1/\epsilon)$ time, and then $selectnode(Pos(i))$ over *par* yields the node position. As in the worst case this mapping is carried out *occ* times, the overall time is $O(\frac{occ}{\epsilon})$. Finally we traverse the subtrees of these nodes in *par* and report all the identifiers found, in constant time per occurrence as done with the usual LZ-index.

Occurrences of Type 2. To solve occurrences of type 2, for every possible partition $P[1..i]$ and $P[i+1..m]$ of P, we traverse the *xbw* from the root, using operation $child(x, \alpha)$ with the symbols of $P[i+1..m]$. This takes $O(m^2)$ time overall for the $m-1$ partitions of P. In this way we are simulating the work done on *LZTrie* when solving occurrences of type 2 in the original scheme. Once this is found, say at *xbw* position j, we switch to the preorder tree (parentheses) using $selectnode(Pos(j))$ over *par*, to get the node v_{lz} whose subtree has preorder interval $[y_1..y_2]$ of all the nodes that start with $P[i+1..m]$. This takes overall $O(\frac{m}{\epsilon})$ time, for the $m-1$ partitions of P. Next we perform a subpath query for $P[1..i]$ in *xbw*, and get the *xbw* interval $[x_1..x_2]$ of all the nodes that finish with $P[1..i]$ (actually we have to perform $x_1 \leftarrow rank_1(S_{last}, x_1)$ and $x_2 \leftarrow rank_1(S_{last}, x_2)$ to avoid counting the same node multiple times, see [FLMM05]). This also takes $O(m^2)$ time overall. Finally, we search the *Range* data structure for $[x_1..x_2] \times [y_1..y_2]$ to get all phrase identifiers t such that phrase B_t finishes with $P[1..i]$ and phrase B_{t+1} starts with $P[i+1..m]$, in $O((m+occ)\log n)$ time overall.

Occurrences of Type 3. For occurrences of type 3, one proceeds mostly as with the original *LZTrie* (navigating the *xbw* instead), so as to find all the nodes equal to substrings of P in $O(m^2)$ time. Then, for each maximal concatenation of phrases $P[i..j] = B_t \dots B_\ell$ we must check that phrase $B_{\ell+1}$ starts with $P[j+1..m]$ and that phrase B_{t-1} finishes with $P[1..i-1]$. The first check can be done in $O(1/\epsilon)$ time by using ids^{-1}: as we have searched for all substrings of P in the trie, we know the preorder interval of the descendants of $P[j+1..m]$, thus we check whether the node at preorder position $ids^{-1}(\ell+1)$ belongs to that interval. The second check can be done in $O(1/\epsilon)$ time, by determining whether $t-1$ lies in the *xbw* interval of $P[1..i-1]$ (that is, B_{t-1} finishes with $P[1..i-1]$). For this, we need Pos^{-1}, so that the position is $Pos^{-1}(ids^{-1}(t-1))$.

Summarizing, occurrences of type 1 cost $O(m + \frac{occ}{\epsilon})$, occurrences of type 2 cost

$O(m^2 + \frac{m}{\epsilon} + (m + occ) \log n)$, and type 3 cost $O(\frac{m^2}{\epsilon})$. Thus, we have achieved Theorem 5.2 again with radically different means. The same complexities are also achieved for counting and existencial queries. We can also get a version requiring $(1 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits and $O(m^2)$ average reporting time if $m \geqslant 2 \log_\sigma n$ (as in Theorem 5.1) if we solve occurrences of type 2 by using a procedure similar to that used to solve occurrences of type 3.

5.3 Faster and Still Small LZ-indexes

In Section 5.1 we have shown how to use suffix links in *RevTrie* to reduce the space requirement of LZ-index. Russo and Oliveira [RO07] show how to use suffix links to reduce the locating time of their LZ-index to $O((m + occ) \log u)$; yet, they do not use suffix links to reduce the space of their index (recall Section 3.4). On the other hand, Ferragina and Manzini [FM05] combine the backward-search concept with a Lempel-Ziv-based scheme to achieve optimal $O(m + occ)$ locating time, without restrictions on m or *occ*. Yet, their index is even larger, requiring $O(uH_k(T) \log^\gamma u)$ bits of space, for any constant $\gamma > 0$ (recall Section 3.2).

In this section we use suffix links to speed up occurrences of type 2, using an idea similar to that of [RO07], and we solve occurrences of type 3 as a particular case of occurrences of type 2, using a similar idea to that of [FM05]. In this way we manage to avoid the $O(m^2)$ term in the locating complexity of LZ-index, achieving the same locating time as [RO07], while reducing their space requirement of $(5 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits.

5.3.1 Index Definition.

We build basically on the LZ-index of Theorem 5.1, composed of *LZTrie*, *RevTrie*, and the R mapping (compressed using suffix links φ). We add to *LZTrie* the data structure of Jansson et al. [JSS07b] to compute *level ancestor queries*, $LA(x, d)$, which gets the ancestor at depth d of node x. This requires $o(n)$ extra bits and supports LA queries in constant time. Therefore, the overall space requirement of the three above data structures is $(1 + \epsilon)uH_k + o(u \log \sigma)$ bits.

To avoid the $O(m^2)$ term in the locating complexity, we should avoid occurrences of type 3, since they make us check the $O(m^2)$ possible candidates. We cannot use the same procedure as for occurrences of type 2 (using the *Range* data structure) because *LZTrie* is only able to index whole phrases, and not text suffixes. Then, by using *LZTrie* to query the *Range* data structure we are only capable to get the phrases starting with a given suffix $P[i + 1..m]$ of the pattern, an therefore we can find only occurrences spanning two consecutive phrases (i.e., occurrences of type 2).

Hence we add the *alphabet friendly FM-index* [FMMN07] of T (AF-FMI(T) for short) to our index. By itself this self-index is able to search for pattern occurrences, requiring $uH_k(T)+o(u\log\sigma)$ bits of space. However, its locate time per occurrence is $O(\log^{1+\epsilon} u)$, for any constant $\epsilon > 0$, which is greater than the $O(\log u)$ time per occurrence of LZ-indexes.

As AF-FMI(T) is based on the *Burrows-Wheeler Transform* [BW94] of T (*bwt*(T) for short), it can be (conceptually) thought of as the suffix array SA_T of T (see Section 2.4.2). The AF-FMI(T) indexes text suffixes. In particular, we will be interested in those suffixes which are aligned with the LZ78 phrase beginnings. By using this structure to query the *Range* data structure (instead of using *LZTrie*) we will be able to find those text suffixes which are aligned with LZ78 phrases and that have $P[i+1..m]$ as a prefix. Thus, $P[i+1..m]$ can span more than two consecutive phrases, and therefore we will consider occurrences of type 3 as a special case of occurrences of type 2.

To find occurrences spanning several phrases we re-define *Range*, the data structure for 2-dimensional range searching. Now it will operate on the grid $[1..u] \times [1..n]$. For each LZ78 phrase with identifier id, for $0 < id \leqslant n$, assume that the *RevTrie* node for id has preorder j', and that phrase $(id+1)$ starts at position p in T. Then we store the point (i', j') in *Range*, where i' is the lexicographic order of the suffix of T starting at position p, i.e., $SA_T[i'] = p$ holds.

Suppose that we search for a given string s_2 in AF-FMI(T) and get the interval $[i_1, i_2]$ in the *bwt*(T) (equivalently, in the suffix array of T), and that the search for string s_1^r in *RevTrie* yields a node such that the preorder interval for its subtree is $[j_1, j_2]$. Then, a search for $[i_1, i_2] \times [j_1, j_2]$ in *Range* yields all phrases ending with s_1 such that the next phrase is aligned with an occurrence of s_2 in T.

We transform the grid $[1..u] \times [1..n]$ indexed by *Range* to an equivalent grid $[1..n] \times [1..n]$ by defining a bit vector $V[1..u]$, which indicates (with a `1`) which positions of AF-FMI(T) point to an LZ78 phrase beginning. We represent V with the data structure of Lemma 2.4 (2) allowing *rank* queries, and requiring $uH_0(V) + o(u)$ bits of space, which according to Lemma 2.2 is $uH_0(V) + o(u) \leqslant n\log\frac{u}{n} + o(u) \leqslant \frac{u\log\log u}{\log_\sigma u} + o(u) = o(u\log\sigma)$ bits of storage. Thus, instead of storing the point (i', j') as in the previous definition of *Range*, we store the point $(rank_1(V, i'), j')$. The same search of the previous paragraph now becomes $[rank_1(V, i_1), rank_1(V, i_2)] \times [j_1, j_2]$.

As there is only one point per row and column of *Range*, we can use the data structure of Lemma 2.9, which can be implemented by using $n\log n + O(n\log\log n) = uH_k(T) + o(u\log\sigma)$ bits [MN07]. As a result, the overall space requirement of our LZ-index is $(3+\epsilon)uH_k(T) + o(u\log\sigma)$, for any $k = o(\log_\sigma u)$ and any $0 < \epsilon < 1$.

5.3.2 Search Algorithm

For `exists` and `count` queries we can achieve $O(m)$ time by just using the AF-FMI(T). We focus now on `locate` queries. Assume that $P[1..m] = p_1 \dots p_m$, for $p_i \in \Sigma$. As explained, we need to consider only occurrences of P in T of type 1 and 2. Those of type 1 are solved just as for the original LZ-index, in $O(m + \frac{occ_1}{\epsilon})$ time. The rest of the section is devoted to those of type 2.

To find the pattern occurrences spanning two or more consecutive phrases we must consider the $m-1$ partitions $P[1..i]$ and $P[i+1..m]$ of P, for $1 \leqslant i < m$. For every partition we must find all phrases terminated with $P[1..i]$ such that the next phrase starts at the same position as an occurrence of $P[i+1..m]$ in T. Hence, as explained before, we must search for $P^r[1..i]$ in *RevTrie* and for $P[i+1..m]$ in AF-FMI(T). Thus, every partition produces two one-dimensional intervals, one in each of the above structures.

If the search in *RevTrie* for $P^r[1..i]$ yields the preorder interval $[j_1, j_2]$, and the search for $P[i+1..m]$ in AF-FMI(T) yields interval $[i_1, i_2]$, the two-dimensional range $[rank_1(V, i_1), rank_1(V, i_2)] \times [j_1, j_2]$ in *Range* yields all pattern occurrences for the given partition of P. For every pattern occurrence we get a point (i', j') from *Range*. The corresponding phrase identifier can be found as $t = ids(R(j'))$, to finally report a pattern occurrence $[\![t, i]\!]$.

Overall, occurrences of type 2 are found in $O((m + occ_2) \log n)$ time. Yet, we still have to show how to find efficiently the intervals in AF-FMI(T) and in *RevTrie*.

The $m-1$ intervals for $P[i+1..m]$ in AF-FMI(T) can be found in $O(m)$ time thanks to the *backward search* concept, since the process to count the number of occurrences of $P[2..m]$ proceeds in $m-1$ steps, each one taking constant time if $\sigma = O(\text{polylog}(u))$ [FM05]: in the first step we find the BWT interval for p_m, then we find the interval for occurrences of $p_{m-1}p_m$, then $p_{m-2}p_{m-1}p_m$, and so on to finally find the interval for $p_2 \dots p_m = P[2..m]$.

However, the work in *RevTrie* can take time $O(m^2)$ if we search for strings $P^r[1..i]$ separately, as done for the indexes of Section 5.1. Fortunately, some work done to search for a given $P^r[1..i]$ can be reused to search for other strings. We have to search for strings $p_{m-1}p_{m-2} \dots p_1$; $p_{m-2} \dots p_1$;...; and p_1 in *RevTrie*. Note that every $p_j \dots p_1$ is the longest proper suffix of $p_{j+1}p_j \dots p_1$. Suppose that we successfully search for $P^r[1..m-1] = p_{m-1}p_{m-2} \dots p_1$, reaching the node with preorder i' in *RevTrie*, hence finding the corresponding preorder interval in *RevTrie* in $O(m)$ time. Now, to find the node representing suffix $p_{m-2} \dots p_1$ we only need to follow suffix link $\varphi(i')$ (which takes $O(1)$ time) instead of searching for it from the *RevTrie* root (which would take $O(m)$ time again). The process of following suffix links can be repeated $m-1$ times up to reaching the node corresponding to string p_1, with total time $O(m)$. This is the main idea to get

the $m-1$ preorder intervals in *RevTrie* in time less than quadratic. The general case is slightly more complicated and corresponds to the *descend and suffix walk* method used in [RO07].

In the sequel we explain the way we implement descend and suffix walk in our data structure. However, we must prove a couple of properties for *RevTrie* in order to be able to apply this method. First, we know that every non-empty node in *RevTrie* has a suffix link (see Lemma 5.1), yet we need to prove that every *RevTrie* node (including empty-non-unary nodes) has also a suffix link.

Lemma 5.7. *Every empty non-unary node in RevTrie has a suffix link.*

Proof. Assume that node v_r in *RevTrie* is empty non-unary, and that it represents string ax, for $a \in \Sigma$ and $x \in \Sigma^*$. As node v_r is empty non-unary, the node has at least two children. In other words, there exist at least two strings of the form axy and axz, for y, $z \in \Sigma^*$, $y \neq z$, both strings corresponding to non-empty nodes, and hence these nodes have a suffix link. These suffix links correspond to strings xy and xz in *RevTrie*. Thus, there must exist a non-unary node for string x, which is the suffix link of node v_r. □

The descent process in *RevTrie* will be a little bit different from the one described in the proof of Lemma 4.1. This time, we are going to reuse the work done for a string already searched in *RevTrie*, so we have to be sure that every time we arrive to a *RevTrie* node, the string represented by that node matches the corresponding pattern prefix (the usual skipping process of a Patricia tree does not ensure that). Thus, the second property is that, although *RevTrie* is a Patricia tree and hence we store only the first symbol of each edge label, we can get all of it.

Lemma 5.8. *Any edge label of length l in RevTrie can be extracted in $O(l)$ time.*

Proof. Assume that we are at node v_r in *RevTrie*, and want to extract the label for edge $e_{v_r v'_r}$ between nodes v_r and v'_r in *RevTrie*. Since we arrive at a node in *RevTrie* by descending from the root, the length of the string represented by a given node can be computed by summing up the skips we have seen in the descent. Let l_{v_r} and $l_{v'_r}$ be the length of strings represented by nodes v_r, and v'_r respectively. Then, $l_{v'_r} - l_{v_r}$ is the length of the label of edge $e_{v_r v'_r}$.

If we assume that node v'_r has preorder j_1 in *RevTrie*, we can access the *LZTrie* node from where to start the extraction of the label by $v'_{lz} = LA(R[j_1], l_{v'_r} - l_{v_r})$, in constant time [JSS07b]. The label of $e_{v_r v'_r}$ is the label of the v'_{lz}-to-root path. Notice that with the level-ancestor query on *LZTrie* we avoid to extract the string represented by node v_r in *RevTrie*, as it has been already extracted before descending to v_r.

In case that v'_r is an empty node, recall that the corresponding value $R[j_1]$ is undefined. However, just as in the proof of Lemma 4.1, we can use any non-empty

node within the subtree of v'_r to map to the *LZTrie*. For instance, we can use the next non-empty node within the subtree of v'_r: let $j_2 = rank_1(B, j_1) + 1$, then the length of the corresponding string can be computed as $depth(R[j_2])$ in *LZTrie*, and we compute $v'_{lz} = LA(R[j_2], depth(R[j_2]) - l_{v_r})$, to finally extract the edge label by moving to the parent $l_{v'_r} - l_{v_r}$ times. □

Thus, we search *RevTrie* as in a normal trie, comparing *every* symbol as we descend, without skipping as it is done in Lemma 4.1. In this way, every time we arrive to a *RevTrie* node, the string represented by that node will match the corresponding prefix of the pattern.

Previously we showed that it is possible to search for all strings $P^r[1..i]$ in time $O(m)$, assuming that $P^r[1..m-1]$ exists in *RevTrie* (therefore all $P^r[1..i]$ exist in *RevTrie*). The general case is as follows.

Let $P^r[1..m-1] = p_{m-1} \dots p_1$ be the longest string that we need to search for in *RevTrie*. We define three integer indices on $P^r[1..m-1]$, which guide the search:

i_1: marks the beginning of the pattern suffix we are currently searching for. It is initialized at 1 since we start searching for $p_{m-1}p_{m-2} \dots p_1$;

i_2: indicates the current symbol in the pattern that is being compared with a symbol in an edge label, with the aim of descending to a child of the current node. Notice that $(P^r)[i_1..i_2 - 1]$ is the part of the current pattern that has been matched with the edge labels of *RevTrie*; and

i_3: delimits the string corresponding to the current node, which represents string $(P^r)[i_1..i_3]$ in *RevTrie*. Thus $(P^r)[i_3 + 1..i_2 - 1]$ will be the part of the pattern that has been compared with the label of the edge leading to the node we are trying to descend to.

Our descend and suffix walk will be composed of three basic operations: *descend*, *suffix*, and *retraverse*.

Descend. We start searching for $p_{m-1}p_{m-2} \dots p_1$ from the *RevTrie* root, using the method of Lemma 5.8 and using i_2 to indicate the current symbol being compared in the descent. Every time we can descend to a non-empty-unary child node (after matching all the characters of an edge), we set $i_3 \leftarrow i_2$ and continue descending in the same way from this node. If, when trying to descend to a child node, we find an empty-unary node (which were added to limit the skips in *RevTrie*, see Section 4.1.2), the index i_3 is not updated as explained before. In this case, we continue the descent with i_2 from the empty-unary node, using Lemma 5.8.

Suffix. Now assume that, being at current node v_r (with preorder j_1 in *RevTrie* and representing string ax, for $a \in \Sigma$, $x \in \Sigma^*$), we cannot descend to a child node v'_r (with preorder j_2 in *RevTrie* and representing string $axyz$, for $y, z \in \Sigma^*$, such that $|yz| > 0$). Let $e_{v_r v'_r}$ be the edge between nodes v_r and v'_r, with label yz, where $y = (P^r)[i_3 + 1..i_2 - 1]$ and $(P^r)[i_2] \neq z_1$. Hence there are no phrases ending with $P[1..m - i_1]$.

Then, we go on to consider the next suffix $P^r[1..m - i_1 - 1]$. To reuse the work done up to node v_r (i.e., $(P^r)[i_1..i_3] = ax$), we follow the suffix link to get the node $\varphi(j_1)$ representing string x, setting $i_1 \leftarrow i_1 + 1$.

Retraverse. We have reused the work up to x, but we had actually worked up to xy. Notice that suffix xy exists for sure in *RevTrie*, yet it could be represented by an empty unary node which has been compressed in an edge. Therefore, from node $\varphi(j_1)$ we descend using $y = (P^r)[i_3 + 1..i_2 - 1]$. The edge $e_{v_r v'_r} = yz$ could be split into a path of several nodes between nodes $\varphi(j_1)$ and $\varphi(j_2)$. As substring y has been already checked in the previous step, the descent from node $\varphi(j_1)$ is done by skipping and checking only the first symbols of the edge labels (advancing i_3 accordingly as we reach new nodes). If being at node v''_r and trying to descend to the next node we find an empty-unary node, we directly jump to the position of the next non-empty-unary node (with preorder j_3) and then compute the length l of the string represented by that node.

For this direct jump we need a bit vector E marking the empty-unary nodes, in preorder. We preprocess E with a data structure for *rank* and *select* queries, so this requires $n' + o(n')$ extra bits. The node with preorder j_3 can be found by using *rank* and *select* on E. The length l can be computed as the sum of the length of the current node plus $n_e \cdot \log u$, where n_e is the number of empty-unary nodes between the current node and the one with preorder j_3 (which can be computed as the number of `1`s between the corresponding positions in E), and $\log u$ comes from the skips of empty-unary nodes (recall Section 4.1.2). In case that $l > |xy|$, we resume the *suffix* mode from v''_r. Otherwise we stay in *retraverse* mode from the node with preorder j_3. This process is carried out up to fully consuming string y, and then we resume the *descend* mode from the corresponding node.

After we find the first suffix $P^r[1..i]$ in *RevTrie* (if any), we are sure that every suffix of it also exists in *RevTrie* (because this trie is suffix-closed). The nodes corresponding to these suffixes are found by following suffix links.

Lemma 5.9. *Given a string P of length m, we can search for strings $P^r[1..i]$, for $1 \leqslant i < m$, in RevTrie in $O(m)$ time.*

Proof. Consider the method just described. Indices i_1, i_2, and i_3 grow from 1 to at most m. For every constant-time action we carry out, at least one of those indexes increases. Thus the total work is $O(m)$. □

Therefore, we have proved:

Theorem 5.3. *Given a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, there exists a compressed full-text self-index requiring $(3+\epsilon)uH_k(T)+o(u\log\sigma)$ bits of space, for $\sigma = O(\text{polylog}(u))$, any $k = o(\log_\sigma u)$, and any $0 < \epsilon < 1$. Given a search pattern $P[1..m]$, this index is able to:*

(1) report the occ occurrences of pattern P in text T in $O((m+\frac{occ}{\epsilon})\log u)$ worst-case time;

(2) count pattern occurrences in $O(m)$ worst-case time; and

(3) determine whether pattern P exists in T in $O(m)$ worst-case time.

5.4 Optimal Displaying of Text Substrings

5.4.1 Reporting Text Positions with LZ-index

As we said before, LZ-index is able to report occurrences in the format $[\![t, o]\!]$, where t is the phrase in which the occurrence starts and o is the distance between the beginning of the occurrence and the end of the phrase, and therefore so are our indexes of Sections 5.1, 5.2, and 5.3. However, we can report occurrences as *text positions* by adding a bit vector $TPos[1..u]$ that marks the n phrase beginnings. Given a text position i, then $rank_1(Tpos, i)$ is the phrase number i belongs to. Given a phrase identifier j, $select_1(TPos, j)$ yields the text position at which the j-th phrase starts. Therefore, given an occurrence in the format $[\![t, o]\!]$, the text position for that occurrence can be computed as $select_1(TPos, t+1) - o$.

Such *TPos* can be represented with $uH_0(TPos) + o(u) \leqslant n\log\frac{u}{n} + o(u) \leqslant \frac{u\log\log u}{\log_\sigma u} + o(u) = o(u\log\sigma)$, see Lemma 2.4 (2) and Lemma 2.2.

The algorithm for `extract` queries (of whole LZ78 phrases) described in Section 3.3.5 can be also used on the indexes of Theorems 5.1, 5.2, and 5.3, yet this time providing the text positions from where to extract (rather than the phrase identifiers), since these positions can be transformed into phrase identifiers by using data structure *TPos*. As the *Node* data structure is simulated by using ids^{-1}, it takes $O(\ell(1+\frac{1}{\epsilon\log_\sigma \ell}))$ time to extract any text substring of length ℓ, since we perform ℓ *parent* operations to get the ℓ symbols we want to display, and we must pay $O(1/\epsilon)$ time to use ids^{-1} each time we go on to extract the next phrase, which in the (very) worst case is done $O(\ell/\log_\sigma \ell)$ times.

To extract the text with *xbw*-based LZ-index of Section 5.2, we use *TPos* to transform the text positions into phrase identifiers, and then we use ids^{-1} to find the preorder position of the corresponding phrase, to finally map to the *xbw* representation of *LZTrie* by using Pos^{-1} in $O(1/\epsilon)$ time. Then we move to the parent in the *xbw*, displaying the

corresponding symbol stored in S_α. When we reach the tree root, we use ids^{-1} again to consider the next phrase, and map to the *xbw* again. The time is therefore $O(\ell(1+\frac{1}{\epsilon \log_\sigma \ell}))$.

The restriction of displaying only whole phrases can be avoided by adding a data structure for *level-ancestor* queries on *LZTrie*. The data structure defined in [JSS07b] builds on DFUDS, allows constant time computation of level-ancestor queries, and requires $o(n)$ extra bits of space. Thus, the part of a phrase that we do not need to display is skipped by using the appropriate level-ancestor query. Yet, the displaying time is not optimal, since we work $O(1)$ per extracted symbol and on a RAM we are able to handle $\Theta(\log u)$ bits per access, which means $\Theta(\log u/\log \sigma) = \Theta(\log_\sigma u)$ symbols per access.

5.4.2 Achieving Optimal Displaying Time

Instead we will describe a technique that can be plugged to any of the indexes proposed in Sections 5.1, 5.2, and 5.3, and those of Chapter 4, for displaying any text substring $T[i..i+\ell-1]$, in optimal $O(1+\ell/\log_\sigma u)$ time. A compressed data structure [GS06] to display any text substring of length $\Theta(\log_\sigma u)$ in constant time, turns out to have similarities with LZ-index. We take advantage of this similarity to plug it within our indexes, with some modifications, and obtain improved time to display text substrings. In [GS06], they added auxiliary data structures of $o(u \log \sigma)$ bits to *LZTrie* to support this operation efficiently. Given a position i of the text, we first find the phrase including the position i by using $rank_1(TPos, i)$, then find the node of *LZTrie* that corresponds to the phrase using *Node* (that is, the corresponding implementation of it). Then displaying a phrase is equivalent to outputting the path going from the node to the root of *LZTrie*. The auxiliary data structure, of size $O(n \log \sigma) = o(u \log \sigma)$ bits, permits outputting the path by chunks of $\Theta(\log_\sigma u)$ symbols in $O(1)$ time per chunk. As explained before, we can also display not only whole phrases, but any text substring within this complexity. Thus the displaying can start backwards from anywhere in a phrase, and of course it can stop at any point as well.

We modify this method to plug it into our indexes. In their original method [GS06], if more than one consecutive phrases have length less than $(\log_\sigma u)/2$ each, their phrase identifiers are not stored. Instead the substring of the text including those phrases are stored without compression. This guarantees efficient displaying operation without increasing the space requirement. However this will cause the problem that we cannot find patterns including those phrases. Therefore in our modification we store, for those short phrases, both the phrases themselves and their phrase identifiers. The search algorithm remains as before. To decode short phrases we can just output the explicitly stored substring including the phrases. For each phrase with length at most $(\log_\sigma u)/2$, we store a text substring of length $\log u$ containing the phrase. Because there are at most $O(\sqrt{u})$ such phrases in the text (recall that all LZ78 phrases are different), we can store all these

substrings in $O(\sqrt{u}\log u) = o(u)$ bits. These auxiliary structures work as long as we can convert a phrase identifier into a preorder position in $LZTrie$ (that is, compute ids^{-1}). Hence they can be applied to all the data structures in Sections 5.1, 5.2, and 5.3.

Theorem 5.4. *The indexes of Theorem 5.1, Theorem 5.2 and Theorem 5.3 (and also thar of Section 5.2) can be adapted to extract a text substring of length ℓ surrounding any text position in optimal $O(1+\frac{\ell}{\epsilon \log_\sigma u})$ worst-case time, using only $o(u\log\sigma)$ extra bits of space, for any $0 < 1 < \epsilon$.*

5.5 Handling Larger Alphabets

For simplicity, throughout this chapter we have assumed $\sigma = O(\text{polylog}(u))$, or equivalently $\log\sigma = O(\log\log u)$. Here we study the cases $\log\sigma = o(\log u)$ and $\log\sigma = \Theta(\log u)$.

5.5.1 The Case $\log\sigma = o(\log u)$

As long as $\log\sigma = o(\log u)$ holds, we can still have $k = o(\log_\sigma u) > 0$, while it also holds that $n\log n = uH_k(T) + o(u\log\sigma)$ [KM99]. Therefore, the space requirements of the indexes of Theorems 5.1 to 5.3 stay the same.

Index of Section 5.1. The data structure of Lemma 2.5 (1), which we use to represent *letts* and array S_W, has a time complexity of $O(\frac{\log\sigma}{\log\log u})$ for *rank* and *select* queries; thus, we lose the constant time for operations $child(x,\alpha)$ and $\varphi'(x,\alpha)$ on the tries, which would increase the time complexity of the whole index. Yet, we can represent *letts* with the (more complicated) data structure used in [BDM$^+$05], thus ensuring constant time for $child(x,\alpha)$ for any σ and retaining the same time complexity in our theorems. In the case of S_W we can use the scheme used to represent array S_α of the *xbw* in [FLMM05], in this way achieving constant time to compute *rank* over S_W, and requiring $n\log\sigma + o(u\log\sigma)$ bits of space. None of the remaining data structures of the index are affected by the alphabet size. As a result, Theorem 5.2 can be extended for the case $\log\sigma = o(\log u)$, rather than only for $\sigma = O(\text{polylog}(u))$.

Index of Section 5.2. The times for the operations on the *xbw* representation of *LZTrie* are affected by the alphabet size, depending on the representation used for S_α. If we use the data structure of Lemma 2.5 (2), occurrences of type 1 are found in $O(m\log\log\sigma + \frac{occ}{\epsilon})$ time, because of the subpath query we perform on *LZTrie*; occurrences of type 2 are found in $O(m^2\log\log\sigma + \frac{m}{\epsilon} + (m+occ)\log n)$ time, where the first term comes from searching for the $m-1$ partitions of P in *xbw*; and occurrences of type 3 are found in $O(m^2\log\log\sigma + \frac{m^2}{\epsilon})$, where the first term comes from searching for the $O(m^2)$ pattern substrings in the *xbw* representation of *LZTrie*. Overall, the time for `locate` is $O(m^2(\frac{1}{\epsilon} +$

$\log\log\sigma) + (m + occ)\log u)$. We can also replace $O(\log\log\sigma)$ for $O(\frac{\log\sigma}{\log\log u})$ in all these figures. Alternatively, we can use for S_α the representation given in [FLMM05], which allows constant time *rank* and *select* over S_α (though doubling the space for this array, since the data structure does not provide operation *access*). In this way, we retain the original complexities.

Index of Section 5.3. For this index the only affected part is the Alphabet-Friendly FM-index, AF-FMI(T), which still has a space requirement of $uH_k(T) + o(u\log\sigma)$ bits. The counting time is increased to $O(m(1+\frac{\log\sigma}{\log\log u}))$. Thus, the time for `locate` of this version of LZ-index now becomes $O(m(1+\frac{\log\sigma}{\log\log u})+(m+\frac{occ}{\epsilon})\log u)$, which is still $O((m+\frac{occ}{\epsilon})\log u)$, the same as stated by Theorem 5.3, since $m\frac{\log\sigma}{\log\log u} = O(m\log u)$. The counting time, on the other hand, now becomes $O(m(1+\frac{\log\sigma}{\log\log u}))$. Thus, we have a more general version of Theorem 5.3:

Theorem 5.5. *Given a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, there exists a compressed full-text self-index requiring $(3+\epsilon)uH_k(T) + o(u\log\sigma)$ bits of space, for any $k = o(\log_\sigma u)$, any $0 < \epsilon < 1$, and such that $\log\sigma = o(\log u)$. Given a search pattern $P[1..m]$, this index is able to:*

(*1*) *report the occ occurrences of pattern P in text T in $O((m+\frac{occ}{\epsilon})\log u)$ worst-case time;*

(*2*) *count pattern occurrences in $O(m(1+\frac{\log\sigma}{\log\log u}))$ worst-case time;*

(*3*) *determine whether pattern P exists in T in $O(m(1+\frac{\log\sigma}{\log\log u}))$ worst-case time; and*

(*4*) *extract any text substring of length ℓ in $O(\ell/(\epsilon\log_\sigma u))$ worst-case time.*

5.5.2 The Case $\log\sigma = \Theta(\log u)$

For the case $\log\sigma = \Theta(\log u)$, because of Lemma 2.1 we have that $n\log n = uH_k(T) + O(u(1 + k\log\sigma))$ bits of space, which is $\Theta(u\log\sigma)$ even for $k = 1$. Thus, high-order compression is lost. For $k = 0$ the space is $uH_0(T) + o(u\log\sigma)$ bits of space, so zero-order compression is retained. On the other hand, all the time complexities obtained for the case $\log\sigma = o(\log u)$ are valid for this larger σ.

However, it has been shown that the empirical-entropy model is not so adequate for such a large alphabet [Gag06].

5.6 Final Comments

We have found affirmative answers for Question 5.1, showing that the space of LZ-index can be squeezed to $(1+\epsilon)uH_k(T)+o(u\log\sigma)$ bits, for any $k=o(\log_\sigma u)$ and any $0<\epsilon<1$, with $O(m^2/\epsilon)$ average-case search time if $m \geqslant 2\log_\sigma n$. This space approaches as much as desired the optimal $uH_k(T)$ under the k-th order empirical entropy model for all k. However, this index does not provides worst-case guarantees at search time.

Then, we also found an affirmative answer for Question 5.2, achieving $(2+\epsilon)uH_k(T)+o(u\log\sigma)$ bits of space, for any $k=o(\log_\sigma u)$ and any $0<\epsilon<1$. This is about half the space the original LZ-index needs to operate [Nav04]. Moreover, we also improved the search time of the LZ-index, achieving stronger compressed self-indexes based on the Lempel-Ziv compression algorithm [ZL78]: Our indexes are able to search for the *occ* occurrences of a pattern $P[1..m]$ in T in $O(\frac{m^2}{\epsilon}+(m+occ)\log u)$ worst-case time, as well as extracting any text substring of length ℓ in optimal $O(\frac{\ell}{\epsilon\log_\sigma u})$ time. Thus, we achieve the same locate time as the index of Kärkkäinen and Ukkonen [KU96a] (see Lemma 3.1), yet with a much smaller index that does not need the text to operate.

In Table 1.1 (see page 9) we summarize the space and time complexities of some of the best existing compressed self-indexes (other less competitive ones are ignored [NM07]). We conclude that ours are the smallest existing compressed self-indexes based on Lempel-Ziv compression. As we argued in previous chapters, operations `extract` and `display` are the most useful in most practical scenarios. As it can be seen from the table, our LZ-indexes are superior in this respect. Total locate times in the table are after counting the pattern occurrences. The fast locating ($O(\log u)$ time per occurrence found) is also a strong point of our structure. Other data structures achieving the same or better complexity for locating occurrences either are of size $O(uH_0(T))$ bits plus a non-negligible extra space of $O(u\log\log\sigma)$ [Sad03], or they achieve this locating time for constant-size alphabets [FM05]. Finally, the CSA of Grossi, Gupta, and Vitter [GGV03] requires $\epsilon^{-1}uH_k(T)+o(u\log\sigma)$ bits of space, with a locating time of $O((\log u)^{\frac{\epsilon}{1-\epsilon}}(\log\sigma)^{\frac{1-2\epsilon}{1-\epsilon}})$ per occurrence, after a counting time of $O(\frac{m}{\log_\sigma u}+(\log u)^{\frac{1+\epsilon}{1-\epsilon}}(\log\sigma)^{\frac{1-3\epsilon}{1-\epsilon}})$, where $0<\epsilon<1/2$ is a constant. When ϵ approaches $1/2$, the space requirement approaches (from above) $2uH_k(T)+o(u\log\sigma)$ bits, with a counting time of $O(\frac{m}{\log_\sigma u}+\frac{\log^3 u}{\log\sigma})$ and a locating time per occurrence of still $\omega(\log u)$.

We also showed how to use an LZ-index to achieve $O((m+occ)\log u)$ time to locate the pattern occurrences, requiring $(3+\epsilon)uH_k(T)+o(u\log\sigma)$ bits of space. This is about half the space required by other LZ-indexes having the same search time [RO07] (see also Lemma 3.5).

Overall, we have achieved LZ-indexes with space requirements ranging from $(1+\epsilon)$ to $(3+\epsilon)$ times the size of the compressed text (plus lower-order terms), with different

achievements in the time complexities according with the space requirement of the index. These indexes are very competitive with state-of-the-art indexes, both in time and space requirement.

Chapter 6

Space-Efficient Construction of Lempel-Ziv Text Indexes

Many works on compressed full-text self-indexes do not consider the space-efficient construction of the indexes. Yet, this aspect becomes crucial when implementing the index in practice. For example, the original construction of *Compressed Suffix Arrays* (CSA) [GV05, Sad03] and *FM-index* [FM05] involves building first the suffix array of the text, using for example the algorithm of Larsson and Sadakane [LS99] or the one by Manzini and Ferragina [MF04]. Similarly, Navarro's LZ-index is constructed over a non-compressed intermediate representation [Nav04]. In both cases one needs in practice about 5 times the text size (in the case of CSA and the FM-index, by using the deep-shallow algorithm [MF04]). For example, the Human Genome (of 3×10^9 base pairs) may fit in less than 1 GB of main memory using these indexes (and thus it can be operated entirely in RAM on a desktop computer), but 15 GB of main memory is needed to build the indexes! Using secondary memory for the construction is nowadays the most practical alternative in the case of suffix arrays [DKMS08]. In this chapter we are interested in indexing algorithms for the LZ-index, that work directly in compressed space in main memory. The idea is to perform as much as we can of the indexing process in main memory.

Model of Computation. In this chapter we assume the standard *word* RAM model of computation, just as stated in Section 2.1. However, we make some extra assumptions here. Usually, after an indexing algorithm builds a text index in main memory, the index is stored on disk along with the text database, for persistence purposes. In the case of compressed self-indexes, the index by itself represents the database. At query time, the index is loaded into main memory in order to answer (many) user queries. Thus, by saving the index the (usually costly) indexing process is amortized over several queries. Yet, in other scenarios, one builds the index in main memory and answers queries on the fly.

We will initially assume that there is enough main memory to hold the final index. Later we will consider reduced-main-memory scenarios, where we will resort to secondary memory to hold the intermediate results. In this case, as the final index must reside on disk, we will assume that there is enough secondary memory to hold the index we build.

Since, depending on the scenario, we might or might not have to read the text from disk, and we might or might not have to write the final index to disk, and because those costs are fixed, we will not mention them. Yet, in the reduced-main-memory scenarios we will use the disk to read/write intermediate results, and in this case we will also consider the amount of extra I/O performed.

When accessing the disk, we assume the standard model [Vit08] where a disk page of B bits can be transferred to/from secondary storage with each access.

Finally, the space required by the text is not accounted for in the space required by the indexing algorithms. If it resides on disk one can process it sequentially so it does not require any significant main memory. Moreover, in most of our algorithms one could erase the text at an early stage of the construction process.

6.1 Related Work

Next we review related work dealing with the space-efficient construction of text indexes:

— An important research path is to try building the suffix array directly in compressed space in main memory. Hon et al. [HSS03a] present an algorithm to construct suffix arrays (and also suffix trees) using $O(u \log \sigma)$ bits of storage, in $O(u \log \log \sigma) = o(u \log u)$ time for suffix arrays, and $O(u \log^{\epsilon} u)$ time for suffix trees, where $0 < \epsilon < 1$. From this they derive an alternative algorithm to construct the CSA and the FM-index using $O(u \log \sigma)$ bits of storage and $O(u \log \log \sigma)$ time, in the case of FM-index assuming $\log \sigma = o(\log u)$. However, the space requirement to construct the CSA is still bigger than the space needed by the final index.

— The works of Lam et al. [LSSY02] and Hon et al. [HLSS03, HLS$^+$07] deal with the space (and time) efficient construction of CSA. The former work presents an algorithm that uses $(2H_0(T)+1+\epsilon)u+o(u \log \sigma)$ bits of space to build the CSA, where ϵ is any positive constant; the construction time is $O(\sigma u \log u)$, which is good enough if the alphabet is small (as in the case of DNA sequences), but may be impractical in the case of larger alphabets such as proteins and Oriental languages. The second work [HLS$^+$07] addresses this problem by requiring $(H_0(T)+2+\epsilon)u+o(u \log \sigma)$ bits of space and $O(u \log u)$ time to build the CSA. Also, they show how to build the FM-index from CSA using negligible extra space in $O(u)$ time. In practice they are able to construct the CSA for the Human Genome in about 24 hours and requiring

about 3.6 GB of main memory [Hon04], on a 1.7 GHz CPU. The FM-index can be constructed from the CSA in about 4 extra hours, for a total of about 28 hours.

— Finally, Na and Park [NP07] construct the CSA in $O(u \log \sigma \log_\sigma^\epsilon u)$ bits of space and $O(u)$ time, for $\epsilon = \log_3 2$. This is the most space-efficient linear-time algorithm for constructing the CSA. They leave open, however, the question of whether the CSA can be constructed in linear time and requiring $O(u \log \sigma)$ bits of space.

As it can be noticed, many works study the space-efficient construction of the CSA and the FM-index. However, the space-efficient construction of LZ-indexes has not been addressed in the literature. As we have shown in previous chapters, the LZ-indexes are competitive for locating pattern occurrences and extracting text substrings. Since this is very important in the scenario of self-indexes, the space-efficient construction of LZ-indexes is also an important issue.

In this chapter we present an efficient algorithm to construct the LZ-index using little space. We look for an indexing algorithm requiring at most the same space the final LZ-index needs to operate (within a lower-order additive term).

6.2 Space-Efficient Construction of the LZ-index

As we have seen, the LZ-index is a compressed full-text self-index, and as such it allows large texts to be indexed and stored in main memory. However, the construction process requires a large amount of main memory, mainly to support the pointer-based tries used to build the final versions of *LZTrie* and *RevTrie* (recall Section 3.3.4). So our problem is: given a text $T[1..u]$ over an alphabet of size σ, construct the LZ-index for T using as little space as possible and within reasonable time.

In this chapter we aim at an efficient algorithm to build those tries in little memory, by replacing the pointer-based tries with space-efficient data structures that support insertions. These can be seen as hybrids between pointer-based tries and the final succinct representations. The space-efficient construction algorithm for LZ-index presented in [AN05] has a construction time of the form $O(\sigma u)$. This makes the construction algorithm impractical for moderately-large alphabets. In the sequel we shall achieve $O(u(\log \sigma + \log \log u))$ time by using an improved dynamic representation.

In Sections 6.2.1 to 6.2.5 we assume that we have enough main memory to store the final LZ-index. A very important aspect is that of the management of the available memory done by our algorithm. In Section 6.2.6 we study how to manage the memory dynamically, using a standard model of memory allocation [RR03] . In Section 6.3, we shall adapt our algorithm to the cases in which there is no enough space to store the whole final index in main memory.

We show next how to space-efficiently construct the LZ-index components. From now on we assume $\sigma \geqslant 2$, as otherwise the whole indexing problem is trivial.

6.2.1 Space-Efficient Construction of $LZTrie$

The space-efficient construction of *LZTrie* is based on a compact representation supporting a fast incremental construction as we traverse the text. In either the BP and DFUDS representations (see Lemma 2.10 and Lemma 2.11 respectively), the insertion of a new node at any position of the sequence implies to rebuild the sequence from scratch, which is expensive. To avoid this we define a *hierarchical representation*, such that we rebuild only a small part of the entire original sequence upon the insertion of a new node.

We incrementally cut the trie into disjoint *blocks* such that every block stores a subset of nodes representing a connected component of the whole trie. We arrange these blocks in a tree by adding some *inter-block* pointers, and thus the entire trie is represented by a tree of blocks.

If a node x is a leaf of a block p, but is not a leaf of the whole trie, then node x stores an inter-block pointer to the representation of its subtree. Let us say that this pointer is pointing to block q. We say that q is a child block of p. In our representation, node x is also stored in block q, as a fictitious root node. Thus, every block is a tree by itself, which shall simplify the navigation on our representation, as well as the management of each block.

To summarize, every such node x has two representations: (1) as a leaf in block p; (2) as the root node of block q. Note that the number of extra nodes introduced by duplicating nodes equals the number of blocks in the representation (minus one), and also that we are enforcing that every node is stored in the same block of its children, which also means that sibling nodes are all stored in the same block.

Rather than using a static representation for the blocks of the tries [AN05], which are rebuilt from scratch upon insertion of new nodes, we represent each block by using dynamic data structures, which can be updated in time less than linear in the block size. We adapt the approach used in [Arr08] to represent succinct dynamic σ-ary trees: We first reduce the size of the problem by dividing the trie into small blocks, and then represent every block (i.e., smaller trie) with a dynamic data structure to avoid the total rebuilding of blocks upon trie updates.

Defining Block Sizes. We divide the *LZTrie* into blocks of N nodes each, where $N_m \leqslant N \leqslant N_M$, for minimum block size $N_m = \Theta(\log^2 u)$ nodes and maximum block size $N_M \geqslant 2\sigma N_m$ nodes. We also need $N_M = (\sigma \log u)^{O(1)}$, for example $N_M = \Theta(\sigma \log^3 u)$ (we do not show the rounding in the block sizes, but it should be clear that these must be integers).

In this way, notice that we shall have one inter-block pointer out of at least N_m nodes. Since each pointer is represented with $\log u$ bits, and since we have n nodes in the tree, we have at most $\frac{n}{N_m} \log u = O(n/\log u)$ bits overall for inter-block pointers.

The definition of N_M, on the other hand, is such that it ensures that a block p has room to store at least the potential σ children of the block root (recall that sibling nodes must be stored all in the same block). Also, when we insert a node in a block of maximal size N_M (i.e., the block overflows), we should be able to split the block into two blocks, each of size at least N_m. By defining N_M as we do, in the worst case (i.e., the case where the overflown block has the smallest possible size) the root of the block has some child with at least N_m nodes, as $N_M \geqslant 1 + \sigma N_m$. Thus, upon an overflow, we can create a new block of size at least N_m from such subtree, requiring little space for inter-block pointers and maintaining the properties of our data structure. The stricter factor 2 shall be useful for our amortized analysis of block partitioning, whereas the polylog upper bound is necessary to ensure short enough pointers within blocks, and because our operation times will be $O(\log N_M)$.

Defining the Block Layout. Each block p of N nodes consists of (we will explain later how to represent each block component):

— The representation T_p of the topology of the block, using any suitable tree representation.

— A bit-vector $F_p[1..N]$ (the *flags*) such that $F_p[j] = \texttt{1}$ iff the j-th node of T_p (in preorder) has associated an inter-block pointer. We shall represent F_p by using a data structure supporting *rank* and *select* queries, requiring $N + o(N)$ bits.

— $\log N_M$ bits to count the current number N of nodes stored in the block.

— The sequence $ids_p[1..N]$ of LZ78 phrase identifiers for the nodes of T_p, in preorder. Except for the *LZTrie* root, every block root is replicated as a leaf in its parent block, as explained. In that case we store the corresponding phrase identifier only in the leaf of the parent block. That is, fictitious roots in each block do not store phrase identifiers. We use $\log u$ bits per phrase identifier, instead of using $\log n$ bits as in the final representation of *ids*. This is because, before constructing the LZ78 parsing of the text, we do not know n, the number of phrase identifiers.

— The symbols ($letts_p$) labeling the edges in the block (the order of the symbols depends on the representation used for T_p, recall Section 2.5.2). Each symbol uses $\log \sigma$ bits of space.

— A variable number of inter-block pointers, stored in data structure ptr_p. The number of inter-block pointers varies from 0 to N, and it corresponds to the number of flags with value 1 in F_p.

Example 6.1. In Fig. 6.1 we show an example of hierarchical representation of *LZTrie* for the running example text, assuming that BP is used to represent the trie topology of each block.

If the subtree of the j-th node (in preorder) of block p is stored in block q, then q is a child block of p and the j-th flag in p has the value 1. If the number of flags with value 1 before the j-th flag in p is h, then the h-th inter-block pointer of p points to q. Note that h can be computed as $rank_1(F_p, j)$.

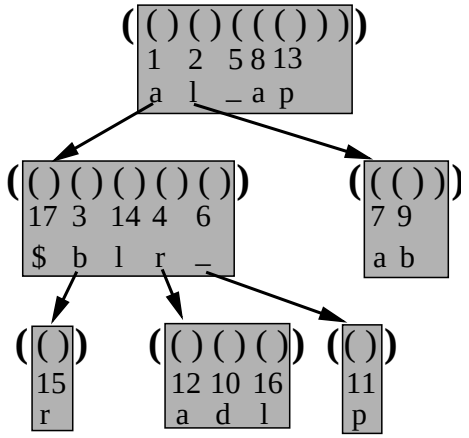


Figure 6.1: Hierarchical representation of the *LZTrie* of Fig. 2.3(c). The trie topology inside the blocks is represented with BP. Nodes having an inter-block pointer are duplicated as the root of the child block, and shown outside each block.

Since blocks are tries by themselves, inside a block p we use the traditional trie-like descent process, using operation $child_p(x, \alpha)$ on T_p. From now on we use the subscript p with the trie operations, to indicate operations which are local to a block p, i.e., disregarding the inter-block structure (e.g., $preorder_p$ computes the preorder of a node within block p, and not within the whole trie, and so on). When we reach a block leaf (with preorder j inside the block), we check the j-th flag in p. If $F_p[j] = 1$ holds in that block, we compute $h = rank_1(F_p, j)$ and follow the h-th inter-block pointer in p to reach the corresponding child block q. Then we follow the descent inside q as before. Otherwise, if $F_p[j] = 0$, then we are in a leaf of the whole trie, and we cannot descend anymore.

We represent the above components for block p in the following way.

Representation of the Trie Topology, T_p. To represent the trie topology of block p we use the data structure for dynamic balanced parentheses of [CHLS07] to represent the DFUDS of the block. The main idea of Chan et al. is to divide the original parentheses sequence into segments S_i of $O(\log N)$ bits, which in our case also means $O(\log N)$ nodes per segment (by identifying each node with its first parenthesis). Every segment S_i is stored in the leaves of a balanced binary tree T'_p, such that concatenating the leaves from left to right gives us back the original sequence T_p.

Some information is stored in the internal nodes of T'_p in order to support the operations on the parentheses sequence, as well as support insertions and deletions of *pairs of matching parentheses*. All the operations of Section 2.5.2 on balanced-parentheses sequences are supported in $O(\log N)$ time by navigating T'_p. In addition, we store in every internal node of T'_p the number of opening parentheses within the left subtree, as well as the total number of parentheses within the left subtree, such as in [MN08b], in order to support operations $rank_($, $rank_)$, $select_($, and $select_)$ over T_p in $O(\log N)$ time.

All these operations on the sequence of parentheses allow us to support the DFUDS operations (recall Section 2.5.2): $parent_p$, $child_p(x, i)$, $subtreesize_p$, $degree_p$, $preorder_p$, $selectnode_p$, etc., all of them in $O(\log N) = O(\log N_M)$ time. As we shall explain later in this section, the insertion of a new node in DFUDS can be simulated by inserting a new pair of matching parentheses in T_p, and thus we can handle it in a straightforward way with the data structure of [CHLS07]. Deletions of leaves are handled in a similar way. The space requirement is $O(N)$ bits per block, which adds up to $O(n) = o(u)$ bits overall[1].

Representation of the Flags, F_p. We represent the flags of block p in preorder and using a dynamic data structure for *rank* and *select* over a binary sequence [MN08b]. This data structure supports *rank*, *select*, and updates on F_p in $O(\log N)$ worst-case time, and requires $N + o(N)$ bits of space. This data structure can be connected with T_p via operations $preorder_p$ and $selectnode_p$: Given a node x in p, the corresponding flag is $F_p[preorder_p(x)]$. Given $F_p[j]$, on the other hand, the corresponding node in T_p is $selectnode_p(j)$. When we insert a new node in T_p, we insert a new flag (with value `0` because the new node is inserted with no related inter-block pointer) at the corresponding position (given by $preorder_p$). This data structure adds $n + o(n) = o(u)$ extra bits to our representation. A more involved representation for F_p, requiring $o(n)$ bits, is given in [Arr08], yet the one we are using here is simpler yet adequate for our purposes.

Representation of the Symbols, $letts_p$. We represent the symbols labeling the edges of the block according to a DFUDS traversal on T_p (see Section 2.5.2), yet this time we store

[1]The space requirement of the trie topology can be reduced to $2n + o(n)$ bits overall, see [Arr08]. However, $O(n)$ bits is sufficient for our purposes.

them in differential form, except for the symbol of the first child of every node, which is represented in absolute form. We then represent this sequence of N integers of $k' = \log \sigma$ bits each with the dynamic data structure for searchable partial sums of Lemma 2.6, which supports all the operations (including insertions and deletions) in $O(\log N)$ time, and requiring $Nk' + O(N) = N \log \sigma + O(N)$ bits of space (recall Lemma 2.6), adding overall $n \log \sigma + O(n) = o(u \log \sigma)$ extra bits of space. This is more efficient (both in time and extra sublinear space) than using a dynamic data structure supporting *rank* and *select* [GN08b].

We can connect $letts_p$ with T_p by using $rank_\iota$ over T_p. Given a node x in T_p, the subsequence $letts_p[rank_\iota(T_p, x)..rank_\iota(T_p, x) + degree_p(x) - 1]$ stores the symbols labeling the children of x. To support operation $child_p(x, \alpha)$, which shall be used to descend in the trie at construction time, we first compute $i \leftarrow rank_\iota(T_p, x)$ to obtain the position in $letts_p$ for the first child of x. We then compute $s \leftarrow Sum(letts_p, i - 1)$, which is the sum of the symbols in $letts_p$ up to position $i - 1$ (i.e., the sum before the first child of x). To compute the position of symbol α within the symbols of the children of node x, we perform $j \leftarrow Search(letts_p, s + \alpha)$. Thus, the node we are looking for is the $(j - i + 1)$-th child of x, which can be computed by $child_p(x, j - i + 1)$, in $O(\log N)$ time overall. To make sure j is a valid answer, we use operation $degree_p(x)$ to check whether $j - i + 1$ is smaller or equal to the degree of x, and then we check whether $Sum(letts_p, j - i + 1) - s = \alpha$ actually holds.

Representation of the Phrase Identifiers, ids_p. To store the phrase identifiers of the trie nodes, we define a *list* L_{ids_p} for block p, storing the identifiers in preorder. Given a new inserted node x in T_p, we must insert the corresponding phrase identifier at position $preorder_p(x)$ within L_{ids_p}, so we must support the efficient search of this position.

The linked-list functionality to represent L_{ids_p} is easily achieved by simplifying, for example, the dynamic partial sums data structure of Lemma 2.6, so that only accesses and insertions are permitted. For a list of N elements, this data structure is a balanced tree storing circular arrays of $\Theta(\log N)$ list elements at the leaves, and subtree sizes at internal nodes. It carries out all the operations in $O(\log N)$ time and poses an extra space overhead of $O(N)$ bits.

We need $N \log u + O(N)$ bits of space to maintain the identifiers, which adds up to $n \log u + O(n)$ bits overall. This is $uH_k(T) + o(u \log \sigma)$ bits of space according to Lemma 2.1. Recall that $N = O(N_M)$ in our case, and therefore the time to manipulate the list is $O(\log \sigma + \log \log u)$ per operation.

Representation of the Inter-Block Pointers, ptr_p. For the inter-block pointers, we use also a linked list L_{ptr_p}, managed in a similar way as for L_{ids_p}. Since blocks have

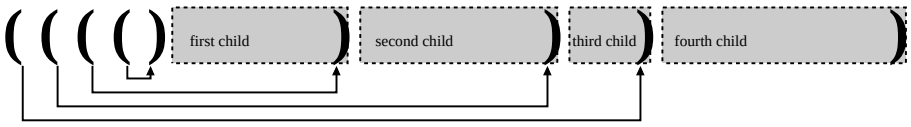
at least N_m nodes, we have at most one pointer for every $\Theta(\log^2 u)$ nodes, which adds $O(n/\log u) = o(u/\log u)$ bits overall.

Construction Process. The construction of *LZTrie* proceeds as explained in Section 3.3.3, using the symbols in the text to descend in the trie, until we cannot descend anymore. This indicates that we have found the longest prefix of the rest of the text that equals a phrase B_ℓ already in the phrase dictionary of LZ78. Thus, we form a new phrase $B_t = B_\ell \cdot c$, where c is the next symbol in the text, and then insert a new leaf representing this phrase. However, this time the nodes are inserted in a hierarchical representation of *LZTrie*, instead of a pointer-based trie.

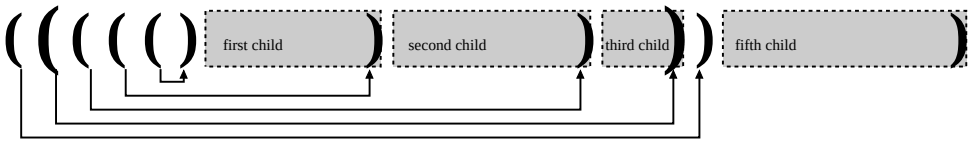
The insertion of a new node for the LZ78 phrase B_t in the trie implies to update only the block p in which the insertion is carried out. Assume that the new leaf must become the j-th node (in preorder) within the block p, and that the new leaf is a new child of node x in block p (i.e., node x represents phrase B_ℓ). We explain next how to carry out the insertion of the new leaf within the DFUDS of T_p.

We must insert a new '`(`' within the representation of x (which simulates the increase of the degree of node x, because of the insertion of the new child), and we must insert also a new '`)`' to represent the new leaf we are inserting. Assume that the new leaf will become the new i-th child of node x, assuming first that $i \leqslant degree(x)$ (i.e., the new node is not inserted as the last child of x). Therefore the new '`(`' must be inserted to the right of the opening parenthesis already at position $i' = x + degree(x) - i$ (recall from Section 2.5.2 how operation $child(x, i)$ uses the opening parentheses defining node x to descend to the i-th child). Then, the new '`)`' must be inserted at position $i'' = findclose(T_p, i' + 1)$, shifting to the right the last '`)`' in the subtree of the $(i-1)$-th child of x, which now becomes the new leaf. As a result, the two inserted parentheses form a matching pair, which can be handled in a straightforward way with the data structure of [CHLS07]. See Fig. 6.2 for an illustration.

If, on the other hand, the new node is inserted as the last child of x (i.e., $i = degree(x)+1$), notice that there is no opening parenthesis at position $i' = x + degree(x) - i = x - 1$, because that corresponds to the closing parenthesis finishing the representation of the previous node in preorder. So we use a different procedure instead. We need to find the position of the rightmost closing parenthesis in the subtree of x. In the example of Fig. 6.2(a), we must find the rightmost '`)`' in the subtree of the fourth child of node x. The new leaf '`)`' must be inserted at the right of this parenthesis. Notice that the rightmost closing parenthesis in the subtree of node x matches the opening parenthesis at position $i' = enclose(T_p, x)$. Then, if we add an opening parenthesis at position x, and a closing parenthesis at position $i'' = findclose(T_p, i')$, we will have a new pair of matching parentheses, which can be handled with the data structure of [CHLS07].



(a) A node x of degree 4 and its corresponding subtree in the DFUDS representation of the *LZTrie*. Notice the relation among the four opening parentheses in the definition of x and the subtrees of the children of node x.



(b) Insertion of a new child of node x. The new leaf is inserted as the new fourth child of x, and thus it is represented by the new bold pair of matching parentheses. Notice how the degree of x is increased to 5 with the new opening parenthesis. The last closing parenthesis in the subtree of the third child of x is shifted to the right and now represents the new inserted leaf.

Figure 6.2: Illustration of the insertion of a new leaf node in the DFUDS representation of *LZTrie*.

Then, we add a new flag `0` at position j in F_p. Also, c is inserted at the corresponding position within $letts_p$, and t is inserted at position j within the identifiers of block p (since these are stored in preorder). All this takes $O(\log N_M) = O(\log \sigma + \log \log u)$ time.

Managing Block Overflows. A *block overflow* occurs when, at construction time, the insertion of a new node must be carried out within a block p of N_M nodes. In such a case, we need to make room in p for the new node by selecting a subset of nodes to be copied to a new child block (of p) and then will be deleted from p. We explain this procedure in detail.

First we select a node z in p whose local subtree (along with z itself) will be copied to a new child block. In this way we ensure that a node and its children (and therefore all sibling nodes) are always stored in the same block (recall that a copy of z, as a leaf, will be kept in p).

Suppose that we have selected in this way the subtree of the j-th node (in preorder) in the block. Both the selected node z and its subtree are copied to a new block p', via insertions in $T_{p'}$. We must also copy to p' the flags $F_p[preorder_p(z)+1..preorder_p(z)+subtreesize_p(z)-1]$ (via insertions in $F_{p'}$) as well as the corresponding inter-block pointers within the subtree of the selected node z, which are stored in array ptr_p from position $rank_1(F_p, preorder_p(z))+1$ up to $rank_1(F_p, preorder_p(z)+subtreesize_p(z)-1)$.

Next we add in p a pointer to p'. The new pointer belongs to z, the j-th opening parenthesis in p (because we selected its subtree). We compute the position for the new pointer as $rank_1(F_p, j)$, adding the pointer at this position in L_{ptr_p}, and then we set to 1 the j-th flag in F_p, updating accordingly the $rank/select$ data structure for F_p (the portion copied to $F_{p'}$ must be deleted from F_p). Finally, we delete in p the subtree of z (via the corresponding deletions in T_p), leaving z as a leaf in p.

Thus, the reinsertion process can be performed in time proportional to the size of the reinserted subtree (times $O(\log N_M)$), by using the insert and delete operations on the corresponding dynamic data structures that form a block. However, we must be careful with the selection of node z, which can be performed in two different ways:

(1) Upon a block overflow, we traverse block p to select node z, which takes $O(N_M)$ time in the worst case; or

(2) we look for z in advance to overflows, as we perform the insertion of new nodes (using the insertion path to look for possible candidates).

We choose the latter option, since in this way we can obtain a good amortized cost for updates, as we will see later in our analysis.

To quickly select node z, we maintain in each block p a *candidate list* C_p [Arr08], storing the local preorders of the nodes that can be copied to a new child block p' upon block overflow. With *selectnode* we can obtain the candidate node corresponding to such a preorder. A subtree must have size at least N_m to be considered a candidate. Thus, after a number of insertions we will find that a node (within the insertion path) becomes a candidate. Let us think for a moment that we only maintain a candidate per block, and not a list of them. It can be the case that a few children of the block root have received (almost) all the insertions, so we have a few large subtrees within the block. When block p overflows, we reinsert the only candidate to a new child block, so we have no candidate anymore for p. We have to use the next insertions in order to find a new one. However, it can be also the case that different children of the root of p receive the new insertions, and hence block p could overflow again within a few insertions, without finding a new subtree large enough so as to be considered a candidate (recall that we just use the insertion path to look for candidates). By maintaining a list of candidates in each block, instead of a unique candidate per block, we can keep track of all the nodes in p whose subtree is large enough, avoiding this problem.

Since the preorder of a node within a block p can change after the insertion of a new node in p, we must update C_p in order to reflect these changes. In particular, we must update the preorders stored in C_p for all candidate nodes whose preorder is greater than that of the new inserted node. To perform these updates efficiently, we represent C_p using a searchable partial sum data structure (see Lemma 2.6). Thus, the original preorder $C_p[i]$

is obtained by performing $Sum(C_p, i)$ in $O(\log N)$ time. Let x be the new inserted node. Then, with $j = Search(C_p, preorder_p(x))$ we find the first candidate (in preorder) whose preorder must be updated, and we perform operation $Update(C_p, j, 1)$. In this way, we are increasing $C_p[j]$ by 1, and hence we are automatically updating all the preorders in C_p that have changed after the insertion of x, in $O(\log N)$ time overall.

If we keep track of every candidate of size at least N_m, every time p overflows there will be already candidate blocks. The reason is, again, that $N_M \geqslant 1 + \sigma N_m$, and thus that at least one of the children of the root must have size at least N_m, and this is a candidate.

Since we use the descent process to look for candidates, we will find them as soon as their subtrees become large enough. In other words, the subtree of a node becomes larger as we descend through the node many times to insert new nodes, until eventually finding a candidate.

We must also ensure that the space to maintain C_p is small (so we cannot have too many candidates). The size of the local subtree (i.e., only considering the descendant nodes stored in block p) of every candidate must be at least N_m. Also, we enforce that no candidate node descends from another candidate, in order to bound the number of candidates.

To maintain C_p, every time we descend in the trie to insert a new LZ78 phrase, we maintain the last node z in the path such that $subtreesize_p(z) \geqslant N_m$. When we find the insertion point of the new node x, say at block p, before adding z to C_p we first perform $p_1 = Search(C_p, preorder_p(z))$, and then $p_2 = Search(C_p, preorder_p(z) + subtreesize_p(z))$. Then, z is added to C_p whenever:

(1) z is not the root of block p, and

(2) there is no other candidate in the subtree of z (that is, $p_1 = p_2$ holds).

If in the descent we find a candidate node z' which is an ancestor of the prospective candidate z, then after inserting z to C_p we delete z' from C_p. In this way, we keep the lowest possible candidates, avoiding that the subtree of a candidate becomes too large after choosing it as a candidate, which would not guarantee a fair partition into two blocks of size between N_m and N_M upon an overflow. Because of Condition (2) above, there are one candidate out of (at least) N_m nodes; thus, the total space for C_p is $\frac{n}{N_m} \log N_M + O(\frac{n}{N_M})$ bits, which is $o(n/\log u)$.

If C_p becomes empty after solving an overflow, it can be the case that there are still valid candidates in the block. Notice that if this happens, there must be at least one ancestor of node z (the candidate node that was just moved to a new block upon the last overflow) which is a valid candidate. So we look for a valid candidate starting from node z and going successively to the parent node, until eventually finding the candidate. Notice

that we need to perform at most N_m *parent* operations until finding a candidate, because in this way we will arrive to a node that has at least N_m descendants. So the cost of looking for the candidate is subsumed by the cost of solving the overflow.

The reinsertion cost is in this way proportional to the size of p', since finding node z now takes $O(\log N_M)$ time (because of the partial-sum data structure used to represent C_p). Notice that the first time a node is reinserted, the reinsertion cost amortizes with the cost of the original insertion. Unfortunately, there are no bounds on the number of reinsertions for a given node. However, we shall show that multiple reinsertions of a node over time amortize with the insertion of other nodes. We use the following *accounting argument* [CLRS01] to prove the amortized cost of insertions. Let $\hat{c} = 2$ be the amortized cost of normal insertions (without overflows), being $c = 1$ the actual cost of an insertion. Therefore, every insertion spends one unit for the insertion itself, and reserves the remaining unit for future (more costly) operations. Let us think that we have separate reserves, one per block of the data structure. We shall prove that every time a block overflows, it has enough reserves so as to pay for the costly operation of reinserting a set of nodes.

In particular, every time a block overflows, its reserve is $N_M - I$, where I was the initial number of nodes for the block (notice that $I = 0$ holds *only* for the root block). Let I' be the number of nodes of the new block p'. Then we must prove that $N_M - I \geqslant I'$ always holds, that is, $N_M \geqslant I + I'$. We need the following lemma:

Lemma 6.1. *For every candidate node x in block p, it holds that $subtreesize_p(x) < \sigma N_m$.*

Proof. By maintaining the lowest possible candidates, we find the smallest possible ones. If a node cannot be chosen as a candidate, this means that its subtree size is smaller than N_m nodes (another possibility is that there is another candidate within the subtree, yet this case is not interesting here). Therefore, the smallest subtree that can be chosen as a candidate may have up to $N_m - 1$ nodes in each children, and hence its total size is at most $1 + \sigma(N_m - 1) < \sigma N_m$. □

Because of this, blocks are created with $I', I < \sigma N_m$ nodes. As we have chosen $N_M \geqslant 2\sigma N_m$, it follows that $N_M \geqslant I + I'$. This means that every reinsertion of a node has been already paid by some node at insertion time.[2] Thus, the insertion cost is $O(\log N_M)$ amortized. After n insertions, the overall cost amortizes to $O(n \log N_M) = O(n(\log \sigma + \log \log u))$.

Once we solved the overflow, the insertion of the new node is carried out either in p' or in p, depending whether the insertion point lies within the moved subtree or not, respectively. Notice that there is room for the new node in either block.

[2]More generally we could have set $N_M \geq (1 + \alpha)\sigma N_m$ for any constant $\alpha > 0$, and the analysis would have worked with $\hat{c} = 1 + 1/\alpha$.

Hierarchical *LZTrie* Construction Analysis. As the trie has n nodes, we need $O(n)+(n+o(n))+(n\log\sigma+O(n))+(n\log u+O(n))+o(n/\log n)+o(n/\log n)$ bits of storage to represent the trie topology, flags, symbols, identifiers, inter-block pointers, and candidate lists, respectively. Because of Lemma 2.1, the space requirement is $uH_k(T)+o(u\log\sigma)$ bits, for any $k=o(\log_\sigma u)$.

When constructing *LZTrie*, the *navigational cost* per symbol of the text is $O(\log N_M)=O(\log\sigma+\log\log u)$, for a total worst-case time $O(u(\log\sigma+\log\log u))$. On the other hand, the cost of rebuilding blocks after an insertion is $O(\log N_M)$ amortized, and therefore the total cost amortizes to $O(n(\log\sigma+\log\log u))=o(u(\log\sigma+\log\log u))$. Therefore, the total construction time is $O(u(\log\sigma+\log\log u))$.

Representing the Final *LZTrie*. Once we construct the same hierarchical representation for *LZTrie*, we delete the text since this is not anymore necessary, and then use the hierarchical *LZTrie* to build the final version of *LZTrie* in $O(n(\log\sigma+\log\log u))$ time. We perform a preorder traversal on the hierarchical tree, transcribing the nodes to a linear representation. Every time we copy a node, we check the corresponding flag, and then decide whether to descend to the corresponding child block or not. We also allocate $n\log\sigma=o(u\log\sigma)$ bits of space for the final array *letts*, and $n\log n$ bits for array *ids*.

Thus, the maximum amount of space used is $2uH_k(T)+o(u\log\sigma)$, since at some point we store both the hierarchical and final versions of *LZTrie*. We then free the hierarchical *LZTrie*, thus we end up with a representation requiring $uH_k(T)+o(u\log\sigma)$ bits.

Thus, we have proved:

Lemma 6.2. *There exists an algorithm to construct the LZTrie for a text $T[1..u]$ over an alphabet of size σ and with k-th order empirical entropy $H_k(T)$, in $O(u(\log\sigma+\log\log u))$ time and using $2uH_k(T)+o(u\log\sigma)$ bits of space, for any $k=o(\log_\sigma u)$.*

6.2.2 Space-Efficient Construction of *RevTrie*

For the space-efficient construction of *RevTrie*, we use the hierarchical technique of Section 6.2.1, to represent not the original reverse trie but its *Patricia tree* [Mor68], which compresses *empty* unary paths of the reverse trie, yielding an important saving of space. However, as we still maintain empty non-unary nodes, the number of nodes in the reverse trie is $n'\leqslant 2n$.

Throughout the construction process we store in the nodes of the reverse trie pointers to *LZTrie* nodes, instead of the corresponding phrase identifiers *rids* stored by the final *RevTrie*. Each pointer uses $\log 2n$ bits, since the *LZTrie* parentheses representation has $2n$ positions (either in BP or DFUDS representations, recall that *LZTrie* is already in final

static form). We store these pointers to *LZTrie* in the same way as for array ids_p in Section 6.2.1, in preorder according to *RevTrie* and spending $O(1)$ extra bits per element for the linked list functionality. The aim is to obtain the text of the phrase represented by a *RevTrie* node, since we are compressing empty-unary paths and the string represented by a node is not available otherwise (unlike what happens with the traditional Patricia trees). This connection is given by *Node* in the final LZ-index. However, at construction time we avoid accessing *Node* when building the reverse trie, so we can build *Node* after both tries have been built, thus reducing the maximum indexing space.

Empty non-unary nodes are marked by storing in each block p a bit vector B_p (represented in the same way as F_p, with a dynamic data structure supporting *rank* and *select* queries). We store pointers to *LZTrie* nodes only for non-empty *RevTrie* nodes, so we store n of them. This shall reduce the indexing space of the preliminary definition of the algorithm [AN05], which shall be useful later when constructing reduced versions of LZ-index, yet introducing some additional problems in our hierarchical representation, as we shall see below.

As we compress empty-unary paths, the edges of the trie are labeled with strings instead of single symbols. The Patricia tree stores only the first symbol of the string that labels the edge, using the same partial sum approach as for *LZTrie*. We store the Patricia-tree skips of every trie node in a linked list $skips_p$, in preorder and using the linked-list approach used for ids_p in *LZTrie*, using $\log\log u$ bits per node. To enforce this limit, we insert *empty* unary nodes when the skip exceeds $\log u$. In this way, one out of $\log u$ empty unary nodes could be explicitly represented. In the worst case there are $O(u)$ empty unary nodes, of which $O(\frac{u}{\log u})$ can be explicitly represented. This means $O(\frac{u}{\log u}(O(1) + \log\log u + \log\sigma)) = o(u\log\sigma)$ extra bits overall in the hierarchical representation (this is for the space of T_p, F_p, B_p, $skips_p$, and $rletts_p$, plus their overheads). Since we use a linked list for $skips_p$, it takes $O(\log N_M)$ time to find the skip corresponding to a given node.

Construction Process. To construct the reverse trie we traverse the final *LZTrie* in depth-first order, generating each LZ78 phrase B_i stored in *LZTrie*, and then inserting its reverse B_i^r into the reverse trie.

When searching for a given string s in *RevTrie*, we descend in the trie checking only the first symbols stored in the trie edges, using the skips to know which symbol of s to use at each node. When the longest possible prefix of string s is thus consumed, say upon arriving at node v_r of *RevTrie*, we must compare the string represented by v_r against s, in order to determine whether the prefix is actually present in *RevTrie* or not. To compute the string corresponding to node v_r we use the connection with the *LZTrie*: we follow the pointer to the corresponding *LZTrie* node, and go up to the *LZTrie* root, extracting the symbols in the upward path. However, we are storing some empty nodes in *RevTrie*, for which we do not store pointers to *LZTrie*.

Assume that node v_r in block p is empty, and represents string s'. Since every descendant of v_r has s' as a suffix, if we map to *LZTrie* from any of these descendants we would find string s' also by reading the upward path in *LZTrie* (we know the length of the string we are looking for, so we know when to stop going up in *LZTrie*). Notice that there exists at least one non-empty descendant v'_r of v_r since *RevTrie* leaves cannot be empty (because they always correspond to an LZ78 phrase). So we can use the *LZTrie* pointer of v'_r to find s'. Since we only store pointers for non-empty nodes, the pointer of v'_r can be found at position $rank_1(B_p, preorder_p(v_r)) + 1$ within the pointer array.

However, there exists an additional problem in our hierarchical representation: the local subtree of node v_r can be exclusively formed by empty nodes, in which case finding the non-empty node v'_r is not as straightforward as explained before, since v'_r is stored in a descendant block. This problem comes from the fact that, upon a block overflow in the past, we might have chosen empty nodes z descending from v_r, whose subtrees were reinserted into new blocks.

To solve this problem, we store in every block p a pointer to *LZTrie*, which is representative for the nodes stored in the block p. If a block is created from a non-empty node, then we can store the pointer of that node. In case of creating a new block p' from an empty node, if the new block p' is going to be a leaf in the tree of blocks, then it will contain at least a non-empty node. Thus, we associate with p' the pointer to *LZTrie* of this non-empty node. If, otherwise, p' is created as an internal node in the tree of blocks, then it can be the case that all of the nodes in p' are empty. In this case, we choose any of the descendants blocks of p' and copy its pointer to p'. This pointer has been "inherited" (in one or several steps) from a leaf block, thus this corresponds to a non-empty *RevTrie* node.

Thus, in case that the local subtree of v_r is formed only by empty nodes, we take one of the blocks descending from v_r (say the first in preorder) and use the *LZTrie* pointer associated to that block, in order to compute string s'.

An important difference with the *LZTrie* construction is that in *RevTrie* we do not necessarily insert new leaves: there are cases where we insert a new non-empty *unary* internal node (this is non-empty because it corresponds to the phrase we are inserting in *RevTrie*). Notice that in DFUDS, the representation of a unary node is '`()`', which is a matching pair and hence the insertion can be handled by the data structure of [CHLS07] representing T_p. If we insert the new node as the parent of an existing node x, then the insertion point is just before the representation of x in the DFUDS sequence.

Hierarchical *RevTrie* Construction Analysis. The hierarchical representation of the reverse trie requires $O(n') + (n' + o(n')) + (n' + o(n')) + (n \log 2n + O(n)) + (n' \log \sigma + O(n')) + (n' \log \log u + O(n')) + o(n'/\log n') + o(n'/\log n')$ bits of storage to represent the

trie topology, flags, bit vector of empty nodes, pointers to *LZTrie* stored in the nodes, symbols, skips, pointers (both inter-block and extra *LZTrie* pointers associated to each block), and candidates, respectively. As we compress empty unary paths, $n \leqslant n' \leqslant 2n$ holds, and thus, the space is upper bounded by $n \log n + o(u \log \sigma)$, which according to Lemma 2.1 is $uH_k(T) + o(u \log \sigma)$ bits, for any $k = o(\log_\sigma u)$.

For each reverse phrase B_i^r to be inserted in the reverse trie, $1 \leqslant i \leqslant n$, the navigational cost is $O(|B_i^r| \log N_M)$ (this subsumes the $O(|B_i^r|)$ time needed to extract the string from *LZTrie*, in order to do the final check in the Patricia tree). Since $\sum_{i=1}^{n} |B_i^r| = u$, the total navigational cost to construct the hierarchical *RevTrie* is $O(u \log N_M)$. Since the number of node insertions is $n' = O(n)$, the total cost stays $O(u(\log \sigma + \log \log u))$, just as for *LZTrie*.

Constructing the Final *RevTrie* After we construct the hierarchical reverse trie, we construct *RevTrie* directly from it in $O(n' \log N_M) = o(u \log \sigma)$ time, replacing the pointers to *LZTrie* by the corresponding phrase identifiers (*rids*). This raises the space to $3uH_k(T) + o(u \log \sigma)$ bits. We then free the hierarchical trie, dropping the space to $2uH_k(T) + o(u \log \sigma)$ bits.

Thus, we have proved:

Lemma 6.3. *Given the LZTrie for a text $T[1..u]$ over an alphabet of size σ and with k-th order empirical entropy $H_k(T)$, there exists an algorithm to construct the corresponding RevTrie in $O(u(\log \sigma + \log \log u))$ worst-case time and using a total space of $2uH_k(T) + o(u \log \sigma)$ bits of space on top of the space required by the final LZTrie, for any $k = o(\log_\sigma u)$.*

6.2.3 Space-Efficient Construction of *Range*

To construct the *Range* data structure, recall that for every LZ78 phrase B_t of T we must store the point $(preorder_r(v_r), preorder_{lz}(v_{lz}))$, where v_r is the *RevTrie* node corresponding to B_t^r, and v_{lz} is the *LZTrie* node corresponding to phrase B_{t+1}. We allocate memory space for a temporary array $RQ[1..n]$ of $n \log n$ bits, storing the points to be represented by *Range*. Array *RQ* is initially sorted by the first coordinates of the points. Notice that since there is a point for every first coordinate $1 \leqslant i \leqslant n$, the first coordinate of every point is represented simply by the index of array *RQ*, thus saving space. In other words, $RQ[i] = j$ represents the point (i, j). Notice also that *RQ* is a permutation of $\{0, \ldots, n\}$.

To generate the points, we first notice that for a *RevTrie* preorder $i = 0, \ldots, n$ (corresponding only to non-empty nodes) representing the reverse phrase B_t^r, we can obtain the corresponding phrase identifier $t = rids[i]$, and then with the inverse permutation

$ids^{-1}[t+1]$ we obtain the *LZTrie* preorder for the node corresponding to phrase B_{t+1}. Thus, we define $RQ[i] = ids^{-1}[rids[i]+1]$.

Therefore, we start by computing ids^{-1} on the same space of ids, using the algorithm of Lemma 2.8, requiring $O(n)$ time and n extra bits of space. Then, we allocate $n \log n$ bits for array RQ, and traverse *RevTrie* in preorder. For every non-empty node with preorder i we set RQ as defined above. The total space is thus raised to $3uH_k(T) + o(u \log \sigma)$ bits. Next, we recover ids from ids^{-1}, using again Lemma 2.8.

After building RQ, to construct *Range* we must sort the points in RQ by the second coordinate (recall Section 2.5.1, see Lemma 2.9), which in our space-efficient representation of the points means using the second coordinates as array indexes, and storing the first coordinates as array values [3]. This means sorting the current values stored in array RQ. However, since these values along with the corresponding array indexes represent points, after sorting the points we must recall the original array index for every value, so as to store that value in the array. This is straightforward if we store both coordinates of the points, requiring $2n \log n$ bits of space. However, we are trying to reduce the indexing space, and therefore use an alternative approach.

Notice that since $RQ[i] = j$ represents the point (i, j), $RQ^{-1}[j] = i$ shall also represent the point (i, j), yet the points in the inverse permutation RQ^{-1} are sorted by their second coordinate. In other words, in RQ^{-1} the second coordinates are used as array indexes. Thus, we use the algorithm of Lemma 2.8 to construct RQ^{-1} on top of the space for RQ, in $O(n)$ time and requiring n extra bits of space.

Now, we can finally build *Range* from RQ^{-1}. We allocate space for $\log n$ bit vectors of n bits each, requiring $n \log n$ extra bits, thus raising the space usage to $4uH_k(T)+o(u \log \sigma)$ bits. Then, we construct *Range* just as explained in Section 2.5.1 (see Lemma 2.9) and using the points represented by RQ^{-1}. This takes $O(n \log n)$ time, which in the worst case is $O(\frac{u \log u}{\log_\sigma u}) = O(u \log \sigma)$. We then free RQ^{-1}, dropping the space to $3uH_k(T) + o(u \log \sigma)$ bits.

Lemma 6.4. *Given a text $T[1..u]$ over an alphabet of size σ and with k-th order empirical entropy $H_k(T)$, and given the corresponding LZTrie and RevTrie data structures, there exists an algorithm to construct the Range data structure requiring a maximum total space of $2uH_k(T) + o(u \log \sigma)$ extra bits on top of the space for LZTrie and RevTrie, and takes $O(u \log \sigma)$ time in the worst case.*

[3] We could choose to define RQ in a different way, storing the first coordinate of the points and using the second coordinate as array index. However, by using our approach we can construct array RQ with a sequential scan over arrays $rids$ and R itself. The importance of this fact shall be made clear later in this chapter.

6.2.4 Construction of the *Node* Mapping and Remaining Data Structures

After building *RevTrie*, we proceed to construct the *Node* mapping, which given a phrase identifier, yields the corresponding $LZTrie$ node. *Node* is constructed as follows: we traverse *LZTrie* in preorder, and for every node x with LZ78 identifier i, we store in $Node[i]$ a "pointer" to node x, which corresponds to the node position within the corresponding parentheses sequence. This increases the total space requirement to $4uH_k(T) + o(u \log \sigma)$ bits, which is the final space required by the LZ-index. The process can be carried out in $O(n)$ time.

As we said in Section 3.3.7, in a practical implementation the *Range* data structure is replaced by the *RNode* mapping [Nav08]. This is built from *rids* in the same way as *Node* is built from *ids*. The process explained in Section 6.2.3 is not carried out in such a case.

Finally, it is important to note that the data structures defined in Chapters 4 and 5, in order to transform the occurrences from the original LZ-index format into actual text positions, can be constructed without requiring any extra space, and thus to simplify we omit them in this chapter.

6.2.5 The Whole Compressed Indexing Process

The whole compressed construction of LZ-index is summarized in the following steps:

(1) We build the hierarchical $LZTrie$ from the text. We can then erase the text.

(2) We build $LZTrie$ from its hierarchical representation. We then free the hierarchical $LZTrie$.

(3) We build the hierarchical representation of the reverse trie from $LZTrie$.

(4) We build $RevTrie$ from its hierarchical representation, and then free the hierarchical $RevTrie$.

(5) We build *Range*.

(6) We build *Node* from *ids*.

In Table 6.1 we show the total space and time requeriment at each step. The meaning of the third column in the table shall be made clear later in Section 6.3.

Table 6.1: Space and time requirements of each step in the whole compressed indexing process. We assume $k = o(\log_\sigma u)$, and that the tree topology of blocks is represented with DFUDS.

Indexing step	Maximum total space	Maximum main-memory space	Indexing time
1	$uH_k(T) + o(u \log \sigma)$	$uH_k(T) + o(u \log \sigma)$	$O(u(\log \sigma + \log\log u))$
2	$2uH_k(T) + o(u \log \sigma)$	$uH_k(T) + o(u \log \sigma)$	$O(u(\log \sigma + \log\log u))$
3	$2uH_k(T) + o(u \log \sigma)$	$uH_k(T) + o(u \log \sigma)$	$O(u(\log \sigma + \log\log u))$
4	$3uH_k(T) + o(u \log \sigma)$	$uH_k(T) + o(u \log \sigma)$	$O(u(\log \sigma + \log\log u))$
5	$4uH_k(T) + o(u \log \sigma)$	$uH_k(T) + o(u \log \sigma)$	$O(u \log \sigma)$
6	$4uH_k(T) + o(u \log \sigma)$	$uH_k(T) + o(u \log \sigma)$	$O(u/\log_\sigma u)$

6.2.6 Managing Dynamic Memory

The model of memory allocation is a fundamental issue of succinct dynamic data structures, since we must be able to manage the dynamic memory fast and without requiring much extra memory space due to memory fragmentation [RR03]. We assume a standard model where the memory is regarded as an array, with words numbered 0 up to $2^w - 1$. The space usage of an algorithm at a given time is the highest memory word currently in use by the algorithm. This corresponds to the so-called $\mathcal{M}_B$ memory model [RR03], which is the most restrictive one. Note $w = \log n + o(\log n)$, as we need $\Theta(n \log n)$ bits of space to build our index[4].

We manage the memory of every trie block separately, each in a "contiguous" memory space. However, trie blocks are dynamic due to insertion of new nodes, therefore the memory space for trie blocks must grow accordingly. If we use an *Extendible Array* (EA) [BCD+99] to manage the memory of a given block, we end up with a collection of at most $O(n/N_m) = O(n/\log^2 u)$ EAs, which must be maintained under the operations:

— create, which creates a new empty EA in the collection;

— destroy, which destroys an EA from the collection;

— grow(A), which increases the size of array A by one;

— shrink(A), which shrinks the size of array A by one; and

— access(A, i), which access the i-th item in array A.

Raman and Rao [RR03] show how operation access can be supported in $O(1)$ worst-case time, create, grow and shrink in $O(1)$ amortized time, and destroy in $O(s'/w)$ time, where s' is the nominal size (in bits) of array A to be destroyed. The space requirement for the

[4]Note this is consistent with our earlier $w = \Theta(\log u)$ assumption for the RAM model (see Section 2.1), as $\log u = \Theta(\log n)$ (see Property 2.7).

whole collection is $s+O(a^*w+\sqrt{sa^*w})$ bits, where a^* is the maximum number of EAs that ever existed simultaneously in the collection, and s is the nominal size of the collection.

To simplify the analysis we store every component of a block in different EA collections (i.e., we have a collection for T_ps, a collection for $letts_p$s, and so on). The memory for $letts_p$, F_p, C_p, T_p, L_{ids_p}, etc. inside the corresponding EAs is managed as in the original work [MN08b].

Thus, we use operation grow on the corresponding EA every time we insert a node in the tree, and operation create to create a new block upon block overflows, both in $O(1)$ amortized time. Operation shrink, on the other hand, is used by our representation after we reinsert the subtree upon block overflow, in $O(1)$ amortized time. Finally, operation destroy over the blocks is used when destroying the whole hierarchical trie. As the cost to build the trie is $O(\log N_M)$ per element inserted, which adds $\Theta(\log u)$ bits to the data structure, the amortized cost per bit inserted is $O(\frac{\log\sigma+\log\log u}{\log u})$. The amortized cost for destroy is just $O(1/w) = O(\frac{1}{\log u})$ per bit, which is subsumed by the earlier construction cost.

Let us analyze the space overhead due to EAs for the case of T_p. Since we only insert nodes into our tries, we have that the maximum number of blocks that we ever have is $a^* = O(n/N_m)$. As the nominal size of the EA collection for T_p is $O(n)$ bits, the EA requires $O(n)+O(\frac{nw}{N_m}+n\sqrt{\frac{w}{N_m}}) = O(n)$ bits of space. A similar analysis can done for the collections supporting F_p and C_p. The nominal size of the collection for $letts_p$ is $n\log\sigma + O(n)$, and thus we have $n\log\sigma + O(n) + O(\frac{nw}{N_m} + n\sqrt{\frac{w\log\sigma}{N_m}}) = n\log\sigma + O(n)$ bits overall. For the collection supporting ids_p we obtain $n\log u + O(n) + O(\frac{nw}{N_m} + n\sqrt{\frac{w\log u}{N_m}}) = n\log u + O(n)$ bits of space. In general, the whole space overhead due to memory management is $O(n)$ bits.

To complete the definition of our memory allocation model, it remains to say that we can store the EAs representing the block components within a unique EA. In this case, the number of EAs in the collection is $a^* = O(1)$, since we have a constant number of block components. The nominal size of the whole collection is $s = n\log u + n\log\sigma + O(n)$ bits (notice that the $O(n)$ term includes the space for the collections of T_p, F_p, etc., as well as the space overhead due to the EA memory management of these collections). Hence, the space overhead to manage this collection is $O(w + \sqrt{wn\log u})$ bits, which is $O(\sqrt{n}\log u) = O(\sqrt{n}\log n) = o(n)$ bits.

Now that we have defined our memory allocation model, we can conclude:

Theorem 6.1. *There exists an algorithm to construct the LZ-index for a text $T[1..u]$ over an alphabet of size σ and with k-th order empirical entropy $H_k(T)$, using $4uH_k(T)+$*

$o(u \log \sigma)$ bits of space and $O(u(\log \sigma + \log\log u))$ time. This holds for any $k = o(\log_\sigma u)$. The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.

6.3 Constructing the LZ-index in Reduced-Memory Scenarios

In this section we assume a model where we have restrictions in the amount of main memory available to build our indexes, such that we cannot maintain the whole index in main memory. So, we aim at reducing as much as possible the main memory usage of our algorithms. We shall prove that the LZ-index can be constructed as long as the available memory is $uH_k(T) + o(u \log \sigma)$ bits. This means that we have enough main-memory space to store just the compressed text. This has applications, for instance, in text search engines, where new versions of the text must be indexed, while providing indexed access to the current text version. Thus, we can use a less powerful computer to carry out the indexing process, devoting a more powerful one (i.e., one able to accommodate the whole index in main memory) to answer user queries. Thus, we have the advantage that every machine is devoted to a specific task, besides the fact that the cost of the indexing technology can be reduced.

Since we have assumed that we have enough secondary storage space so as to store the final index (see Section 6), we will use that space to temporarily store on disk certain LZ-index components which will not be needed in the next indexing step, and then possibly loading them back to main memory when needed. It is important to note that this does not mean that the index is built on secondary storage, but that in certain cases we use the available secondary memory to store an index component which is not currently needed, thus reducing the peak of main memory usage. However, and as we have seen before throughout Section 6.2, our indexing algorithm is independent of this fact, and we can choose not to use the disk at all when enough main memory is available.

In the following, we show how to adapt our original algorithm to this scenario. At every step we will show the space requirement in two ways: the *maximum amount of main memory* used at that step and the *total amount of memory* used at that step (main-memory plus secondary-memory space). The latter corresponds to the amount of main memory used at every step if we do not use the disk along the construction process.

Step (1) We build the hierarchical $LZTrie$ from the text. We can then erase the text. The total and main-memory space is $uH_k(T) + o(u \log \sigma)$ bits.

Step (2) We build $LZTrie$ from its hierarchical representation. To construct the final *ids* array while trying to reduce the maximum main-memory space, we do not allocate space for it at once. Since this array is indexed by preorder, and since we perform a preorder traversal on the trie, the values in array *ids* are produced by a linear scan.

Thus, we only allocate main-memory space for a constant number of components of the array, e.g., a constant number of disk pages, which are stored on disk upon filling them. This process performs $(n \log n)/B$ (sequential) disk accesses. The symbols (*letts*) and the trie topology, however, are maintained in main memory for the next step, requiring $2n + n \log \sigma + o(n \log \sigma) = o(u \log \sigma)$ bits of space (recall the text is read by small chunks into main memory).

Thus, the maximum main-memory space is $uH_k(T) + o(u \log \sigma)$ bits, while the maximum total amount of space is $2uH_k(T)+o(u \log \sigma)$ bits, since we store the hierarchical *LZTrie* in main memory and array *ids* on disk. We then free the hierarchical *LZTrie*, ending up with a representation requiring $o(u \log \sigma)$ bits of main-memory space, and a total of $uH_k(T) + o(u \log \sigma)$ bits.

Step (3) We build the hierarchical representation of the reverse trie from *LZTrie*. Recall that every non-empty *RevTrie* node stores a pointer to the corresponding *LZTrie* node. This raises the total space requirement to $2uH_k(T)+o(u \log \sigma)$ bits of space. The maximum main-memory usage is $uH_k(T) + o(u \log \sigma)$ bits of space (recall that array *ids* is on disk).

Step (4) We build *RevTrie* from its hierarchical representation as follows. We store the pointers to *LZTrie* associated with *RevTrie* nodes in a linear array on disk, in the same way as done in Step (2) for array *ids* in *LZTrie*. In this way we do not need extra main-memory space on top of the hierarchical *RevTrie*. After storing the pointers on disk and representing the remaining components of *RevTrie*, the total space is raised to $3uH_k(T) + o(u \log \sigma)$ bits, since we have at the same time the final *LZTrie* (array *ids* is on disk), the hierarchical *RevTrie* (in main memory), and the final *RevTrie* (pointers to *LZTrie* are on disk). Then, we free the hierarchical *RevTrie*, thus reducing the total and main-memory space.

Then, we proceed to replace the pointers by the corresponding phrase identifiers. We first load array *ids* to main memory (leaving a copy of it on disk, for further use). Then, we perform a sequential scan on the array of pointers, bringing to main memory just a constant number of disk pages, then following these pointers to *LZTrie* to get the phrase identifier stored in *ids* (note this means that the accesses to *ids* are at random, hence we need *ids* in main memory) and storing these identifiers in the same space of the pointers, writing them to disk and loading the next portion of the pointer array. Finally, we leave a copy of array *ids* in main memory (this shall be useful for the next step).

The maximum main-memory space needed along this step is $uH_k(T) + o(u \log \sigma)$ bits, which corresponds to the space of the hierarchical *RevTrie*, and we end up with a representation requiring $uH_k(T)+o(u \log \sigma)$ bits of main memory, and $3uH_k(T)+o(u \log \sigma)$ bits overall. The number of disk accesses performed is $(4n \log n)/B$.

Step (5) We build *Range*, basically using the procedure of Section 6.2.3, yet with some changes in the memory management in order to reduce the peak of memory usage. Therefore, we compute ids^{-1} on the same space required by *ids*, using the algorithm of Lemma 2.8, requiring $O(n)$ time and n extra bits of space. Then, we traverse *rids* in preorder and for every non-empty node with preorder i we set $RQ[i] \leftarrow ids^{-1}[rids[i]+1]$. Notice that both arrays *rids* and *RQ* are accessed sequentially, which means that we can maintain just a constant number of components of these arrays in main memory. Array ids^{-1}, on the other hand, is accessed randomly, so we maintain it in main memory. In this way, the maximum main-memory space needed along this process is $uH_k(T) + o(u \log \sigma)$ bits.

When this process finishes, the total space is raised to $4uH_k(T) + o(u \log \sigma)$ bits, and then we free array ids^{-1} (recall that we have a copy of the original array *ids* still on disk), dropping the space to $3uH_k(T) + o(u \log \sigma)$ bits of space, and the main-memory space to $o(u \log \sigma)$ bits, since we maintain just the trie topology and symbols of both *LZTrie* and *RevTrie*. This process takes $O(n)$ time overall.

After building *RQ*, to construct *Range* we must sort the points in *RQ* by the second coordinate, by means of constructing RQ^{-1}. Thus, we bring *RQ* to main memory (and delete it on disk), and use the algorithm of Lemma 2.8 to construct RQ^{-1} on top of the space for *RQ*, in $O(n)$ time and requiring n extra bits of space on top of RQ^{-1}. To build *Range* from RQ^{-1}, instead of allocating memory for the $\log n$ bit vectors of n bits each, which would require $n \log n$ extra bits of space on top of RQ^{-1}, we just allocate memory level per level (i.e., we allocate just n bits per level), construct that level from RQ^{-1}, just as explained in Section 2.5.1 (see Lemma 2.9), and then we save that level to disk. Thus, the maximum main-memory space requirement to construct *Range* is $n \log n + o(u \log \sigma) = uH_k(T) + o(u \log \sigma)$ bits of space. The maximum total space is $2n \log n + o(u \log \sigma) = 2uH_k(T) + o(u \log \sigma)$ extra bits on top of the space for *LZTrie* and *RevTrie*, which means a total space of $4uH_k(T) + o(u \log \sigma)$ bits. The construction process takes $O(n \log n)$ time, which in the worst case is $O(\frac{u \log u}{\log_\sigma u}) = O(u \log \sigma)$. After getting *Range*, we free array RQ^{-1} and we are done in this step with a partial representation of LZ-index requiring $3uH_k(T) + o(u \log \sigma)$ bits. The number of disk accesses is $(4n \log n)/B$.

Step (6) We build *Node* from *ids*, by traversing *LZTrie* in preorder. In this way, array *ids* is sequentially traversed, while *Node* is randomly accessed. Thus, we allocate $n \log 2n$ bits of space for *Node*, and maintain it in main memory. Array *ids*, on the other hand, is brought by parts to main memory, according to a sequential scan. Finally, we save *Node* to disk. The number of disk accesses is $(2n \log n)/B$.

Thus, we need only $uH_k(T) + o(u \log \sigma)$ bits of main-memory space to construct *Node*, and this increases the total space requirement to $4uH_k(T) + o(u \log \sigma)$ bits, which is the

final space required by the LZ-index. The process can be carried out in $O(n)$ time. We use the same procedure in case of using the *RNode* data structure instead of *Range*.

In the third column of Table 6.1 we show the maximum main-memory space requirement at each step. The overall number of disk accesses is $(11n \log n)/B = (11uH_k(T) + o(u \log \sigma))/B$. Thus, we have proved:

Theorem 6.2. *There exists an algorithm to construct the LZ-index for a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, using a maximum main-memory space of $uH_k(T) + o(u \log \sigma)$ bits and $O(u(\log \sigma + \log \log u))$ time, for any $k = o(\log_\sigma u)$. The algorithm performs $(7uH_k(T) + o(u \log \sigma))/B$ disk accesses, plus those to write the final index. The total space used by the algorithm is $4uH_k(T) + o(u \log \sigma)$ bits. The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.*

6.4 Space-Efficient Construction of Reduced LZ-indexes

We have shown first how to construct the original LZ-index by using our space-efficient algorithm. However, in Chapters 4 and 5 we defined new versions of LZ-index, some of which are able to replace the original LZ-index in many practical scenarios. Henceforth, in this section we adapt our space-efficient algorithm to build the most promising members of the LZ-index family defined in this thesis.

Throughout this section, and just as in Chapter 5, we assume that the final tries are represented with DFUDS. We also assume the reduced-memory scenario as in Section 6.3. Recall that we present the space usage of our algorithms in two ways: the total maximum main-memory space and the maximum total space (main-memory plus secondary-memory space) at every step.

6.4.1 Space-Efficient Construction of Scheme 2

We perform the following steps in order to build the reduced Scheme 2 of LZ-index (recall its definition in Section 4.1.2).

(1) We build the hierarchical *LZTrie* from the text. This takes $O(u(\log \sigma + \log \log u))$ time, and the maximum space requirement is $uH_k(T) + o(u \log \sigma)$ bits.

(2) We derive the final *LZTrie* from the hierarchical one, which is then freed. The *LZTrie* stores the trie topology *par*, the symbols *letts*, and the phrase identifiers *ids*, requiring $uH_k(T) + o(u \log \sigma)$ extra bits. This takes $O(u(\log \sigma + \log \log u))$ time because of the traversals on the hierarchical *LZTrie*. We use the approach of Section 6.3 (Step (2)) to construct *ids*, without requiring extra asymptotic space. In practice, we use the same approach to build array *letts*, in order to reduce the peak

of main memory, introducing $o(u \log \sigma)/B$ extra disk accesses. The total space usage is $2uH_k(T) + o(u \log \sigma)$ bits, while the maximum main-memory usage is $uH_k(T) + o(u \log \sigma)$ bits. The main-memory space after freeing the hierarchical trie is $o(u \log \sigma)$ bits. After freeing the hierarchical *LZTrie*, we load array *letts* to main memory. The resulting number of I/Os is $(uH_k(T) + o(u \log \sigma))/B$, because of the construction of array *ids*.

(3) We build the hierarchical *RevTrie* from the *LZTrie*, as in Section 6.2.2. This takes $O(u(\log \sigma + \log \log u))$ time. The total space usage is raised to $2uH_k(T) + o(u \log \sigma)$ bits. The maximum main-memory space is $uH_k(T) + o(u \log \sigma)$ bits.

(4) We build the final *RevTrie* from the hierarchical one, storing the trie topology *rpar*, the Patricia-tree *skips*, the symbols *rletts*, and bit vector B marking the empty nodes, requiring $(n' + \frac{u}{\log u})(3 + \log \log u + \log \sigma) = o(u \log \sigma)$ extra bits of space. In order to reduce the indexing space, array $rids^{-1}$ is built later. Array R is built from the pointers to *LZTrie*, replacing them by the corresponding *LZTrie* preorder (recall that we apply *rank* on *par* to get the *LZTrie* preorder of a node). We construct R by using the same approach as for array *ids* in Step (2), performing $(uH_k(T) + o(u \log \sigma))/B$ extra I/Os. The total time is $O(u(\log \sigma + \log \log u))$. We then free the space of the hierarchical *RevTrie*. The maximum total space is $3uH_k(T) + o(u \log \sigma)$ bits, while the maximum main-memory space is $uH_k(T) + o(u \log \sigma)$ bits. We end up using $o(u \log \sigma)$ bits of main-memory space.

(5) To space-efficiently construct array $rids^{-1}$, we first construct *rids* in the following way: we start by loading array *ids* to main memory and erasing it from disk. Then, for every non-empty *RevTrie* node with preorder j we store $rids[j] \leftarrow ids[R[j]]$. In this way, arrays *rids* and R are traversed sequentially, for increasing values of j. Then, we can store/load them to/from disk by parts (respectively), without requiring extra main-memory space. After we build *rids*, the total space has raised to $3uH_k(T) + o(u \log \sigma)$ bits. We then store array *ids* to disk, and free its main-memory space (hence dropping the total space). Finally, we load *rids* to main memory, and use the procedure of Lemma 2.8 to construct $rids^{-1}$ on top of *rids*, to finally store $rids^{-1}$ on disk. The overall time is $O(n)$. The maximum total space is $3uH_k(T) + o(u \log \sigma)$ bits, while the maximum main-memory space is $uH_k(T) + o(u \log \sigma)$ bits. The total number of disk accesses performed by this process is $(6uH_k(T) + o(u \log \sigma))/B$.

Notice that this is a practical version of the LZ-index, and thus we do not store the *Range* data structure. Thus, we conclude:

Theorem 6.3. *There exists an algorithm to construct Scheme 2 of the LZ-index for a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, using*

a total space of $3uH_k(T) + o(u \log \sigma)$ *bits and* $O(u(\log \sigma + \log \log u))$ *time, for any* $k = o(\log_\sigma u)$. *The maximum main-memory space used at any time to construct Scheme 2 can be reduced to* $uH_k(T) + o(u \log \sigma)$ *bits, in such a case performing* $(5uH_k(T) + o(u \log \sigma))/B$ *disk accesses, plus those to write the final index. The space and time bounds are valid in the standard model* $\mathcal{M}_B$ *of memory allocation.*

6.4.2 Space-Efficient Construction of Scheme 3

Scheme 3 of LZ-index can be constructed in a straightforward way, using some of the techniques we have developed up to now. We first build the *LZTrie* in $O(u(\log \sigma + \log \log u))$ time, storing *par*, *letts*, and *ids*. This requires a maximum of $2uH_k(T) + o(u \log \sigma)$ bits of space, and ends up with a representation requiring $uH_k(T) + o(u \log \sigma)$ bits (i.e., the final *LZTrie*). The maximum main-memory space is $uH_k(T) + o(u \log \sigma)$ bits, using the same procedure as in Section 6.3, Step (2). This requires $(uH_k(T) + o(u \log \sigma))/B$ disk accesses.

We then construct the hierarchical *RevTrie*, storing pointers to *LZTrie* nodes to provide the connectivity among tries. Thus, the space requirement raises to $2uH_k(T) + o(u \log \sigma)$ bits. We build the final *RevTrie* storing just *rpar*, *skips*, and *rletts*, and discard the pointers to *LZTrie*, thus temporarily losing the connectivity between tries. We then free the hierarchical *RevTrie*, which drops the space used to $uH_k(T) + o(u \log \sigma)$ bits.

Next we allocate memory space for array $rids[1..n]$, requiring $n \log n$ extra bits. We traverse the *LZTrie* in preorder, and generate every phrase B_t stored in it (assuming that i is the preorder of the corresponding *LZTrie* node). We then look for B_t^r in the *RevTrie*. Recall that at this point we do not have the connectivity between tries, which is generally used to search in the *RevTrie*. However, since this string exists for sure in *RevTrie* (because it exists as an LZ78 phrase in *LZTrie*), we only need to descend in the *RevTrie* using the skips, up to consuming B_t^r. At this point we have arrived at the node for B_t^r, which has preorder j in *RevTrie*, without the need of accessing the *LZTrie* to extract the string. Then we set $rids[j] \leftarrow ids[i]$ (notice the sequential scan on *ids*, which is brought to main memory by parts). Then, we store array *rids* on disk, and free its main memory space. This requires $(2uH_k(T) + o(u \log \sigma))/B$ extra disk accesses.

Now, we go on to compute the inverse permutations for *ids* and *rids* arrays. We first load *ids* from disk, performing $(uH_k(T) + o(u \log \sigma))/B$ extra disk acceses, and construct on it the data structure for permutations of Lemma 2.7, in order to support the computation of ids^{-1}. This requires $\epsilon n \log n + O(n)$ extra space, for $0 < \epsilon < 1$, and takes $O(n)$ time if we use the following procedure.

Let $A_{ids}[1..n]$ be an auxiliary bit vector, and let $B_{ids}[1..n]$ be a bit vector marking which elements of *ids* have an associated backward pointer. Both bit vectors are initialized to all zeros.

We start from the first position of ids, and follow the cycles of the permutation. We mark every visited position i of the permutation as $A_{ids}[i] \leftarrow \mathtt{1}$. We also mark one out of $1/\epsilon$ elements when following the cycles, by setting to $\mathtt{1}$ the appropriate position in B_{ids}. We stop following the current cycle upon arriving to a position j such that $A_{ids}[j] = \mathtt{1}$; then, we move sequentially from position j to the next position j' such that $A_{ids}[j'] = \mathtt{0}$, and repeat the previous process.

Each element in ids is visited twice in this process (this is similar to the process done in the proof of Lemma 2.8), thus this first scan takes $O(n)$ time.

Then, we go on a second scan on the cycles of ids. We set A_{ids} to all zeros again, and allocate array Bwd of $\epsilon n \log n$ bits of space, which shall store the backward pointers of the permutation. We preprocess array B_{ids} with data structures to support operation $rank$ 2.4 (1). We start from the first element and follow the cycles once again. Visited elements are marked in A_{ids}, as before. Every time we reach a position i in the permutation such that $B_{ids}[i] = \mathtt{1}$, we store a backward pointer to the previously visited position j in the cycle, such that $B_{ids}[j] = \mathtt{1}$ (this means that there are $1/\epsilon$ elements between these two positions within the cycle). In other words, we set $Bwd[rank_1(B_{ids}, i)] \leftarrow j$.

This second scan takes also $O(n)$ time, thus the overall process takes $O(n)$ time. We finally free the space of A_{ids} and maintain bit vector B_{ids} as a marker of the positions storing the backward pointers.

Then, we store ids and the data structure for ids^{-1} on disk, and free its main-memory space. This yields $((1+\epsilon)uH_k(T) + o(u \log \sigma))/B$ disk accesses. Finally, we build on $rids$ the data structure of Lemma 2.7, to support the efficient computation of $rids^{-1}$, with $((2+\epsilon)uH_k(T) + o(u \log \sigma))/B$ extra disk accesses. Thus, we conclude:

Theorem 6.4. *There exists an algorithm to construct Scheme 3 of the LZ-index for a text $T[1..u]$ over an alphabet of size σ, and k-th order empirical entropy $H_k(T)$, using $(2+\epsilon)uH_k(T)+o(u \log \sigma)$ bits of space and $O(u(\log \sigma + \log\log u))$ time. This holds for any $0 < \epsilon < 1$ and any $k = o(\log_\sigma u)$. The main-memory space used at any time to construct Scheme 3 can be reduced to $(1+\epsilon)uH_k(T) + o(u \log \sigma)$ bits, in such a case performing $(5uH_k(T) + o(u \log \sigma))/B$ disk accesses, plus those to write the final index. The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.*

6.4.3 Space-Efficient Construction of Index of Theorem 5.1 and Relatives

To construct the LZ-index of Theorem 5.1 without (asymptotically) requiring extra space, we will need two passes over the text, and several traversals over the *LZTrie* and *RevTrie* (yet the number of traversals is a constant). This is because we must be careful not to surpass the reduced space requirement of this index, $(1+\epsilon)uH_k(T) + o(u \log \sigma)$ bits. We carry out the following steps in order:

(1) We build the hierarchical *LZTrie*, just storing the trie topology and the symbols labeling the edges of the trie, without storing the phrase identifiers ids_p in each trie block p. This requires $O(n \log \sigma) = o(u \log \sigma)$ bits of space, and takes $O(u(\log \sigma + \log\log u))$ time. We cannot yet erase the text, as we need it at a later step.

(2) We build the final *LZTrie* from its hierarchical representation, in $O(u(\log \sigma + \log\log u))$ time and requiring extra $O(n \log \sigma)$ bits of space. Recall that we do not store the phrase identifiers *ids*. We then free the hierarchical *LZTrie*.

(3) We traverse *LZTrie* in preorder, generating each LZ78 phrase B_i in constant time per string, and insert B_i^r into a hierarchical *RevTrie*. We store pointers to *LZTrie* nodes in the *RevTrie* nodes, just as in Section 6.2. This requires a maximum of $uH_k(T) + o(u \log \sigma)$ bits of space after the hierarchical *RevTrie* is built, and takes $O(u(\log \sigma + \log\log u))$ time.

(4) We build the final *RevTrie* from its hierarchical representation, storing just the tree topology *rpar*, the Patricia-tree skips, and the symbols *rletts*, requiring $o(u \log \sigma)$ extra bits of space. The pointers to *LZTrie* nodes are not stored, but these were used just to provide the connectivity between tries while constructing *RevTrie*. We then free the hierarchical *RevTrie*. This takes $O(u(\log \sigma + \log\log u))$ time. The maximum space requirement is $uH_k(T) + o(u \log \sigma)$ bits (before freeing the hierarchical *RevTrie*), and we end up with a representation using just $o(u \log \sigma)$ bits of space.

(5) We allocate memory for array $R[1..n]$, of $n \log n$ bits of space, which is constructed as follows. We traverse the *LZTrie* in preorder, and for every phrase B_i, we look for B_i^r in *RevTrie*, which exists for sure and therefore we do not need the connection between tries in order to search. This takes $O(|B_i^r| \log \sigma)$ time. Let v_{lz} be the *LZTrie* node corresponding to B_i. Then we store $R[preorder(v_r)] \leftarrow preorder(v_{lz})$. The overall work on *LZTrie* is $O(n \log \sigma)$, since each string is generated in $O(\log \sigma)$ time (because of the data structure used to represent *letts*). For the *RevTrie*, on the other hand, we have that $\sum_{i=1}^{n} |B_i^r| = u$, and therefore the overall time is $O(u \log \sigma)$. We sample ϵn values of R, as explained in Section 5.1.4.

(6) We allocate space for arrays V_W and S_W, which are used to compute function φ' in *RevTrie*. This adds $O(n \log \sigma) = o(u \log \sigma)$ extra bits. We use the procedure explained in Section 5.1.4 in order to construct these arrays, traversing the *RevTrie* in preorder and using R to map to the *LZTrie*. This takes $O(n)$ time overall. Then we preprocess V_W and S_W with data structures to support *rank* and *select* on them.

(7) We build on R the data structure for inverse permutations of Lemma 2.7, using the same procedure as in Section 6.4.2, raising the overall space requirement to

$(1+\epsilon)uH_k(T) + o(u \log \sigma)$ bits. This takes $O(n)$ time. We then sample ϵn values of R^{-1}, as explained in Section 5.1.4.

(8) We reuse the space allocated for array R to build the uncompressed representation of function φ. Just as in Step (5), we do not need the connection between tries in order to navigate the *RevTrie*, and hence we do not need the information of array R. Recall from Section 5.1 that φ acts as a suffix link in *RevTrie*, and we only store suffix links for the n non-empty nodes. Henceforth, we traverse again the *LZTrie* in preorder, and generate each phrase $B_i = xa$ in $O(\log \sigma)$ time, for $x \in \Sigma^*$, and $a \in \Sigma$. Then we search for ax^r and x^r in *RevTrie*, obtaining non-empty nodes v_r and v'_r respectively. Thus, we store $\varphi[preorder(v_r)] \leftarrow preorder(v'_r)$, and go on with the next phrase in *LZTrie*.

Thus, the work for phrase $B_i^r = xa$ takes $O((|ax^r| + |x^r|) \log \sigma) = O(|B_i^r| \log \sigma)$ time, and thus the overall time is $O(\sum_{i=1}^{n} |B_i^r| \log \sigma) = O(u \log \sigma)$.

(9) We build the compressed version of φ, requiring only extra $O(n \log \sigma) = o(u \log \sigma)$ bits for the final compressed representation of φ. Recall that the representation of φ is the same as Ψ function of CSA, see Sections 2.4.2 and 5.1.3. We then free the uncompressed φ.

We could alternatively use the approach of [CHLS07] to construct φ, which is originally defined to construct function Ψ of CSA [GV05, Sad03] in $O(u \log u)$ time and requiring only $O(u \log \sigma)$ bits of space. In the case of constructing $\varphi = R^{-1}(parent_{lz}(R[i]))$, for every *RevTrie* preorder $i = 1, \ldots, n$, this alternative approach would take $O(n \log n + \frac{n}{\epsilon}) = O(u \log \sigma + \frac{u \log \sigma}{\epsilon \log u})$ time, for any $0 < \epsilon < 1$, requiring no asymptotic extra space (just the $o(u \log \sigma)$ bits for φ). The $O(\frac{n}{\epsilon})$ factor comes from computing n times the inverse R^{-1}. In our case, however, we have previously allocated space for array R, which we use to construct φ much faster. At the end of this step we drop the overall space requirement to $\epsilon uH_k(T) + o(u \log \sigma)$ bits.

(10) We finally allocate memory for array $ids[1..n]$, and set it with all zeros. We also set $i \leftarrow 1$. We perform a second pass on the text T to enumerate the LZ78 phrases (this yields $(u \log \sigma)/B$ extra disk accesses in case the text is stored on disk in plain form), descending in the *LZTrie* with the symbols of T as done when constructing the LZ78 parsing for the first time. Every time we reach a node v_{lz} in *LZTrie*, we check whether $ids[preorder(v_{lz})]$ is `0` or not. In the affirmative case, this means that the corresponding phrase has not yet been enumerated, and thus we store $ids[preorder(v_{lz})] \leftarrow i$ and set $i \leftarrow i + 1$. We go back to the *LZTrie* root and go on with the next symbol of T. In case we arrive at a node v_{lz} with $ids[preorder(v_{lz})] \neq 0$, then we continue the descent from this node, since its phrase

has been already enumerated. This takes $O(n \log \sigma)$ time provided the *LZTrie* is represented with DFUDS. Finally, we can erase the text.

Theorem 6.5. *There exists an algorithm to construct the LZ-index of Theorem 5.1 for a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, using $(1+\epsilon)uH_k(T) + o(u \log \sigma)$ bits of space and $O(u(\log \sigma + \log\log u))$ time. This holds for any $0 < \epsilon < 1$ and any $k = o(\log_\sigma u)$. The algorithm performs two passes over text T, thus requiring $(u \log \sigma)/B$ disk accesses in addition to those for writing the final index to disk. The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.*

We can use this algorithm to construct the LZ-index of Theorem 5.2, which only adds the *Range* data structure, which in turn can be constructed with the same procedure used in Section 6.3, Step (5). Since this requires $2uH_k(T) + o(u \log \sigma)$ bits of space to be constructed, we build *Range* before Step (5) of the previous algorithm. Thus we conclude:

Corollary 6.1. *There exists an algorithm to construct the LZ-index of Theorem 5.2 for a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, using $(2+\epsilon)uH_k(T)+o(u \log \sigma)$ bits of space and $O(u(\log \sigma+\log\log u))$ time. This holds for any $0 < \epsilon < 1$ and any $k = o(\log_\sigma u)$. The algorithm requires $(u \log \sigma+2uH_k(T)+o(u \log \sigma))/B$ disk accesses in addition to those to write the final index to disk. The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.*

Finally, the LZ-index of Theorem 5.3 adds the *Alphabet-Friendly FM-index* [FMMN07], which according to [GN08b] can be constructed with $uH_k(T) + o(u \log \sigma)$ bits of space in $O(u \log u(1 + \frac{\log \sigma}{\log\log u}))$ time. Then, we have:

Corollary 6.2. *There exists an algorithm to construct the LZ-index of Theorem 5.3 for a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, using $(3+\epsilon)uH_k(T)+o(u \log \sigma)$ bits of space and $O(u \log u(1+\frac{\log \sigma}{\log\log u}))$ time. This holds for any $0 < \epsilon < 1$ and any $k = o(\log_\sigma u)$. The algorithm requires $(u \log \sigma + uH_k(T)+o(u \log \sigma))/B$ disk accesses, in addition to those to write the final index. The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.*

6.5 Experimental Results

We implemented a simplification of the algorithm presented in Sections 6.2 and 6.3, which shall be tested in this section. We run our experiments on an Intel(R) Pentium(R) 4 processor at 3 GHz, 4 GB of RAM and 1MB of L2 cache, running version 2.6.13-gentoo of Linux kernel. We compiled the code with `gcc 3.3.6` using full optimization. Times were averaged over 10 repetitions.

6.5.1 A Practical Implementation of Hierarchical Tries

We implement our construction algorithms for Scheme 2 and Scheme 3, and use a simpler representation for the hierarchical trie, just as defined in our original work [AN05].

In this simpler representation, every block in the tree uses contiguous memory space, which stores all the block components. We define different block capacities $N_1 < N_2 \ldots < N_t$, and say that a block of size N_i is able to store up to N_i nodes. When we want to insert a node in a block p of size $N_i < N_t$ which is already full, we first create a new block of size N_{i+1}, copy the content of p to the new one, and then insert the new node within this block. This is called a grow operation. If the full block p is of size N_t, we say that p overflows. In such a case we proceed as explained in Section 6.2.1, with the only difference that the subtree to be reinserted is searched by traversing the whole block (we choose the subtree of maximum size not exceeding $N_t/2$ nodes, just as in [AN05]).

To ensure a minimum fill ratio $0 < \alpha < 1$ in the trie blocks, thus controlling the wasted space, we define $N_i = N_{i-1}/\alpha$, for $i = 2, \ldots, t$, and $1 \leqslant N_1 \leqslant 1/\alpha$. Notice that parameter α allows us for time/space trade-offs: smaller values of α yield a poor utilization of blocks, yet they trigger a smaller number of grow operations (which are expensive) as we insert new nodes. The opposite occurs for large values of α.

The block representation is completely static. This means that the whole block is rebuilt from scratch when inserting new nodes, or upon block overflows. We do not store information to quickly navigate the parentheses within each block. So, we navigate them by brute force (using precomputed tables to avoid a bit-per-bit scan, just as for the simple balanced parentheses data structure implemented by Navarro, see Section 2.5.2). In this way, the navigations can be a little bit slower, yet we save space and time reconstructing these data structures after every insertion. We will show, however, that this is a very efficient representation for our intermediate tries, achieving competitive results in practice.

We use the following parameters throughout our experiments: $N_1 = 2$, $N_t = 1024$, and $\alpha = 0.95$, according to the preliminary results obtained in [AN05]. We assume the reduced-memory model presented in Section 6.3. We also show the results for the model in which only main memory is used, where in most cases the maximum total space coincides with the size of the final LZ-index. We use the `memusage` application by Ulrich Drepper[5] to measure the peaks of main memory usage. Since our algorithms need to use the disk to store intermediate partial results, we measure the user time plus the system time of our algorithms.

We show the results only for Scheme 2 and Scheme 3. This is because these are the most competitive schemes in practice (recall Chapter 4), and also because the most critical points along the indexing algorithm (i.e., the construction of the hierarchical tries) is the

[5] `http://pizzachili.dcc.uchile.cl/utils/memusage/memusage-2.2.2.tar.gz`

same for all schemes (including the original LZ-index). In case of Scheme 3, we choose parameters $1/\epsilon = 1$ and $1/\epsilon = 15$ for the inverse-permutation data structures. Remember from Section 4.3.2 that for $\epsilon = 1$ we simply store the inverse permutation explicitly. These values represent the extreme cases (both for time and space requirements) tested in Chapter 4; all intermediate values offer interesting results as well. Notice that when $1/\epsilon = 1$ the space requirement of Scheme 3 is the same as that of the original LZ-index.

6.5.2 Indexing English Texts

For the experiments with English texts we use the 1-GB file provided in the *Pizza&Chili Corpus* [FN05], downloadable from `http://pizzachili.dcc.uchile.cl/texts/nlang/english.1024MB.gz`.

In Table 6.2 we show the experimental results for English text. As it can be seen, the most time-consuming tasks along the construction process are that of building the hierarchical representations of the tries. For *LZTrie*, the construction rate is about 1.01 MB/sec, while for *RevTrie* the result is about 0.39 MB/sec. Thus, *RevTrie* is much slower than *LZTrie* to be built. The overall average indexing rate is 0.29 MB/sec for Scheme 2, 0.29 MB/sec for Scheme 3 ($1/\epsilon = 1$), and 0.28 MB/sec for Scheme 3 ($1/\epsilon = 15$). As it can be seen, the sample rate of the inverse permutations in Scheme 3 does not affect much the indexing speed.

For Scheme 2, the maximum main-memory peak is reached at Step 3, and it is of about 548 MB. This means that we need about 0.54 times the size of the original text to construct Scheme 2 for the English text. This is 0.59 times the space of the final Scheme 2. When comparing the space required by the hierarchical trie representations with that required by the final trie representations, we have 411,928,076 bytes for the hierarchical *LZTrie* and 408,876,348 bytes for the hierarchical *RevTrie*, versus 410,873,083 bytes for *LZTrie* and 309,412,004 bytes for *RevTrie*. This means that the hierarchical *LZTrie* requires about 1.01 times the size of the final *LZTrie*, while the hierarchical *RevTrie* requires about 1.32 times the size of the final *RevTrie*. The bigger difference between *RevTrie* representations comes from the fact that the hierarchical *RevTrie* stores the symbols labeling the arcs and the Patricia-tree skips, while in practice the final *RevTrie* does not.

Table 6.3 summarizes all these results. Numbers in boldface in the table indicate the final index size in every case, not including the text-position data structure of Section 4.2.2 (although this can be constructed without affecting the current memory peak). Notice that the index sizes (seen as a fraction of the original text size) are smaller than the corresponding sizes reported in Table 4.1 (on page 84) for shorter prefixes of the same texts: 1.03 times the text size for Scheme 1 (including the size for the text-position data structure), versus 1.38 times the text size as reported in Table 4.1 for the shorter text. For Scheme 2 we have now 0.96–1.26 times the text size, versus 1.13–1.69 for the shorter

Table 6.2: Experimental results for English text. The numbers in boldface indicate the final index size in every case.

Index	Indexing step	Main-memory space (bytes)	Total space (bytes)	Time (secs)
Scheme 2	1	411,928,076	411,928,076	909.37
	2	505,729,592	822,801,159	17.55
	3	574,548,639	819,749,431	2,554.07
	4	454,026,216	883,576,755	15.01
	5 & 6	491,169,360	**965,869,767**	52.19
	Total (peak)	574,548,639	965,869,767	3,549.20
Scheme 3	1	411,928,076	411,928,076	898.40
$1/\epsilon = 1$	2	505,729,592	822,801,159	17.51
	3	574,548,639	819,749,431	2,590.78
	4	454,026,216	883,576,755	14.86
	5 & 6	491,169,360	**1,204,608,375**	62.00
	Total (peak)	574,548,639	1,204,608,375	3,583.56
Scheme 3	1	411,928,076	411,928,076	896.88
$1/\epsilon = 15$	2	505,729,592	822,801,159	17.46
	3	574,548,639	819,749,431	2,588.83
	4	454,026,216	883,576,755	14.81
	5 & 6	274,463,684	**771,197,007**	102.80
	Total (peak)	574,548,639	883,576,755	3,620.87

text. To understand this difference we compute the ratio $\frac{n \log n}{u}$, which should give us an idea on how well LZ78 compresses the texts (see Section 2.4.1). The result is that the ratio is about 2.55 for the shorter text (which is close to the third-order empirical entropy of the text, according to the results shown in the *Pizza&Chili Corpus* [6]), while for the longer text the ratio es about 1.83 (which is close to the fifth-order empirical entropy of the text). So we can conclude that this is a realization of the fact that LZ78 compression converges to the entropy of the source as the text length grows. Also, recall that with LZ78 we achieve $nH_k(T) + o(u \log \sigma)$ bits for $k = o(\log_\sigma u)$, and since $\log_\sigma u$ grows with u, it is natural to achieve higher order compression (i.e., higher values of k) for longer texts. In addition, there are several terms of the form $O(n)$ or $O(n \log \sigma)$, which vanish asymptotically compared to $n \log n$.

The results are very similar for Scheme 3 and $1/\epsilon = 1$. For $1/\epsilon = 15$, however, the peak of memory usage when considering the total indexing space at each step is reached at Step 4, and it is slightly greater than the space needed by the final Scheme 3 (more

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

precisely, 1.15 times the size of the final Scheme 3).

As a comparison, we indexed a 500-MB prefix of this text with the original construction algorithm of Scheme 2, using an approach similar to that used in [Nav08], with non-space-efficient intermediate representation for the tries. The peak of main memory is 1,566 MB (this means 3.13 times the size of the original text)[7], with an indexing rate of about 1.29 MB/sec (see Table 6.4). This means that our indexing algorithm is 4.60 times slower than the original indexing algorithm (see column "Slowdown" in Table 6.4), yet we require 5.80 times less memory (see column "Space reduction" in Table 6.4). The intermediate *LZTrie* required 751,817,455 bytes (extrapolating, this is 3.66 times the size of our hierarchical *LZTrie*, see column "Intermediate *LZTrie*" in Table 6.4), while the intermediate *RevTrie* required 1,185,969,250 bytes (extrapolating, this is 5.79 times the size of our hierarchical *RevTrie*, see column "Intermediate *RevTrie*" in Table 6.4). Note the bigger difference among *RevTrie* representations. This is because we are not only using a space-efficient representation, but also because we are compressing empty unary paths at reverse-trie construction time. Thus, we can conclude that our space-efficient trie representations are effective to reduce the indexing space of LZ-index schemes. The price is, on the other hand, a slower construction.

Table 6.3: Statistics for our space-efficient indexing algorithm for Scheme 2. The results for Scheme 3 are similar.

Text	Main-memory peak	Size hierarchical *LZTrie* (bytes)	Size hierarchical *RevTrie* (bytes)
English	0.54 times text size 0.59 times size of final Scheme 2	411,928,076 (1.01 times size of final *LZTrie*)	309,412,004 (1.32 times size of final *RevTrie*)
Human Genome	0.50 times text size 0.44 times size of final Scheme 2	1,233,336,206 (1.02 times size of final *LZTrie*)	1,209,073,218 (1.27 times size of final *RevTrie*)
XML	0.40 times text size 0.61 times size of final Scheme 2	90,563,835 (1.07 times size of final *LZTrie*)	84,591,900 (1.29 times size of final *RevTrie*)
Proteins	1.05 times text size 0.51 times size of final Scheme 2	839,446,471 (0.99 times size of final *LZTrie*)	807,660,745 (1.28 times size of final *RevTrie*)

[7]It is important to note that the original algorithm uses just main memory to construct Scheme 2

Table 6.4: Main statistics for the construction of Scheme 2 versus the non-space-efficient original algorithm. Column "Slowdown" shows the slowdown obtained by using our space-efficient algorithm instead of the original one. "Space reduction" indicates the factor of space reduction gained by using our algorithm instead of the original one. Finally, columns "Intermediate *LZTrie*" and "Intermediate *RevTrie*" show the size of the intermediate data structures used to build the final tries, as a fraction of the size of the final trie representations.

Text	Main-memory peak	Indexing rate (MB/secs)	Slowdown	Space reduction	Intermediate *LZTrie*	Intermediate *RevTrie*
English (500 MB)	1,566 MB (3.13 times text size)	1.29	4.60	5.80	3.66	5.74
Genome (500 MB)	1,275 MB (2.55 times text size)	1.86	9.78	5.10	3.22	5.95
XML	862 MB (3.02 times text size)	2.31	5.25	7.50	2.68	9.02
Proteins (500 MB)	1,781 MB (3.56 times text size)	1.82	9.58	3.39	2.41	3.04

6.5.3 Indexing the Human Genome

For the test on DNA data we indexed the Human Genome[8], whose size is about 3,182MB. In Table 6.5 we show the experimental results obtained with our construction algorithm. The indexing rate for the hierarchical *LZTrie* is about 1.30 MB/sec, while for *RevTrie* it is about 0.23 MB/sec. The total indexing time (user time plus system time) is about 4.63 hours, which means and overall indexing rate of about 0.19 MB/sec.

See Table 6.3 for the statistics regarding the memory peak of the algorithm, as well as a comparison between intermediate and final trie representations. See Table 6.4 for a comparison with the original construction algorithm for Scheme 2, indexing a 500-MB prefix of the Human Genome.

We show in Table 6.6 the practical results for the best existing indexing algorithms we know of. The results have been taken from the original papers indicated in the table. As a comparison, W.-K. Hon et al. [HLSS03, Hon04] index the Human Genome with the CSA in about 24 hours, using a Pentium IV processor at 1.7 GHz with 512 KB of L2 cache,

[8] `http://hgdownload.cse.ucsc.edu/goldenPath/hg18/bigZips/est.fa.gz`.

Table 6.5: Experimental results for the Human Genome. The numbers in boldface indicate the final index size in every case.

Index	Indexing step	Main-memory space (bytes)	Total space (bytes)	Time (secs)
Scheme 2	1	1,233,336,206	1,233,336,206	2,440.33
	2	1,428,595,278	2,442,409,424	51.73
	3	1,677,938,853	2,467,406,392	13,966.22
	4	1,405,350,330	2,665,257,752	45.00
	5 & 6	1,579,033,696	**2,985,958,274**	181.96
	Total (peak)	1,677,938,853	2,985,958,274	16,685.28
Scheme 3	1	1,233,336,206	1,233,336,206	2,443.83
$1/\epsilon = 1$	2	1,428,595,278	2,442,409,424	51.98
	3	1,677,938,853	2,467,406,392	13,791.08
	4	1,405,350,330	2,665,257,752	44.93
	5 & 6	1,579,033,696	**3,775,475,122**	211.81
	Total (peak)	1,677,938,853	3,775,475,122	16,543.63
Scheme 3	1	1,233,336,206	1,233,336,206	2,445.02
$1/\epsilon = 15$	2	1,428,595,278	2,442,409,424	51.61
	3	1,677,938,853	2,467,406,392	13,812.29
	4	1,405,350,330	2,665,257,752	44.92
	5 & 6	841,516,932	**2,300,440,426**	365.18
	Total (peak)	1,677,938,853	2,665,257,752	16,719.02

and 4 GB of main memory, running Solaris 9 operating system. Despite the difference in CPU rate of our machine compared to Hon et al.'s, the difference in indexing time suggests that the LZ-index can be space-efficiently constructed in much less time than CSAs. Hon et al. are also able to construct the FM-index in about 4 extra hours, for a total of about 28 hours. The algorithm of [DKMS08], on the other hand, allows us to index the Human Genome in about 8.52 hours on a machine with 80-GB ATA IBM 120-GPX disks, and about 5.56 hours on a machine with 73-GB SCSI Seagate 15,000 RPM ST373453LC disks, in both cases using secondary storage and just a constant amount of main memory. Ours is a relevant practical result, specifically for biological research, since it demonstrates that it is feasible to index the Human Genome within a few hours (less than 5) and in the main memory of a desktop computer.

6.5.4 Indexing XML Data

For XML data we indexed the file `http://pizzachili.dcc.uchile.cl/texts/xml/dblp.xml.gz` of about 285 MB provided in the *Pizza&Chili* Corpus. This text has shown

Table 6.6: Comparison of indexing algorithms to construct an index for the Human Genome. For suffix trees, Kurtz estimated the indexing time on his machine, whose CPU is 10 times slower than ours. In case of suffix arrays, we estimate the indexing space according to the space used with other texts; we do not have time estimations for these. In both cases the indexing algorithms are probably faster than our algorithms for the LZ-index (provided they have the given amount of main memory available).

Index	Construction algorithm	Indexing time	Maximum indexing space (RAM)
Suffix tree	[Kur99]	< 9 hours (*)	45.31 GB
Suffix array	[LS99]	—	27.96 GB
Suffix array	[MF04]	—	18.64 GB
Suffix array	[DKMS08]	5.56 – 8.52 hours	sec. storage
CSA	[Hon04]	24 hours	3.60 GB (¶)
FM-index	[Hon04]	28 hours	3.60 GB
Scheme 2 of LZ-index	This thesis	4.63 hours	2.78 GB
Scheme 2 (reduced-memory model)	This thesis	4.63 hours	1.56 GB (†)

(*) Estimated, on a Sun-UltraSparc 300 MHz, 192 MB of main memory, under Solaris 2.
(¶) Hon [Hon04] reported a size of 2.88 GB for the Human Genome, whereas ours is of size 3.11 GB. They use a 1.7 GHz CPU.
(†) Just regarding main-memory space.

to be highly compressible.

In Table 6.7 we show the experimental results for XML text. The indexing rate for *LZTrie* is about 1.43 MB/sec, while for *RevTrie* it is about 0.65 MB/sec. The overall indexing rate is about 0.44 MB/sec.

See Table 6.3 for the statistics regarding the memory peak of the algorithm, as well as a comparison between intermediate and final trie representations. See Table 6.4 for a comparison with the original construction algorithm for Scheme 2.

6.5.5 Indexing Proteins

We indexed the text `http://pizzachili.dcc.uchile.cl/texts/protein/proteins.gz` of about 1 GB of protein sequences, provided in the *Pizza&Chili* Corpus. This has shown to be a not so compressible text.

In Table 6.8 we show the experimental results for proteins. The indexing rate for

Table 6.7: Experimental results for XML text. The numbers in boldface indicate the final index size in every case.

Index	Indexing step	Main-memory space (bytes)	Total space (bytes)	Time (secs)
Scheme 2	1	90,563,835	90,563,835	199.74
	2	111,467,467	175,009,211	3.82
	3	120,592,538	169,037,276	435.20
	4	98,337,536	185,878,936	3.23
	5 & 6	97,231,032	**198,518,068**	9.29
	Total (peak)	120,592,538	198,518,068	651.28
Scheme 3	1	90,563,835	90,563,835	201.43
$1/\epsilon = 1$	2	111,467,467	175,009,211	3.88
	3	120,592,538	169,037,276	441.91
	4	98,337,536	185,878,936	3.24
	5 & 6	97,231,032	**245,871,260**	11.02
	Total (peak)	120,592,538	245,871,260	661.41
Scheme 3	1	90,563,835	90,563,835	200.91
$1/\epsilon = 15$	2	111,467,467	175,009,211	3.79
	3	120,592,538	169,037,276	441.34
	4	98,337,536	185,878,936	3.20
	5 & 6	54,641,864	**160,692,920**	18.66
	Total (peak)	120,592,538	185,878,936	667.91

the hierarchical *LZTrie* is about 0.92 MB/sec, while for *RevTrie* it is about 0.24 MB/sec. Note that the indexing rate for *RevTrie* is much slower than for other texts. This could be mainly because proteins are not so compressible, and therefore the tries have a greater number of nodes to be inserted, which makes the process slower. The overall indexing rate is about 0.19 MB/sec.

See Table 6.3 for the statistics regarding the memory peak of the algorithm, as well as a comparison between intermediate and final trie representations. See Table 6.4 for a comparison with the original construction algorithm for Scheme 2, indexing a 500-MB prefix of proteins.

6.6 Final Comments

The main result of this chapter is that both the original LZ-index and the whole family of LZ-indexes defined in this thesis (i.e., the LZ-indexes defined in Chapters 4 and 5) can be constructed without requiring extra space on top of that required by the index. This

Table 6.8: Experimental results for proteins. The numbers in boldface indicate the final index size in every case.

Index	Indexing step	Main-memory space (bytes)	Total space (bytes)	Time (secs)
Scheme 2	1	839,446,471	839,446,471	1,087.58
	2	1,018,660,027	1,681,050,175	33.82
	3	1,133,180,292	1,649,264,449	4,105.11
	4	895,675,465	1,766,181,601	27.83
	5 & 6	1,032,374,144	**1,990,895,000**	112.75
	Total (peak)	1,133,180,292	1,990,895,000	5,374.88
Scheme 3	1	839,446,471	839,446,471	1,095.56
$1/\epsilon = 1$	2	1,018,660,027	1,681,050,175	33.49
	3	1,133,180,292	1,649,264,449	4,113.27
	4	895,675,465	1,766,181,601	27.55
	5 & 6	1,032,374,144	**2,502,718,500**	134.72
	Total (peak)	1,133,180,292	2,502,718,500	5,404.62
Scheme 3	1	839,446,471	839,446,471	1,097.09
$1/\epsilon = 15$	2	1,018,660,027	1,681,050,175	33.86
	3	1,133,180,292	1,649,264,449	4,117.30
	4	895,675,465	1,766,181,601	27.62
	5 & 6	575,948,072	**1,589,866,364**	232.25
	Total (peak)	1,133,180,292	1,766,181,601	5,508.14

is a relevant aspect regarding the practicality of our indexes, since wherever these LZ-indexes can be used, we will be able to build them without the need of accessing secondary storage. The construction time is $O(u(\log \sigma + \log \log u))$, which is linear on the text size on polylog-sized alphabets.

Given the data structures that compose the LZ-index, the space-efficient construction is highly related to the representation of succinct dynamic σ-ary trees (or tries). Thus, the basic idea is to construct the tries of LZ-index using space-efficient intermediate representations supporting fast incremental insertion of nodes. This gives us, as a spin off, an efficient representation of succinct dynamic σ-ary trees, with basic operation times $O(\log \sigma + \log \log u)$ (amortized in the case of updates), where u is the number of nodes in the tree [Arr08]. This is the first such representation with operation times related to $O(\log \sigma)$ rather than just to $O(\log u)$ [CHLS07], so we are able to take advantage of smaller values of σ.

The space-efficient construction of the LZ-index tries is combined, however, with a careful construction of the remaining index components, in order to not to surpass the space

requirement of the final index. Many combinatorial properties of the index components are exploited in order to achieve this goal.

We also defined an alternative model in which we have a reduced amount of main memory to perform the indexing process (perhaps less memory than that needed to accommodate the whole index). We show that the LZ-indexes can be constructed within $(1+\epsilon)uH_k(T)+o(u\log\sigma)$ bits of space, in $O(u(\log\sigma+\log\log u))$ time. This means that the LZ-indexes can be constructed within asymptotically the same space than that required to store the compressed text.

On the practical side, our experimental results indicate that all LZ-index versions can be constructed within the same amount of memory as needed by the final index. Under the reduced-memory scenario, we have that the LZ-index versions can be constructed requiring 0.40 – 1.05 times the size of the original text, depending on the compressibility of the text. This means about 3.39 – 7.50 times less space as that needed by the original construction algorithm (which works assuming that there is enough memory to store the whole index in main memory). Our indexing rate is about 0.19 – 0.44 MB/sec., which is 4.60 – 9.58 times slower than the original construction algorithm. In conclusion, our algorithm requires much less memory than the original one, in exchange for a slower construction algorithm. However, our indexing algorithm is still competitive with existing indexing technologies. For example, we are able to construct the LZ-index for the Human Genome in less than 5 hours, while Dementiev et al. [DKMS08] and Hon et al. [HLS$^+$07] require 5.6–8.5 and 24 hours to construct the suffix array and CSA for the Human Genome, respectively.

Given the similarities in the composition of the indexes, we believe that our methods could be extended to construct related LZ-indexes [FM05, RO07] within limited space.

6.6.1 A Further Application

An interesting (and direct) application of our indexing algorithm is in the construction of the LZ78 parsing of a text T. Grossi and Sadakane [GS06] define an alternative representation for the LZ78 parsing, which has the nice property of supporting optimal time to access any text substring. The parsing consists basically of the *LZTrie* (the trie topology and array of symbols labeling the trie), plus an array that, for any phrase identifier i, stores the preorder of the corresponding *LZTrie* node (see *Variant 2* to represent the LZ78 parsing, in Section 2.4.1 of this thesis). Using our notation, the latter array is just ids^{-1}.

Jansson et al. [JSS07a] propose an algorithm to construct the parsing in $O(\frac{u}{\log_\sigma u}\frac{(\log\log u)^2}{\log\log\log u})$ time and requiring $uH_k(T)+o(u\log\sigma)$ bits of space. The algorithm, however, needs two passes over the text, which means $(u\log\sigma)/B$ extra disk accesses if it

is stored on disk, which can be expensive. We can reduce the number of disk accesses as follows, mainly when the text is compressible:

— We construct the hierarchical *LZTrie* for T, storing the phrase identifier for each node. We can erase T since it is not anymore necessary. This takes $O(u(\log\sigma + \log\log u))$ time.

— We build the final *LZTrie*, storing array *ids* on disk, as it was explained in Section 6.3. This takes extra $O(u(\log\sigma + \log\log u))$ time, and performs $(uH_k(T) + o(u\log\sigma))/B$ extra disk accesses.

— We then free the hierarchical *LZTrie* and load array *ids* back to main memory, performing $(uH_k(T) + o(u\log\sigma))/B$ extra disk accesses.

— We compute ids^{-1} in place, using the algorithm of Lemma 2.8, and this way we complete the representation for the LZ78 parsing of text T.

As we can see, we exchange the $u\log\sigma/B$ extra disk accesses of [JSS07a] by $(2uH_k(T) + o(u\log\sigma))/B$ disk accesses. This can be much better, specifically in the case of large compressible texts. The total time is $O(u(\log\sigma + \log\log u))$, and the maximum main-memory space required is $uH_k(T) + o(u\log\sigma)$ bits.

Chapter 7

A Secondary-Memory LZ-index

The initial statement in behalf of compressed full-text self-indexes was that larger texts could be indexed and stored in main memory, without accessing secondary storage. However, some texts are so large that their corresponding indexes do not fit entirely in main memory, even compressed. In these cases the index must be stored on secondary memory and the search proceeds by loading the relevant parts into main memory. Because of its high cost, the problem here consists in reducing the number of accesses to secondary storage at search time.

Unlike what happens with sequential text searching, which speeds up with compression, because the compressed text is transferred faster to main memory [MNZBY00], working on secondary storage with a compressed index usually requires *more* disk accesses in order to find the pattern occurrences. This is because of the overhead introduced to retrieve the pattern occurrences from the compressed index, since most compressed indexes use non-local compression methods: somehow, they rearrange the data in order to achieve compression, thus locality is usually destroyed. Yet, these indexes require less space, which in addition can reduce the seek time incurred by a larger index because seek time is roughly proportional to the size of the data. It is worth reminding that seek time is the most expensive component in the disk-access time.

Model of Computation. We assume in this chapter a model of computation where a *disk page* of size B (able to store $b = \omega(1)$ integers of $\log u$ bits, i.e., $B = b \log u$ bits) can be transferred to main memory in a single disk access [Vit08]. Because of their high cost, the performance of our algorithms is measured as the number of disk accesses performed to solve a query. We count *every* disk access, which is an upper bound to the real number of accesses, as we disregard the disk caching due to the operating system. Assume we can only hold a constant number of disk pages in main memory. As in all of this chapter, we assume that our text T is static, i.e., there are no insertions nor deletions of text symbols.

7.1 Related Work

In what follows we depict the most important works on disk-based full-text indexes.

— The classical suffix arrays can be adapted to work on disk, following the idea of Baeza-Yates et al. [BYBZ96]: we divide the suffix array into blocks of $h \leqslant b$ elements (pointers to text suffixes), and move to main memory the first l text symbols of the first suffix of each block, that is, we store $\frac{u}{h}l$ extra symbols. At search time, we carry out two binary searches [MM93] to delimit the interval $[i..j]$ of the pattern occurrences. Yet, the first part of the binary search is done over the samples without accessing the disk. Once the blocks where i and j lie are identified, we bring them to main memory and finish the binary search, this time accessing the text on disk at each comparison. Therefore, the total cost is $2 + 2\log h$ disk accesses. We must pay at most $1 + \lceil \frac{occ-1}{b} \rceil$ extra accesses to report the occurrences of P within those two positions. The space requirement including the text is $(5 + \frac{l}{h})$ times the text size, assuming 4-byte integers.

— One of the best known indexes for secondary memory is the *String B-tree* [FG99], although this is not a compressed data structure. It requires $O(\log_b u + \frac{m+occ}{b})$ disk accesses in searches and $O(u/b)$ disk pages of space. This value is, in practice, about 12.5 times the text size (not including the text) [FG96], which is prohibitive for very large texts. Its static version takes about 3–5 times the text size (including the text).

— Clark and Munro [CM96] present a compact representation of suffix trees on secondary storage (the *Compact Pat Trees*, or *CPT* for short). This is not a compressed index, and also needs the text to operate. Although not providing worst-case guarantees, the representation is organized in such a way that the number of disk accesses is reduced to 3–4 per query. The authors claim that the space requirement of their index is comparable to that of suffix arrays, needing about 4–5 times the text size (not including the text).

— Mäkinen et al. [MNS04] propose a technique to store a *Compressed Suffix Array* on secondary storage, based on *backward searching* [Sad02]. This is the only proposal to store a (*zero*-th order) compressed full-text self-index on secondary memory, requiring $u(H_0 + O(\log\log\sigma))$ bits of storage and a counting cost of at most $2(1 + m\lceil\log_B u\rceil)$ disk accesses. Locating the occurrences of the pattern would need $O(\log u)$ extra accesses *per occurrence*, which is prohibitively costly.

— González and Navarro [GN08a] present a suffix-array-based scheme working on secondary storage. When the text is compressible, their scheme takes significantly less space than the corresponding suffix array. Given the definition of function Ψ of

Compressed Suffix Arrays as in Section 2.4.2, the idea is to represent Ψ in differential form, transforming its regularities into true repetitions. These repetitions are then compressed with an algorithm like Re-Pair [LM99], which has the advantage of providing fast local decompression. This is called the *Locally Compressed Suffix Array* (LCSA) [GN07]. The advantage of this scheme is that we can store in every disk page a greater number of suffix-array entries (i.e., occurrences). Therefore, with each disk access we are able to report many more occurrences than with suffix arrays. The result is a very competitive index that requires between $2(m-1)$ to $4(m-1)$ disk accesses for `count` (this depends on the available main memory). Operation `locate` takes $1+\lceil\frac{occ-1}{b}\rceil$ extra I/Os in the worst case, and $cr \cdot occ/b$ I/Os on average, being $0 < cr \leqslant 1$ the suffix array compression ratio achieved. Thus, unlike most other indexes, the performance of this index improves with compression. The space requirement of this index is $O(H_k(T)\log(1/H_k(T))u\log u)$ bits (if sufficient main memory is available), which in practice can be up to 4 times smaller than classical suffix arrays.

— Recently Sinha et al. [SPMT08] presented the *LOF SA*, a disk-based full-text index which is a new variant to arrange suffix arrays on disk. The index is basically divided into two parts: the LOF trie and the LOF array. The former is a truncated suffix trie, in order to make it compact (sometimes this can be maintained in main memory). The latter is an interleaving of suffix-array entries and LCP (longest common prefix [MM93]) values, plus some extra symbols from the text (which are used in the same way as for the disk-based suffix array of [BYBZ96]). These components are stored on disk as an array of triples. The search starts in the LOF trie, and then proceeds into the LOF array. The idea is to use the LOF trie to narrow as much as possible the suffix array portion that must be searched, thus reducing the number of accesses performed by the binary search on the suffix array. The authors show a good performance in practice, being up to three times faster than the best existing suffix array on disk (see [SPMT08] for further details). The space requirement is, however, excessive: 13 times the text size.

As it can be seen from the related works, except for the disk-based LCSA [GN08a], all the existing full-text indexes on secondary storage are either non-compressed (and they require a considerable amount of storage), or they are compressed yet have a poor performance for `extract`, `display`, and `locate` queries, as for example Compressed Suffix Arrays on disk [MNS04].

In this chapter we propose a version of Navarro's LZ-index that can be efficiently handled on secondary storage, mostly from a practical point of view.

7.2 The LZ-index on Secondary Storage

Navarro's LZ-index (see Section 3.3) was originally designed to work in main memory, and hence it has a non-regular access pattern to the index components. As a result, it is not suitable to work on secondary storage. This can be noticed with the navigational-scheme approach of Chapter 4: every time we follow an arrow in the scheme, this means a random access to an index component. We shall show how to achieve locality in the access to the LZ-index components, so as to have good secondary storage performance. In this process we introduce some redundancy over main-memory proposals of previous chapters.

7.2.1 Supporting the Basic Trie Operations

To represent the tries of the index we use a space-efficient representation similar to the hierarchical representation of Chapter 6, which now we make searchable by adding a more complete set of operations. We cut the trie into disjoint *blocks* of size B such that every block stores a connected component of the whole trie. We arrange these blocks in a tree by adding some *inter-block* pointers.

We cut the trie in a *bottom-up* fashion, trying to maximize the number of nodes in each block. This is the same partition used by Clark and Munro [CM96], and so we also suffer of very small blocks. To achieve a better fill ratio and reduce the space requirement, we store several trie blocks within each disk page.

Just as in Chapter 6, every trie node x in this representation is either a leaf of the whole trie, or it is an internal node. For internal nodes there are two cases: the node x is internal to a block p or x is a leaf of block p (but not a leaf of the whole trie). In the latter case, x stores a pointer to the representation q of its subtree. The leaf is also stored as a fictitious root of q, so that every block is a subtree. Therefore, every such node x has two representations: (1) as a leaf in block p; (2) as the root node of the child block q.

Each block p of N nodes and root node x consists basically of:

— The trie topology T_p of block p, using the *balanced parentheses* (BP) representation (Lemma 2.10) of the subtree, requiring $2N + o(N)$ bits.

— A bit-vector $F_p[1..N]$ (the *flags*) such that $F_p[j] = \texttt{1}$ iff the j-th node of the block (in preorder) is a leaf of p, but not a leaf of the whole trie. In other words, the j-th node has a pointer to the representation of its subtree. We represent F_p using the data structure of Lemma 2.4 (1) to allow *rank* and *select* queries in constant time and requiring $N + o(N)$ bits.

— The sequence $letts_p[1..N]$ of symbols labeling the arcs of the subtree, in preorder. The space requirement is $N\lceil \log \sigma \rceil$ bits.

— Only in the case of *LZTrie*, the sequence $ids_p[1..N]$ of phrase identifiers in preorder. The space requirement is $N \log n$ bits.

— A pointer to the leaf representation of x in the parent block.

— The depth and preorder of x within the whole trie.

— A variable number of pointers to child blocks. The number of child blocks of a given block can be known from the number of `1`s in F_p.

— An array $Size_p$ such that each pointer to child block stores the size of the corresponding subtree.

Using this information, given node x with preorder j within block p, we are able to support the following operations (see Section 2.5.2 for a definition of the operations):

Operation parent(x). If x is not the root of some trie block, then both x and $parent(x)$ are stored in the same block and $parent(x)$ is found in constant time using the BP representation of T_p. Otherwise, $parent(x)$ is stored in the parent block q of p (the block which is pointing to p). We follow the pointer to the leaf representation of x in q and finally use the *parent* operation on the BP representation of q. As a result, in the worst case we need one disk access to solve the operation.

Operation child(x, α). If x is not a leaf of some block, the desired child is also stored in block p. Therefore, we use the $child(x, i)$ operation (which gets the i-th child of node x) provided by BP for $i = 1, \ldots, \sigma$, until we reach the child by symbol α (or until we discover that such child does not exist). This takes $O(\sigma)$ CPU time in the worst case, but it is free of disk accesses. If x is a leaf of block p, the child we are looking for is a child of the root representation of x in the corresponding child block q. If $i = rank_1(F_p, j)$, then q is the i-th child block of p. We then move to q and look for the child of its root by label α. Hence, in the worst case we need only one disk access to solve this operation.

Operation depth(x). It can be computed as the depth of the root of p plus the depth of x within p, which can be computed in constant time by using BP [MR01]. Therefore, this operation can be supported in constant time and free of disk accesses.

Operation subtreesize(x). The BP representation allows us to support the *subtreesize* operation in constant time, so the operation can be supported trivially if the subtree of x is completely contained in block p (we call $subtreesize_p$ that operation). However, the subtree of x can span more than its block. We use array $Size_p$ to solve this problem. The portion of $Size_p$ corresponding to x goes from position $p_1 = rank_1(F_p, j-1) + 1$ to $p_2 = rank_1(F_p, j + subtreesize_p(x) - 1)$. If p_1 equals p_2, the subtree of x is completely stored in p. Otherwise, $subtreesize(x)$ can be computed as $subtreesize_p(x) + \sum_{i=p_1}^{p_2} Size_p[i]$. We do not need extra accesses to support this operation.

Operation preorder(x). This is computed from three main components: the preorder of the root of block p within the whole trie, plus the preorder of x within p (computed using BP), plus the number of nodes stored in child blocks of p, such that all nodes in these child blocks have preorder less than x. We compute the last term as $\sum_{i=1}^{r} Size_p[i]$, where $r = rank_1(F_p, j-1)$.

Operation ancestor(x, y). This is trivially solved provided we can solve operations *subtreesize* and *preorder*: node x is an ancestor of node y iff $preorder(x) \leqslant preorder(y) \leqslant preorder(x) + subtreesize(x)$.

Improving CPU Time. To achieve more efficient CPU time we can represent the subtrees in each block using the DFUDS representation of Lemma 2.11 (which allows $child(x, \alpha)$ in $O(1)$ time). The only consideration is with operation *depth*, which is not originally supported by DFUDS, yet we can use the data structure of [JSS07b] to support it. We can also represent $Size_p$ using a *searchable partial sum* data structure [HSS03b].

Analysis of Space Complexity. In the case of *LZTrie*, as the number of nodes is n, the space requirement is $2n + n + n\log\sigma + n\log n + o(n)$ bits, for the BP representation, the flags, the symbols, and phrase identifiers respectively. To this we must add the space required for the inter-block pointers and the extra information added to each block, such as the depth of the root, etc. If the trie is represented by a total of K blocks, these data add up to $O(K\log n)$ bits. The bottom-up partition of the trie ensures $K = O(\sigma n/b)$, because the trie is σ-ary and thus we can only guarantee an utilization of $O(1/\sigma)$ of every block. However, we have that $K = \min\{n, \sigma n/b\}$, because the number of blocks is smaller or equal to the number of nodes. Hence, the extra information stored in each block requires $O(uH_k(T)) + o(u\log\sigma)$ bits of space in the worst case. If $b = \omega(\sigma)$, we have $o(n\log n) = o(u\log\sigma)$ bits of space for the extra information.

In the case of *RevTrie*, as there can be empty nodes, we represent the trie using a *Patricia tree* [Mor68], compressing empty unary paths so that there are $n \leqslant n' \leqslant 2n$ nodes. In the worst case the space requirement is $4n + 2n + 2n\log\sigma + o(n)$ bits, plus the extra information as before.

As we pack several trie blocks in a disk page, we ensure a utilization ratio of 50% at least. Hence the space of the tries can be at most doubled on disk.

7.2.2 Reducing the Navigation between Structures

We add the following data structures with the aim of reducing the number of disk accesses required by the LZ-index at search time:

— $ids^{-1}[1..n]$: a mapping from phrase identifiers to the corresponding *LZTrie* preorder, requiring $n \log n$ bits of space.

— $Rev[1..n]$: a mapping from *RevTrie* preorder positions to the corresponding *LZTrie* node. Since *LZTrie* has $K = O(\frac{\sigma n}{b})$ blocks, every such pointer requires $\log n + \log \sigma + O(1) - \log b$ bits of space to indicate the block in which the node is stored, and $\log b + O(1)$ bits o indicate the position of the node within the block. The total per pointer is, therefore, $\log n + \log \sigma + O(1)$ bits of space. The total space for this array is $n \log n + n \log \sigma + O(n)$ bits. We use one extra bit per pointer, for a total of n extra bits. Later in this section we explain why we need n extra bits of space.

— $TPos_{lz}[1..n]$: if the phrase corresponding to the node with preorder i in *LZTrie* starts at position j in the text, then $TPos_{lz}[i]$ stores the value j. This array requires $n \log u$ bits and is used just for `locate` queries.

— $LR[1..n]$: an array requiring $n \log n$ bits. If the node with preorder i in *LZTrie* corresponds to the LZ78 phrase B_k, then $LR[i]$ stores the preorder of the *RevTrie* node for B_{k-1}.

— $S_r[1..n]$: an array requiring $n \log u$ bits, storing in $S_r[i]$ the subtree size of the *LZTrie* node corresponding to the i-th *RevTrie* node (in preorder). This array is used just for `count` queries.

— $Node[1..n]$: the mapping from phrase identifiers to the corresponding *LZTrie* node, as defined in Section 3.3.1, requiring $n \log n + n \log \sigma + O(n)$ bits. This is used to support `extract` queries.

As the size of these arrays depends on the compressed text size, we do not need that much space to store them: they require $2n \log u + 4n \log n + 2n \log \sigma + O(n)$ bits, which summed to the space of the tries yields $O(uH_k(T)) + o(u \log \sigma)$ bits of space, for any $k = o(\log_\sigma u)$. If $b = \omega(\sigma)$, the space requirement is $8uH_k(T) + o(u \log \sigma)$ bits of space, for any $k = o(\log_\sigma u)$ [1]. Later in our experiments we will show estimations for the constant in the main component of the space requirement.

If the index is used *only* for `count` queries, we basically need arrays ids^{-1}, LR, S_r, and the tries, plus an array $RL[1..n]$, which is similar to LR but mapping from a *RevTrie* node for B_k to the *LZTrie* preorder for B_{k+1}. All these add up to $6uH_k + o(u \log \sigma)$ bits if $b = \omega(\sigma)$.

After searching for all pattern substrings $P[i..j]$ in *LZTrie* (recording in $C_{lz}[i, j]$ the phrase identifier, the preorder, and the subtree size of the corresponding *LZTrie* node,

[1] The space requirement is $16uH_k(T) + o(u \log \sigma)$ bits in the worst case, if $b = \omega(\sigma)$ does not hold, for any $k = o(\log_\sigma u)$.

along with the node itself) and all reversed prefixes $P^r[1..i]$ in *RevTrie* (recording in array $C_r[i]$ the preorder and subtree size of the corresponding *RevTrie* node), the pattern occurrences are found as follows:

Occurrences of Type 1. Assume that the search for P^r in *RevTrie* yields node v_r. For every node with preorder i, such that $preorder(v_r) \leqslant i \leqslant preorder(v_r) + subtreesize(v_r)$ in *RevTrie*, with $Rev[i]$ we get the node v_{lz_i} in *LZTrie* representing a phrase B_t ending with P. The length of B_t is $d = depth(v_{lz_i})$, and the occurrence starts at position $d - m$ inside B_t. Therefore, if $p = preorder(v_{lz_i})$, the exact text position can be computed as $TPos_{lz}[p] + d - m$. We then traverse all the subtree of v_{lz_i} and report, as an occurrence of type 1, each node contained in this subtree, accessing $TPos_{lz}[p..p + subtreesize(v_{lz_i})]$ to find the text positions. Note that the offset $d - m$ stays constant for all nodes in the subtree.

Note that every node in the subtree of v_r produces a random access in *LZTrie*. In the worst case, the subtree of v_{lz_i} has only one element to report (v_{lz_i} itself), and hence we have occ_1 random accesses in the worst case. To reduce the worst case to $occ_1/2$, we use the n extra bits in *Rev*: in front of the $\log u$ bits of each *Rev* element, a bit indicates whether we are pointing to a *LZTrie* leaf. In such a case we do not perform a random access to *LZTrie*, but we use the corresponding $\log u$ bits to store the exact text position of the occurrence.

To avoid accessing the same *LZTrie* page more than once, even for different trie blocks stored in that page, for each $Rev[i]$ we solve all the other $Rev[j]$ that need to access the same *LZTrie* page. As the tries are space-efficient, many random accesses could need to access the same page.

For `count` queries we traverse the S_r array instead of *Rev*, summing up the sizes of the corresponding *LZTrie* subtrees without accessing them, therefore requiring $O(occ_1/b)$ disk accesses.

Occurrences of Type 2. For occurrences of type 2 we consider every possible partition $P[1..i]$ and $P[i+1..m]$ of P. Suppose the search for $P^r[1..i]$ in *RevTrie* yields node v_r (with preorder p_r and subtree size s_r), and the search for $P[i + 1..m]$ in *LZTrie* yields node v_{lz} (with preorder p_{lz} and subtree size s_{lz}). Then we traverse sequentially $LR[j]$, for $j = p_{lz}$ $..p_{lz} + s_{lz}$, reporting an occurrence at text position $TPos_{lz}[j] - i$ iff $LR[j] \in [p_r..p_r + s_r]$. This algorithm has the nice property of traversing arrays LR and $TPos_{lz}$ sequentially, yet the number of elements traversed can be arbitrarily larger than occ_2.

For `count` queries, since we have also array RL, we choose to traverse $RL[j]$, for $j = p_r..p_r + s_r$, when the subtree of v_r is smaller than that of v_{lz}, counting an occurrence only if $RL[j] \in [p_{lz}..p_{lz} + s_{lz}]$.

To reduce the number of accesses from $2\lceil 1 + \frac{s_{lz}}{b} \rceil$ to $\lceil \frac{2s_{lz}+1}{b} \rceil$, we interleave arrays

LR and $TPos_{lz}$, such that we store $LR[1]$ followed by $TPos_{lz}[1]$, then $LR[2]$ followed by $TPos_{lz}[2]$, etc.

Occurrences of Type 3. We find all the maximal concatenations of phrases using the information stored in C_{lz} and C_r. If we found that $P[i..j] = B_t \dots B_\ell$ is a maximal concatenation, we check whether phrase $B_{\ell+1}$ has $P[j+1..m]$ as a prefix, and whether phrase B_{t-1} has $P[1..i-1]$ as a suffix. Note that, according to the LZ78 properties, $B_{\ell+1}$ starting with $P[j+1..m]$ implies that there exists a previous phrase $B_{t'}$, $t' < \ell+1$, such that $B_{t'} = P[j+1..m]$. In other words, $C_{lz}[j+1, m]$ must not be null (i.e., phrase $B_{t'}$ must exist) and the phrase identifier stored at $C_{lz}[j+1, m]$ must be smaller than $\ell+1$ (i.e., $t' < \ell+1$). If these conditions hold, we check whether $P^r[1..i-1]$ exists in *RevTrie*, using the information stored at $C_r[i-1]$. Only if all these condition hold, we check whether $ids^{-1}[\ell+1]$ descends from the *LZTrie* node corresponding to $P[j+1..m]$ (using the preorder and subtree size stored at $C_{lz}[j+1, m]$), and if we pass this check, we finally check whether $LR[ids^{-1}[t]]$ (which yields the *RevTrie* preorder of the node corresponding to phrase $t-1$) descend from the *RevTrie* node for $P^r[1..i-1]$ (using the preorder and subtree size stored at $C_r[i-1]$). Fortunately, we have a high probability that $ids^{-1}[\ell+1]$ and $ids^{-1}[t]$ need to access the same disk page. If we find an occurrence, the corresponding position is $TPos_{lz}[ids^{-1}[t]] - (i-1)$.

***Extract* Queries.** For $\texttt{extract}(T, i, j)$ we have the following problem: given any text position i, we need to compute the corresponding *LZTrie* node so as to extract the text from there. We store in a *B-tree* the starting text position for every phrase $B_{p \cdot h}$, for $p = 1, \dots, \frac{n}{h}$ and $h > 0$. This requires $\frac{n}{h} \log u$ bits of space. We hold the root block of the *B-tree* always in main memory. Using the *B-tree* we can search for the rightmost phrase B_t with starting text position p_t, such that $p_t \leqslant i$ and p_t has been stored in the *B-tree*. From p_t we get the disk page at which we access *Node*, using the fact that the sampling in the text positions is regular. We then repeatedly access $Node[t+s]$ in *LZTrie*, for $s \geqslant 0$, adding $depth(Node[t+s])$ to p_t, so as to get the text position p_{t+s} of phrase B_{t+s}, until the sum is greater than i. At this point, we are in the *LZTrie* node corresponding to the phrase B_{t+s} containing text position i. The text can be extracted by going successively to the parent in *LZTrie*, getting the symbol at each node. Once we reach the *LZTrie* root, we go on to extract the text in the next phrase. For any $h = \omega(1)$ we have $o(u \log \sigma)$ extra bits, and we perform $\omega(1)$ extra disk accesses.

For $\texttt{display}(T, P, \ell)$ queries, we first solve $\texttt{locate}(T, P)$. Then, for each occurrence starting at position i we carry out operation $\texttt{extract}(T, i-\ell, i+m+\ell-1)$.

7.3 Experimental Results

For the experiments of this chapter we consider two text files: the text WSJ (Wall Street Journal) from the TREC collection [Har95], of 128 megabytes, and the XML file provided in the *Pizza&Chili Corpus*, downloadable from `http://pizzachili.dcc.uchile.cl/texts/xml/dblp.xml.200MB.gz`, of 200 megabytes. Although these texts are smaller than the ones used in the experiments of previous chapters, we use them in order to compare against state-of-the-art indexes, which use similar texts to perform their experiments. We searched for 5,000 random patterns, of length from 5 to 50, generated from these files. As in [FG96], we assume a disk page size of 32 kilobytes. We compared our results against the following state-of-the-art indexes for secondary storage:

***Suffix Arrays* (SA):** The suffix arrays of [BYBZ96], as explained in Section 7.1. Recall that we move to main memory the first l text symbols of the first suffix of each block (of size h), so we have $\frac{u}{h}l$ extra symbols. We assume in our experiments that $l = m$ holds, which is the best situation. The total cost is $2 + 2\log h$ disk accesses, plus $\lceil 1 + \frac{occ-1}{b} \rceil$ extra accesses to report the occurrences of P. The space requirement including the text is $(5 + \frac{m}{h})$ times the text size. In the experiments, we use $h = 1, 2, 4, 8, 16,$ and 32.

***String B-trees* [FG99]:** in [FG96] they pointed out that an implementation of *String B-trees* for static texts would require about $2 + \frac{2.125}{t}$ times the text size (where $t > 0$ is a constant) and the height h of the tree is 3 for texts of up to 2 gigabytes, since the branching factor (number of children of each tree node) is $b' \approx \frac{b}{8.25}$. The experimental number of disk accesses given by the authors is $O(\log t)(\lfloor \frac{m}{b} \rfloor + 2h) + \lceil \frac{occ}{b'} \rceil$. Notice that this model assumes that the trie root is kept in main memory. We assume a constant of 1 for the $O(\log t)$ factor, since this is not clear in the paper [FG96, Sect. 2.1]. We use $t = 2$, 4, 8, 16, and 32.

***Compact Pat Trees* (CPT) [CM96]:** we assume that the tree has height 3, according to the experimental results of Clark and Munro, and assume that the trie root is kept in main memory. We need $1 + \lceil \frac{occ-1}{b} \rceil$ extra accesses to locate the pattern occurrences. According to the original paper, the space is about 4–5 times the text size (plus the text). Hence, the space requirement of this index is shown in our plots as a line between 5 and 6 times the text size, meaning that the space requirement lies within this range.

***Locally-Compressed Suffix Arrays* (LCSA) [GN08a]:** we show the same results as in the original work [GN08a]. It is important to note that this index needs some data structures to reside in main memory, as for instance the Re-Pair dictionary used to decompress the suffix-array entries. According to the experiments in [GN08a],

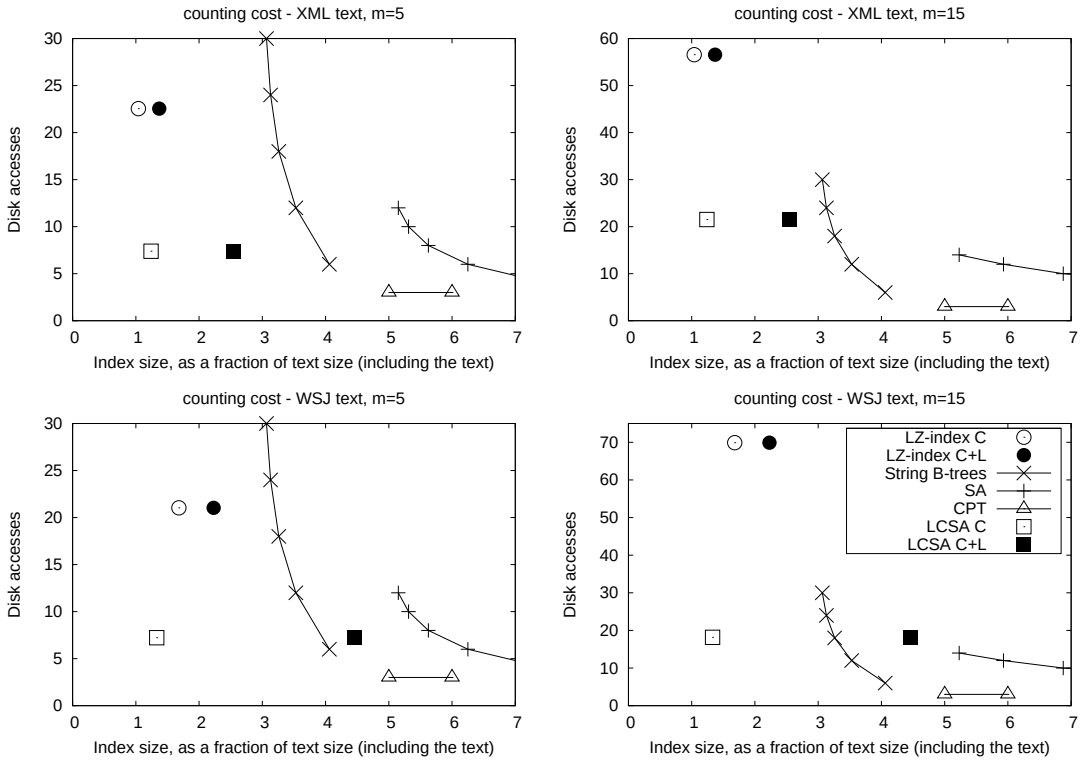


Figure 7.1: `Count` cost versus space requirement for the different indexes we tested.

the amount of main memory used is 19.15 MB for XML, and 12.54 MB for WSJ. Though this index includes enough information so as to be able to extract any text substring, this is not efficient on disk. So one would need to add an extra data structure representing the compressed text [GN07] in order to support efficient text extraction on disk. We do not account for this data structure in our experiments.

We restrict our comparison to indexes that have been implemented, or at least simulated, in the literature. Hence we exclude the *Compressed Suffix Arrays* (CSA) [MNS04] since we only know that it needs at most $2(1 + m\lceil \log_b u \rceil)$ accesses for `count` queries. This index requires about 0.22 and 0.45 times the text size for the XML and WSJ texts respectively, which, as we shall see, is smaller than ours. However, CSA requires $O(\log u)$ accesses to report *each* pattern occurrence, which is prohibitively costly.

Fig. 7.1 shows the time/space trade-offs of the different indexes for `count` queries, for patterns of length 5 and 15. For the LZ-index and the LCSA we show two version of the indexes: "`LZ-index C`" and "`LCSA C`" are the versions allowing just `count` queries, while "`LZ-index C+L`" and "`LCSA C+L`" are the versions supporting both `count` and `locate`

queries.

As it can be seen, our LZ-index requires about 1.04 times the text size for the (highly compressible) XML text, and 1.68 times the text size for the WSJ text. For $m = 5$, the counting requires about 23 disk accesses, and for $m = 15$ it needs about 69 accesses. Note that for $m = 5$, there is a difference of 10 disk accesses among the LZ-index and *String B-trees*, but the latter requires 3.39 (XML) and 2.10 (WSJ) times the space of the LZ-index. For $m = 15$ the difference is greater in favor of *String B-Trees*. The SA outperforms the LZ-index in both cases, but the latter requires about 20% the space of SA. Finally, the LZ-index needs (depending on the pattern length) about 7–23 times the number of accesses of CPTs, but the latter requires 4.9–5.8 (XML) and 3–3.6 (WSJ) times the space of LZ-index. When comparing with the LCSA in the case of supporting just `count` queries, we can see that our index is competitive only for XML, the LCSA requiring 1.19 times the size of the LZ-index. When, on the other hand, we consider an scenario where both `count` and `locate` queries need to be supported, our LZ-index is much more competitive, the LCSA requiring 1.85 (XML) and 2.00 (WSJ) times the size of the LZ-index, in both texts with a difference of about 15 disk accesses in favor of the LCSA (for $m = 5$), and a difference of about 35 disk accesses (XML) and 50 disk accesses (WSJ) for $m = 15$.

Fig. 7.2 shows the time/space trade-offs for `locate` queries, this time showing the average number of occurrences reported per disk access. The LZ-index requires about 1.37 (XML) and 2.23 (WSJ) times the text size, and is able of reporting about 597 (XML) and 63 (WSJ) occurrences per disk access for $m = 5$, and about 234 (XML) and 10 (WSJ) occurrences per disk access for $m = 15$. We estimate the constant in the space requirement of our LZ-index, dividing the index size in bits (we previously subtract the lower order terms corresponding to the space for parentheses, symbols, etc.) by $n \log n$. We get the constants 7.71 for XML and 7.62 for WSJ, which indicate that in practice the constant is smaller than 8, the worst-case constant that we predicted in theory when $b = \omega(\sigma)$, since σ is not so big in the texts we are using, and also $b \approx 8,192$. The average number of occurrences found for $m = 5$ is 293,038 (XML) and 27,565 (WSJ); for $m = 15$ there are 45,087 and 870 occurrences on average. *String B-trees* report 3,449 (XML) and 1,450 (WSJ) occurrences per access for $m = 5$, and for $m = 15$ the results are 1,964 (XML) and 66 (WSJ) occurrences per access, but they require 2.57 (XML) and 1.58 (WSJ) times the space of the LZ-index.

Fig. 7.3 shows the cost for the different parts of the LZ-index search algorithm, for `locate` queries and in the case of XML (WSJ yields similar results): the work done in the tries (labeled "`tries`"), the different types of occurrences, and the total cost ("`total`"). The total cost can be decomposed in three components: a part linear on m (trie traversal), a part linear in *occ* (type 1), and a constant part (type 2 and 3).

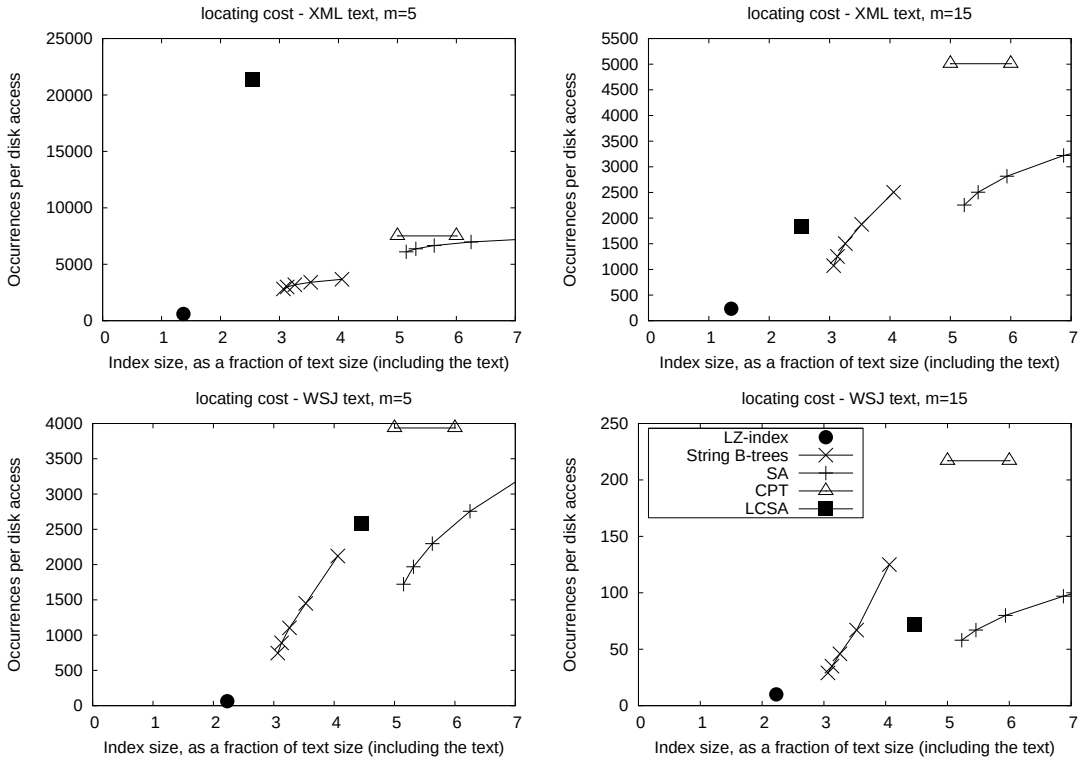


Figure 7.2: `Locate` cost versus space requirement for the different indexes we tested. Higher means better `locate` performance.

Now that we compared our LZ-index against existing alternatives, we run experiments with larger texts, in order to test more cases. We test with a 500-MB prefix of the English text from the *Pizza&Chili Corpus*, and a 500-MB prefix of the Human Genome. For the English text, our LZ-index requires 2.49 times the text size for the complete index, and 1.78 times the text size if we only want to use the index for `count` queries. The constant in the main component of the space requirement is 7.40. At search time, we are able to locate on average about 687 occurrences per disk access for patterns of length 5, and about 34 occurrences per access for patterns of length 15. For the Human Genome, our LZ-index requires 2.07 times the text size, and 1.48 times the text size if we only need to count occurrences. The constant in the main component of the space requirement is 7.38. At search time, we are able to locate on average about 1,984 occurrences per access for patterns of length 5, and about 81 occurrences per access for patterns of length 15.

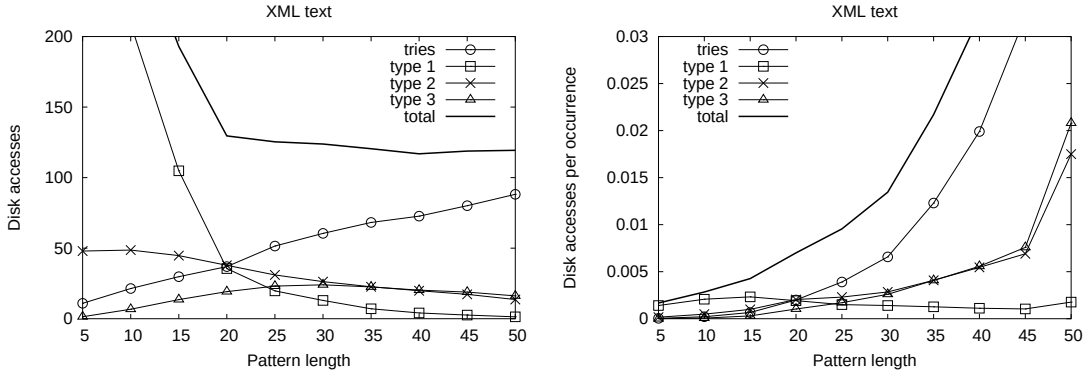


Figure 7.3: Cost for the different parts of the LZ-index search algorithm, for `locate` queries and variable pattern length.

7.4 Final Comments

We can conclude that the LZ-index can be adapted to work on secondary storage, requiring $O(uH_k) + o(u \log \sigma)$ bits of space, for any $k = o(\log_\sigma u)$. In practice, this value is about 1.4–2.3 times the text size, including the text, which means 39%–65% the space of *String B-trees* [FG99]. Saving space in secondary storage is important not only by itself (space is very important for storage media of limited size, such as CD-ROMs), but also to reduce the high seek time incurred by a larger index, which usually is the main component in the cost of accessing secondary storage, and is roughly proportional to the size of the data.

Our index is significantly smaller than any other practical secondary-memory data structure. In exchange, it requires more disk accesses to locate the pattern occurrences. For XML text, we are able to report (depending on the pattern length) about 597 occurrences per disk access, versus 3,449 occurrences reported by *String B-trees*. For English text (WSJ file from [Har95]), the numbers are 63 versus 1,450 occurrences per disk access. In many applications, it is important to find quickly a few pattern occurrences, so as to find the remaining while processing the first ones, or on user demand, recall partial `locate` queries, defined in Section 1.1.3. Fig. 7.3 (left, see the line "`tries`") shows that for $m = 5$ we need about 11 disk accesses to report the first pattern occurrence, while *String B-trees* need about 12. If we only want to count the pattern occurrences, the space can be dropped to $6uH_k + o(u \log \sigma)$ bits; in practice 1.0–1.7 times the text size. This means 29%–48% the space of *String B-trees*, with a slowdown of 2–4 in the time.

We have considered only the number of disk accesses in our work, ignoring seek times. Random seeks cost is roughly proportional to the size of the data. If we multiply the number of accesses by the index size, we get a very rough idea of the overall seek times. The smaller size of our LZ-index should favor it in practice. For example, it is very close

to *String B-trees* for counting on XML and $m = 5$ (Fig. 7.1). This model is optimistic, but counting only accesses is pessimistic.

Now that we know that the LZ-index can be adapted to work on secondary storage, the construction of our index becomes an important issue. An alternative would be to construct our index in main memory of a bigger machine, then generating the secondary-memory version, to finally move the index to the (smaller) machine used for queries. Since the index components are similar to the ones defined for the LZ-index versions of previous chapters, we can adapt to our index the reduced-memory construction algorithm of Section 6.3. The difference is that this time we cannot provide guarantees in the number of disk accesses needed to construct the index, since we need to navigate the tries. So we could construct our LZ-index using just $uH_k(T) + o(u \log \sigma)$ bits. Notice that the machine used for queries will have less than $uH_k(T) + o(u \log \sigma)$ bits of main memory available, because otherwise we would use the index of Theorem 5.1 directly in main memory. As a future work, we plan to study the direct construction on secondary storage of our LZ-index. We think that this is feasible, adapting the method of Chapter 6 to work on disk.

An important line of future research is to achieve good worst-case guarantees at search time. For example, we should represent the tries with a data structure supporting secondary-storage access [FGG$^+$08], as well as efficient support for occurrences of type 1 (recall the random pattern of accesses from *RevTrie*). Occurrences of type 2, on the other hand, should be found by using the *Range* data structure on disk. Another interesting line of research is that of studying the best ways to use the available main memory in order to reduce the number of disk accesses performed by the index at search time. Another plan for future work is to handle dynamism.

Chapter 8

Conclusions and Further Work

Since about a decade ago, the track of compressed full-text self-indexes is a very active area of research, aiming at both theoretical and practical results with diverse applications. As a consequence, the area has grown very fast, achieving unexpected (and sometimes even unsuspected) results, as for instance the fact that we are able to replace a text by a compressed representation of it which, besides supporting indexed-text-search capabilities, is also supporting the extraction of any text substring in optimal time (i.e., in the same time as with the text at hand). Undoubtedly, this has been a breakthrough in the areas of text compression and string matching.

However, given the popularity of classical text indexes like suffix trees and suffix arrays, the attention of researchers has been biased towards suffix-array-based compressed indexes. As an example, most of the literature only considers this kind of indexes when studying compressed full-text self-indexes (see, e.g., [MN08a]). There are, however, other kind of compressed full-text self-indexes which we believe deserve attention. In this thesis we made a deep study on Lempel-Ziv compressed full-text self-indexes (LZ-indexes). This was the first family of compressed indexes [KU96a, Kär99], although they did not receive much attention from researchers, until recently [FM05, Nav04, RO07]. We base our study on Navarro's LZ-index [Nav04].

As a result from our study, we get a new family of LZ-indexes, which are space efficient, competitive at search time with the best existing alternatives, practical, able to be built within compressed space, and adaptable to work on secondary storage when the indexed text is very large. Thus, we can conclude that the LZ-indexes can be also effective and competitive, outperforming the suffix-array based compressed indexes in many key aspects, as for example more efficient support for operations `extract` and `display`, which are essential for self-indexes since the text is not available otherwise. The faster construction of our indexes makes them suitable for scenarios where the index must be built and queried on the fly, as for instance in cases where there are updates in the text and one wants to

quickly give access to the latest text version that has arrived (e.g., for indexing blogs or online newspapers).

8.1 Summary of Main Contributions

Let $T[1..u]$ be a text over an alphabet of size σ, and let $H_k(T)$ denote the k-th order empirical entropy of T. Let $P[1..m]$ be a search pattern over the same alphabet. From our study, we can draw the following general conclusions (see the final comments at the end of each chapter for more specific ones):

8.1.1 A method to reduce the space requirement of LZ-indexes

We have defined in Chapter 4 a method to reduce the space requirement of LZ-indexes. Our method provides a way to add space/time trade-offs to the LZ-index, alleviating this drawback of the original index. Thus, we obtain LZ-indexes whose size is up to 2/3rd of the size of the original LZ-index, while still retaining much of the good features of it (e.g., fast extraction of text substrings and fast locating of pattern occurrences). Though we are not able to provide worst-case guarantees at search time, our indexes are very effective in practice: they are in most cases the most efficient for extracting text substrings, as well as displaying occurrence contexts, which we argued are the most basic operations a self-index must support, since the text is not available otherwise and displaying queries are the most frequent ones. The extracting rate is about 1 to 1.5 million symbols per second, being about twice as fast as the most competitive alternatives. Our indexes are also competitive (though not always the best) with the best existing compressed self-indexes for locating pattern occurrences. However, in scenarios where one does not need to retrieve all the pattern occurrences, but just a fraction of them, we show that our indexes are competitive, specifically in cases of short patterns, small alphabets, or retrieving just a few occurrences. We also conclude that our indexes are very competitive in cases of highly compressible texts, since we obtain smaller indexes which are still fast. In these cases, suffix-array-based compressed indexes need to add extra non-compressible information in order to compete, thus requiring much more space.

We made the prototypes of our LZ-indexes publicly available in the *Pizza&Chili* corpus, throughout the site `http://pizzachili.dcc.uchile.cl/indexes/LZ-index/`.

8.1.2 A family of stronger LZ-indexes

Pushing further the previous approaches to reduce the space requirement of LZ-indexes, we studied in Chapter 5 a further reduction in space requirement (which was not possible anymore by using the approach of Chapter 4). The result is an even smaller index, which can be then augmented to provide worst-case guarantees at search time. Moreover, we

improve the search time of the original LZ-index. In summary, starting with the original space requirement of $4uH_k(T) + o(u \log \sigma)$ bits of space, for any $k = o(\log_\sigma u)$, and original locate time $O(m^3 \log \sigma + (m + occ) \log u)$, we obtain LZ-indexes with the following features:

— $(1 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits of space and $O(\frac{m^2}{\epsilon})$ average locate time for $m \geqslant 2 \log_\sigma u$. This is the smallest existing LZ-index, yet we cannot provide worst-case guarantees at search time.

— $(2 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits of space and $O(\frac{m^2}{\epsilon} + (m + occ) \log u + \frac{occ}{\epsilon})$ worst-case locate time. This is the smallest LZ-index that provides worst-case guarantees at search time. We showed that this index is competitive against state-of-the-art indexes, outperforming them while requiring about the same space.

— $(3 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits of space and $O((m + \frac{occ}{\epsilon}) \log u)$ worst-case locate time. This index requires about half of the space required by competing schemes and achieves the same time complexity.

So we have a novel family of LZ-indexes, with space ranging from $(1 + \epsilon)$ to $(3 + \epsilon)$ times the size of the compressed text (plus lower-order additive terms). In all cases, our indexes are competitive with state-of-the-art indexes.

8.1.3 A space-efficient method to construct the LZ-indexes

We have defined a space-efficient algorithm to build the LZ-indexes, which allows us to construct each of the indexes of the previous paragraph in $O(u(\log \sigma + \log \log u))$ time, and without requiring extra space on top of that required by the final index. This is very important for the applicability of the indexes, since wherever these can be used, we will be able to construct them. We also defined a method to construct the indexes when the main-memory space available for the indexing process is smaller than that required by the final index. Our result is that we can construct our LZ-indexes in $O(u(\log \sigma + \log \log u))$ time while requiring just $(1+\epsilon)uH_k(T)+o(u \log \sigma)$ bits of main-memory space. This means that we are able to construct our LZ-indexes whenever we have slightly more main-memory space on top of the space needed to store the compressed text. This has applications in text search engines, where we can use a less powerful computer to carry out the indexing process, devoting a more powerful one to answer user queries.

On the practical side, we achieve an indexing algorithm that is very competitive with the best existing indexing technologies. As an example, we are able to index the Human Genome (of about 3×10^9 base pairs) in less than 5 hours (on a 3GHz CPU), and requiring only 1.6 GB of main memory space (which is about half of the space required by the genome if we assume that ASCII codes are used to represent the bases). This is much faster than the space-efficient indexing algorithms for competing schemes: 24 hours for

Compressed Suffix Arrays [Hon04] and 28 hours for FM-index [Hon04] (both on a 1.7GHz CPU), and less than 9 hours for suffix arrays, using secondary storage to construct them [DKMS08]. There are scenarios where one needs to construct the index and query it on the fly. Our indexes are superior in such scenarios.

Cardinal trees, or tries, are a fundamental data structure for pattern-matching applications (among many other in Computer Science). Since the space-efficient construction of LZ-index is highly related to the maintenance of succinct dynamic cardinal trees, we achieve the first representation for that kind of trees reaching space close to the information-theoretic lower bound of $2n + n\log\sigma + o(n\log\sigma)$ bits, supporting basic operations in $O(\log\sigma + \log\log u)$ time (amortized in the case of update operations).

We summarize the results of Sections 8.1.2 and 8.1.3 in the following Corollaries, which characterize the new family of LZ-indexes:

Corollary 8.1. *Given a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, and let n be the number of phrases in the LZ78 parsing of T, there exists a compressed full-text self-index requiring $(1+\epsilon)uH_k(T) + o(u\log\sigma)$ bits of space, for $\log\sigma = o(\log u)$, any $k = o(\log_\sigma u)$ and any $0 < \epsilon < 1$. Given a search pattern $P[1..m]$, this index is able to:*

(1) locate (and count) the occ occurrences of pattern P in text T in $O(\frac{m^2}{\epsilon} + \frac{n}{\epsilon\sigma^{m/2}})$ average time, which is $O(\frac{m^2}{\epsilon})$ if $m \geqslant 2\log_\sigma n$; and

(2) extract a text substring of length ℓ surrounding any text position in optimal $O(1 + \frac{\ell}{\epsilon\log_\sigma u})$ worst-case time.

This index can be constructed in $O(u(\log\sigma + \log\log u))$ time and without requiring any extra space.

Then, we have:

Corollary 8.2. *Given a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, there exists a compressed full-text self-index requiring $(2+\epsilon)$ $uH_k(T) + o(u\log\sigma)$ bits of space, for $\log\sigma = o(\log u)$, any $k = o(\log_\sigma u)$ and any $0 < \epsilon < 1$. Given a search pattern $P[1..m]$, this index is able to:*

(1) locate the occ occurrences of pattern P in text T in $O(\frac{m^2}{\epsilon} + (m + occ)\log u + \frac{occ}{\epsilon})$ worst-case time;

(2) count the number of pattern occurrences in $O(\frac{m^2}{\epsilon} + m\log u + \frac{occ}{\epsilon})$ worst-case time;

(3) determine whether pattern P exists in T in $O(\frac{m^2}{\epsilon} + m\log u)$ worst-case time; and

(*4*) *extract a text substring of length ℓ surrounding any text position in optimal $O(1+\frac{\ell}{\epsilon \log_\sigma u})$ worst-case time.*

This index can be constructed in $O(u(\log\sigma + \log\log u))$ time and without requiring any extra space. The indexing space can be reduced to $(1+\epsilon)uH_k(T) + o(u\log\sigma)$ bits, in such a case performing $(u\log\sigma + 4uH_k(T) + o(u\log\sigma))/B$ disk accesses (where B is the size of a disk page, in bits).

It is important to note that with the index of Corollary 8.2 we are achieving the same result as the classical LZ-index of Kärkkäinen and Ukkonen [KU96a, Kär99], yet with a much smaller index which does not need the text to operate. Notice, however, that we are still able to access any text substring in optimal time, without having to store the text as the classical index does.

Finally, we have:

Corollary 8.3. *Given a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, there exists a compressed full-text self-index requiring $(3+\epsilon)uH_k(T) + o(u\log\sigma)$ bits of space, for $\log\sigma = o(\log u)$, any $k = o(\log_\sigma u)$ and any $0 < \epsilon < 1$. Given a search pattern $P[1..m]$, this index is able to:*

(*1*) *locate the occ occurrences of pattern P in text T in $O((m + \frac{occ}{\epsilon})\log u)$ worst-case time;*

(*2*) *count the number of pattern occurrences in $O(m(1 + \frac{\log\sigma}{\log\log u}))$ worst-case time;*

(*3*) *determine whether pattern P exists in T in $O(m(1 + \frac{\log\sigma}{\log\log u}))$ worst-case time; and*

(*4*) *extract a text substring of length ℓ surrounding any text position in optimal $O(\ell/(\epsilon\log_\sigma u))$ worst-case time.*

This index can be constructed in $O(u\log u(1 + \frac{\log\sigma}{\log\log u}))$ time and without requiring any extra space. The indexing space can be reduced to $(1+\epsilon)uH_k(T) + o(u\log\sigma)$ bits, in such a case performing $(u\log\sigma + 4uH_k(T) + o(u\log\sigma))/B$ disk accesses (where B is the size of a disk page, in bits).

8.1.4 An efficient LZ-index working on secondary storage

We have defined an LZ-index version which is able to efficiently work on secondary storage. Based on the study done for reducing the space of LZ-indexes, we added some redundancy to the index in order to transform random accesses to index components into sequential scanning of arrays, which is much more efficient for secondary storage devices. We end

up with an index requiring $O(uH_k(T)) + o(u \log \sigma)$ bits of space. Though not providing worst-case guarantees at search time, we obtained in practice the smallest existing full-text index working on disk, with a very competitive performance, specifically for locating pattern occurrences: we are able to report about 65–600 (depending on the indexed text) occurrences per access, versus 1,450–3,450 occurrences per access of *String B-Trees* [FG99]. Yet, our index is significantly smaller, requiring 39%–65% the space of *String B-Trees*.

8.2 Lines for Future Research

These are the most important lines for future research related to our work:

— Since the results obtained in the definition of our disk-based LZ-index are promising, it would be interesting to get an actual implementation of it (currently we only have a simulator). As usual, it should be accompanied of a complete engineering process, in order to get a competitive implementation. This prototype should be used also to study the best ways to profit from the available main-memory space, reducing the number of disk accesses. Some possibilities could be to use the available space to store the first levels of the tries in main memory, thus reducing the cost of navigating the tries. Another possibility could be to store some *LZTrie* subtrees in main memory in order to reduce the amount of random accesses due to occurrences of type 1.

— It would be interesting to study a way of adding worst-case guarantees to our disk-based LZ-index. For example, we could represent the tries with a data structure supporting secondary-storage access [FGG$^+$08], as well as efficient support for occurrences of type 1 (recall the random pattern of accesses from *RevTrie*). Occurrences of type 2, on the other hand, should be found by using a representation of the *Range* data structure on disk. Another important issue for future research is the direct construction of the LZ-index on disk.

— Adding dynamism to full-text indexes is an area of research that has not been explored in depth, though it has an important number of applications. The idea of a dynamic LZ-index would be to improve the $O(\log^2 n)$ time per occurrence incurred by existing dynamic compressed full-text self-indexes. To represent the dynamic LZ-index, we could use the dynamic representation defined for the space-efficient construction algorithm. However, there are many issues that must be solved, as for instance how to efficiently update the external pointers to the nodes of a given trie block, after we insert a new node in that block (e.g., how to efficiently update the *Node* data structure in such a case).

— Nowadays, many applications produce texts which are highly self-repetitive (e.g., think of a collection of genomes or a collection with the source codes for the different

versions of a given program). Hence, it is important to have techniques which profit from this property to reduce the space of the resulting self-index. It is well known that LZ78 is not good in catching long repetitions in the text, since the very restrictive parsing destroys most of them [SVMN08]. However, it could be more amenable to work on indexes based on the LZ77 compression algorithm. This seems to be the most promising future line for LZ-indexes, yet it is full of challenges for self indexing.

— LZ78 can be seen as a grammar-based compressor [CLL+05]. It would be interesting to study other kinds of grammar compression techniques to define compressed self-indexes. For example, indexes based on straight-line programs [CN08a]. This kind of compression was shown to be very effective for local decompression [GN07], which is very important when dealing with indexes stored on disk.

— In this thesis we showed different representations for the tries that compose the LZ-index. In particular, we showed that the DFUDS representation is effective when working with larger alphabets. An example of application dealing with large alphabets is that of indexing natural-language texts, since in that case the words that compose the text are mapped to symbols in a larger alphabet. Compressed Suffix Arrays have already been adapted to work on natural-language texts [BFN+08]. Since the LZ-index is very effective for queries with short patterns, an LZ-index on words could profit from the usually short queries given by users. Notice that a query with only one word consists only of occurrences of type 1 in the LZ-index, which can be found very fast in $O(1)$ time per occurrence. A query with a phrase of length 2, on the other hand, consists only of occurrences of types 1 and 2. Let n_1 be the number of occurrences of the first word composing the phrase, and let n_2 be the number of occurrences of the second word. The occurrences of the whole phrase can be determined in $O(\min\{n_1, n_2\})$ time, by using the process to find occurrences of type 2 in practice (i.e., checking by hand the possible candidate occurrences one by one). This should be compared with the time achieved by classical solutions to the problem of natural-language indexing, as for instance inverted indexes. These indexes must solve this kind of queries by means of intersecting the posting lists for the occurrences of the first and second word of the phrase, which takes time $O(\min\{n_1, n_2\} \cdot \log(\max\{n_1, n_2\}))$ with the approach of [CM07].

— Classical full-text search (i.e., the one assumed in this thesis, where the exact pattern occurrences need to be found) is not enough in some applications, like biological research, music and image retrieval, etc. In these cases, more complex search capabilities are needed, as for instance approximate full-text search, regular-expression search, etc. It would be interesting to add these capabilities to our indexes [RNO07].

Bibliography

[ABR00] S. Alstrup, G. Brodal, and T. Rahue. New data structures for orthogonal range searching. In *Proc. 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 198–207, 2000.

[AE99] P. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, pages 1–56. Contemporary Mathematics 223, American Mathematical Society Press, 1999.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AN05] D. Arroyuelo and G. Navarro. Space-efficient construction of LZ-index. In *Proc. 16th Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS 3827, pages 1143–1152, 2005.

[AN07a] D. Arroyuelo and G. Navarro. A Lempel-Ziv text index on secondary storage. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 83–94, 2007.

[AN07b] D. Arroyuelo and G. Navarro. Smaller and faster LZ-indices. In *Proc. International Workshop on Combinatorial Algorithms (IWOCA)*, pages 11–20. College Publications, 2007.

[AN08] D. Arroyuelo and G Navarro. Practical approaches to reduce the space requirement of Lempel-Ziv-based compressed text indices. Technical Report TR/DCC-2008-9, Department of Computer Science, University of Chile, 2008. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/smallerlzpract.ps.gz`. Submitted.

[AN09] D. Arroyuelo and G. Navarro. Space-efficient construction of Lempel-Ziv compressed text indexes. Technical Report TR/DCC-2009-2, Department of Computer Science, University of Chile, 2009.

[ANS06] D. Arroyuelo, G. Navarro, and K. Sadakane. Reducing the space requirement of LZ-index. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009, pages 319–330, 2006.

[ANS08] D. Arroyuelo, G. Navarro, and K. Sadakane. Stronger Lempel-Ziv based compressed text indexing. Technical Report TR/DCC-2008-2, Department of Computer Science, University of Chile, 2008.

[AOK02] M. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In *Proc. 9th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 2476, pages 31–43, 2002.

[Apo85] A Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.

[Arr08] D. Arroyuelo. An improved succinct representation for dynamic k-ary trees. In *Proc. 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 5029, pages 277–289, 2008.

[BCD+99] A. Brodnik, S. Carlsson, E. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In *Proc. 6th International Workshop on Algorithms and Data Structures (WADS)*, LNCS 1663, pages 37–48, 1999.

[BCW90] T. Bell, J. Cleary, and I. Witten. *Text Compression*. Prentice–Hall, 1990.

[BDM+05] D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[BFN+08] N. Brisaboa, A. Fariña, G. Navarro, A. Places, and E. Rodríguez. Self-indexing natural language. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 121–132, 2008.

[BHMR07] J. Barbay, M. He, J. I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 680–689, 2007.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[BYBZ96] R. Baeza-Yates, E. F. Barbosa, and N. Ziviani. Hierarchies of indices for text searching. *Information Systems*, 21(6):497–514, 1996.

[Cha88] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[CHLS07] H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms (TALG)*, 3(2):article 21, 2007.

[CHSV08] Y.-F. Chien, W.-K. Hon, R. Shah, and J. S. Vitter. Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In *Proc. Data Compression Conference (DCC)*, pages 252–261, 2008.

[CLL+05] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. Prentice–Hall, second edition, 2001.

[CM96] D. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 383–391, 1996.

[CM07] S. Culpepper and A. Moffat. Compact set representation for information retrieval. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 137–148, 2007.

[CN08a] F. Claude and G. Navarro. Indexing straight line programs. Technical Report TR/DCC-2008-15, Department of Computer Science, University of Chile, 2008.

[CN08b] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187, 2008.

[dBCvKO08] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, third edition, 2008.

[DKMS08] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *ACM Journal of Experimental Algorithmics (JEA)*, 12:1–24, article 3.4, 2008.

[Eli75] P. Elias. Universal codeword sets and representation of integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[FG96] P. Ferragina and R. Grossi. Fast string searching in secondary storage: theoretical developments and experimental results. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 373–382, 1996.

[FG99] P. Ferragina and R. Grossi. The String B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[FGG+08] P. Ferragina, R. Grossi, A. Gupta, R. Shah, and J. S. Vitter. On searching compressed string collections cache-obliviously. In *Proc. 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 181–190, 2008.

[FGGV06] L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms (TALG)*, 2(4):611–639, 2006.

[FGNV08] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice! *ACM Journal of Experimental Algorithmics (JEA)*, 2008. To appear. See also `http://pizzachili.dcc.uchile.cl/resources/cti_jea07.pdf`.

[FKS84] M. Fredman, J. Komlós, and E. Szemeréd. Storing a sparse table with 0(1) worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[FLMM05] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 184–196, 2005.

[FM00] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.

[FM01] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 269–278, 2001.

[FM05] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 54(4):552–581, 2005.

[FM07] G. Franceschini and S. Muthukrishnan. In-place suffix sorting. In *Proc. 34th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 4596, pages 533–546, 2007.

[FMMN04] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proc. 11th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 3246, pages 150–160, 2004.

[FMMN07] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):article 20, 2007.

[FMN06] K. Fredriksson, V. Mäkinen, and G. Navarro. Flexible music retrieval in sublinear time. *International Journal of Foundations of Computer Science (IJFCS)*, 17(6):1345–1364, 2006.

[FMP95] F. Fich, J. I. Munro, and P. Poblete. Permuting in place. *SIAM Journal on Computing*, 24(2):266–278, 1995.

[FN05] P. Ferragina and G Navarro. Pizza&Chili Corpus — Compressed indexes and their testbeds, 2005. `http://pizzachili.dcc.uchile.cl`.

[Fre91] K. Frenkel. The human genome project and informatics. *Communications of the ACM*, 34(11):41–51, 1991.

[Gag06] T. Gagie. Large alphabets and incompressibility. *Information Processing Letters*, 99(6):246–251, 2006.

[GBYS92] G. Gonnet, R. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In W. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, chapter 5, pages 66–82. Prentice-Hall Publishers, 1992.

[GGMN05] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Experimental and Efficient Algorithms (WEA)*, pages 27–38. CTI Press and Ellinika Grammata, 2005.

[GGV03] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[GGV04] R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: experiments with compressing suffix arrays and applications. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 636–645, 2004.

[GMR06] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: A tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

[GN07] R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 216–227, 2007.

[GN08a] R. González and G. Navarro. A compressed text index on secondary memory. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 2008. To appear.

[GN08b] R. González and G. Navarro. Rank/select on dynamic compressed sequences and applications. *Theoretical Computer Science*, 2008. To appear.

[GRR06] R. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms (TALG)*, 2(4):510–534, 2006.

[GRRR06] R. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368(3):231–246, 2006.

[GS06] R. Grossi and K. Sadakane. Squeezing succinct data structures into entropy bounds. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1230–1239, 2006.

[GV00] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 397–406, 2000.

[GV05] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[Har95] D. Harman. Overview of the third text REtrieval conference. In *Proc. 3rd Text REtrieval Conference (TREC-3)*. NIST Special Publication 500-207, 1995.

[HLS+04a] W.-K. Hon, T.-W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yu. Compressed index for dynamic text. In *Proc. Data Compression Conference (DCC)*, pages 102–111, 2004.

[HLS+04b] W.-K. Hon, T.-W. Lam, W.-K. Sung, W.-L. Tse, C.-K. Wong, and S.-M. Yiu. Practical aspects of Compressed Suffix Arrays and FM-index in searching DNA sequences. In *Proc. 6th Workshop on Algorithm Engineering and Experimentation (ALENEX)*, pages 31–38, 2004.

[HLS+07] W.-K. Hon, T. W. Lam, K. Sadakane, W.-K. Sung, and M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 48(1):23–36, 2007.

[HLSS03] W. K. Hon, T. W. Lam, K. Sadakane, and W. K. Sung. Constructing compressed suffix arrays with large alphabets. In *Proc. 14th Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS 2906, pages 240–249, 2003.

[HMR05] M. He, J. I. Munro, and S. S. Rao. A categorization theorem on suffix arrays with applications to space-efficient text indexes. In *Proc. the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 23–32, 2005.

[Hon04] W.-K. Hon. *On the construction and application of compressed text indexes*. PhD thesis, University of Hong Kong, Hong Kong, 2004.

[HSS03a] W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 251–260, 2003.

[HSS03b] W.-K. Hon, K. Sadakane, and W.-K. Sung. Succinct data structures for searchable partial sums. In *Proc. 14th Annual International Symposium on Algorithms and Computation (ISAAC)*, pages 505–516, 2003.

[HTSW03] J. Healy, E. Thomas, J. Schwartz, and M. Wigler. Annotating large genomes with exact word matches. *Genome Research*, 13:2306–2315, 2003.

[Hum05] The Human Genome Project, 2005. `http://www.ornl.gov/sci/techresources/Human_Genome/home.shtml`.

[Jac89] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[JSS07a] J. Jansson, K. Sadakane, and W.-K. Sung. Compressed dynamic tries with applications to LZ-compression in sublinear time and space. In *Proc. 27th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 4855, pages 424–435, 2007.

[JSS07b] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 575–584, 2007.

[KA03] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 2719, pages 200–210, 2003.

[Kär95] J. Kärkkäinen. Suffix cactus: a cross between suffix tree and suffix array. In *Proc. 6th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 937, pages 191–204, 1995.

[Kär99] J. Kärkkäinen. *Repetition-based text indexes*. PhD thesis, Department of Computer Science, University of Helsinki, Finland, 1999.

[KM99] R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.

[KNKP05] D. Kim, J. Na, J. Kim, and K. Park. Efficient implementation of rank and select functions for succinct representation. In *Proc. 4th Workshop on Experimental and Efficient Algorithms (WEA)*, pages 315–327. LNCS 3503, 2005.

[Knu73] D. Knuth. *The Art of Computer Programming – Volume 3: Sorting and Searching*. Addison-Wesley, 1973.

[KR03] J. Kärkkäinen and S. Rao. Full-text indexes in external memory. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies*, volume LNCS 2625, chapter 7, pages 149–170. Springer-Verlag Berlin, 2003.

[KS00] C. Knessl and W. Szpankowski. Height in a digital search tree and the longest phrase of the Lempel-Ziv scheme. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 187–196, 2000.

[KS03] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 2719, pages 943–955, 2003.

[KSPP03] D. Kim, J. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 2676, pages 186–199, 2003.

[KU96a] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP)*, pages 141–155. Carleton University Press, 1996.

[KU96b] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. 2nd Annual International Conference on Computing and Combinatorics (COCOON)*, LNCS 1090, pages 219–230, 1996.

[Kur99] S. Kurtz. Reducing the space requeriments of suffix trees. *Software - Practice and Experience*, 29(13):1149–1171, 1999.

[LM99] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Proc. Data Compression Conference (DCC)*, pages 296–305, 1999.

[LS99] N. J. Larsson and K. Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99–214, Lund University, 1999.

[LSSY02] T. W. Lam, K. Sadakane, W. K. Sung, and S. M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. In *Proc. 8th Annual International Conference on Computing and Combinatorics (COCOON)*, pages 401–410, 2002.

[LY08] H.-I. Lu and C.-C. Yeh. Balanced parentheses strike back. *ACM Transactions on Algorithms (TALG)*, 4(3):1–13, 2008.

[LZ76] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

[Mäk00] V. Mäkinen. Compact suffix array. In *Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 305–319, 2000.

[Mäk03] V. Mäkinen. Compact Suffix Array — a space-efficient full-text index. *Fundamenta Informaticae*, 56(1–2):191–210, 2003.

[Man01] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[Mat94] J. Matoušek. Geometric range searching. *ACM Computing Surveys*, 26(4):422–461, 1994.

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[MF04] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

[MM93] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[MN05] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

[MN07] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.

[MN08a] V. Mäkinen and G. Navarro. Compressed text indexing. In M.-Y. Kao, editor, *Encyclopedia of Algorithms*, pages 176–178. Springer, 2008.

[MN08b] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 4(3):article 32, 2008. 38 pages.

[MNS04] V. Mäkinen, G. Navarro, and K. Sadakane. Advantages of backward searching — efficient secondary memory and distributed implementation of compressed suffix arrays. In *Proc. 15th Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS 3341, pages 681–692, 2004.

[MNZBY00] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139, 2000.

[Mor68] D. R. Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[MR01] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[MRRR03] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 2719, pages 345–356, 2003.

[MRS01] J. I. Munro, V. Raman, and A. Storm. Representing dynamic binary trees succinctly. In *Proc. 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 529–536, 2001.

[Mun96] I. Munro. Tables. In *Proc. 16th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.

[Nav04] G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.

[Nav08] G. Navarro. Implementing the LZ-index: Theory versus practice. *ACM Journal of Experimental Algorithmics (JEA)*, 13:article 1.2, 49 pages, 2008.

[NM07] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[NP07] J. Na and K. Park. Alphabet-independent linear-time construction of compressed suffix arrays using $o(n \log n)$-bit working space. *Theoretical Computer Science*, 385(1–3):127–136, 2007.

[NR02] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings — Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002.

[NR04] G. Navarro and M. Raffinot. Practical and flexible pattern matching over ziv-lempel compressed text. *Journal of Discrete Algorithms (JDA)*, 2(3):347–371, 2004.

[OS07] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. Workshop on Algorithm Engineering and Experimentation (ALENEX)*, pages 60–70, 2007.

[Ove88] M. Overmars. Efficient data structures for range searching on a grid. *Journal of Algorithms*, 9:254–275, 1988.

[Ram96] R. Raman. Priority queues: small, monotone and trans-dichotomous. In *Proc. 4th European Symposium on Algorithms (ESA)*, LNCS 1136, pages 121–137, 1996.

[RNO07] L. Russo, G. Navarro, and A. Oliveira. Approximate string matching with Lempel-Ziv compressed indexes. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 264–275, 2007.

[RO06] L. Russo and A. Oliveira. A compressed self-index using a ziv-lempel dictionary. In *Proc. 13th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4209, pages 163–180, 2006.

[RO07] L. Russo and A. Oliveira. A compressed self-index using a Ziv-Lempel dictionary. *Journal of Information Retrieval*, 5(3):501–513, 2007.

[RR03] R. Raman and S. S. Rao. Succinct dynamic dictionaries and trees. In *Proc. 30th International Colloquium on Automata, Languages and Computation (ICALP)*, LNCS 2719, pages 357–368, 2003.

[RRR02] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[Sad00] K. Sadakane. Compressed text databases with efficient query algorithms based on the Compressed Suffix Array. In *Proc. 11th Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS 1969, pages 410–421, 2000.

[Sad02] K. Sadakane. Succinct representations of *lcp* information and improvements in the Compressed Suffix Arrays. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 225–232, 2002.

[Sad03] K. Sadakane. New text indexing functionalities of the Compressed Suffix Arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[Sad08] K. Sadakane. The ultimate balanced parentheses. Technical report, Department of Computer Science and Communication Engineering, Kyushu University, 2008.

[SPMT08] R. Sinha, S. Puglisi, S. Moffat, and A. Turpin. Improving suffix array locality for fast pattern matching on disk. In *Proc. the ACM SIGMOD International Conference on Management of Data*, pages 661–672. ACM, 2008.

[SVMN08] J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 164–175. Springer, 2008.

[Ukk95] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[Vit08] J. S. Vitter. *Algorithms and Data Structures for External Memory*. Series on Foundations and Trends in Theoretical Computer Science, now Publishers, 2008.

[w3c94] The World Wide Web Consortium (W3C), 1994. `http://www.w3c.org`.

[Wei73] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1–11, 1973.

[Wel84] T. Welch. A technique for high performance data compression. *IEEE Computer Magazine*, 17(6):8–19, 1984.

[WMB99] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, New York, second edition, 1999.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.