UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

AUTO-ÍNDICE DE TEXTO BASADO EN LZ77

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

SEBASTIÁN ANDRÉS KREFT CARREÑO

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

MIEMBROS DE LA COMISIÓN:
DIEGO ARROYUELO BILLIARDI
JÉRÉMY BARBAY
NIEVES BRISABOA

Este trabajo ha sido financiado en parte por la Beca Conicyt de Magíster Nacional y por el Instituto Milenio de Dinámica Celular y Biotecnología.

SANTIAGO DE CHILE
AGOSTO 2010

Resumen

Los dominios como bioinformática, sistemas de versionamiento de código, sistemas de edición colaborativos (wikis), y otros, producen grandes colecciones de texto que son sumamente repetitivas. Esto es, existen pocas diferencias entre los elementos de la colección. Esto permite que la compresibilidad de la colección sea extremadamente alta. Por ejemplo, una colección con versiones de un mismo artículo de Wikipedia puede ser comprimida a un 0.1% de su espacio original, utilizando el esquema de compresión Lempel-Ziv de 1977 (LZ77).

Muchas de estas colecciones repetitivas contienen grandes volúmenes de texto. Es por eso que se requiere un método que permita almacenarlas eficientemente y a la vez operar sobre ellas. Las operaciones más comunes son extraer porciones aleatorias de la colección y encontrar todas las ocurrencias de un patrón dentro de la colección.

Un auto-índice es una estructura que almacena un texto en forma comprimida y permite encontrar eficientemente las ocurrencias de un patrón. Adicionalmente los auto-índices permiten extraer cualquier porción de la colección. Uno de los objetivos de estos índices es que puedan ser almacenados en memoria principal. Esta característica es sumamente importante ya que el disco puede llegar a ser un millón de veces más lento que la memoria principal.

La mayoría de los auto-índices existentes están basados en un esquema de compresión que predice los símbolos siguientes en base a los k símbolos anteriores. Este esquema, sin embargo, no funciona con textos repetitivos, ya que no es capaz de reconocer elementos repetidos en la colección. Un esquema que sí captura las repeticiones es el LZ77, pero tiene el problema de no poder acceder aleatoriamente el texto.

En este trabajo se presenta un esquema para extraer substrings de un texto comprimido con un esquema Lempel-Ziv. Adicionalmente se presenta una variante de LZ77, LZ-End, que permite extraer el texto eficientemente usando espacio cercano al de LZ77. LZ77 extrae del orden de 1 millón de caracteres por segundo, mientras que LZ-End extrae más del doble.

Nuestro resultado más importante es el primer auto-índice orientado a textos repetitivos basado en LZ77/LZ-End que supera al estado del arte (el auto-índice RLCSA). La compresión de nuestros índices llega a ser dos veces mejor en ADN y colecciones de Wikipedia que la del RLCSA. Cabe destacar que nuestro índice se construye en 35% del tiempo requerido por el RLCSA, usando el 60% de espacio de construcción. La búsqueda de patrones cortos es más rápida que en el RLCSA, y para patrones largos la relación entre espacio y tiempo es favorable a nuestros índices.

Finalmente, se presenta también una colección de textos repetitivos provenientes de diversos dominios. Esta colección está disponible públicamente con el objetivo que se pueda convertir en un referente en experimentación.

UNIVERSITY OF CHILE
FACULTY OF PHYSICS AND MATHEMATICS
DEPARTMENT OF COMPUTER SCIENCE

SELF-INDEX BASED ON LZ77

SUBMITTED TO THE UNIVERSITY OF CHILE IN FULFILLMENT OF THE THESIS REQUIREMENT TO OBTAIN THE DEGREE OF MSC. IN COMPUTER SCIENCE

SEBASTIAN KREFT

ADVISOR:
GONZALO NAVARRO

COMMITTEE:
DIEGO ARROYUELO
JÉRÉMY BARBAY
NIEVES BRISABOA

This work was partially funded by Conicyt's Master Scholarship and by the Millennium Institute for Cell Dynamics and Biotechnology (ICDB).

SANTIAGO - CHILE
AUGUST 2010

Abstract

Domains like bioinformatics, version control systems, collaborative editing systems (wiki), and others, are producing huge data collections that are very repetitive. That is, there are few differences between the elements of the collection. This fact makes the compressibility of the collection extremely high. For example, a collection with all different versions of a Wikipedia article can be compressed up to the 0.1% of its original space, using the Lempel-Ziv 1977 (LZ77) compression scheme.

Many of these repetitive collections handle huge amounts of text data. For that reason, we require a method to store them efficiently, while providing the ability to operate on them. The most common operations are the extraction of random portions of the collection and the search for all the occurrences of a given pattern inside the whole collection.

A self-index is a data structure that stores a text in compressed form and allows to find the occurrences of a pattern efficiently. On the other hand, self-indexes can extract any substring of the collection, hence they are able to replace the original text. One of the main goals when using these indexes is to store them within main memory. This characteristic is very important, as the disk may be 1 million times slower than main memory.

Most current self-indexes are based on a compression scheme that predicts the following symbol based on the previous k symbols. However, this scheme is not well suited for repetitive texts as it does not capture long-range repetitions. The LZ77 compression scheme does capture such repetitions, but it is not able to access the text at random.

In this thesis we present a scheme for random text extraction from text compressed with a Lempel-Ziv parsing. Additionally, we present a variant of LZ77, called LZ-End, that efficiently extracts text using space close to that of LZ77. LZ77 extracts around 1 million characters per second, while LZ-End extracts over 2 million.

The main contribution of this thesis is the first self-index based on LZ77/LZ-End and oriented to repetitive texts, which outperforms the state of the art (the RLCSA self-index) in many aspects. The compression of our indexes is better than that of RLCSA, being two times better for DNA and for Wikipedia articles. Our index is built using just 60% of the space required by the RLCSA and within 35% of the time. Searching for short patterns is faster than on the RLCSA, and for longer patterns the space/time trade-off is in favor of our indexes.

Finally, we present a corpus of repetitive texts, coming from several application domains. We aim at providing a standard set of texts for research and experimentation, hence this corpus is publicly available.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent times we have seen a rise in the amount of digital information. This may be attributable to the drop of the data acquisition and storage costs. Most of this information is text, that is, symbol sequences representing natural language, music, source code, time series, biological sequences like DNA and proteins, and others.

Despite that the examples presented above seem very different, there is an operation that arises in most applications handling those types of sequences. This operation is called *text search* and consists in finding all positions on the text where a given pattern appears. This operation serves as a basis for building more complex and meaningful operations, like finding the most common words, or finding approximate patterns.

Text search can be solved by two different approaches. The first scans the text sequentially looking for matches of the pattern. Classical examples of this type of search are Knuth-Morris-Pratt [KMP77] and Boyer-Moore [BM77] algorithms. The second way of searching is by querying an *index* of the text, a data structure we have to build before performing the queries. This structure allows us to find the occurrences of a given pattern without scanning the whole text.

To index the text we need enough space in order to store the index, and most importantly we need to be able to access it efficiently. Nowadays, storage is not a difficult problem, however efficient access is. In the last years the speed of hard-drives has not experienced significant improvements. Hard-drive access times are around $10ms = 10^7ns$, while main memory access (RAM) is around $10ns$; in other words, accessing secondary storage is 1 million times slower than accessing main memory. This problem is still present despite the appearance of solid state drives (SSD), which have access times around $0.1ms = 10^5ns$, being 10 thousand times slower than main memory. For this reason, indexes using space proportional to the compressed text have been proposed, aiming at storing them in main memory and handling the data directly in compressed form, rather than decompressing before using it [ZdMNBY00, NM07]. There are some indexes that, within that compressed space, are able to replace the original text; these are called self-indexes and are obviously preferable as one can discard the original text.

A particular kind of texts not yet fully benefited by current self-indexes are repetitive ones. These arise from domains that handle huge collections of very similar entries or documents. For example, in a DNA collection of human genomes of different individuals, the similarity between any two DNA sequences would be close to 99.9% [B$^+$08]. Source code collections are also very repetitive, as the changes between one version and the next are not substantial, except in the case of a major release. Versioning systems, like *wikis*, also generate very repetitive collections because each revision is very similar to the previous one. The main problem is that existing self-indexes do not sufficiently exploit these repetitions, being the self-index orders of magnitude larger than the space achievable with a compression scheme that does exploit the repetitions, like LZ77 [ZL77]. LZ77 parses the text into *phrases* so that each phrase, except its last letter, appears previously in the text (these previous occurrences are called *sources*)). It compresses by essentially replacing each phrase by a backward pointer. A recent work, aiming at adapting current self-indexes to handle large DNA databases of the same species [SVMN08] found that LZ77 compression was still much superior to capture this repetitiveness, yet it was inadequate as a format for compressed storage because of its inability to retrieve individual sequences from the collection. Another work [CN09, CFMPN10] shows that grammar-based compression can allow extraction of substrings while capturing such repetitions, yet LZ77 compression is superior to grammar compression [Ryt03, CLL$^+$05].

For these reasons in this thesis we focus on the definition of a self-index oriented to repetitive texts and based on LZ77-like compression schemes. Our main contributions are two: (1) a scheme for random text extraction in LZ77-like parsing, as well as a space-competitive variant, called LZ-End, achieving faster text extraction; (2) a self-index based on LZ77/LZ-End that achieves a better space/time trade-off than the best self-indexes oriented to repetitive texts.

1.1 Contributions of the Thesis

Chapter 3 : We create a public corpus of highly repetitive texts. The corpus is composed of texts coming from different real domains like biology, source code repositories, document repositories, and others, as well as artificial texts having interesting combinatorial properties. This corpus is available at `http://pizzachili.dcc.uchile.cl/repcorpus.html`.

Chapter 4 : The worst-case extraction time of a substring of length m in an LZ77 parsing is $O(mH)$, where H is the maximum number of times a character is transitively copied in the parsing. We present an alternative parsing, called LZ-End, that performs very close to LZ77 in terms of compression but permits faster text extraction, $O(m + H)$ worst-case time. This work was published in the *20th Data Compression Conference* [KN10].

Chapter 5 : We introduce a new self-index oriented to repetitive texts and based on the LZ77, LZ-End, and similar parsings. Let n' be the number of phrases of the parsing (for highly repetitive texts, n' will be a small value). This index uses in theory $2n' \log n + n' \log n' + n' \log D + O(n' \log \sigma) + o(n)$ bits of space, where σ is the size of the alphabet and D is upper-bounded by the maximum number of sources covering each other. It finds the *occ* occurrences of a pattern of length m in time $O(m^2 H + m \log n' + occ \cdot D \log n')$. We present several practical variants that achieve better results, both in time and space, than the *Run-length Compressed Suffix Array* (RLCSA) [SVMN08] and the *Grammar-based Self-index* [CN09, CFMPN10], the state-of-the art self-indexes oriented to repetitive texts.

1.2 Outline of the Thesis

Chapter 2 describes basic concepts and related work relevant to this thesis.

Chapter 3 presents a text corpus intended for repetitive text.

Chapter 4 explains the Lempel-Ziv (LZ) parsing and some of its properties. It also introduces a new LZ variant called LZ-End, able to extract an arbitrary substring in constant time per extracted symbol in some cases.

Chapter 5 presents a new self-index based on LZ77-like parsings. It covers the theoretical proposal and the considerations we made when implementing the index.

Chapter 6 shows the experimental results of our proposed index, comparing it with the state-of-the-art self-indexes for repetitive texts.

Chapter 7 presents our conclusions and gives some lines of research that can be further investigated.

Chapter 2

Basic Concepts

In this chapter we introduce the basic concepts and notation used through this thesis. Then, we present the data structures used to build our index. Finally, we present two self-indexes oriented to repetitive texts. All logarithms in this thesis will be in base 2 and we will assume that $0 \log 0 = 0$.

2.1 Strings

Definition 2.1. A *string* T is a sequence of characters drawn from an alphabet Σ. The alphabet is an ordered and finite set of size $|\Sigma| = \sigma$. The i-th character of a string is represented as $T[i]$. The symbol ε represents the empty string of length 0.

Definition 2.2. Given a string T, and positions i and j, the *substring* of T starting at i and ending at j is defined as $T[i,j] = T[i]T[i+1]\ldots T[j]$. If $i > j$, then $T[i,j] = \varepsilon$.

Definition 2.3. Let T be a string of length n. The *prefixes* of T are the strings $T[1,j], \forall\, 0 \leq j \leq n$ and its *suffixes* are the strings $T[i,n], \forall\, 1 \leq i \leq n+1$.

Definition 2.4. Let T_1, T_2 be strings of length n_1 and n_2, respectively. We define the concatenation of these strings as $T_1T_2 = T_1[1]\ldots T_1[n_1]T_2[1]\ldots T_2[n_2]$.

Definition 2.5. Given a string T of length n, the reverse of T is $T^{rev} = T[n]T[n-1]\ldots T[2]T[1]$.

Definition 2.6. The *lexicographic order* ($<$) between strings is defined as follows: Let a, b be characters in Σ and X, Y be strings over Σ.

$$\varepsilon < X, \forall X \neq \varepsilon$$
$$aX < bY \;\; \textit{if}\; a < b \vee (a = b \wedge X < Y)$$

2.2 Search Problems

Definition 2.7. Given a string T and a pattern P (a string of length m) both over an alphabet Σ, the *occurrence positions* of P in T are defined as $O = \{1+|X|,\ \exists X, Y,\ T = XPY\}$.

Definition 2.8. Given a string T and a pattern P, the following search problems are of interest:

- $\mathit{exists}(P, T)$ returns true iff P is in T, i.e., returns true iff $O \neq \emptyset$.
- $\mathit{count}(P, T)$ counts the number of occurrences of P in T, i.e., returns $\mathit{occ} = |O|$.
- $\mathit{locate}(P, T)$ finds the occurrences of P in T, i.e., returns the set O in some order.
- $\mathit{extract}(T, l, r)$ extracts the text substring $T[l, r]$.

Remark 2.9. Note that *exists* and *count* can be answered after performing a *locate* query.

2.3 Entropy

Definition 2.10. Let T be a string of length n. The zero-th order empirical entropy is defined as

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

where n_c is the number of times the character c appears in T, that is, n_c/n is the empirical probability of appearance of character c.

It is worth noticing that the zero-th order entropy is invariant to permutations in the order of the text characters. The value $nH_0(T)$ is the least number of bits needed to represent T using a compressor that gives each character a fixed encoding.

Definition 2.11. Let T be a string of length n. The k-th order empirical entropy [Man01] is defined as

$$H_k(T) = \sum_{S \in \Sigma^k} \frac{\left|T^S\right|}{n} H_0\left(T^S\right)$$

where T^S is the sequence composed of all characters preceded by string S in T.

The value $nH_k(T)$ is the least number of bits needed to represent T using a compressor that encodes each character taking into account the k preceding characters in T. This value assumes the first k characters are encoded for free, thus it gives a relevant lower bound only when $n \gg k$.

H_k is a decreasing function in k, that is,

$$0 \leq H_k(T) \leq H_{k-1}(T) \leq \ldots \leq H_1(T) \leq H_0(T) \leq \log \sigma.$$

The following lemma yields the ground to show that the empirical entropy H_k is not a good lower-bound measure for the compressibility of repetitive texts.

Lemma 2.12. *Let T be a string of length n. For any $k \leq n$ it holds $H_k(TT) \geq H_k(T)$.*

Proof. As new relevant contexts may have arisen in the concatenation TT, we denote by $C(T,k)$ the contexts of length k present in T, and $C(TT,k)$ the contexts of TT. We have that $C(T,k) \subseteq C(TT,k)$. The number of new contexts in TT is at most k. For each $S \in C(T,k)$, we have $(TT)^S = T^S A^S T^S$, for some A^S such that $|A^S| \leq k$. Then,

$$\begin{aligned}
H_k(TT) &= \frac{1}{|TT|} \sum_{S \in C(TT,k)} |(TT)^S| H_0((TT)^S) \\
&\geq \frac{1}{2|T|} \sum_{S \in C(T,k)} |T^S A^S T^S| H_0(T^S A^S T^S) \\
&\geq \frac{1}{2|T|} \sum_{S \in C(T,k)} |T^S T^S| H_0(T^S T^S) \\
&= \frac{1}{|T|} \sum_{S \in C(T,k)} |T^S| H_0(T^S) \\
&= H_k(T).
\end{aligned}$$

In the first step we used $C(T,k) \subseteq C(TT,k)$; in the second we used $|T|H_0(T) \leq |TA|H_0(TA)$, for $T = T^S T^S$ and $A = A^S$ (since $|T^S A^S T^S| H_0(T^S A^S T^S) = |T^S T^S A^S| H_0(T^S T^S A^S)$); and in the third we used $H_0(TT) = H_0(T)$. The second property holds because

$$\begin{aligned}
|TA|H_0(TA) &= \sum_{c\in\Sigma}(n_c^T + n_c^A)\log\frac{n^T+n^A}{n_c^T+n_c^A} \\
&\geq n^T\sum_{c\in\Sigma}\frac{n_c^T}{n^T}\log\frac{n^T+n^A}{n_c^T+n_c^A} \\
&\geq n^T\sum_{c\in\Sigma}\frac{n_c^T}{n^T}\log\frac{n^T}{n_c^T} \\
&= |T|H_0(T)
\end{aligned}$$

where n_c^X is the number of occurrences of character c in string X, and $n^X = |X|$ for $X = T$ or A. The last line is justified by the Gibbs inequality [Ham86]. □

It follows that $|TT|H_k(TT) \geq 2|T|H_k(T)$, that is, to encode TT this model uses at least twice the space of the one used to encode T. An LZ77 encoding would need just one more phrase, as seen later.

2.4 Encodings

Most data structures need to represent symbols and numbers. Classic data structures use a fixed amount of space to store them, for example 1 byte for characters and 4 bytes for integers. Instead, compressed data structures aim to use the minimum possible space, thus they represent symbols using variable-length prefix-free codes or just using a fixed amount b of bits, where b is as small as possible. Table 2.1 shows different encodings for the integers 1,. . . ,9, which we describe next.

Unary Codes This representation is the simplest and serves as a basis for other coders. It represents a positive n as $\texttt{1}^{n-1}\texttt{0}$, thus it uses exactly n bits.

Gamma Codes It represents a positive n by concatenating the length of its binary representation in unary and the binary representation of the symbol, omitting the most significant bit. The space is $2\lfloor\log n\rfloor + 1$, $\lfloor\log n\rfloor + 1$ for the length and $\lfloor\log n\rfloor$ for the binary representation.

Delta Codes This is an extension of γ-codes that works better on larger numbers. It represents the length of the binary representation of n using γ-codes and then n in binary without its most significant bit, thus using $\{2\lfloor\log(\lfloor\log n\rfloor + 1)\rfloor + +1\} + \lfloor\log n\rfloor$ bits.

Symbol	Unary Code	γ-Code	δ-Code	Binary($b = 4$)	Vbyte($b = 2$)
1	0	0	0	0001	001
2	10	100	1000	0010	010
3	110	101	1001	0011	011
4	1110	11000	10100	0100	001100
5	11110	11001	10101	0101	001101
6	111110	11010	10110	0110	001110
7	1111110	11011	10111	0111	001111
8	11111110	1110000	11000000	1000	001100100
9	111111110	1110001	110000001	1001	001100101

Table 2.1: Example of different coders

Vbyte Coding [WZ99] It splits the $\lfloor \log(n+1) \rfloor$ bits needed to represent n into blocks of b bits and stores each block in a chunk of $b+1$ bits. The highest bit is 0 in the chunk holding the most significant bits of n, and 1 in the rest of the chunks. For clarity we write the chunks from most to least significant, just like the binary representation of n. For example, if $n = 25 = 11001$ and $b = 3$, then we need two chunks and the representation is $0011 \cdot 1001$. Compared to an optimal encoding of $\lfloor \log(n+1) \rfloor$ bits, this code loses one bit per b bits of n, plus possibly an almost empty final chunk. Even when the best choice for b is used, the total space achieved is still worse than δ-encoding's performance. In exchange, Vbyte codes are very fast to decode.

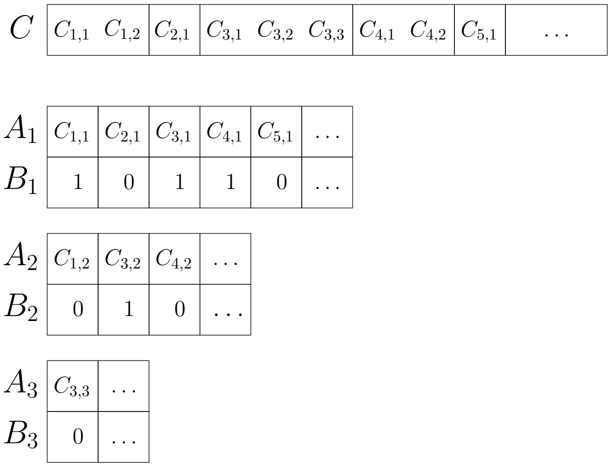
2.4.1 Directly Addressable Codes

In many cases we need to store a set of numbers using the least possible space, yet providing fast random access to each element. Variable-length codes complicate this task, as they require storing, in addition, pointers to sampled positions of the encoded sequence.

A simple solution that shows good performance in practice is the so-called *Directly Addressable Codes* (DAC) [BLN09], a variant of *Vbytes* [WZ99]. They start with a sequence $C = C_1, \ldots, C_n$ of n integers. Then they compute the Vbyte encoding of each number. The least significant blocks are stored contiguously in an array A_1, and the highest bits of the least significant chunks are stored in a bitmap B_1. The remaining chunks are organized in the same way in arrays A_i and bitmaps B_i, storing contiguously the i-th chunks of the numbers that have them. Note that arrays A_i

store contiguously the bits $(i-1)\cdot b+1,\ldots,i\cdot b$ and bitmaps B_i store whether a number has further chunks or not, hence the name *Reordered Vbytes*.

Figure 2.1 shows an example of the resulting structure. The first element is represented with two blocks, thus, $A_1[0]=C_{1,1}$, $A_2[0]=C_{1,2}$, $B_1[0]=\mathtt{1}$ and $B_2[0]=\mathtt{0}$.



C	$C_{1,1}$ $C_{1,2}$	$C_{2,1}$	$C_{3,1}$ $C_{3,2}$ $C_{3,3}$	$C_{4,1}$ $C_{4,2}$	$C_{5,1}$	$\ldots$

A_1	$C_{1,1}$	$C_{2,1}$	$C_{3,1}$	$C_{4,1}$	$C_{5,1}$	$\ldots$
B_1	1	0	1	1	0	$\ldots$

A_2	$C_{1,2}$	$C_{3,2}$	$C_{4,2}$	$\ldots$
B_2	0	1	0	$\ldots$

A_3	$C_{3,3}$	$\ldots$
B_3	0	$\ldots$

Figure 2.1: Example of Directly Addressable Codes structure

To access the element at position $i=i_1$ we check whether $B_1[i_1]$ is set. If it is not set, this is the last chunk and we already have the value $C[i]=A_1[i_1]$, otherwise we have to fetch the following chunks. In that case, we recompute the position as $i_2=rank_1(B_1,i_1)$, where $rank_1(B_1,i_1)$ is the number of ones up to position i_1 on bitmap B_1 (see Section 2.5 for further details). If $B_2[i_2]$ is not set we are done with $C[i]=A_1[i_1]+A_2[i_2]\cdot 2^b$, otherwise we set $i_3=rank_1(B_2,i_2)$ and continue in the following levels. Accessing a random element takes $O(\log(M)/b)$ worst case time, where $M=\max C_i$. However, the access time is lower for elements with shorter codewords, which are usually the most frequent ones.

We will use the implementation of Susana Ladra[1] (available by personal request) in this thesis.

2.5 Bitmaps

Let B a binary sequence over $\Sigma=\{\mathtt{0},\mathtt{1}\}$ (a bitmap) of length n and assume it has m ones. We are interested in solving the following operations:

[1] Universidade da Coruña, Spain. `sladra@udc.es`

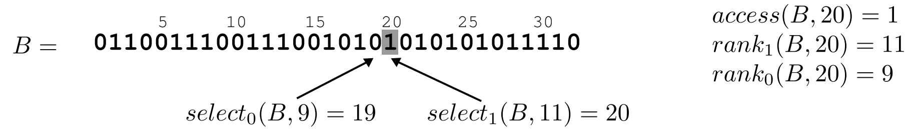


Figure 2.2: Example of rank and select

Variant	Size	Rank	Select
Clark	$n + o(n)$	$O(1)$	$O(1)$
RRR	$nH_0(B) + o(n)$	$O(1)$	$O(1)$
esp	$nH_0(B) + o(n)$	$O(1)$	$O(1)$
recrank	$1.44m \log \frac{n}{m} + m + o(n)$	$O\left(\log \frac{n}{m}\right)$	$O\left(\log \frac{n}{m}\right)$
vcode	$m \log(n/\log^2 n) + o(n)$	$O(\log^2 n)$	$O(\log n)$
sdarray	$m \log \frac{n}{m} + 2m + o(m)$	$O\left(\log \frac{n}{m} + \frac{\log^4 m}{\log n}\right)$	$O\left(\frac{\log^4 m}{\log n}\right)$

Table 2.2: Complexities for binary rank and select

- $rank_b(B, i)$**:** How many b's are up to position i (included).
- $select_b(B, i)$**:** The position of the i-th b bit.

Example 2.13. Figure 2.2 shows an example of the operations rank and select. We show the values of both $rank_1(B, 20) = 11$ and $rank_0(B, 20) = 9$. Note that these two values add up to 20, since the former returns the number of ones up to position 20, and the latter the number of zeroes. Also, *access* simply returns the bit stored at that position, in our case at position 20 there is a 1. Finally, we show the value of $select_1(B, 11) = 20$, which was expected since $access(B, 20) = 1$. The value of $select_0(B, 9)$ is 19.

Several solutions have been proposed to address this problem. The first solution able to solve both kinds of queries in constant time uses $n + O(\frac{n \log \log n}{\log n})$ bits of space [Cla96]. Raman, Raman and Rao's solution (RRR) [RRR02] achieves $nH_0(B) + O(\frac{n \log \log n}{\log n})$ bits and answer the queries in constant time. Okanohara and Sadakane [OS07] proposed several alternatives tailored to the case of small m (sparse bitmaps): `esp`, `recrank`, `vcode`, and `sdarray`. Table 2.2 shows the time and space complexities of these solutions. Note that the reported spaces include the representation of the bitmap.

2.5.1 Practical Dense Bitmaps

The extra $o(n)$ space of theoretical solutions [Cla96] is large in practice. González *et al.* [GGMN05] proposed a solution with good results in practice and small space overhead (up to 5%). This implementation is very simple, yet its practical performance is better than classical solutions. They store the plain bitmap in an array B and have a table R_s where they store $rank_1(B, i \cdot s)$, where $s = 32k$, where k is a parameter for the frequency of the sampling of the bit vector. They use a function called *popcount* that counts the number of `1` bits in a word (4 bytes). This operation can be solved bit by bit, but it is easy to improve it, using either bit parallelism or precomputed tables, requiring thus just a few operations. They solve the operations as follows ($rank_0$ and $select_0$ are obvious variations):

- $rank_1(B, i)$: They start in the last entry of R_s that precedes i ($R_s[\lfloor i/s \rfloor]$), and then sequentially scan the array B, popcounting consecutive words, until reaching the desired position. The popcounting of the last word is done by first setting all bits after position i to zero, which is done in constant time using a mask. Thus the time is $O(k)$.
- $select_1(B, i)$: They first binary search the R_s table for the last position p where $R_s[p] \leq i$. Then they scan B sequentially using *popcount* looking for the word where the desired select position is. Finally they find the desired position in the word by sequentially scanning the word bit by bit. Thus the time is $O(k + \log \frac{n}{k})$.

We will use the implementation of Rodrigo González (available at `http://code.google.com/p/libcds`) in this thesis.

2.5.2 Practical Sparse Bitmaps

When the bitmap is very sparse (i.e., the number of ones in the bitmap is very low) one practical solution is to δ-encode the distances between consecutive ones. Additionally we need to store absolute sample values $select_1(B, i \cdot s)$ for a sampling step s, plus pointers to the corresponding positions in the δ-encoded sequence. We solve the operations as follows:

- $select_1(B, i)$ is solved within $O(s)$ time by going to the last sampled position preceding i and decoding the δ-encoded sequence from there.

- $rank_1(B,i)$ is solved in time $O(s+\log\frac{m}{s})$. First, we binary search the samples looking for the last sampled position such that $select_1(B,\ell\cdot s)\leq i$. Starting from that position we sequentially decode the bitmap and stop as soon as $select_1(B,p)\geq i$.

- $access(B,i)$ is solved in time $O(s+\log\frac{m}{s})$ in a way similar to rank.

The space needed by the structure is $W+n/s(\lfloor\log m\rfloor+1+\lfloor\log W\rfloor+1)$, where W is the number of bits needed to represent all the δ-codes. In the worst case $W=2m\lfloor\log(\lfloor\log\frac{n}{m}\rfloor+1)\rfloor+m\lfloor\log\frac{n}{m}\rfloor+m=m\log\frac{n}{m}+O(m\log\log\frac{n}{m})$.

This structure allows a space-time trade-off related to s and also has the property that several operations cost $O(1)$ after solving others. For example, $select_1(B,p)$ and $select_1(B,p+1)$ cost $O(1)$ after solving $p\leftarrow rank_1(B,i)$.

2.6 Wavelet Trees

A wavelet tree [GGV03] is an elegant data structure that stores a sequence S of n symbols from an alphabet Σ of size σ. This structure supports some basic queries and is easily extensible to support others.

We split the alphabet into two halves L and R, so that the elements of L are lexicographically smaller than those of R. Then, we create a bitmap B of size n setting $B[i]=\texttt{0}$ if the symbol at position i belongs to L and $B[i]=\texttt{1}$ otherwise. This bitmap is stored at the root of the tree. Afterward, we extract from S all symbols belonging to L, generating sequence S_L, and all symbols belonging to R, generating sequence S_R (these sequences are not stored). Finally, we recursively generate the left subtree on S_L and the right subtree on S_R. We continue until we get a sequence over a one-letter alphabet. Figure 2.3 shows the wavelet tree for the example text `alabar_a_la_alabarda`. Only the bitmaps (black color) are stored in the tree. The labels of the tree show (gray color) the subsets L and R and the strings over the bitmaps (gray color) show the conceptual subsequences S_L and S_R.

The resulting tree has σ leaves, height $\lceil\log\sigma\rceil$, and n bits per level. Thus the space occupancy is $n\log\sigma$ bits, plus $o(n\log\sigma)$ (more precisely, $O(\frac{n\log\sigma\log\log n}{\log n})$) additional bits to support *rank* and *select* queries on the bitmaps.

In the following we explain how this structure supports the operations *access*, *rank* and *select* on S. The last two operations are just a generalization for larger alphabets of those defined in Section 2.5.

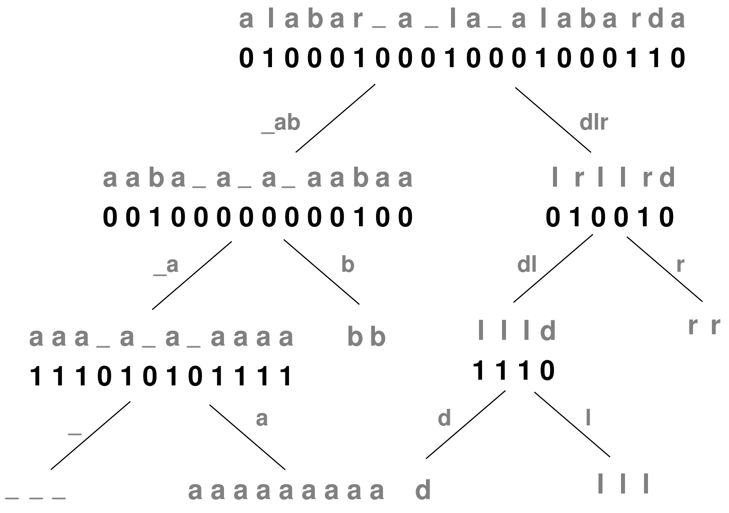


Figure 2.3: Example of a wavelet tree for the text `alabar_a_la_alabarda`

- **Access:** To retrieve the symbol $S[i]$ we look at $B[i]$ at the root. If it is a `0` we go to left subtree, otherwise to the right subtree. The new position is $i \leftarrow rank_0(B, i)$ if we go to the left and $i \leftarrow rank_1(B, i)$ if we go to the right. This procedure continues recursively until we reach a leaf. The bits read in the path from the root to the leaf represent the symbol sought.

- **Rank:** To count how many c's are up to position i we go to the left if c is in L and otherwise to the right. The new position is $i \leftarrow rank_0(B, i)$ if we go to the left and $i \leftarrow rank_1(B, i)$ if we go to the right, where B is the bitmap of the root. When we reach a leaf the answer is i.

- **Select:** To find the i-th symbol c we first go to the leaf corresponding to c and then go upwards to the root. Let B the bitmap of the parent. If the current node is a left child then the position at the parent is $i \leftarrow select_0(B, i)$, otherwise it is $i \leftarrow select_1(B, i)$. When we reach the root the answer is the current i value.

The running time of these operations is $O(\log \sigma)$, since we use a bitmap supporting constant-time rank, select and access.

Example 2.14. Figure 2.4 shows an example of how we retrieve the 11th symbol of sequence S ($access(S, 11) = \texttt{a}$). First we access the bitmap of the root and see that at position 11 there is a 0. Hence we descend to the left. Then using $rank_0(B, 11) = 8$ we count how many zeroes are up to position 11. This value is our new position in

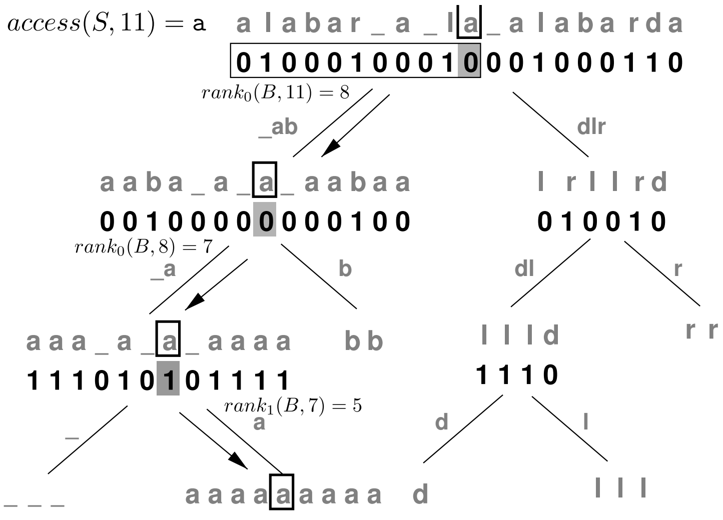


Figure 2.4: Example of access in a wavelet tree

the next level. Then we continue the process until we reach a leaf; in that case the symbol stored in that lead is the symbol sought, in our case an ‘a’.

Example 2.15. Figure 2.5 shows step by step how we compute $rank_{\mathtt{l}}(S, 11) = 2$. Since symbol ‘l’ is mapped to a 1 we descend from the root to the right child. Using $rank_1(B, 11) = 3$ we count the number of ones up to that position. This is our new position in the next level. Then we continue the process until we reach a leaf. The value sought is the last value of rank, in our case 2.

Example 2.16. Figure 2.6 shows an example of how to select the second ‘b’ in the sequence S ($select_{\mathtt{b}}(S, 2)$). First we descend to the leaf representing symbol ‘b’. Since that symbol was last mapped to a 1, we go to the parent and compute our new position as $select_1(B, 2) = 12$. In that level, ‘b’ was mapped to a 0, so we go to the parent and the new position is $select_0(B, 12) = 16$, and that is the value sought.

2.6.1 Range Search

A direct application of wavelet trees is to answer range search queries [MN07]. This method is very similar to the idea of Chazelle [Cha88].

Definition 2.17. Given a subset R ($|R| = t$) of the discrete range $[1, n] \times [1, \sigma]$, a *range query* returns the points $p \in R$ belonging to a range $[x_1, x_2] \times [y_1, y_2]$.

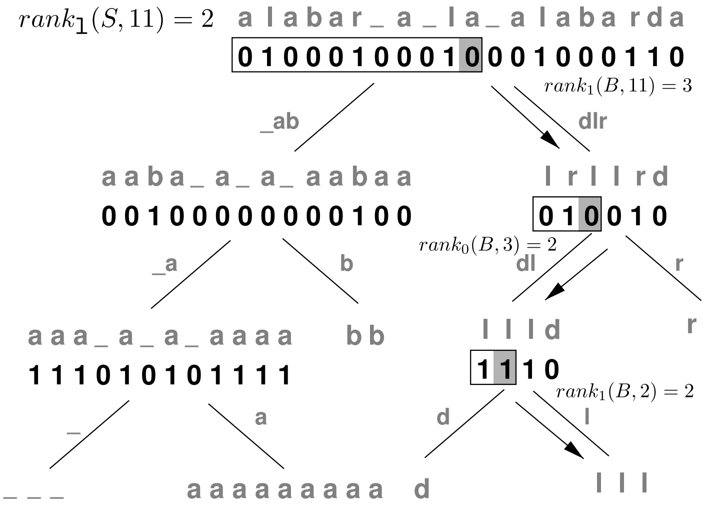


Figure 2.5: Example of rank in a wavelet tree

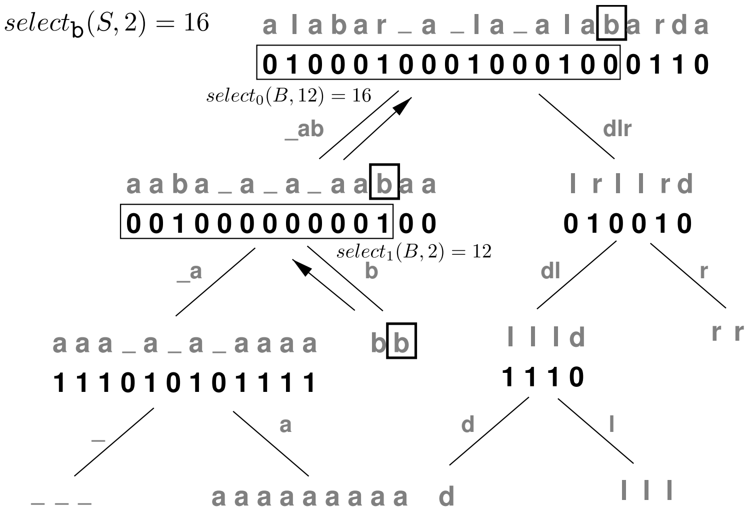


Figure 2.6: Example of select in a wavelet tree

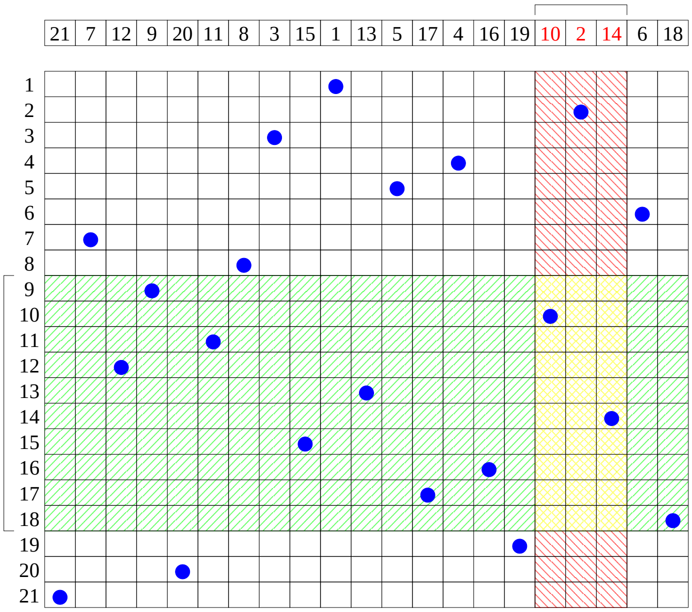


Figure 2.7: Example of 2-dimensional range query

An extension of the wavelet tree supports range queries using $n + t \log n + o(n + t \log n)$ bits, counting the number of points within the range in time $O(\log n)$ and reporting each occurrence in time $O(\log n)$. We will use a modified version of the implementation of Gonzalo Navarro[2].

We explain here a simplified version for the case in which there exists exactly one point for each value of x. We order the points of R by their x coordinate and create the sequence $S[1, n]$, such that for each $(x, y) \in R$, $S[x] = y$. Then we build the wavelet tree of S.

Example 2.18. Figure 2.7 shows a grid, with exactly one y value for each value of x. The figure shows in yellow the range $[17, 19] \times [9, 18]$, containing two occurrences; in red and yellow, the range $[17, 19] \times [1, 21]$, containing 3 occurrences; and in green and yellow, the range $[1, 21] \times [9, 18]$, containing 10 occurrences.

[2] Check the LZ77-index source code (`http://pizzachili.dcc.uchile.cl/indexes/LZ77-index`) for the updated version

Projecting

A range in S represents a range along the x coordinate and the splits made by the wavelet tree define ranges along the y coordinate. Every time we descend to a child of a node we need to know where the range represented in that child is. The operation of determining the range defined by a child, given the range of the parent, is called *projecting.* Using rank we project a range downwards. Given a node with bitmap B the left projection of $[x, x']$ is $[1 + rank_0(B, x - 1), rank_0(B, x')]$ and the right projection is $[1+rank_1(B, x-1), rank_1(B, x')]$. A range $[y, y']$ along the y coordinate is projected to the left as $[y, \lfloor(y + y')/2\rfloor]$ and to the right as $[\lfloor(y + y')/2\rfloor + 1, y']$.

Counting

We start from the root with the one-dimensional ranges $[x, x'] = [x_1, x_2]$ and $[y, y'] = [1, n]$ and project them in both subtrees. We do this recursively until:

1. $[x, x'] = \emptyset$;
2. $[y, y'] \cap [y_1, y_2] = \emptyset$; or
3. $[y, y'] \subseteq [y_1, y_2]$, in which case we add $x' - x + 1$ to the total.

As the interval $[y_1, y_2]$ is covered by $O(\log n)$ maximal wavelet tree nodes, the total time to count the occurrences is $O(\log n)$.

Example 2.19. Figure 2.8 shows the wavelet tree that represents the range of Figure 2.7. The figure represents how to count the occurrences in the range $[17, 19] \times [9, 18]$. The figure shows in red how the range $[17, 19]$ in the x coordinate is projected downwards. The nodes below the blue line are those whose y range is contained in the range $[9, 18]$. Additionally, the nodes with a circle next to them are those in which the counting process ends. The blue circle represents rule number 1 (see above), the green one represents rule number 2 and finally rule number 3 is represented by the red circle. In our case, at each node marked with red, we report one occurrence, yielding a total of 2 occurrences.

Locating

To locate the actual points we start from each node in which we were counting. If we want to know the x coordinate we go up using select and if we want to know the

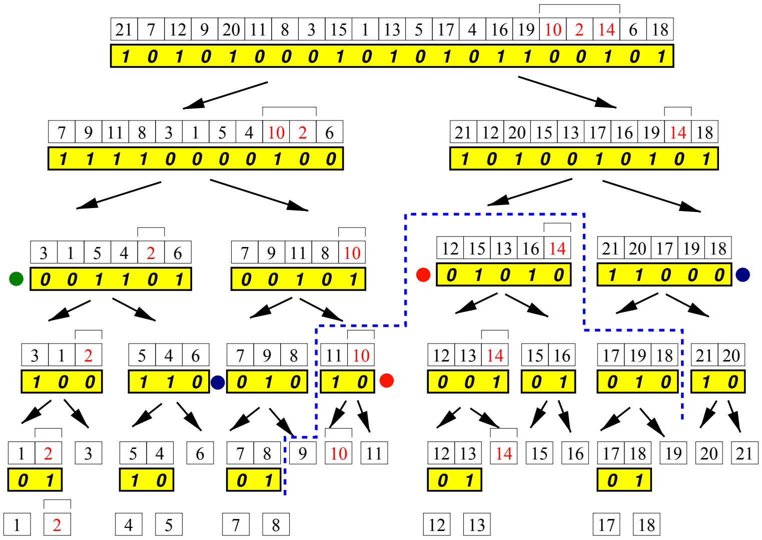


Figure 2.8: Example of counting the occurrences in a 2-dimensional range query using a wavelet tree

y coordinate we go down using rank. This operation takes $O(\log n)$ for each point located.

2.7 Permutations

A permutation is a bijection $\pi : [1, n] \rightarrow [1, n]$, and we are interested in computing efficiently both $\pi(i)$ and $\pi^{-1}(i)$ for any $1 \leq i \leq n$. The permutation can be represented in a plain array using $n \log n$ bits, by storing $P = [\pi(1), \ldots, \pi(n)]$. This answers $\pi(i)$ in constant time. Solving $\pi^{-1}(i)$ can be done by sequentially scanning P for the position j where $\pi(j) = i$. A more efficient solution [MRRR03] is based on the cycles of a permutation. A cycle is a sequence $i, \pi(i), \pi^2(i), \ldots, \pi^k(i)$ such that $\pi^{k+1}(i) = i$. Every i belongs to exactly one cycle. Then, to compute π^{-1} we repeatedly apply π over i, finding the element e of the cycle such that $\pi(e) = i$. These solutions do not require any extra space to compute $\pi^{-1}(i)$, but they take $O(n)$ time in the worst case.

Representing the sequence $\pi[1, n]$ with a wavelet tree one can answer both queries using $O(\log n)$ time and $n \log n + o(n \log n)$ bits of space. A faster solution [MRRR03] is based on the cycles of the permutation. By introducing shortcuts in the cycles, it uses $(1 + \varepsilon)n \log n + O(n)$ bits and solves $\pi(i)$ in constant time and $\pi^{-1}(i)$ in $O(1/\varepsilon)$ time, for any $\varepsilon > 0$.

We will use the implementation of Munro *et al.*'s shortcut technique by Diego Arroyuelo[3], available at `http://code.google.com/p/libcds`.

2.8 Tree Representations

A classical representation of a general tree of n nodes requires $O(nw)$ bits of space, where $w \geq \log n$ is the bit length of a machine pointer. Typically only operations such as moving to the first child and to the next sibling, or to the i-th child, are supported in constant time. By further increasing the constant, some other simple operations are easily supported, such as moving to the parent, knowing the subtree size, or the depth of the node. However, the $\Omega(n \log n)$ bit space complexity is excessive in terms of information theory. The number of different general trees of n nodes is $C_n \approx 4^n/n^{3/2}$, hence $\log C_n = 2n - \Theta(\log n)$ bits are sufficient to distinguish any one of them.

There are several succinct tree representations that use $2n+o(n)$ bits of space and answer most queries in constant time (see the review by Arroyuelo *et al.* [ACNS10] for a detailed exposition); here we explain the DFUDS [BDM$^+$05] representation as this is the one that meets our requirements.

Definition 2.20. A sequence S drawn from alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ is said to be *balanced* if: (1) there are as many `0`s as `1`s and (2) at any position i the number of zeroes to the left is greater or equal than the number of ones (i.e., $rank_{\mathtt{0}}(S, i) \geq rank_{\mathtt{1}}(S, i)$). Usually a balanced sequence is referred as *balanced parentheses* by identifying `0` as '(' and `1` as ')', as the nesting of parentheses satisfies the above definition.

The operations defined over a balanced sequence are: (1) *findclose(S,i) (findopen(S,i))* finds the matching `1` (`0`) of the `0` (`1`) at position i, and (2) *enclose(S,i)* is the position of tightest `0` enclosing node i.

Definition 2.21 ([BDM$^+$05])**.** The Depth-first unary degree sequence (DFUDS) is generated by a depth-first traversal of the tree, at each node appending the degree of the node in unary. Additionally a leading `1` is prepended to the sequence to make it balanced and allow the concatenation of several such encodings into a forest.

The DFUDS sequence represents the topology of the tree using $2n$ bits. Tree nodes are identified in the DFUDS sequence according to their rank in the order given by the depth-first traversal (more precisely, the i-th node is identified by position $select_{\mathtt{1}}(i)$

[3] Yahoo! Research, Chile. `darroyue@dcc.uchile.cl`

in the DFUDS encoding). Figure 2.9 shows the DFUDS bit-sequence for the example tree. The red 1 in the sequence is the preceding 1 added to make the sequence balanced. The green node is represented by the 10th 1 in the sequence, as it is the 10th node visited during a depth-first traversal. The blue sequence of five 1s and one 0 is the degree of the blue node.

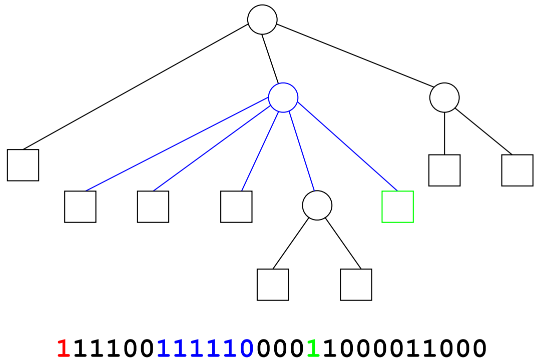


Figure 2.9: Example of DFUDS representation

To solve the common operations over trees two data structures are built over the DFUDS sequence: a bitmap data structure supporting rank and select (Section 2.5) and a data structure solving operations *findclose*, *findopen* and *enclose* [Jac89, MR01, Nav09]. These structures allow one to compute the most common operations in constant time using $o(n)$ additional bits of space. Additionally, if we use labeled trees we need to store the labels of the edges in an array $chars$, using $n \log \sigma$ additional bits, where σ is the labels' alphabet size. The label of the edge pointing to the i-th child of node x is at $chars[rank_1(dfuds, x) + i]$. The operations we are interested in for this thesis are:

- *degree(x)*: number of children of node x.
- *isLeaf(x)*: whether node x is a leaf.
- *child(x,i)*: i-th child of node x.
- *labeledChild(x,c)*: child of node x labeled by symbol c.
- *leftmostLeaf(x)*: leftmost leaf of the subtree starting at node x.
- *rightmostLeaf(x)*: rightmost leaf of the subtree starting at node x.
- *leafRank(x)*: number of leaves to the left of node x.

- *preorder(x)*: preorder position of node x.

All these operations can be solved theoretically in constant time; however, in practice *labeledChild* is solved by binary searching the labels of the children, because it is much easier to implement and fast enough in practice. To solve *leftmostLeaf*, *rightmostLeaf* and *leafRank* we need to solve the queries $rank_{\texttt{00}}(i)$ and $select_{\texttt{00}}(i)$. $Rank_{\texttt{00}}(i)$ returns the number of occurrences of the substring `00` in the bitmap up to position i and $select_{\texttt{00}}(i)$ returns the position p of the i-th occurrence of the substring `00` in the bitmap. Solving these queries requires an additional data structure that uses $o(n)$ bits. It uses the same ideas as the one for solving rank and select for binary alphabets.

We will use a modified version of the implementation of Diego Arroyuelo available at `http://code.google.com/p/libcds`, adding support for leaf-related operations.

2.9 Tries

A *trie* or digital tree is a data structure that stores a set of strings. It can find the elements of the set prefixed by a pattern in time proportional to the pattern length.

Definition 2.22. A *trie* for a set S of distinct strings is a tree where each node represents a distinct prefix in the set. The root node represents the empty prefix ε. A node v representing prefix Y is a child of node u representing prefix X iff $Y = Xc$ for some character c, which labels the edge between u and v.

We suppose that all strings are ended by a special symbol `$`, not present in the alphabet. We do this in order to ensure that no string S_i is a prefix of some other string S_j. This property guarantees that the tree has exactly $|S|$ leaves. Figure 2.10 shows an example of a trie.

A trie for the set $S = \{S_1, \ldots, S_n\}$ is easily built on $O(|S_1| + \ldots + |S_n|)$ time by successive insertions (assuming we can descend to any child in constant time). A pattern P is searched for in the trie starting from the root and following the edges labeled with the characters of P. This takes a total time of $O(|P|)$.

A *compact trie* is an alternative representation that reduces the space of the trie by collapsing unary nodes into a single node and labeling the edge with the concatenation of all labels. A *PATRICIA tree* [Mor68], an alternative that uses even less space, just stores the first character of the label string and its length. This variant is used when the strings S are available separately, as not all information is stored in the edges. In this variant, after the search we need to check if the prefix found

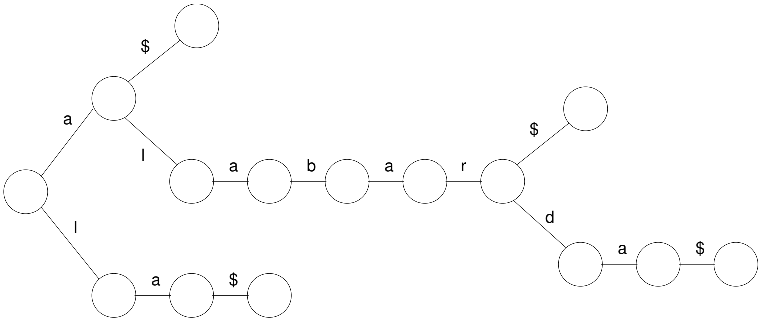


Figure 2.10: Example of a trie for the set $S = \{$‘alabar’, ‘a’, ‘la’, ‘alabarda’$\}$.

actually matches the pattern. For doing so, we have to extract the text corresponding to any string with the prefix found and compare it with the pattern. It they are equal then all leaves will be occurrences (i.e., strings prefixed with the pattern), otherwise none will be an occurrence. Figure 2.11 shows an example of this kind of trie.

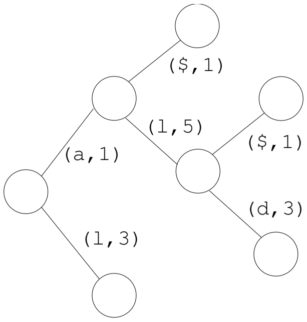


Figure 2.11: Example of a PATRICIA trie for the set $S = \{$‘alabar’, ‘a’, ‘la’, ‘alabarda’$\}$. The values in parentheses are respectively the first character of the label and the length of the label.

Definition 2.23. A *suffix trie* is a trie composed of all the suffixes $T[i, n]$ of a given text $T[1, n]$. The leaves of the trie store the positions where the suffixes start.

2.10 Suffix Trees

Definition 2.24 ([Wei73, McC76])**.** A *suffix tree* is a PATRICIA tree built over all the suffixes $T[i, n]$ of a given text $T[1, n]$. The leaves in the tree indicate the text positions where the corresponding suffixes start.

Figure 2.12 shows the suffix tree for the text ‘alabar_a_la_alabarda$’.

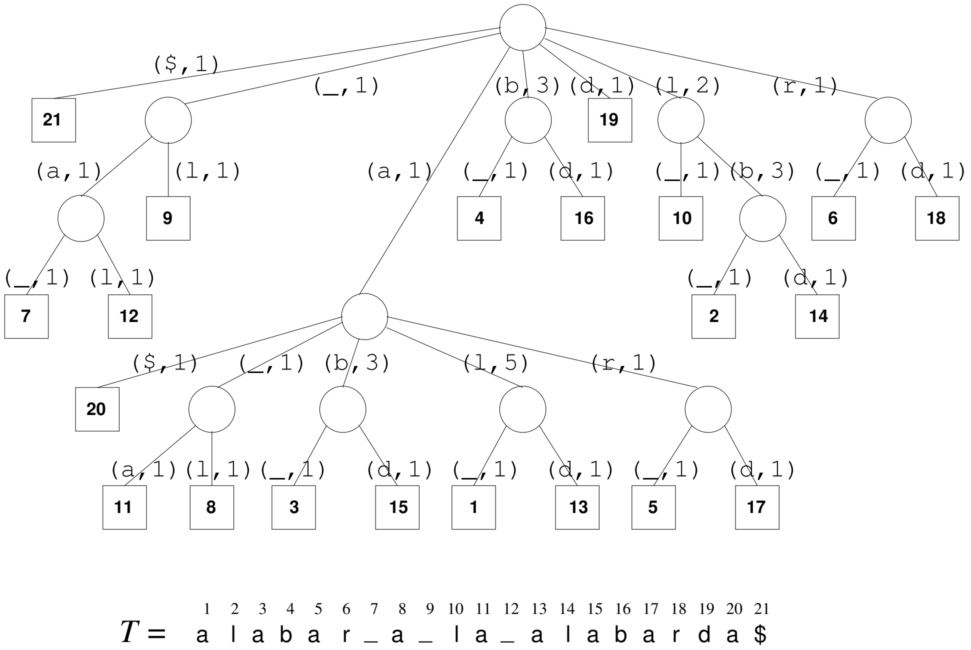


Figure 2.12: The suffix tree for the text ‘alabar_a_la_alabarda$’

The suffix tree can be built in $O(n)$ time using $O(n \log n)$ bits of space [McC76, Ukk95].

A suffix tree is able to find all the *occ* occurrences of a pattern P of length m in time $O(m + occ)$, i.e., to solve the *locate* query described in Section 2.2. After descending by the tree according to the characters of the pattern, we could be in three different cases: i) we reach a point in which there is no edge labeled with the current character of P, which means that the pattern does not occur in T; ii) we finish reading P in an internal node (or in the middle of an edge), in which case the suffixes of the corresponding subtree are either all occurrences or none, therefore we only need to check if one of those suffixes matches the pattern P; iii) we end up in a leaf without consuming all the pattern, in which case at most one occurrence is found after checking the suffix with the pattern. As a subtree with *occ* leaves has $O(occ)$ nodes, the total time for reporting the occurrences is as stated above.

The suffix tree can solve the queries *count* and *exists* in $O(m)$ time. The process is similar to that of *locate*. First we descend the tree according to the pattern. Then, we check if one of the suffixes of the subtree is a match. If it is a match the answer of *count* is the number of leaves of the subtree (for which we need to store in each internal node the number of leaves that descend from it), otherwise it is zero.

2.11 Suffix Arrays

Definition 2.25 ([MM93, GBYS92])**.** A *suffix array* $A[1,n]$ is a permutation of the integer interval $[1,n]$, holding $T[A[i],n] < T[A[i+1],n]$ for all $1 \leq i < n$. In other words, it is a permutation of the suffixes of the text such that the suffixes are lexicographically sorted.

Figure 2.13 shows the suffix array for the text `‘alabar_a_la_alabarda$’`. The character `$` is the smallest one in lexicographical order. The zone highlighted in gray represents those suffixes starting with `‘a’`.

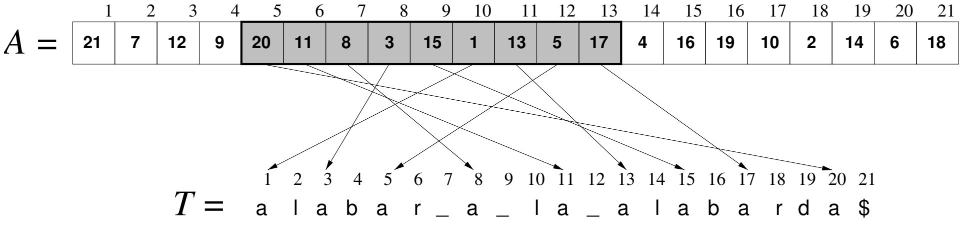


Figure 2.13: The suffix array for the text `‘alabar_a_la_alabarda$’`

Note that the suffix array could be computed by collecting the values at the leaves of the suffix tree. However, several methods exist that compute the suffix array in $O(n)$ or $O(n \log n)$ time, using significantly less space. For a complete survey see [PST07].

The suffix array can solve *locate* queries in $O(m \log n + occ)$ time, and *count* and *exists* queries in $O(m \log n)$ time. First, we search for the interval $A[sp_1, ep_1]$ of the suffixes starting with $P[1]$. This can be done via two binary searches on A. The first binary search determines the starting position sp for the suffixes lexicographically larger than or equal to $P[1]$, and the second determines the ending position ep for suffixes that start with $P[1]$. Then, we consider $P[2]$, narrowing the interval to $A[sp_2, ep_2]$, holding all suffixes starting with $P[1,2]$. This process continues until P

is fully consumed or the current interval becomes empty. Note that this algorithm searches for the pattern from left to right. For each character of the pattern, we do two binary searches taking at most time $O(\log n)$, hence the total time is $O(m \log n)$. Then *locate* reports all occurrences in $O(occ)$ time and the answer to *count* is $ep_m - sp_m + 1$. We can also directly search for the interval $A[sp, ep]$ where the suffixes start with the pattern P using just two binary searches on A, which find the first and last position where the suffixes start with P. Each comparison between the pattern and a suffix will take at most $O(m)$ time, hence the total running time is also $O(m \log n)$. Yet, this is faster in practice than the previous method and is what we use in this thesis.

2.12 Backward Search

Backward search is an alternative method for finding the interval $[sp, ep]$ corresponding to a pattern P in the suffix array. It searches for the pattern from right to left, and is based on the Burrows-Wheeler transform.

Definition 2.26 ([BW94])**.** Given a text T terminated with the special character $T[n] = \$$ smaller than all others, and its suffix array $A[1, n]$, the *Burrows-Wheeler transform (BWT)* of T is defined as $T^{bwt}[i] = T[A[i] - 1]$, except when $A[i] = 1$, where $T^{bwt}[i] = T[n]$. In other words, the transformation is conceptually built first by generating all the cyclic shifts of the text, then sorting them lexicographically, and finally taking the last character of each shift. In practice it can be built in linear time by building the suffix array first.

We can think of the sorted list of cyclic shift as a conceptual matrix $M[1, n][1, n]$. Figure 2.14 shows an example of how the BWT is computed for the text `‘alabar_a_la_alabarda$’`. This transformation has the advantage of being easily compressed by local compressors [Man01]. It can be reversed as follows.

Definition 2.27. The LF-mapping $LF(i)$ maps a position i in the last column of M ($L = T^{bwt}$) to its occurrence in the first column of M (F).

Lemma 2.28 ([FM05])**.** *It holds*

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

where $c = T^{bwt}[i]$ and $C[c]$ is the number of symbols smaller than c in T.

Lemma 2.29 ([BW94])**.** *The LF-mapping allows one to reverse the Burrows-Wheeler transform.*

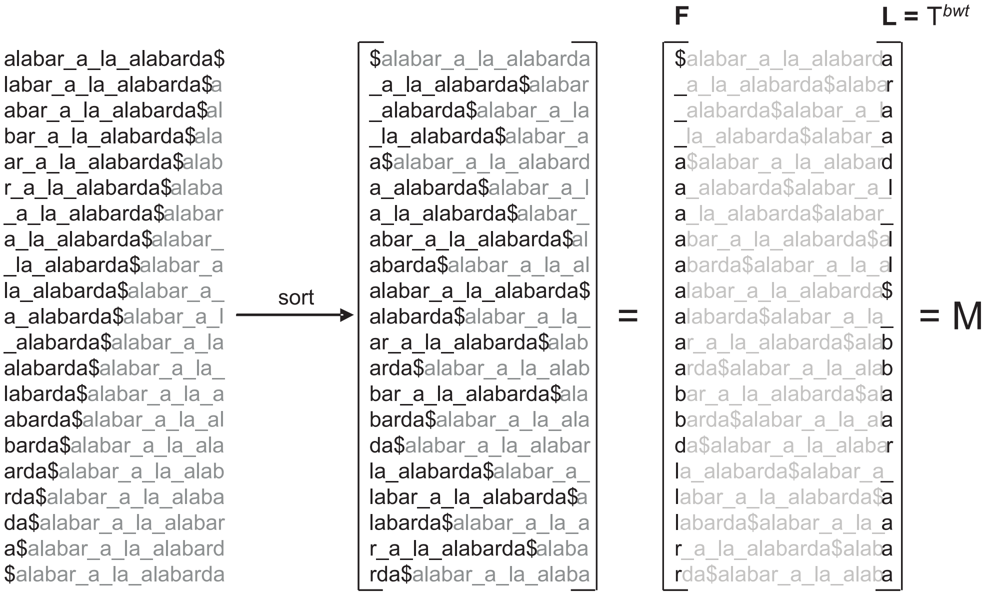


Figure 2.14: The BWT of the text `alabar_a_la_alabarda$'

Proof. We know that $T[n] = \$$ and since $ is the smallest symbol, $T[n] = F[1] = \$$ and thus $T[n-1] = L[1] = T^{bwt}[1]$. Using the LF-mapping we compute $i = LF(1)$; knowing that $T[n-1]$ is at $F[i]$, we have $T[n-2] = L[i]$, as $L[i]$ always precedes $F[i]$ in T. In general, it holds $T[n-k] = T^{bwt}[LF^{k-1}(1)]$. □

Given the close relation between the suffix array and the BWT, it is natural to expect that a search algorithm can work on top of the BWT. Such algorithm is called *backward search* (BWS), and at each stage it narrows the interval $[sp_i, ep_i]$ of the suffix array in which the suffixes start with $P[i,m]$, starting from $i = m$ and ending with $i = 1$. Narrowing the interval $A[sp, ep]$ with a new character c is called a $BWS(sp, ep, c)$ step and it is done very similarly to the LF-mapping (Lemma 2.28). BWS searches a pattern from right to left, opposite to the search on suffix arrays, that searches for a pattern from left to right.

Figure 2.15 shows the backward search algorithm. Lines 5-7 correspond to the BWS step.

```
**BWS**($P$)
$i \leftarrow len(P)$
$sp \leftarrow 1$
$ep \leftarrow n$
**while** $sp \leq ep$ **and** $i \geq 1$ **do**
  $c \leftarrow P[i]$
  $sp \leftarrow C[c] + rank_c(T^{bwt}, sp-1) + 1$
  $ep \leftarrow C[c] + rank_c(T^{bwt}, ep)$
  $i \leftarrow i - 1$
**if** $sp > ep$ **then**
  **return** $\emptyset$
**return** $(sp, ep)$
```

Figure 2.15: Backward Search algorithm (BWS)

2.13 Lempel-Ziv Parsings and Repetitions

Lempel and Ziv proposed in the seventies a new compression method [LZ76, ZL77, ZL78]. The basic idea is to replace a repeated portion of the text with a pointer to some previous occurrence of that portion. To find the repetitions they keep a dictionary representing all the portions that can be copied. Many variants of these algorithms exist [SS82, Wel84, Wil91] which differ in the way they parse the text or the encoding they use.

The LZ77 [ZL77] parsing is a dictionary-based compression scheme in which the dictionary used is the set of substrings of the preceding text. This definition allows it to get one of the best compression ratios for repetitive texts.

Definition 2.30 ([ZL77])**.** The *LZ77 parsing* of text $T[1,n]$ is a sequence $Z[1,n']$ of *phrases* such that $T = Z[1]Z[2]\ldots Z[n']$, built as follows. Assume we have already processed $T[1,i-1]$ producing the sequence $Z[1,p-1]$. Then, we find the longest prefix $T[i,i'-1]$ of $T[i,n]$ which occurs in $T[1,i-1]$,[4] set $Z[p] = T[i,i']$ and continue with $i = i'+1$. The occurrence in $T[1,i-1]$ of the prefix $T[i,i'-1]$ is called the *source* of the phrase $Z[p]$.

Note that each phrase is composed of the content of a source, which can be the empty string ε, plus a trailing character. Note also that all phrases of the parsing

[4] The original definition allows the source of $T[i,i'-1]$ to extend beyond position $i-1$, but we ignore this feature in this thesis.

are different, except possibly the last one. To avoid that case, a special character `$` is appended at the end, $T[n] = \$$.

Typically a phrase is represented as a triple $Z[p] = (start, len, c)$, where $start$ is the start position of the source, len is the length of the source and c is the trailing character.

Example 2.31. Let $T =$ `'alabar_a_la_alabarda$'`; the LZ77 parsing is as follows:

a	l	ab	ar	_	a_	la_	alabard	a$

In this example the seventh phrase copies two characters starting at position 2 and has a trailing character '_'.

One of the greatest advantages of this algorithm is the simple and fast scheme of decompression, opposed to the construction algorithm which is more complicated. Decompression runs in linear time by copying the source content referenced by each phrase and then the trailing character. However, random text extraction is not as easy.

The LZ78 [ZL78] compression scheme is also dictionary-based. Its dictionary is the set of all phrases previously produced. Because of this definition of the dictionary the construction process is much simpler than that of LZ77.

Definition 2.32 ([ZL78])**.** The *LZ78 parsing* of text $T[1, n]$ is a sequence $Z[1, n']$ of *phrases* such that $T = Z[1]Z[2] \dots Z[n']$, built as follows. Assume we have already processed $T[1, i-1]$ producing the sequence $Z[1, p-1]$. Then, we find the longest phrase $Z[j]$, for $j \le p-1$, that is a prefix of $T[i, n]$, set $Z[p] = Z[j]T[i + |Z[j]|]$ and continue with $i = i + |Z[j]| + 1$.

Typically a phrase is represented as $Z[p] = (j, c)$, where j is the phrase number of the source and c is the trailing character.

Example 2.33. Let $T =$ `'alabar_a_la_alabarda$'`; the LZ78 parsing is as follows:

a	l	ab	ar	_	a_	la	_a	lab	ard	a$

In this example the ninth phrase copies two characters starting at position 2 and has a trailing character '`b`'.

With respect to compression, both LZ77 and LZ78 converge to the entropy of stationary ergodic sources [LZ76, ZL78]. It also converges below the empirical entropy (Section 2.3), as detailed next.

Definition 2.34 ([KM99])**.** A parsing algorithm is said to be *coarsely optimal* if its compression ratio $\rho(T)$ differs from the k-th order empirical entropy $H_k(T)$ by a quantity depending only on the length of the text and that goes to zero as the length increases. That is, $\forall k\, \exists f_k, \lim_{n\to\infty} f_k(n) = 0$, such that for every text T, $\rho(T) \leq H_k(T) + f_k(|T|)$.

Theorem 2.35 ([KM99, PWZ92])**.** *The LZ77 and LZ78 parsings are coarsely optimal.*

As explained in Section 2.3, however, converging to $H_k(T)$ is not sufficiently good for repetitive texts. Repetitive texts are originated in applications where many similar versions of one base text are generated (i.e., DNA sequences); or where successive versions, each one similar to the preceding one (i.e., wiki), are generated. Statistical compressors are not able to capture this characteristic, because they predict a symbol based only on a short previous context, and such statistics do not change when the text is replicated many times (see Section 2.3 for the relation between $H_k(T)$ and $H_k(TT)$). Compressors based on repetitions, such as Lempel-Ziv parsings or grammar based ones, do exploit this repetitiveness.

2.14 Self-Indexes

Definition 2.36. A *self-index* [NM07] is an index that uses space proportional to that of the compressed text and solves the queries *locate* and *extract*. As this kind of indexes can reproduce any text substring, they replace the original text. Additionally, some indexes provide more efficient ways of computing *exists* and *count* queries.

There are several general-purpose self-indexes, however most of them do not achieve high compression for repetitive texts, as they are only able to compress up to the k-th order empirical entropy (Section 2.3). Most are based on the BWT or suffix array (see [NM07] for a complete survey). In the last years some self-indexes oriented to repetitive texts have been proposed. We cover these now.

2.14.1 Run-Length Compressed Suffix Arrays (RLCSAs)

The Run-Length Compressed Suffix Array (RLCSA) [SVMN08] is based on the Compressed Suffix Array of Sadakane [Sad03]. This is built around the so called Ψ function.

Definition 2.37 ([GV05])**.** Let $A[1,\ldots,n]$ be the suffix array of a text T. Then $\Psi(i)$ is defined as

$$\Psi(i) = A^{-1}[(A[i] \bmod n) + 1]$$

The Ψ function is the inverse of the LF mapping. Ψ maps suffix $T[A[i], n]$ to suffix $T[A[i]+1, n]$, allowing one to scan the text from left to right. A *run* in the Ψ array is an interval $[a, b]$ for which it holds $\forall i \in [a, b-1]$, $\Psi(i+1) = \Psi(i) + 1$.

In the RLCSA, one run-length encodes the differences $\Psi[i] - \Psi[i-1]$ and store absolute samples of the array Ψ. This structure is very fast for *count* and *exists* queries. Its major drawback is the sampling it requires for *locate* and *extract* queries, as it takes $(n \log n)/s$ extra bits to achieve locating time $O(s)$, and time $O(s + r - l)$ for $extract(l, r)$, where s is the sampling step.

The number of runs may be much smaller than $nH_k(T)$ (for example $runs(T) = runs(TT)$, whereas $|TT|H_k(TT) \geq 2|T|H_k(T)$ as shown in Section 2.3). However, the difference between the number of runs and the number of phrases in an LZ77 parsing [ZL77] may be a multiplicative factor as high as $\Theta(\sqrt{n})$.[5] For these reasons, the RLCSA seems to be an intermediate solution between LZ77 and empirical-entropy-based indexes.

2.14.2 Indexes based on sparse suffix arrays

In this section we present two indexes [KU96b, KU96a] by Kärkkäinen and Ukkonen. Although these are not self-indexes, they set the ground for several self-indexes proposed later.

- First, they choose some indexing positions of the text. These can be evenly spaced points [KU96b] or the points defined by a Lempel-Ziv parsing [KU96a].
- The suffixes starting at those points are indexed in a suffix trie, and the reversed prefixes in another trie.
- The index in principle only allows one to find occurrences crossing an indexing point.
- To find a pattern P of length m, they partition it in all $m + 1$ combinations of prefix and suffix.

[5] Veli Mäkinen, personal communication

- For each partition, they search for the suffix in the suffix trie and for the prefix of the pattern in the reverse prefix trie.
- The previous searches define a 2-dimensional range in a grid that relates each indexed text prefix (in lexicographic order) with the text suffix that follows (in lexicographic order). That is, related prefixes and suffixes are consecutive in the text.
- A data structure supporting 2-dimensional range queries [Cha88], finds all pairs of related suffixes and prefixes, finding in this way the actual occurrences.
- Additionally, using a Lempel-Ziv parsing they are able to find all the occurrences of the pattern. The occurrences are either found in the grid by the process described above (primary occurrences), or by considering the copies detected by the parsing (secondary occurrences), for which an additional method tracking the copies finds the remaining occurrences.

All following indexes can be thought as heirs of this general idea, which was improved by replacing or adding additional compact data structures to decrease the space usage. In most cases, the parsing was restricted only to LZ78 (Section 2.14.3), since it simplifies the index, and in others to text grammars (SLPs, Section 2.14.4). In the following two subsections we list the results obtained in those cases. This thesis can also be thought as a heir of this fundamental scheme: For the first time compact data structures supporting the LZ77 parsing have been developed in this thesis, which show better performance on repetitive texts.

2.14.3 LZ78-based Self-Indexes

In this section we present the space and running times of two indexes based on LZ78. Although they offer decent upper bounds and competitive performance on typical texts, experiments [SVMN08] have demonstrated that LZ78 is too weak to profit from highly repetitive texts. There are other such self-indexes [FM05], not implemented as far as we know.

Arroyuelo *et al.*'s LZ-Index

Navarro's LZ-Index [Nav04] is the first self-index based on the LZ78 parsing using $O(nH_k(T))$ bits of space (it is also the first implemented in practice). It uses $4n' \log n'(1 + o(1))$ bits and takes $O(m^3 \log \sigma + (m + occ) \log n')$ time to *locate* the *occ* occurrences of a pattern of length m, where σ is the size of the alphabet, and n' is the number of phrases of the parsing.

Arroyuelo *et al.* later improved the time and space of the index, achieving $(2+\epsilon)n'\log n'(1+o(1))$ bits and $O(m^2+(m+occ)\log n')$ locate time [ANS10], or $(3+\epsilon)n'\log n'(1+o(1))$ bits and $O((m+occ)\log n')$ locate time [AN07].

Russo and Oliveira's ILZI

Russo and Oliveira present a self-index based on the so-called *maximal parsing*, called ILZI [RO08].

Definition 2.38 ([RO08])**.** Given a suffix trie $\mathcal{T}$ (of a set of strings), the $\mathcal{T}$*-maximal parsing* of string T is the sequence of nodes $v_1,\ldots,v_f$ such that $T=v_1\ldots v_f$ and, for every j, v_j is the largest prefix of $v_j\ldots v_f$ that is a node of T.

First, they compute the LZ78 parsing of T^{rev}, and then generate a suffix tree $\mathcal{T}_{78}$ over the set of the reverse phrases. Next they build the maximal parsing of T using $\mathcal{T}_{78}$. This parsing improves the compression of LZ78, as shown by the following lemma.

Lemma 2.39 ([RO08])**.** *If the number of phrases of the LZ78 parsing of T is n' then the $\mathcal{T}_{78}$-maximal parsing of T has at most n' phrases.*

Their index uses at most $5n'\log n'(1+o(1))$ bits and takes $O((m+occ)\log n')$ time to locate the *occ* occurrences of a pattern of length m (n' is the number of blocks of the maximal parsing).

2.14.4 Straight Line Programs (SLPs)

Claude and Navarro [CN09] proposed a self-index based on *straight-line programs* (SLPs). SLPs are grammars in which the rules are either $X_i \rightarrow \alpha \in \Sigma$ or $X_i \rightarrow X_lX_r$, for $l,r<i$. The LZ78 [ZL78] parsing may produce an output exponentially larger than the smallest SLP. However, the LZ77 [ZL77] parsing outperforms the smallest SLP [CLL$^+$05]. On the other hand producing the smallest SLP is an NP-complete problem [Ryt03, CLL$^+$05]. However, Rytter [Ryt03] has shown how to generate in linear time a grammar using $O(\ell\log\ell)$ rules and height $O(\log\ell)$, where ℓ is the size of the LZ77 parsing. Again, SLPs are intermediate between LZ77 and other methods.

The index [CN09] uses $n' \log n + O(n' \log n')$ bits of space, where n' is the number of rules of the grammar. It solves $extract(l, r)$ in $O((r - l + h) \log n')$ time and *locate* in $O((m(m + h) + h \cdot occ) \log n')$ time, where h is the height of the derivation tree of the grammar and m the length of the pattern.

Claude *et al.* [CFMPN10] evaluated a practical implementation using the grammar produced by Re-Pair [LM00]. The results are competitive with the RLCSA only for extremely repetitive texts and short patterns.

Chapter 3

A Repetitive Corpus Testbed

In this chapter we present a corpus of repetitive texts. These texts are categorized according to the source they come from into the following: Artificial Texts, Pseudo-Real Texts and Real Texts. The main goal of this collection is to serve as a standard testbed for benchmarking algorithms oriented to repetitive texts. The corpus can be downloaded from `http://pizzachili.dcc.uchile.cl/repcorpus.html`.

3.1 Artificial Texts

This subset is composed of highly repetitive texts that do not come from any real-life source, but are artificially generated through some mathematical definition and have interesting combinatorial properties.

3.1.1 Fibonacci Sequence (F_n)

This sequence is defined by the recurrence

$$\begin{aligned} F_1 &= \mathtt{0} \\ F_2 &= \mathtt{1} \\ F_n &= F_{n-1}F_{n-2} \end{aligned} \tag{3.1}$$

The length of the string F_n is the Fibonacci number f_n and the sequence is a *sturmian word* [Lot02], which means it has $i+1$ different substrings (factors) of length i.

3.1.2 Thue-Morse Sequence (T_n)

This sequence [AS99] is defined by the recurrence

$$\begin{aligned} T_1 &= \mathtt{0} \\ T_n &= T_{n-1}\overline{T_{n-1}} \end{aligned} \tag{3.2}$$

where $\bar{F}$ is the bitwise negation operator (i.e., all 0 get converted to 1 and all 1 to 0). Because of the construction scheme of this sequence, there are many substrings of the form XX, for any string X. However, there are no overlapping squares, i.e., substrings of the form $\mathtt{0}X\mathtt{0}X\mathtt{0}$ or $\mathtt{1}X\mathtt{1}X\mathtt{1}$. Furthermore, this sequence is strongly cube-free, i.e., there are no substrings of the form XXx, where x is the first character of the string X. Another interesting property of this string is that it is recurrent. That is, given any finite substring w of length n, there is some length n_w (often much longer than n) such that w is contained in every substring of length n_w. The length of these strings is $|T_n| = 2^n$.

3.1.3 Run-Rich String Sequence (R_n)

A measure of string complexity, related to the regularities of the text and strongly related to the LZ77 parsing [KK99], is the number of runs.

Definition 3.1. A *period* of string $T[1,n]$ is a positive integer p holding that $\forall\, 1 \leq i \leq n-p$, $T[i] = T[i+p]$. A string is said to be *periodic* if its minimum period p is such that $p \leq n/2$.

Definition 3.2 ([Mai89])**.** The substring $T[i,j]$ is a *run* in a string T iff $T[i,j]$ is periodic and $T[i,j]$ is not extendable to the right ($j = n$ or $T[j+1] \neq T[j-p+1]$) or left ($i = 1$ or $T[i-1] \neq T[i+p-1]$).

The higher the number of runs in a string, the more regularities it has.

It has been shown that the maximum number of runs in a string is greater than $0.944n$ [MKI$^+$08] and lower than $1.029n$ [CIT08]. Franek *et al.* [FSS03] show a constructive and simple way to obtain strings with many runs; the n-th of those strings is denoted R_n. The ratio of the runs of their strings to the length approaches $3/(1+\sqrt{5}) = 0.92705\ldots$.

3.2 Pseudo-Real Texts

Here we present a set of texts that were generated by artificially adding repetitiveness to real texts, thus we call them *pseudo-real texts*.

To generate the texts, we took a prefix of 1MiB of all texts of Pizza&Chili Corpus[1], we mutated them, and we concatenated all of them in the order they were generated. Our mutations take a random character position and change it to a random character different from the original one.

We used two different schemes for the mutations. The first one, denoted by a '^{1}', generates different mutations of the first text. The second, denoted by a '^{2}', mutates the last text generated. The second scheme resembles the changes obtained through time in a software project or the versions of a document, while the first scheme produces changes analogous to the ones found in a collection of related DNA sequences.

The mutation rate, i.e., percentage of mutated characters, was set to 0.1%, 0.01% and 0.001%.

The base texts (all from the Pizza&Chili corpus) we mutated were the following:

- Sources: This file is formed by C/Java source code obtained by concatenating all the `.c`, `.h`, `.C` and `.java` files of the linux-2.6.11.6 and gcc-4.0.0 distributions.
- Pitches: This file is a sequence of midi pitch values (bytes in 0-127, plus a few extra special values) obtained from a myriad of MIDI files freely available on Internet.
- Proteins: This file is a sequence of newline-separated protein sequences obtained from the Swissprot database.
- DNA: This file is a sequence of newline-separated gene DNA sequences obtained from files `01hgp10` to `21hgp10`, plus `0xhgp10` and `0yhgp10`, from Gutenberg Project.
- English: This file is the concatenation of English text files selected from `etext02` to `etext05` collections of Gutenberg Project.
- XML: This file is an XML that provides bibliographic information on major computer science journals and proceedings and it was obtained from `http://dblp.uni-trier.de`.

[1] `http://pizzachili.dcc.uchile.cl`

3.3 Real Texts

This subset is composed of texts coming from real repetitive sources. These sources are DNA, Wikipedia Articles, Source Code, and Documents.

For the case of DNA we concatenated the texts in random order. For the others, we concatenated the texts according to the date they were created, from oldest to newest.

3.3.1 DNA

Our DNA texts come from different sources.

- The Saccharomyces Genome Resequencing Project[2] provides two text collections: *para*, which contains 36 sequences of *Saccharomyces Paradoxus* and *cere*, which contains 37 sequences of *Saccharomyces Cerevisiae.*
- From the National Center for Biotechnology Information (NCBI)[3] we collected some DNA sequences of the same bacteria. The species we collected are *Escherichia Coli* (23), *Salmonella Enterica* (15), *Staphylococcus Aureus* (14), *Streptococcus Pyogenes* (13), *Streptococcus Pneumoniae* (11) and *Clostridium Botulium* (10). We wrote in parentheses the total number of sequences of each collection. We chose these species as they were the only ones with 10 or more different sequences.
- A collection composed of 78,041 sequences of *Haemophilus Influenzae*[4], also coming from the NCBI.

Remark 3.3. Although there are four bases $\{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$, DNA sequences may have alphabets of size up to $16 = 2^4$ because some characters denote an unknown choice among the four bases. The most common character used is `N`, which denotes a totally unknown symbol.

[2] `http://www.sanger.ac.uk/Teams/Team71/durbin/sgrp`

[3] `http://www.ncbi.nlm.nih.gov`

[4] `ftp://ftp.ncbi.nih.gov/genomes/INFLUENZA/influenza.fna.gz`

3.3.2 Wikipedia Articles

We downloaded all versions of three Wikipedia articles, *Albert Einstein*, *Alan Turing* and *Nobel Prize*. We downloaded them in English (denoted *en*) and German (denoted *de*). We chose these languages as they are among the most widely used on Internet and their alphabet may be represented using standard 1-byte encodings. The versions for all documents are up to January 12, 2010, except for the English article of *Albert Einstein*, which was downloaded only up to November 10, 2006 because of the massive number of versions it has.

3.3.3 Source Code

We collected all versions 5.x of the *Coreutils*[5] package and removed all binary files, making a total of 9 versions. We also collected all 1.0.x and 1.1.x versions of the *Linux Kernel*[6], making a total of 36 versions.

3.3.4 Documents

We took all *pdf* files of CIA World Leaders[7] from January 2003 to December 2009, and converted them to text (using software `pdftotext`).

3.4 Statistics

To understand the characteristics of the texts present in the *Repetitive Corpus*, we provide below some statistics about them. The statistics presented are the following:

- **Alphabet Size:** We give the alphabet size and the inverse probability of matching (IPM), which is the inverse of the probability that two characters chosen at random match. IPM is a measure of the effective alphabet size. On a uniformly distributed text, it is precisely the alphabet size.

- **Compression Ratio:** Since we are dealing with compressed indexes it is useful to have an idea of the compressibility of the texts using general-purpose

[5] `ftp://mirrors.kernel.org/gnu/coreutils`

[6] `ftp://ftp.kernel.org/pub/linux/kernel`

[7] `https://www.cia.gov/library/publications/world-leaders-1`

compressors. The following compressors are used: `gzip`[8] gives an idea of compressibility via dictionaries (an LZ77 parsing with limited window size); `bzip2`[9] gives an idea of block-sorting compressibility (using the BWT transform, Section 2.12); `ppmdi`[10] gives an idea of partial-match-based compressors (related to the k-th order entropy, Section 2.3); `p7zip`[11] gives an idea of LZ77 compression with virtually unlimited window; and `Re-Pair`[12] [LM00] gives an idea of grammar-based compression. All compressors were run with the highest compression options.

- **Empirical Entropy:** Here we give the empirical entropy H_k of the text with k ranging from 0 to 8, measured as compression ratio. We also show, in parentheses, the *complexity function* of T [Lot02] (or the *number of contexts*) which count how many different substrings of a given size does T have. This is exactly our $C(T,k)$ of Lemma 2.12. This measure has the following properties:

$$\begin{aligned} C(T,1) &= \sigma \\ C(T,n+m) &\leq C(T,n)C(T,m) \end{aligned}$$

  The lower this measure, the more repetitive the text is. For example, if $C(T,n) = 1\,\forall n$, then $T = c^m$ for some character c. When $P(C,n) = n+1$ the sequence is said to be *Sturmian* (the Fibonacci sequence is an example of a *Sturmian* string).

Remark 3.4. The compression ratios are given as the percentage of the compressed file size over the uncompressed file size, assuming the original file uses one byte per character. This means that 25% compression can be achieved over a DNA sequence having an alphabet {`A`,`C`,`G`,`T`} by simply using 2 bits per symbol. As seen from the real-life examples given, these four symbols are usually predominant, so it is not hard to get very close to 25% on general DNA sequences as well.

3.4.1 Artificial Texts

Tables 3.1-3.3 give the statistics of artificial texts.

[8] `http://www.gzip.org`

[9] `http://www.bzip.org`

[10] `http://pizzachili.dcc.uchile.cl/utils/ppmdi.tar.gz`

[11] `http://www.7-zip.org`

[12] `http://www.cbrc.jp/~rwan/en/restore.html`

File	**Size**	Σ	**IPM**
F_{41}	256MiB	2	1.894
T_{29}	256MiB	2	2.000
R_{13}	207MiB	2	2.000

Table 3.1: Alphabet statistics for Artificial Collection

File	**p7zip**	**bzip2**	**gzip**	**ppmdi**	**Re-pair**
F_{41}	0.17624%	0.00572%	0.46875%	1.87500%	0.00002%
T_{29}	0.35896%	0.01259%	0.54688%	2.18750%	0.00004%
R_{13}	0.17172%	0.01227%	0.53140%	2.12560%	0.00009%

Table 3.2: Compression statistics for Artificial Collection

File	H_0	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
F_{41}	11.99% (1)	7.41% (2)	4.58% (3)	4.58% (4)	2.83% (5)	2.83% (6)	2.83% (7)	1.75% (8)	1.75% (9)
T_{29}	12.50% (1)	11.48% (2)	8.34% (4)	8.34% (6)	4.16% (10)	4.16% (12)	4.16% (16)	2.09% (20)	2.09% (22)
R_{13}	12.50% (1)	9.85% (2)	8.51% (4)	6.55% (6)	2.56% (8)	2.33% (10)	2.33% (12)	2.33% (14)	2.33% (16)

Table 3.3: Empirical entropy statistics for Artificial Collection

3.4.2 Pseudo-Real Texts

Tables 3.4-3.9 give the statistics of pseudo-real texts.

File	**Size**	Σ	**IPM**
Xml 0.001%[1]	100MiB	89	27.84
Xml 0.01%[1]	100MiB	89	27.84
Xml 0.1%[1]	100MiB	89	27.84
DNA 0.001%[1]	100MiB	5	3.98
DNA 0.01%[1]	100MiB	5	3.98
DNA 0.1%[1]	100MiB	5	3.98
English 0.001%[1]	100MiB	106	15.65
English 0.01%[1]	100MiB	106	15.65
English 0.1%[1]	100MiB	106	15.65
Pitches 0.001%[1]	100MiB	73	33.07
Pitches 0.01%[1]	100MiB	73	33.07
Pitches 0.1%[1]	100MiB	73	33.07
Proteins 0.001%[1]	100MiB	21	16.90
Proteins 0.01%[1]	100MiB	21	16.90
Proteins 0.1%[1]	100MiB	21	16.90
Sources 0.001%[1]	100MiB	98	28.86
Sources 0.01%[1]	100MiB	98	28.86
Sources 0.1%[1]	100MiB	98	28.86

Table 3.4: Alphabet statistics for Pseudo-Real Collection (Scheme 1)

File	**Size**	Σ	**IPM**
Xml 0.001%[2]	100MiB	89	27.84
Xml 0.01%[2]	100MiB	89	27.84
Xml 0.1%[2]	100MiB	89	27.86
DNA 0.001%[2]	100MiB	5	3.98
DNA 0.01%[2]	100MiB	5	3.98
DNA 0.1%[2]	100MiB	5	3.98
English 0.001%[2]	100MiB	106	15.65
English 0.01%[2]	100MiB	106	15.66
English 0.1%[2]	100MiB	106	15.74
Pitches 0.001%[2]	100MiB	73	33.07
Pitches 0.01%[2]	100MiB	73	33.07
Pitches 0.1%[2]	100MiB	73	33.10
Proteins 0.001%[2]	100MiB	21	16.90
Proteins 0.01%[2]	100MiB	21	16.90
Proteins 0.1%[2]	100MiB	21	16.92
Sources 0.001%[2]	100MiB	98	28.86
Sources 0.01%[2]	100MiB	98	28.86
Sources 0.1%[2]	100MiB	98	28.92

Table 3.5: Alphabet statistics for Pseudo-Real Collection (Scheme 2)

File	p7zip	bzip2	gzip	ppmdi	Re-Pair
Xml 0.001%^{1}	0.15%	11.00%	18.00%	3.50%	0.19%
Xml 0.01%^{1}	0.18%	12.00%	18.00%	3.60%	0.46%
Xml 0.1%^{1}	0.46%	12.00%	18.00%	4.10%	2.00%
DNA 0.001%^{1}	0.27%	27.00%	28.00%	11.00%	0.34%
DNA 0.01%^{1}	0.29%	27.00%	28.00%	11.00%	0.58%
DNA 0.1%^{1}	0.51%	27.00%	28.00%	12.00%	2.50%
English 0.001%^{1}	0.31%	28.00%	37.00%	22.00%	0.39%
English 0.01%^{1}	0.35%	28.00%	37.00%	22.00%	0.65%
English 0.1%^{1}	0.59%	28.00%	37.00%	22.00%	2.70%
Pitches 0.001%^{1}	0.47%	54.00%	52.00%	47.00%	0.69%
Pitches 0.01%^{1}	0.50%	54.00%	52.00%	47.00%	0.95%
Pitches 0.1%^{1}	0.75%	54.00%	52.00%	48.00%	3.20%
Proteins 0.001%^{1}	0.32%	41.00%	39.00%	31.00%	0.42%
Proteins 0.01%^{1}	0.35%	41.00%	39.00%	31.00%	0.68%
Proteins 0.1%^{1}	0.59%	41.00%	39.00%	32.00%	2.70%
Sources 0.001%^{1}	0.20%	19.00%	25.00%	12.00%	0.28%
Sources 0.01%^{1}	0.23%	19.00%	25.00%	12.00%	0.56%
Sources 0.1%^{1}	0.50%	20.00%	25.00%	13.00%	2.60%

Table 3.6: Compression statistics for Pseudo-Real Collection (Scheme 1)

File	p7zip	bzip2	gzip	ppmdi	Re-Pair
Xml 0.001%^{2}	0.15%	12.00%	18.00%	3.50%	0.18%
Xml 0.01%^{2}	0.18%	14.00%	19.00%	4.40%	0.39%
Xml 0.1%^{2}	0.39%	25.00%	29.00%	17.00%	2.20%
DNA 0.001%^{2}	0.26%	27.00%	28.00%	11.00%	0.33%
DNA 0.01%^{2}	0.29%	27.00%	28.00%	11.00%	0.52%
DNA 0.1%^{2}	0.46%	27.00%	28.00%	13.00%	2.20%
English 0.001%^{2}	0.31%	28.00%	37.00%	22.00%	0.38%
English 0.01%^{2}	0.34%	29.00%	37.00%	23.00%	0.59%
English 0.1%^{2}	0.55%	38.00%	43.00%	31.00%	2.50%
Pitches 0.001%^{2}	0.46%	54.00%	52.00%	47.00%	0.68%
Pitches 0.01%^{2}	0.49%	54.00%	53.00%	48.00%	0.89%
Pitches 0.1%^{2}	0.71%	59.00%	57.00%	52.00%	2.80%
Proteins 0.001%^{2}	0.31%	41.00%	39.00%	32.00%	0.41%
Proteins 0.01%^{2}	0.34%	42.00%	40.00%	33.00%	0.62%
Proteins 0.1%^{2}	0.54%	47.00%	46.00%	40.00%	2.50%
Sources 0.001%^{2}	0.20%	20.00%	25.00%	13.00%	0.27%
Sources 0.01%^{2}	0.23%	21.00%	26.00%	14.00%	0.49%
Sources 0.1%^{2}	0.44%	34.00%	35.00%	26.00%	2.50%

Table 3.7: Compression statistics for Pseudo-Real Collection (Scheme 2)

File	H_0	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
Xml 0.001%^{1}	65.25% (1)	38.63% (89)	21.00% (3325)	12.50% (20560)	8.13% (56120)	6.00% (98084)	5.25% (134897)	4.75% (168846)	4.13% (200451)
Xml 0.01%^{1}	65.25% (1)	38.63% (89)	21.00% (4135)	12.50% (30975)	8.13% (79379)	6.00% (131811)	5.25% (177924)	4.75% (220923)	4.13% (261651)
Xml 0.1%^{1}	65.25% (1)	38.75% (89)	21.25% (5251)	12.75% (67479)	8.25% (196554)	6.13% (326296)	5.38% (440199)	4.88% (550570)	4.25% (661284)
DNA 0.001%^{1}	25.00% (1)	24.25% (5)	24.13% (18)	24.00% (67)	24.00% (260)	23.75% (1029)	23.50% (4102)	22.88% (16349)	21.25% (62437)
DNA 0.01%^{1}	25.00% (1)	24.25% (5)	24.13% (18)	24.00% (67)	24.00% (260)	23.75% (1029)	23.50% (4102)	22.88% (16368)	21.25% (63204)
DNA 0.1%^{1}	25.00% (1)	24.25% (5)	24.13% (19)	24.00% (70)	24.00% (264)	23.75% (1034)	23.50% (4109)	22.88% (16399)	21.38% (65168)
English 0.001%^{1}	57.25% (1)	45.13% (106)	34.75% (2659)	25.88% (18352)	19.88% (63299)	15.88% (145194)	12.50% (256838)	9.63% (379514)	7.25% (501400)
English 0.01%^{1}	57.25% (1)	45.13% (106)	34.75% (3243)	25.88% (24063)	19.88% (82896)	15.88% (180401)	12.50% (305292)	9.63% (439387)	7.25% (572056)
English 0.1%^{1}	57.25% (1)	45.25% (106)	34.88% (4491)	26.13% (46116)	20.13% (190765)	16.00% (439130)	12.50% (715127)	9.75% (983435)	7.25% (1237512)
Pitches 0.001%^{1}	66.13% (1)	61.00% (73)	53.50% (3549)	37.13% (73664)	16.38% (376958)	6.25% (642406)	2.88% (767028)	1.38% (833456)	0.75% (871970)
Pitches 0.01%^{1}	66.13% (1)	61.00% (73)	53.50% (3581)	37.25% (76900)	16.38% (399435)	6.25% (684445)	2.88% (821533)	1.38% (898126)	0.75% (946219)
Pitches 0.1%^{1}	66.13% (1)	61.13% (73)	53.63% (3733)	37.38% (95838)	16.63% (598394)	6.38% (1096014)	2.88% (1363610)	1.50% (1543086)	0.88% (1687166)
Proteins 0.001%^{1}	52.25% (1)	52.13% (21)	51.63% (422)	47.50% (8045)	25.13% (128975)	4.63% (463357)	0.75% (572530)	0.25% (589356)	0.25% (595906)
Proteins 0.01%^{1}	52.25% (1)	52.13% (21)	51.63% (422)	47.50% (8045)	25.13% (131064)	4.63% (494845)	0.75% (626269)	0.25% (654067)	0.25% (670075)
Proteins 0.1%^{1}	52.25% (1)	52.13% (21)	51.63% (425)	47.50% (8076)	25.50% (143879)	4.88% (768510)	0.88% (1150595)	0.38% (1293347)	0.38% (1403589)
Sources 0.001%^{1}	68.75% (1)	46.88% (98)	30.00% (4557)	19.63% (29667)	14.38% (75316)	11.00% (130527)	8.38% (194105)	6.88% (259413)	5.75% (320468)
Sources 0.01%^{1}	68.75% (1)	46.88% (98)	30.00% (5621)	19.63% (42303)	14.38% (102977)	11.00% (170525)	8.50% (244755)	6.88% (320237)	5.75% (391260)
Sources 0.1%^{1}	68.75% (1)	47.00% (98)	30.25% (7359)	19.88% (104679)	14.63% (299799)	11.13% (498046)	8.50% (687941)	7.00% (872189)	5.88% (1049051)

Table 3.8: Empirical entropy statistics for Pseudo-Real Collection (Scheme 1)

File	H_0	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
Xml 0.001%^{2}	65.25% (1)	38.63% (89)	21.13% (3325)	12.63% (20560)	8.13% (56120)	6.00% (98084)	5.25% (134897)	4.75% (168846)	4.13% (200451)
Xml 0.01%^{2}	65.25% (1)	39.38% (89)	22.00% (4135)	13.25% (31042)	8.63% (79630)	6.50% (132163)	5.63% (178388)	5.13% (221499)	4.50% (262329)
Xml 0.1%^{2}	65.25% (1)	44.00% (89)	28.75% (5255)	18.50% (72227)	12.25% (226418)	9.25% (378994)	8.00% (513539)	7.13% (645141)	6.25% (777226)
DNA 0.001%^{2}	25.00% (1)	24.25% (5)	24.13% (18)	24.00% (67)	24.00% (260)	23.75% (1029)	23.50% (4102)	22.88% (16349)	21.25% (62436)
DNA 0.01%^{2}	25.00% (1)	24.25% (5)	24.13% (18)	24.13% (67)	24.00% (260)	23.88% (1029)	23.50% (4102)	23.00% (16369)	21.38% (63242)
DNA 0.1%^{2}	25.00% (1)	24.50% (5)	24.38% (19)	24.25% (70)	24.25% (264)	24.13% (1034)	23.88% (4109)	23.50% (16400)	22.38% (65387)
English 0.001%^{2}	57.25% (1)	45.13% (106)	34.75% (2659)	26.00% (18353)	20.00% (63300)	15.88% (145195)	12.50% (256838)	9.63% (379514)	7.13% (501400)
English 0.01%^{2}	57.25% (1)	45.50% (106)	35.38% (3243)	26.50% (24079)	20.25% (83037)	15.88% (180592)	12.38% (305458)	9.50% (439539)	7.13% (572186)
English 0.1%^{2}	57.38% (1)	47.75% (106)	39.50% (4482)	31.13% (47357)	23.00% (202366)	16.63% (466838)	12.13% (749065)	8.88% (1015587)	6.38% (1265447)
Pitches 0.001%^{2}	66.13% (1)	61.13% (73)	53.63% (3549)	37.25% (73664)	16.38% (376958)	6.25% (642406)	2.88% (767028)	1.38% (833456)	0.75% (871970)
Pitches 0.01%^{2}	66.13% (1)	61.13% (73)	53.88% (3581)	37.50% (76917)	16.50% (399546)	6.38% (684518)	2.88% (821589)	1.38% (898152)	0.88% (946228)
Pitches 0.1%^{2}	66.13% (1)	62.00% (73)	55.88% (3742)	40.25% (96359)	17.38% (606175)	6.50% (1103560)	3.13% (1367417)	1.88% (1545154)	1.38% (1688526)
Proteins 0.001%^{2}	52.25% (1)	52.13% (21)	51.63% (422)	47.50% (8045)	25.25% (128975)	4.63% (463357)	0.75% (572529)	0.25% (589356)	0.25% (595906)
Proteins 0.01%^{2}	52.25% (1)	52.13% (21)	51.63% (422)	47.63% (8045)	25.75% (131079)	5.00% (494846)	0.88% (626306)	0.50% (654107)	0.38% (670114)
Proteins 0.1%^{2}	52.25% (1)	52.13% (21)	51.75% (426)	48.75% (8072)	30.13% (143924)	7.63% (771311)	2.13% (1154106)	1.50% (1297080)	1.38% (1407901)
Sources 0.001%^{2}	68.75% (1)	47.00% (98)	30.00% (4557)	19.75% (29667)	14.38% (75316)	11.00% (130527)	8.50% (194105)	6.88% (259413)	5.75% (320468)
Sources 0.01%^{2}	68.75% (1)	47.50% (98)	30.75% (5615)	20.13% (42337)	14.63% (103082)	11.13% (170646)	8.63% (244874)	7.00% (320346)	5.88% (391369)
Sources 0.1%^{2}	68.75% (1)	51.25% (98)	36.63% (7372)	24.38% (108997)	16.75% (319310)	12.13% (525914)	9.13% (718657)	7.25% (904022)	6.00% (1080824)

Table 3.9: Empirical entropy statistics for Pseudo-Real Collection (Scheme 2)

3.4.3 Real Texts

Tables 3.10-3.12 give the statistics of real texts.

File	**Size**	Σ	**IPM**
Cere	440MiB	5	4.301
Para	410MiB	5	4.096
Clostridium Botulium	34MiB	4	3.356
Escherichia Coli	108MiB	15	4.000
Salmonella Enterica	66MiB	9	3.993
Staphylococcus Aureus	38MiB	5	3.579
Streptococcus Pneumoniae	23MiB	8	3.836
Streptococcus Pyogenes	24MIB	10	3.800
Influenza	148MiB	15	3.845
Coreutils	196MiB	236	19.553
Kernel	247MiB	160	23.078
Einstein (en)	446MiB	139	19.501
Einstein (de)	89MiB	117	19.264
Nobel (en)	85MiB	126	20.070
Nobel (de)	31MiB	118	17.786
Turing (en)	7.7MiB	103	21.096
Turing (de)	85MiB	100	19.719
World Leaders	45MiB	89	3.855

Table 3.10: Alphabet statistics for Real Collection

File	**p7zip**	**bzip2**	**gzip**	**ppmdi**	**Re-Pair**
Cere	1.14%	2.50%	26.36%	24.09%	1.86%
Para	1.46%	26.34%	27.07%	24.88%	2.80%
Clostridium Botulium	8.53%	25.88%	26.47%	24.12%	20.00%
Escherichia Coli	4.72%	26.85%	28.70%	25.93%	9.63%
Salmonella Enterica	5.61%	27.27%	28.79%	25.76%	12.42%
Staphylococcus Aureus	2.89%	26.32%	28.95%	25.00%	5.26%
Streptococcus Pneumoniae	4.78%	26.52%	27.39%	24.78%	9.57%
Streptococcus Pyogenes	5.00%	26.25%	27.08%	25.00%	9.58%
Influenza	1.35%	6.62%	7.43%	3.78%	3.31%
coreutils	1.94%	16.33%	24.49%	12.76%	2.55%
kernel	0.81%	21.86%	27.13%	18.62%	1.13%
einstein.en	0.07%	5.38%	35.20%	1.61%	0.10%
einstein.de	0.11%	4.38%	31.46%	1.35%	0.16%
nobel.en	0.13%	2.94%	18.82%	1.76%	0.20%
nobel.de	0.18%	3.55%	27.74%	1.68%	0.30%
turing.en	1.09%	36.36%	285.71%	15.58%	1.71%
turing.de	0.03%	0.18%	0.10%	0.11%	0.05%
world leaders	1.29%	7.11%	17.78%	3.56%	1.78%

Table 3.11: Compression statistics for Real Collection

File	H_0	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
Cere	27.38% (1)	22.63% (5)	22.63% (25)	22.50% (125)	22.50% (610)	22.50% (2515)	22.50% (8697)	22.38% (28080)	22.25% (88624)
Para	26.50% (1)	23.50% (5)	23.38% (25)	23.38% (125)	23.38% (625)	23.38% (3125)	23.25% (14725)	23.25% (51542)	23.13% (139149)
Clostridium Botulium	23.25% (1)	23.00% (4)	22.88% (16)	22.75% (64)	22.75% (256)	22.75% (1024)	22.63% (4096)	22.50% (16383)	22.25% (65118)
Escherichia Coli	25.00% (1)	24.75% (15)	24.50% (145)	24.38% (779)	24.25% (2715)	24.25% (7436)	24.13% (15641)	24.13% (32561)	23.88% (85363)
Salmonella Enterica	25.00% (1)	24.75% (9)	24.50% (35)	24.38% (97)	24.25% (299)	24.13% (1077)	24.13% (4159)	24.00% (16457)	23.75% (65618)
Staphylococcus Aureus	23.88% (1)	23.75% (5)	23.75% (18)	23.63% (67)	23.63% (260)	23.63% (1029)	23.50% (4102)	23.25% (16391)	22.75% (65282)
Streptococcus Pneumoniae	24.63% (1)	24.38% (8)	24.38% (31)	24.25% (133)	24.13% (574)	24.13% (2183)	24.00% (6928)	23.75% (21093)	23.13% (71592)
Streptococcus Pyogenes	24.50% (1)	24.38% (10)	24.25% (50)	24.13% (174)	24.13% (456)	24.13% (1291)	24.00% (4418)	23.88% (16758)	23.25% (65919)
Influenza	24.63% (1)	24.13% (15)	24.13% (125)	24.00% (583)	23.88% (2329)	23.50% (7978)	22.00% (21316)	18.63% (44748)	13.25% (101559)
coreutils	68.38% (1)	51.25% (236)	35.88% (18500)	23.88% (169716)	17.00% (606527)	12.88% (1335553)	10.13% (2258650)	8.00% (3258896)	6.50% (4247313)
kernel	67.25% (1)	50.50% (160)	36.63% (7122)	25.75% (90396)	19.25% (351918)	15.13% (773818)	12.13% (1305616)	9.63% (1912604)	7.75% (2553008)
einstein.en	62.00% (1)	46.38% (139)	33.38% (4546)	21.13% (28685)	13.25% (77333)	9.00% (142559)	6.50% (211506)	4.75% (276343)	3.50% (335151)
einstein.de	63.00% (1)	44.88% (117)	32.63% (3278)	20.88% (16765)	13.25% (39010)	9.00% (64884)	6.13% (89914)	4.38% (112043)	3.13% (130473)
nobel.en	62.63% (1)	44.63% (126)	30.50% (3566)	18.25% (18079)	11.50% (42334)	8.13% (69855)	6.00% (95644)	4.50% (119260)	3.38% (140401)
nobel.de	61.13% (1)	43.25% (118)	31.13% (2726)	19.63% (12959)	12.50% (30756)	8.63% (49695)	6.00% (66108)	4.13% (80467)	3.00% (92184)
turing.en	63.25% (1)	45.75% (103)	32.00% (2794)	19.13% (14091)	11.50% (33498)	7.63% (55489)	5.38% (75611)	3.88% (93402)	2.88% (108636)
turing.de	62.38% (1)	43.25% (100)	29.25% (1806)	16.75% (7268)	9.50% (15407)	6.00% (23070)	3.88% (29038)	2.63% (33714)	2.00% (37335)
world leaders	43.38% (1)	24.38% (89)	17.25% (2526)	11.63% (23924)	7.63% (106573)	5.13% (246566)	4.00% (374668)	3.50% (468701)	3.13% (547040)

Table 3.12: Empirical entropy statistics for Real Collection

3.5 Discussion

It can be seen in the tables presented above that only p7zip and Re-Pair capture the repetitiveness of the texts, achieving a compression ratio at least one order of magnitude better than bzip2, gzip or ppmdi. It can also be noted in Tables 3.6 and 3.7 that p7zip is more robust to capture the repetitiveness than Re-Pair, as with mutation ratios of 0.1% p7zip compresses 5 times better than Re-Pair. Table 3.11 also shows that Re-Pair fails to capture some repetitions, as for all DNA texts except *para* and *cere* the compression of p7zip is two times better than that of Re-Pair. Tables 3.6 and 3.8 also show that the compression ratio of bzip2, gzip and ppmdi does not change significantly when increasing the repetitiveness of the text (decreasing mutation ratio). However, Tables 3.7 and 3.9 show that when decreasing the mutation ratio from 0.1% to 0.01% the gain in compression is greater than 10%, but when decreasing the mutation to 0.001% the compression ratio does not improve as much. It can also be seen that the compression ratios of bzip2 and gzip are close to the H_2-H_3, whereas, curiously, ppmdi compression ratios are not well predicted by any H_k. Notice that, since artificial texts are extremely compressible, small constant overheads (usually irrelevant) may produce significant differences in the size of the compressed file.

Chapter 4

LZ-End: A New Lempel-Ziv Parsing

In this chapter we explain some properties of the LZ77 parsing (see Section 2.13) and present a variant that has the advantage of faster text extraction. The results presented in this chapter were published in the *20th Data Compression Conference (DCC)* [KN10].

4.1 LZ77 on Repetitive Texts

An interesting property of the LZ77 parsing is that it captures the repetitions of the text. Text repetitions, as well as single-character edits on a text, alter the number of phrases of the parsing very little. This explains why LZ77 is so strong on highly repetitive collections.

Lemma 4.1. *Given the texts T, T' and the characters a , b; the following statements hold*

$$\begin{aligned} H^{LZ77}(TT) &= H^{LZ77}(T)+1 && (4.1)\\ H^{LZ77}(TT\$) &\leq H^{LZ77}(T\$)+1 && (4.2)\\ H^{LZ77}(TT') &\leq H^{LZ77}(TaT')+1 && (4.3)\\ H^{LZ77}(TaT') &\leq H^{LZ77}(TT')+1 && (4.4)\\ H^{LZ77}(TaT') &\leq H^{LZ77}(TbT')+1 && (4.5) \end{aligned}$$

where $H^{LZ77}(T)$ is the number of phrases of the LZ77 parsing.

Proof. Assume the last phrase of the LZ77 parsing of $T\$$ is $\boxed{\$}$ and that $H^{LZ77}(T\$) = n'$. That means the first $n'-1$ phrases cover the text T. Now, if we have the text $TT\$$, we have that the first $n'-1$ phrases are the same as for the parsing of T and the last phrase would be $\boxed{T\$}$, hence inequality for Equation (4.2) holds. Now, assume the last phrase of the parsing is $\boxed{A\$}$ for some $A \neq \varepsilon$. Therefore the n'-th phrase of the parsing of TT would be $\boxed{AB}$ for some B such that $1 \leq |B| < |T|$, thus this phrase does not completely cover TT. An additional phrase covers the remaining portion of the text, thus equality holds for Equation (4.2). The proof of Equation (4.1) is similar to the second part of Equation (4.2).

Now consider Equation (4.3). Let $Z[p] = XY$ the last phrase covering T, where X is a suffix of T and Y is a prefix of T'. When adding the new character in the middle, in the worst case the phrase gets converted to Xa (this is the phrase that may increase the total number of phrases). Then the following phrase will cover at least the prefix Y, and each successive phrase will cover at least the next phrase of the original parsing. Hence, the number of phrases is at most one more than the original number of phrases. The proofs for Equations (4.4) and (4.5) are similar to the one above. □

The LZ78 parsing [ZL78] described in Section 2.14.3 is not that powerful. On $T = a^n$ it produces $n' = \frac{\sqrt{n}}{2} + O(1)$ phrases, and this increases to $n' = \frac{\sqrt{2n}}{2} + O(1)$ on TT. LZ77, instead, produces $n' = \log_2(n) + O(1)$ phrases on T and just one more phrase on TT.

4.2 LZ-End

In this section we introduce a new LZ-like parsing. Its main characteristic is a faster random text extraction, while its compression is close to that of LZ77.

Definition 4.2. The *LZ-End parsing* of text $T[1,n]$ is a sequence $Z[1,n']$ of *phrases* such that $T = Z[1]Z[2]\ldots Z[n']$, built as follows. Assume we have already processed $T[1,i-1]$ producing the sequence $Z[1,p-1]$. Then, we find the longest prefix $T[i,i'-1]$ of $T[i,n]$ that is a suffix of $Z[1]\ldots Z[q]$ for some $q < p$, set $Z[p] = T[i,i']$ and continue with $i = i'+1$.

Example 4.3. Let $T =$ `alabar_a_la_alabarda$`; the LZ-End parsing is as follows:

a	l	ab	ar	_	a_	la	_a	labard	a$

In this example, when generating the seventh phrase we cannot copy two characters as in Example 2.31, because `‘la’` does not end in a previous end of phrase. However, `‘l’` does end in an end of phrase, hence we generate the phrase `‘la’`. Notice that the number of phrases increased from 9 to 10 with respect to the original LZ77 scheme.

The LZ-End parsing is similar to the one proposed by Fiala and Green [FG89], in that theirs restricts where the sources start, while ours restricts where the sources end. This is the key feature that will allow us extract arbitrary phrases in constant time per extracted symbol and, as shown in Section 4.2.2.

4.2.1 Encoding

The output of an LZ77 compressor is, essentially, the sequence of triplets $z(p) = (j, \ell, c)$, such that the source of $Z[p] = T[i, i']$ is $T[j, j+\ell-1]$, $\ell = i' - i$, and $c = T[i']$. This format allows fast decompression of T, but not decompressing an individual phrase $Z[p]$ in reasonable time (one must basically decompress the whole text).

The LZ-End parsing, although potentially generates more phrases than LZ77, permits a shorter encoding of each, of the form $z(p) = (q, \ell, c)$, such that the source of $Z[p] = T[i, i']$ is a suffix of $Z[1] \dots Z[q]$, and the rest is as above. This representation is shorter because it stores the phrase identifier rather than a text position. We introduce a more sophisticated encoding that will, in addition, allow us to extract individual phrases in constant time per extracted symbol.

- $\mathit{char}[1, n']$ (using $n' \lceil \log \sigma \rceil$ bits) encodes the trailing characters (c above).
- $\mathit{source}[1, n']$ (using $n' \lceil \log n' \rceil$ bits) encodes the phrase identifier where the source ends (q above).
- $B[1, n]$ (using $n' \log \frac{n}{n'} + O(n' + \frac{n \log \log n}{\log n})$ bits in compressed form [RRR02], see Section 2.5) marks the ending positions of the phrases in T.

Thus we have $z(p) = (q, l, c) = (\mathit{source}[p], \mathit{select}_1(B, p+1) - \mathit{select}_1(B, p) - 1, \mathit{char}[p])$. We can also know in constant time that phrase p ends at $\mathit{select}_1(B, \mathit{source}[p])$ and that it starts ℓ positions before. Finally, we can know that the text position i belongs to phrase $Z[\mathit{rank}_1(B, i-1) + 1]$.

```
**Extract**($start, len$)
1  **if** $len > 0$ **then**
2    $end \leftarrow start + len - 1$
3    $p \leftarrow rank_1(B, end)$
4    **if** $B[end] = 1$ **then**
5      Extract($start, len - 1$)
6      **output** $char[p]$
7    **else**
8      $pos \leftarrow select_1(B, p) + 1$
9      **if** $start < pos$ **then**
10       Extract($start, pos - start$)
11       $len \leftarrow end - pos + 1$
12       $start \leftarrow pos$
13     Extract($select_1(B, source[p+1]) - select_1(B, p+1) + start + 1, len$)
```

Figure 4.1: LZ-End extraction algorithm for $T[start, start + len - 1]$.

4.2.2 Extraction Algorithm

The algorithm to extract an arbitrary substring in LZ-End is given in Figure 4.1. The extraction works from right to left. First we compute the last phrase p intersecting the substring. If the last character is stored explicitly, i.e., it is an end of phrase (see Line 4), we output $char[p]$ and recursively extract the remaining substring (line 5). Otherwise we split the substring into two parts. The right one is the intersection of the rightmost phrase covering the substring and the substring itself, and is extracted recursively by going to the source of that phrase (line 10). The left part is also extracted recursively (line 13).

While the algorithm works for extracting any substring, we can prove it takes constant time per extracted symbol when the substring ends at a phrase.

Theorem 4.4. *Function* Extract *outputs a text substring* $T[start, end]$ *ending at a phrase in time* $O(end - start + 1)$.

Proof. If $T[start, end]$ ends at a phrase, then $B[end] = 1$. We proceed by induction on $len = end - start + 1$. The case $len \leq 1$ is trivial by inspection. Otherwise, we output $T[end]$ at line 6 after a recursive call on the same phrase and length $len - 1$. This time we go to line 8. The current phrase (now $p+1$) starts at pos. If $start < pos$, we carry out a recursive call at line 10 to handle the segment $T[start, pos - 1]$. As this segment ends at the end of phrase p, induction shows that this takes time $O(pos - start + 1)$.

Now the segment $T[\max(start, pos), end]$ is contained in $Z[p+1]$ and it finishes one symbol before the phrase ends. Thus a copy of it finishes where $Z[source[p+1]]$ ends, so induction applies also to the recursive call at line 13, which extracts the remaining string from the source instead of from $Z[p+1]$, also in constant time per extracted symbol. □

We have shown that the algorithm extracts any substring by starting from the end of a phrase. Thus, extracting an arbitrary substring may be more expensive than an end-of-phrase aligned one.

Definition 4.5. Let $T = Z[1]Z[2]\ldots Z[n']$ be a LZ-parsing of $T[1,n]$. Then the *height* of the parsing is defined as $H = \max_{1\leq i\leq n} C[i]$, where C is defined as follows. Let $Z[i] = T[a,b]$ be a phrase which source is $T[c,d]$, then

$$\begin{aligned} C[k] &= C[(k-a)+c]+1, \forall a \leq k < b \\ C[b] &= 1 \end{aligned}$$

Array C counts how many times a character was transitively copied from its original source. This is also the extraction cost of that character. Hence, the value H is the worst-case bound for extracting a single character in the LZ parse.

Lemma 4.6. *In an LZ-End parsing it holds that H is smaller than the longest phrase, i.e., $H \leq \max_{1\leq p\leq n'} |Z[p]|$.*

Proof. We will prove by induction that $\forall 1 \leq i < n$, $C[i] \leq C[i+1]+1$. From this inequality the lemma follows. For all positions i_p where a phrase p ends, it holds by definition that $C[i_p] = 1$. Thus, for all positions i in the phrase p, we have $C[i] \leq C[i_p] + i_p - i \leq |Z[p]|$.

The first phrase of any LZ-End parsing is $T[0]$, and the second is either $T[1]$ or $T[1]T[2]$. In the first case, we have $C[1]C[2] = 1,1$, in the latter $C[1]C[2]C[3] = 1,2,1$. In both cases the property holds. Now, suppose the inequality is valid up to position i_p where the phrase $Z[p]$ ends. Let i_{p+1} be the position where the phrase $Z[p+1] = T[a,b]$ ends (so $a = i_p + 1$ and $b = i_{p+1}$) and $T[c,d]$ its source. For all $i_p + 1 \leq i < i_{p+1}$, $C[i] = C[(i-a)+c]+1$, and since $d \leq i_p$, the inequality holds by inductive hypothesis for $i_p + 1 \leq i \leq i_{p+1} - 2$. By definition of the LZ-End parsing the source of a phrase ends in a previous end of phrase, hence $C[i_{p+1}-1] = C[d]+1 = 2 \leq 1+1 = C[i_{p+1}]+1$. For position i_{p+1} (end of phrase) the inequality trivially holds as it has by definition the least possible value. □

The above lemma does not hold for LZ77. Moreover, the LZ-End parsing yields a better extraction complexity.

Lemma 4.7. *Extracting a substring of length ℓ from an LZ-End parsing takes time $O(\ell + H)$.*

Proof. Theorem 4.4 already shows that the cost to extract a substring ending at a phrase boundary is constant per extracted symbol. The only piece of the code in Figure 4.1 that does not amortize in this sense is line 13, where we recursively unroll the last phrase, removing the last character each time, until hitting the end of the substring to extract. By definition of H, this line cannot be executed more than H times. So the total time is $O(\ell + H)$. □

Remark 4.8. On a text coming from an ergodic Markov source of entropy h, the expected value of the longest phrase is $O(\frac{\log n}{h})$. However, as we are dealing with highly repetitive texts this expected length does not hold.

Remark 4.9. Algorithm Extract (Figure 4.1) also works on parsing LZ77, but in this case the best theoretical bound we can prove for extracting a substring of length ℓ is $O(\ell H)$. However, the results in Section 4.5.3 suggest that on average it may be much better.

4.3 Compression Performance

We study now the compression performance of LZ-End, first with respect to the empirical k-th order entropy and then on repetitive texts.

4.3.1 Coarse Optimality

We prove that LZ-End is coarsely optimal. The main tool is the following lemma.

Lemma 4.10. *All the phrases generated by an LZ-End parse are different.*

Proof. Assume by contradiction $Z[p] = Z[p']$ for some $p < p'$. When $Z[p']$ was generated, we could have taken $Z[p]$ as the source, yielding phrase $Z[p']c$, longer than $Z[p']$. This is clearly a valid source as $Z[p]$ is a suffix of $Z[1] \ldots Z[p]$. So this is not an LZ-End parse. □

Lemma 4.11 ([LZ76])**.** *Any parsing of $T[1, n]$ into n' distinct phrases satisfies $n' = O\left(\frac{n}{\log_\sigma n}\right)$, where σ is the alphabet size of T.*

Lemma 4.12 ([KM99])**.** *For any text $T[1, n]$ parsed into n' different phrases, it holds $n' \log n' \leq nH_k(T) + n' \log \frac{n}{n'} + \Theta(n'(1 + k \log \sigma))$, for any k.*

Lemma 4.13. *For any text $T[1, n]$ parsed into n' different phrases, using LZ77 or LZ-End, it holds $n' \log n \leq nH_k(T) + o(n \log \sigma)$ for any $k = o(\log_\sigma n)$.*

Proof. Arroyuelo and Navarro [AN] prove that the property holds for any LZ parsing for which Lemmas 4.11 and 4.12 hold. In particular it holds for LZ77 and for our proposal, the LZ-End. □

Theorem 4.14. *The LZ-End compression is coarsely optimal.*

Proof. The proof is based on the one by Kosaraju and Manzini [KM99] for LZ77. Here we consider in addition our particular encoding (the result holds for triplets (q, ℓ, c) as well). The size of the parsing in bits is

$$\begin{aligned} \textit{LZ-End}(T) &= n'\lceil\log\sigma\rceil + n'\lceil\log n'\rceil + n'\log\frac{n}{n'} + O\left(n' + \frac{n\log\log n}{\log n}\right) \\ &= n'\log n + O\left(n'\log\sigma + \frac{n\log\log n}{\log n}\right). \end{aligned}$$

Thus from Lemmas 4.10 and 4.12 we have

$$\textit{LZ-End}(T) \leq nH_k(T) + 2n'\log\frac{n}{n'} + O\left(n'(k+1)\log\sigma + \frac{n\log\log n}{\log n}\right).$$

Now, by means of Lemma 4.11 and since $n' \log \frac{n}{n'}$ is increasing in n', we get

$$\begin{aligned} \textit{LZ-End}(T) &\leq nH_k(T) + O\left(\frac{n\log\sigma\log\log n}{\log n}\right) + O\left(\frac{n(k+1)\log^2\sigma}{\log n} + \frac{n\log\log n}{\log n}\right) \\ &= nH_k(T) + O\left(\frac{n\log\sigma(\log\log n + (k+1)\log\sigma)}{\log n}\right). \end{aligned}$$

Thus, diving by n and taking k and σ as constants, we get that the compression ratio is

$$\rho(T) \leq H_k(T) + O\left(\frac{n\log\log n}{\log n}\right).$$

□

4.3.2 Performance on Repetitive Texts

We have not found a worst-case bound for the competitiveness of LZ-End compared to LZ77. However, we show, on the negative side, a sequence that produces almost twice the number of phrases when parsed with LZ-End, so LZ-End is at best 2-competitive with LZ77. On the positive side, we show that LZ-End satisfies some of the properties of Lemma 4.1.

Example 4.15. Let $T = \texttt{112} \cdot \texttt{113} \cdot \texttt{214} \cdot \texttt{325} \cdot \texttt{436} \cdot \texttt{547} \cdot \ldots \cdot (\sigma-2)(\sigma-3)\sigma$. The length of the text is $n = 3(\sigma - 1)$. The LZ parsings are:

$$\text{LZ77} \quad \boxed{\texttt{1}}\boxed{\texttt{12}}\boxed{\texttt{113}}\boxed{\texttt{214}}\boxed{\texttt{325}}\boxed{\texttt{436}}\boxed{\texttt{547}}\ldots\boxed{(\sigma-2)(\sigma-3)\sigma}$$

$$\text{LZ-End} \quad \boxed{\texttt{1}}\boxed{\texttt{12}}\boxed{\texttt{11}}\boxed{\texttt{3}}\boxed{\texttt{21}}\boxed{\texttt{4}}\boxed{\texttt{32}}\boxed{\texttt{5}}\boxed{\texttt{43}}\boxed{\texttt{6}}\boxed{\texttt{54}}\boxed{\texttt{7}}\ldots\boxed{(\sigma-2)(\sigma-3)}\boxed{\sigma}$$

The size of LZ77 is $n' = \sigma$ and the size of LZ-End is $n' = 2(\sigma - 1)$.

For LZ-End we can only prove the following lemma regarding the concatenation of texts.

Lemma 4.16. *Given a text T, the following statements hold*

$$H^{LZ-End}(TT) \leq H^{LZ-End}(T) + 2 \tag{4.6}$$
$$H^{LZ-End}(TT\$) \leq H^{LZ-End}(T\$) + 1 \tag{4.7}$$

where $H^{LZ-End}(T)$ is the number of phrases of the LZ-End parsing.

Proof. Assume $H^{LZ-End}(T\$) = n'$ and that the last phrase of the LZ-End parsing of $T\$$ is $\boxed{\$}$. That means the first $n' - 1$ phrases cover the text T. Now, if we have the text $TT\$$, we have that the first $n' - 1$ phrases are the same as for the parsing of T and the last phrase would be $\boxed{T\$}$ (since T ends at the end of the $(n' - 1)$-th phrase), hence inequality for Equation (4.7) holds (actually it holds $H^{LZ-End}(TT\$) = H^{LZ-End}(T\$)$). Now, assume the last phrase of the parsing is $\boxed{A\$}$ for some $A \neq \varepsilon$, and that the prefix of T in the $n' - 1$ first phrases is xX, where x is a character. Therefore the n'-th phrase of the parsing of TT would be at least $\boxed{Ax}$. Then the $(n' + 1)$-th phrase will be $\boxed{XA\$}$, thus equality holds for Equation (4.7). The situation is analogous if the n'-th phrase extends beyond Ax. For Equation (4.6) consider that T is parsed into $n' - 1$ phrases covering the prefix xX and the last phrase is aAb (where x, a and b are characters and $A \neq \varepsilon$ is a string). Then, the $(n' + 1)$-th phrase of TT is at least $\boxed{xXa}$, and thus, the $(n' + 2)$-th phrase is $\boxed{Ab}$. Because there must exist a phrase ending in A for phrase $\boxed{aAb}$ to exist. If, instead, the n'-th phrase is just $\boxed{a}$, then the $(n' + 1)$-th phrase is $\boxed{xXa} = \boxed{T}$. □

```
$\mathcal{F} \leftarrow \{\langle -1, n+1 \rangle\}$
$i \leftarrow 1$, $p \leftarrow 1$
**while** $i \leq n$ **do**
  $[sp, ep] \leftarrow [1, n]$
  $j \leftarrow 0$, $\ell \leftarrow j$
  **while** $i + j \leq n$ **do**
    $[sp, ep] \leftarrow \mathrm{BWS}(sp, ep, T[i+j])$
    $mpos \leftarrow \arg\max_{sp \leq k \leq ep} A[k]$
    **if** $A[mpos] \leq n + 1 - i$ **then**
      **break**
    $j \leftarrow j + 1$
    $\langle q, fpos \rangle \leftarrow \mathrm{Successor}(\mathcal{F}, sp)$
    **if** $fpos \leq ep$ **then**
      $\ell \leftarrow j$
  $\mathrm{Insert}(\mathcal{F}, \langle p, A^{-1}[n+1-(i+\ell)] \rangle)$
  **output** $(q, \ell, T[i+\ell])$
  $i \leftarrow i + \ell + 1$,  $p \leftarrow p + 1$
```

Figure 4.2: LZ-End construction algorithm. $\mathcal{F}$ stores pairs $\langle$phrase identifier, suffix array position$\rangle$ and answers successor queries on the text position. $BWS(sp, ep, c)$ was defined in Section 2.12.

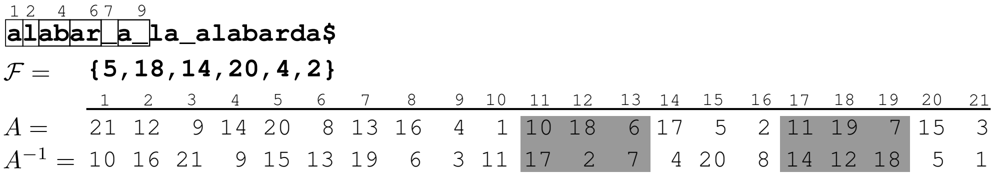
4.4 Construction Algorithm

We present an algorithm to compute the parsing LZ-End, inspired by the algorithm CSP2 by Chen *et al.* [CPS08]. We compute the range of all text prefixes ending with a pattern P, rather than suffixes starting with P [FG89].

We first build the suffix array (Section 2.11) $A[1, n]$ of the *reverse* text, $T^{rev} = T[n-1] \ldots T[2]T[1]\$$, so that $T^{rev}[A[i], n]$ is the lexicographically i-th smallest suffix of T^{rev}. We also build its inverse permutation: $A^{-1}[j]$ is the lexicographic rank of $T^{rev}[j, n]$. Finally, we build the Burrows-Wheeler Transform (BWT) (Section 2.12) of T^{rev}, $T^{bwt}[i] = T^{rev}[A[i] - 1]$ (or $T^{rev}[n]$ if $A[i] = 1$).

On top of the BWT we will apply *backward search* (Section 2.12) to find out whether there are occurrences of a $T[i, i' - 1]$ (Definitions 2.30 and 4.2).

Since, for LZ-End, the phrases must in addition finish at a previous phrase end, we maintain a dynamic set $\mathcal{F}$ where we add the ending positions of the successive phrases we create, mapped to A. That is, once we create phrase $Z[p] = T[i, i']$, we



alabar_a_la_alabarda$

$\mathcal{F} =$ **{5,18,14,20,4,2}**

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
$A =$	21	12	9	14	20	8	13	16	4	1	10	18	6	17	5	2	11	19	7	15	3
$A^{-1} =$	10	16	21	9	15	13	19	6	3	11	17	2	7	4	20	8	14	12	18	5	1

Figure 4.3: Example of LZ-End construction algorithm

insert $A^{-1}[n+1-i']$ into $\mathcal{F}$.

Backward search over T^{rev} adapts very well to our purpose. By considering the patterns $P = (T[i, i'-1])^{rev}$ for consecutive values of i', we are searching backwards for P in T^{rev}, and thus finding the *ending* positions of $T[i, i'-1]$ in T, by carrying out one further BWS step for each new i' value. Thus we can use $\mathcal{F}$ naturally.

As we advance i' in $T[i, i'-1]$, we test whether $A[sp, ep]$ contains some occurrence finishing before i in T, that is, starting after $n+1-i$ in T^{rev}. If it does not, then we stop looking for larger i' values as there are no matches preceding $T[i]$. For this, we precompute a *Range Maximum Query (RMQ)* data structure [FH07] on A, which answers queries $mpos = \arg\max_{sp \leq k \leq ep} A[k]$. Then if $A[mpos]$ is not large enough, we stop.

In addition, we must know if i' finishes at some phrase end, i.e., if $\mathcal{F}$ contains some value in $[sp, ep]$. A *successor* query on $\mathcal{F}$ finds the smallest value $fpos \geq sp$ in $\mathcal{F}$. If $fpos \leq ep$, then it represents a suitable LZ-End source for $T[i, i']$. Otherwise, as the condition could hold again for a later $[sp, ep]$ range, we do not stop but recall the last $j = i'$ where it was valid. Once we stop because no matches ending before $T[i]$ exist, we insert phrase $Z[p] = T[i, j]$ and continue from $i = j+1$. This may retraverse some text since we had processed up to $i' \geq j$. We call $N \geq n$ the total number of text symbols processed.

The algorithm is depicted in Figure 4.2.

Example 4.17. Figure 4.3 shows the structures used during the parsing of the string '`alabar_a_la_alabarda$`'. The array A corresponds to the suffix array of the reversed text and A^{-1} to its inverse permutation. The figure shows the parsing up to the 6th phrase and the values inserted in the dictionary $\mathcal{F}$. The values inserted in $\mathcal{F}$ are $A^{-1}[len - i]$, where i is the ending position of a phrase and len the length of the text. For example, the second phrase ends in position 2, thus the value inserted corresponds to $A^{-1}[21-2] = 18$.

Now we continue the process to generate the next phrase. First, using BWS we find the interval of A that represents the suffixes (of the reverse text) starting with `'l'`, obtaining the range $[17, 18]$ (right gray zone). Then we look in $\mathcal{F}$ for the successor of 17, obtaining the value 18, which is still in the range. Hence, we have found a valid source. Afterward, we continue with the next character. Again, with BWS we find the interval of A representing the suffixes (of the reverse text) starting with `'al'` (left gray zone), which are the prefixes of the text ending with `'la'`. This gives us the range $[11, 13]$. We look for the successor of 11 in $\mathcal{F}$, which is 14. Since this value is outside the interval, there are no valid sources. We continue this process until there are no more possible sources. Finally, we get that the only valid source is `'l'`, generating the new phrase `'la'`.

In theory the construction algorithm can work within bit space (1) $nH_k(T^{rev}) + o(n \log \sigma) = nH_k(T) + o(n \log \sigma)$ (since $nH_k(T) = nH_k(T^{rev}) + O(\log n)$ [FM05, Theorem A.3]) for building the BWT incrementally [GN08]; plus (2) $2n + o(n)$ bits for the RMQ structure [FH07]; plus (3) $O(n' \log n)$ bits for a successor data structure. After building the BWT incrementally in time $O(n \log n \lceil \frac{\log \sigma}{\log \log n} \rceil)$ [GN08], we can make it static, so that it supports access to the successive characters of T in time $O(\lceil \frac{\log \sigma}{\log \log n} \rceil)$, as well to A and A^{-1} in time $O(\log^{1+\epsilon} n)$ for any constant $\epsilon > 0$ [FMMN07]. The RMQ structure is built in $O(n)$ time and within the same final space, and answers queries in constant time. The successor data structure could be a simple balanced search tree, with leaves holding $\Theta(\log n)$ elements, so that the access time is $O(\log n)$ and the space is $n' \log n(1 + o(1))$ [Mun86]. Thus, using Lemmas 4.11 and 4.12, the overall construction space is $2n(H_k(T) + 1) + o(n \log \sigma)$ bits, for any $k = o(\log_\sigma n)$. The time is dominated by the BWT construction, $O(n \log n \lceil \frac{\log \sigma}{\log \log n} \rceil)$, plus the N accesses to A, $O(N \log^{1+\epsilon} n)$. If, instead, we use $O(n \log n)$ bits of space, we can build and store explicitly A and A^{-1} in $O(n)$ time [KS03]. The overall time becomes $O(N \lceil \frac{\log \sigma}{\log \log n} \rceil)$.

Note that a simplification of our construction algorithm, disregarding $\mathcal{F}$ (and thus $N = n$) builds the LZ77 parsing using just $n(H_k(T) + 2) + o(n \log \sigma)$ bits and $O(n \log n(\log^\epsilon n + o(\log \sigma)))$ time, which is less than the best existing solutions [OS08, CPS08].

In practice our implementation of the algorithm works within *byte* space (1) n as we maintain T explicitly; plus (2) $2.02n$ for our implementation of BWT (following Navarro's "large" FM-index implementation [Nav09], where L is maintained explicitly); plus (3) $4n$ for A, which is explicitly maintained; plus (4) $0.7n$ for Fischer's implementation of RMQ [FH07]; plus (5) n for A^{-1}, using a sampling-based implementation of inverse permutations [MRRR03] (Section 2.7); plus (6) $12n'$ for a balanced binary tree implementing the successor structure. This adds up to less than $10n$ bytes in practice. A is built in time $O(n \log n)$ in practice; other construction times are $O(n)$. After this, the time of the algorithm is $O(N \log n') = O(N \log n)$.

As we see soon, N is usually (but not always) only slightly larger than n; we now prove it is limited by the phrase lengths.

Lemma 4.18. *The amount of text retraversed at any step is* $< |Z[p]|$ *for some* p.

Proof. Say the last valid match $T[i, j-1]$ was with suffix $Z[1] \ldots Z[p-1]$ for some p, thereafter we worked until $T[i, i'-1]$ without finding any other valid match, and then formed the phrase (with source $p-1$). Then we will retraverse $T[j+1, i'-1]$, which must be shorter than $Z[p]$ since otherwise $Z[1] \ldots Z[p]$ would have been a valid match. □

Remark 4.19. On ergodic sources with entropy h, $N = O(\frac{n}{h} \log n)$, but as explained this is not a realistic model on repetitive texts.

4.5 Experimental Results

We implemented two different LZ-End encoding schemes. The first is as explained in Section 4.2.1. In the second (*LZ-End2*) we store the starting position of the source, $select_1(B, source[p])$, rather than the identifier of the source, $source[p]$. This in theory raises the $nH_k(T)$ term in the space to $2nH_k(T)$ (and noticeably in practice, as seen soon), yet we save one *select* operation at extraction time (line 13 in Figure 4.1), which has a significant impact in performance. In both implementations, bitmap B is represented by δ-encoding the consecutive phrase lengths (Section 2.5.2). Recall that, in a δ-encoded bitmap, $select_1(B, p)$ and $select_1(B, p+1)$ cost $O(1)$ after solving $p \leftarrow rank_1(B, end)$, thus LZ-End2 does no *select* operations for extracting.

We compare our compressors with *LZ77* and *LZ78* implemented by ourselves. LZ77 triples are encoded in the same way as LZ-End2. We include the best performing compressors of Chapter 3, p7zip and Re-Pair. Compared to p7zip, LZ77 differs in the final encoding of the triples, which p7zip does better. This is orthogonal to the parsing issue we focus on in this thesis. We also implemented *LZB* [Ban09], which limits the distance *dist* at which the phrases can be from their original (not transitive) sources, so one can decompress any window by starting from that distance behind; and *LZ-Cost*, a novel proposal where we limit the number of times any text character can be copied (i.e., its $C[\cdot]$ value in Definition 4.5), thus directly limiting the maximum cost per character extraction. We have found no efficient parsing algorithm for *LZB* and *LZ-Cost*, thus we test them on small texts only. We also implemented *LZ-Begin*, the “symmetric” variant of LZ-End, which also allows random phrase extraction in constant time per extracted symbol. LZ-Begin forces the source of a phrase to start where some previous phrase starts, just like Fiala and Green [FG89], yet phrases have a leading rather than a trailing character. Although the parsing is much simpler, the compression ratio is noticeably worse than that of LZ-End, as we will see in Section 4.5.1.

We used the texts of the Canterbury corpus (`http://corpus.canterbury.ac.nz`), the 50MB texts from the Pizza&Chili corpus (`http://pizzachili.dcc.uchile.cl`), and highly repetitive texts from the previous chapter. We use a 3.0 GHz Core 2 Duo processor with 4GB of main memory, running Linux 2.6.24 and g++ (gcc version 4.2.4) compiler with -O3 optimization.

4.5.1 Compression Ratio

Table 4.1 gives compression ratios for the different collections and parsers. Figure 4.4 shows the same results graphically for one representative text of each collection. For LZ-End we omit the sampling for bitmap B, as it can be reconstructed on the fly at loading time. LZ-End is usually 5% worse than LZ77, and at most 10% over it on general texts and 20% on the highly repetitive collections, where the compression ratios are nevertheless excellent. LZ78 is from 20% better to 25% worse than LZ-End on typical texts, but it is orders of magnitude worse on highly repetitive collections. With parameter $\log(n)/2$, LZ-Cost is usually close to LZ77, yet sometimes it is much worse, and it is never better than LZ-End except by negligible margins. LZB is not competitive at all. Finally, LZ-Begin is about 30% worse than LZ77 on typical texts, and up to 40 times worse for repetitive texts. This is because not all phrases of the parsing are unique (Lemma 4.20). This property was the key to prove the coarse optimality of the LZ parsings.

Lemma 4.20. *Not all the phrases generated by an LZ-Begin parsing are different.*

Proof. We prove this lemma by showing a counter-example. Let $T = A\mathtt{xy}A\mathtt{y}A\mathtt{z}$, where $\mathtt{x}, \mathtt{y}, \mathtt{z}$ are distinct characters and A is a string. Suppose we have parsed up to $A\mathtt{x}$, then the next phrase will be $\mathtt{y}A$, and the following phrase will also be $\mathtt{y}A$. □

The above results show that LZ-End achieves very competitive compression ratios, even in the challenging case of highly repetitive sequences, where LZ77 excells.

		LZ77	**LZ78**	**LZ-End**	**LZ-Cost**	**LZB**	**LZ-Begin**	**Re-Pair**
Canterbury	Size(KiB)							
alice29.txt	148.52	47.17%	49.91%	49.32%	48.51%	61.75%	59.02%	72.29%
asyoulik.txt	122.25	51.71%	52.95%	53.51%	52.41%	66.42%	62.34%	81.52%
cp.html	24.03	43.61%	53.60%	45.53%	46.27%	66.26%	56.93%	78.65%
fields.c	10.89	39.21%	54.73%	41.69%	44.44%	61.32%	60.61%	65.19%
grammar.lsp	3.63	48.48%	57.85%	50.41%	56.30 %	67.02%	67.14%	85.60%
lcet10.txt	416.75	42.62%	46.83%	44.65%	43.44%	56.72%	54.21%	57.47%
plrabn12.txt	470.57	50.21%	49.34%	52.06%	50.83%	63.55%	59.15%	74.32%
xargs.1	4.13	57.87%	65.38%	59.56%	59.45%	86.37%	73.14%	107.33%
aaa.txt	97.66	0.055%	0.51%	0.045%	1.56%	0.95%	0.040%	0.045%
alphabet.txt	97.66	0.110%	4.31%	0.105%	0.23%	1.15%	0.100%	0.081%
random.txt	97.66	107.39%	90.10%	105.43%	107.40%	121.11%	106.9%	219.24%
E.coli	4529.97	34.13%	27.70%	34.72%	-	-	35.99%	57.63%
bible.txt	3952.53	34.18%	36.27%	36.44%	-	-	43.98%	41.81%
world192.txt	2415.43	29.04%	38.52%	30.99%	-	-	41.52%	38.29%
pi.txt	976.56	55.73%	47.13%	55.99%	-	-	57.36%	108.08%

		LZ77	**LZ78**	**LZ-End**	**LZ-Begin**	**Re-Pair**
Pizza Chili	Size(MiB)					
Sources	50	28.50%	41.14%	31.00%	41.95%	31.07%
Pitches	50	44.50%	59.30%	45.78%	57.22%	59.90%
Proteins	50	47.80%	53.20%	47.84%	54.95%	71.29%
DNA	50	31.88%	28.12%	32.76%	34.28%	45.90%
English	50	31.12%	41.80%	31.12%	38.54%	30.50%
XML	50	17.00%	21.24%	17.64%	25.49%	18.50%

		LZ77	**LZ78**	**LZ-End**	**LZ-Begin**	**Re-Pair**
Repetitive	Size(MiB)					
Wikipedia Einstein	357.40	9.97×10^{-2}%	9.29%	1.01×10^{-1}%	4.27%	1.04×10^{-1}%
World Leaders	40.65	1.73%	15.89%	1.93%	7.97%	1.89%
Rich String 11	48.80	3.20×10^{-4}%	0.82%	4.18×10^{-4}%	0.01%	3.75×10^{-4}%
Fibonacci 42	255.50	7.32×10^{-5}%	0.40%	5.32×10^{-5}%	6.07×10^{-5}%	2.13×10^{-5}%
Para	409.38	2.09%	25.49%	2.48%	7.29%	2.74%
Cere	439.92	1.48%	25.33%	1.74%	6.15%	1.86%
Coreutils	195.77	3.18%	27.57%	3.35%	7.33%	2.54%
Kernel	246.01	1.35%	30.02%	1.43%	3.43%	1.10%

Table 4.1: Compression ratio of different parsings, in percentage of compressed over original size. We use parameter $cost = (\log n)/2$ for *LZ-Cost* and $dist = n/5$ for *LZB*.

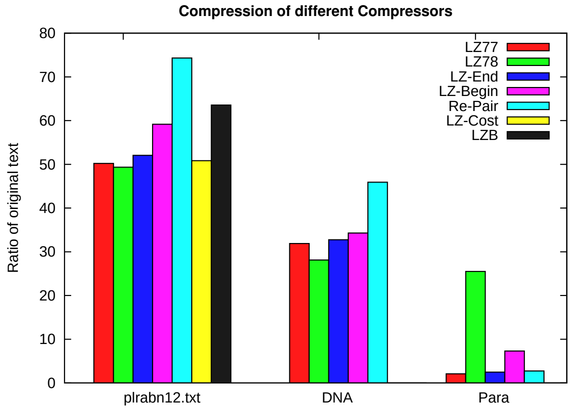


Figure 4.4: Compression ratio for different compressors

Consistently with Chapter 3, Re-Pair results show that grammar-based compression is a relevant alternative. Yet, we note that it is only competitive on highly repetitive sequences, where most of the compressed data is in the dictionary. This implementation applies sophisticated compression to the dictionary, which we do not apply on our compressors. Those sophisticated dictionary compression techniques prevent direct access to the grammar rules, essential for extracting substrings.

4.5.2 Parsing Time

Figure 4.5 shows parsing times on two files for LZ77 (implemented following CSP2 [CPS08]), LZ-End with the algorithm of Section 4.4, and p7zip. We show separately the time of the suffix array construction algorithm we use, `libdivsufsort` (`http://code.google.com/p/libdivsufsort`), common to LZ77 and LZ-End.

Our LZ77 construction time is competitive with the state of the art (p7zip), thus the excess of LZ-End is due to the more complex parsing. Least squares fitting for the nanoseconds/char yields $10.4 \log n + O(1/n)$ (LZ77) and $82.3 \log n + O(1/n)$ (LZ-End) for Einstein text, and $32.6 \log n + O(1/n)$ (LZ77) and $127.9 \log n + O(1/n)$ (LZ-End) for XML. The correlation coefficient is always over 0.999, which suggests that $N = O(n)$ and our parsing takes $O(n \log n)$ time in practice. Indeed, across all of our collections, the ratio N/n stays between 1.05 and 1.37, except on `aaa.txt` and `alphabet.txt`, where it is 10–14 (which suggests that $N = \omega(n)$ in the worst case). Figure 4.6 shows the total text traversed by LZ-End parsing algorithm for two different texts.

The LZ-End parsing time breaks down as follows. For XML: BWS 36%, RMQ

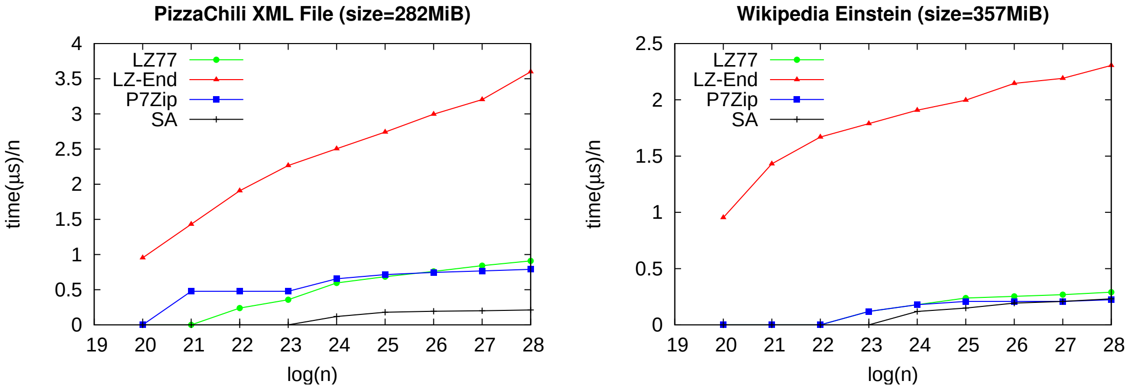


Figure 4.5: Parsing times for XML and Wikipedia Einstein, in microseconds per character.

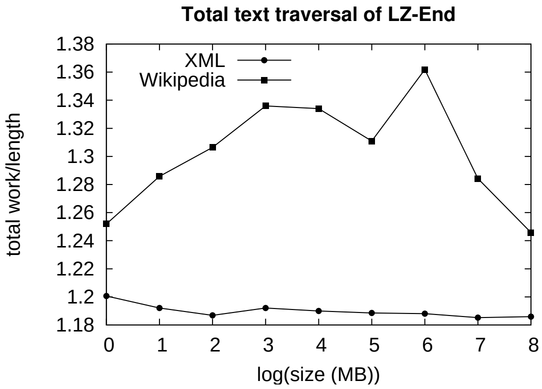


Figure 4.6: Total text traversed during LZ-End construction algorithm.

19%, tree operations 33%, SA construction 6% and inverse SA lookups 6%. For Einstein: BWS 56%, RMQ 19%, tree operations 17%, and SA construction 8% (the inverse SA lookups take negligible time).

4.5.3 Text Extraction Speed

Figure 4.7 shows the extraction speed of *arbitrary* substrings of increasing length. The three parsings (LZ77, LZ-End and LZ-End2) are parameterized to use approximately the same space, 550KiB for Wikipedia Einstein and 64MiB for XML. This is achieved by adjusting the sample step s of the δ-encoded bitmaps (Section 2.5.2). It can be seen that (1) the time per character stabilizes after some extracted length, as expected from Lemma 4.7, (2) LZ-End variants extract faster than LZ77, especially on very repetitive collections, and (3) LZ-End2 is faster than LZ-End, even if the latter invests its better compression in a denser sampling. Least squares fitting for the extraction time of a substring of length m are given in Table 4.2.

Pizza&Chili XML

Scheme	Model
LZ77	$4.44 + 0.33m$
LZ-End	$7.40 + 0.36m$
LZ-End2	$6.41 + 0.27m$

Wikipedia Einstein

Scheme	Model
LZ77	$19.09 + 0.38m$
LZ-End	$5.75 + 0.19m$
LZ-End2	$5.64 + 0.19m$

Table 4.2: Least squares fitting for extraction time. All correlation coefficients are always over 0.999.

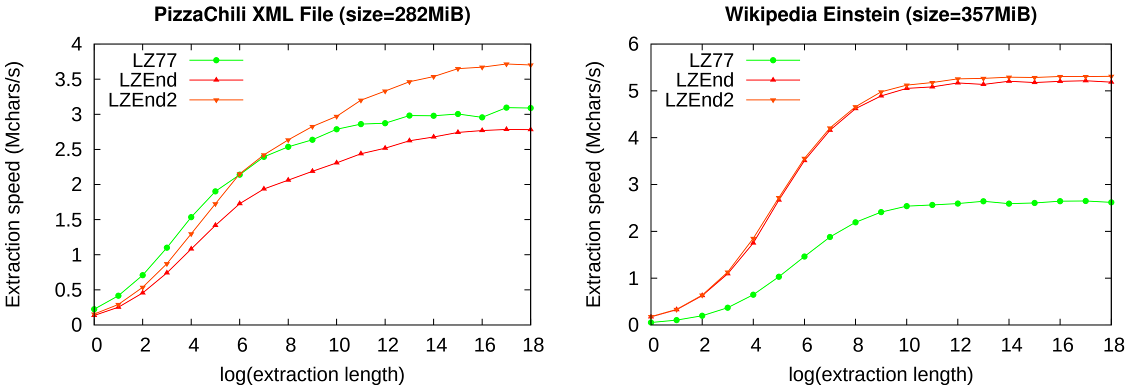


Figure 4.7: Extraction speed vs extracted length, for XML and Wikipedia Einstein.

We now set the extraction length to 1,000 and measure the extraction speed per character, as a function of the space used by the data and the sampling. Here we use bitmap B and its sampling for the other formats as well. LZB and LZ-Cost have also their own space/time trade-off parameter; we tried several combinations and chose the points dominating the others. Figure 4.8 shows the results for small and large files.

It can be seen that LZB is not competitive, whereas LZ-Cost follows LZ77 closely (while offering a worst-case guarantee). The LZ-End variants dominate all the trade-off except when LZ77/LZ-Cost are able of using less space. On repetitive collections, LZ-End2 is more than 2.5 times faster than LZ77 at extraction.

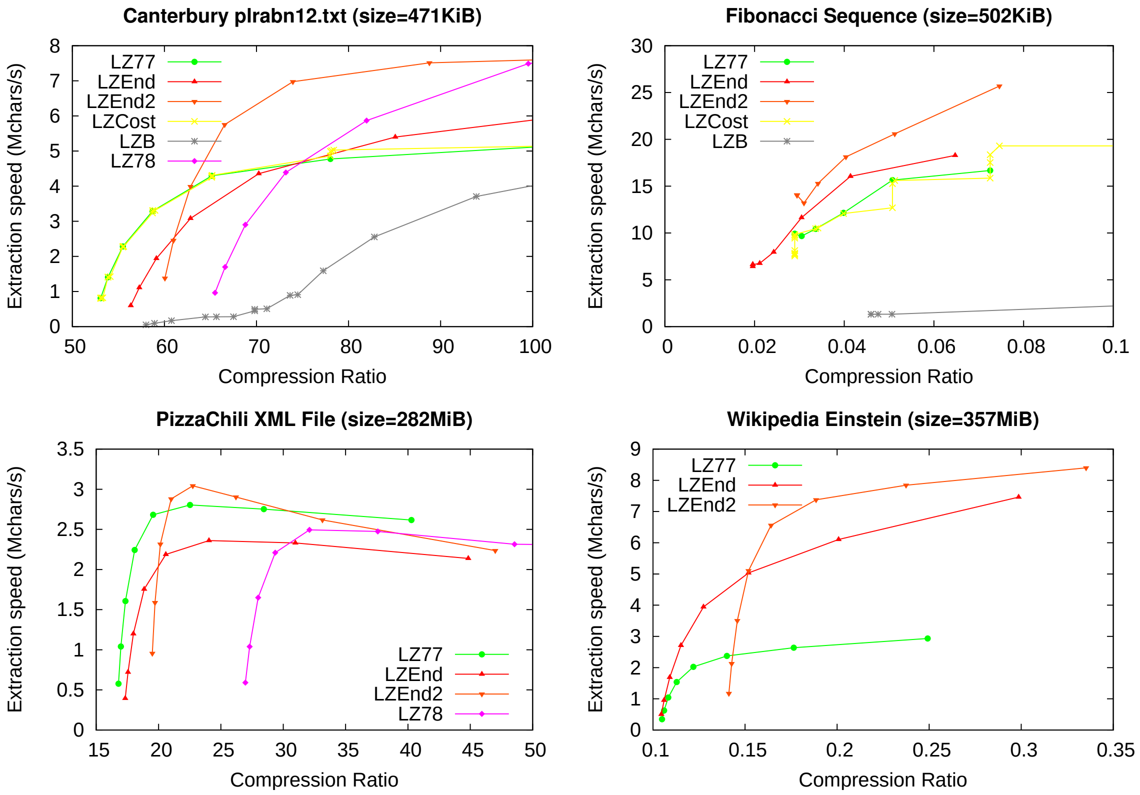


Figure 4.8: Extraction speed vs parsing and sampling size, on different texts.

Chapter 5

An LZ77-Based Self-Index

In this chapter we describe a self-index based on the LZ77 parsing. It builds on the ideas of the original LZ-based index proposed by Kärkkäinen and Ukkonen [KU96a, Kär99] and the ideas presented by Navarro for reducing its space usage [Nav08]. Our index will be mostly independent of the type of Lempel-Ziv parsing used, and we will combine it with LZ77 and LZ-End. We use compact data structures to achieve the minimum possible space. These structures also allow one to convert the original index into a self-index, so that we do not need the text anymore.

As we will show, the index includes all the structures needed to randomly extract any substring from the text, introduced in the previous chapter. The worst-case time to extract a substring of length ℓ is $O(\ell H)$ for LZ77 and $O(\ell + H)$ for LZ-End (see Section 4.2.2). Additionally, the proposed index only supports *count* queries by performing a full *locate*, and *exists* queries by essentially locating one occurrence. For these reasons, in the following we focus only on *locate* queries.

5.1 Basic Definitions

Assume we have a text T of length n, which is partitioned into n' phrases using a LZ77-like compressor (see Chapter 4). Let $P[1, m]$ be a search pattern. We will call *primary occurrences* of P those covering more than one phrase; *special primary occurrences* those ending at the end of a phrase and being completely covered by the phrase; and *secondary occurrences* those occurrences completely covered by a phrase and not ending at an end of phrase.

Example 5.1.

1	2	3 4	5 6	7	8 9	10 11 12	13 14 15 16 17 18 19	20 21
a	l	ab	ar	_	a_	la_	alabard	a$

In this example the occurrence of 'lab' starting at position 2 (red color) is primary as it spans two phrases. The second occurrence, starting at position 14 (blue color) is secondary. The occurrence of 'rd' starting at position 18 (green color) is special primary.

We need to distinguish between these three types of occurrences, as we will find first the primary occurrences (including the special ones), which will be then used to recursively find the secondary ones (which, in turn, will be used to find further secondary occurrences).

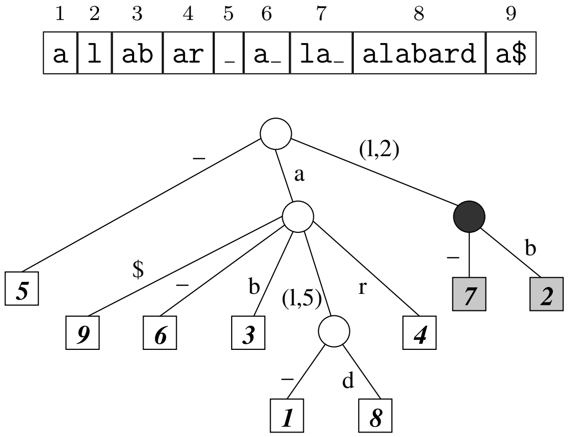
5.2 Primary Occurrences

By definition, a primary occurrence covers at least two phrases. Thus, each primary occurrence can be seen as $P = LR$, where the left side L is a suffix of a phrase and the right side R is the concatenation of zero or more consecutive phrases plus a prefix of the next phrase. For this reason, to find this type of occurrences we partition the pattern in two (in every possible way). Then, we search for the occurrences of the left part of the pattern in the suffixes of the phrases and for the right part in the prefixes of the suffixes of the text starting at beginning of phrases. Then, we need to find which pairs of left and right occurrences actually represent an occurrence of pattern P:

1. Partition the pattern $P[1, m]$ into $P[1, i]$ and $P[i + 1, m]$ for each $1 \leq i < m$.
2. Search for the right part $P[i + 1, m]$ in the prefixes of the suffixes of the text starting at phrases.
3. Search for the left part $P[1, i]$ in suffixes of phrases.
4. Connect both results, generating all primary occurrences.

5.2.1 Right Part of the Pattern

To find the right side $P[i + 1, m]$ of the pattern we use a suffix trie (recall Sections 2.9 and 2.14.2) that indexes all suffixes of T starting at the beginning of a phrase. In the leaves of the tree we store the identifier (id) of the phrases. Conceptually, these form an array *id* that stores the phrase ids in lexicographic order (i.e., the leaves of the suffix trie). As we see later, we do not need to store *id* explicitly.



```
dfuds   (((())((((())))(())))(()))
chars   _a$_bl_drl_b
id      5,9,6,3,1,8,4,7,2
```

Figure 5.1: The suffix trie for the string ‘alabar_a_la_alabarda$’. The dark node is the note at which we stop searching for the pattern ‘la’, and the gray leaves represent the phrases that start with that pattern.

We will represent the suffix trie as a labeled tree using DFUDS (Section 2.8). To search for a pattern we descend through the tree using *labeled_child* (recall Section 2.8), and then discard as many characters of the pattern as the skip of the branch indicates. We continue this process either until we reach a leaf, the pattern is completely consumed, or we cannot descend anymore. Our answer will be an interval of the array of ids, representing all phrases starting with the pattern $P[i+1, m]$. In case we consume the pattern in an internal node, we need to go to the leftmost and rightmost leaves in order to obtain the interval, which is computed using $leaf\_rank$ and represents the start and end positions in the array of the ids.

Example 5.2. Suppose we are looking for the right pattern ‘la’. Figure 5.1 shows in dark the node at which we stop searching for the pattern, and in gray the phrases that start with that pattern. The answer is the range $[8, 9]$ (i.e., the lexicographical order of the phrases).

Remark 5.3. Recall from Section 2.9 that in a PATRICIA tree, after searching for the positions we need to check if they are actually a match, as some characters are not checked because of the skips. In the example presented above, the answer would have been the same if we were searching for any right pattern of the form `l`x, where x is a character distinct from `a`. We use a different method here, which is explained in Section 5.2.3.

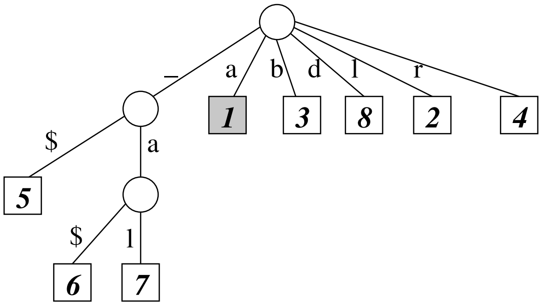
We do not explicitly store the skips in our theoretical proposal, as they can be computed from the tree and the text. Given a node in the trie, if we go to the leftmost and rightmost leaves, we can extract the corresponding suffixes until computing how many characters they share. This value will be the sum of all the skips from the root to the given node. However, we already know they share S characters, where S is the sum of all skips from the root to the previous node (i.e., the parent node). Therefore, to compute the skip, we extract the suffixes of both leaves skipping the first S characters. The amount of symbols shared by both extracted strings will be the skip. Extracting a skip of length s will take at most $O(sH)$ time both for LZ77 and for LZ-End, since the extraction is from left to right and we have to extract one character at a time until they differ. Thus, the total time for extracting the skips as we descend is $O(mH)$.

5.2.2 Left Part of the Pattern

To find the left part $P[1, i]$ of the pattern we have a trie (actually a PATRICIA trie, Section 2.9) that indexes all the reversed phrases, stored as a compact labeled tree (Section 2.8). Thus to find the left part of the pattern in the text we need to search for $(P[1, i])^{rev}$ in this trie. The array that stores the leaves of the trie is called *rev_id* and is stored explicitly.

The search process and the considerations for this tree are exactly the same as the ones for Section 5.2.1. The only difference with the suffix trie is that the computation of the skips is faster now. Our text extraction algorithm works from right to left and since the text is reversed our algorithm outputs the characters in the correct order. Thus, extracting a skip of length s takes $O(sH)$ time for LZ77 and $O(s + H)$ time for LZ-End. However, in the worst case the total time would still be $O(mH)$ as all skips may be of length 1.

Example 5.4. Suppose we are looking for the left pattern `'a'`. Figure 5.2 shows in gray the node at which we stop searching for the pattern. In this case we end up in a leaf, so that is the only phrase that ends with the given pattern. The answer is the range $[4, 4]$.



```
dfuds    ((((((()(())(())))))))) 
chars    _$a$labdlr
rev_id   5,6,7,1,3,8,2,4
```

Figure 5.2: The reverse trie for the string 'alabar_a_la_alabarda$'. The gray leaf is the node at which we stop searching for the pattern 'a'.

5.2.3 Connecting Both Parts

In the previous steps we found two intervals, one in the *id* array and the other in the *rev_id* array. These intervals represent the sets of phrases where the matches of the right side of the pattern start (*id* array interval) and the phrases ending with the left side of the pattern (*rev_id* array interval). Actual occurrences of the pattern are composed of consecutive phrases. Hence, to find the occurrences of the pattern, we need to find which ids in the right interval are consecutive to those rev_ids in the left interval. For doing so we use a range structure (see Section 2.6.1) that connects the consecutive phrases in both trees. Figure 5.3 shows the range data structure connecting both trees for our example string and below the sequence that is represented with the wavelet tree.

This structure is built from a permutation π on $[1, \ldots, n']$. This permutation is just an array containing for each id (column) the corresponding rev_id (row). In other words, the permutation holds that $id[i] = 1 + rev\_id[\pi(i)]$. For our example the permutation array would be {8,6,1,7,0,3,5,2,4} (note that we count from left to right and from bottom to top, and that we assume that $rev\_id[0] = 0$).

Example 5.5. Suppose we are looking for the pattern 'ala'. The possible partitions are (a, la) and (al, a). Figure 5.3 shows in gray the ranges obtained when searching for the left and right part of partition (a, la). Then we look for all points inside those ranges, obtaining the only primary occurrence that starts at phrase 1. The same procedure is carried out for the other partition.

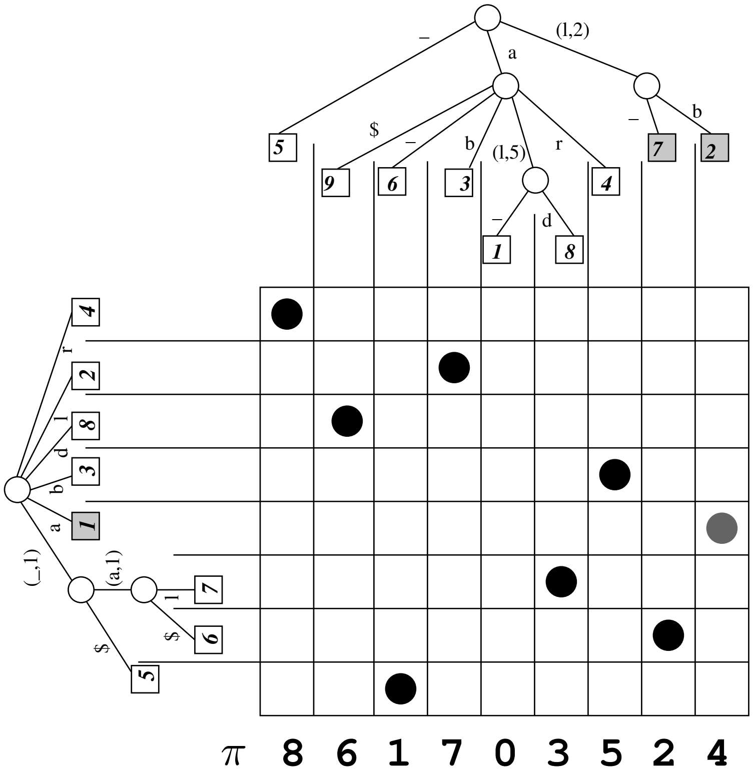


Figure 5.3: The range structure for the string `‘alabar_a_la_alabarda$’`. The gray circle marks the only primary occurrence of the pattern `‘ala’`, and the gray nodes show the ranges defined by the left and right part of the pattern.

Remark 5.6. The range structure allows us to compute $id[i]$, just storing the rev_id array. Say we want to compute $id[i]$. We extract the value $S[i]$ from the wavelet tree, giving us the row p where the corresponding reverse id is. Then we compute $id[i] = 1 + rev\_id[p]$.

Example 5.7. Say we want to compute $id[6]$ (i.e., the phrase id of the 6th lexicographical smallest phrase). We extract from the wavelet tree the 6th symbol, getting the value 3. This value is the lexicographical order of the reversed 5th phrase. Computing $rev\_id[3] = 7$, we know that the 5th phrase is phrase number 7. Hence, the 6th phrase is phrase number 8 (i.e., $id[6] = 8$).

At this stage we also have to validate that the answers returned by the search query are actual occurrences, as the PATRICIA tries by themselves do not guarantee the pattern found is actually a match (see Remark 5.3). For the first occurrence reported by the range data structure we extract the substring of length m starting at the reported position and check if it matches the pattern. If so we can ensure that all the other reported occurrences match the pattern as well, otherwise no occurrence is a match. This process works because all occurrences reported by both tries share all characters, thus all occurrences reported by the range query share all characters. We check the validity of the occurrences here as the range check is cheaper than extracting text and we want to extract text only when a candidate to complete occurrence is found.

This structure adds $O(\log n')$ time to the search phase, and $O(\log n')$ time per primary occurrence found.

Note that we are able to answer *exists* queries with the structures explained so far. If the number of occurrences reported by the range search is greater than one, then we check if one of those queries is an actual match. If there is a match, then the pattern is present in the text.

5.2.4 Special Primary Occurrences

The special primary occurrences could be found using the same steps explained above for primary occurrences, taking the left part of the pattern as the pattern itself and the right side of the pattern as the empty string ε. However, we do know that looking for ε in the suffix trie will return the complete tree, thus making the search in the range structure unnecessary. For this reason we call this type of occurrence special primary, as we search for them slightly differently from the primary ones. For these occurrences we just need to search for P^{rev} in the reverse trie.

Since the search P^{rev} in the reverse trie gives us a range in the rev_id array, we decided to store it explicitly instead of the id array. Furthermore, the result of the range search gives us positions in the rev_id array.

5.2.5 Converting Phrase Ids to Text Positions

From the range structure we obtain the phrase id where an occurrence lies. Then we need to convert it to a real text position. For doing so, we use a bitmap that marks the ends of phrases. This bitmap is the same B used in Chapter 4 for extracting text. Figure 5.4 shows the bitmap for the example string. The bitmap is below the parsing.

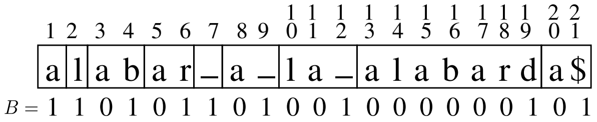

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
a l a b a r _ a _ l a _ a l a b a r d a $
B = 1 1 0 1 0 1 1 0 1 0 0 1 0 0 0 0 0 0 1 0 1

Figure 5.4: The bitmap B of phrases for the string `'alabar_a_la_alabarda'`

The conversion between phrase ids and text positions takes constant time as follows:

- $phrase(pos) = 1 + rank_1(B, pos - 1)$: it gives the phrase id containing any text position pos.
- $first\_pos(id) = select_1(B, id - 1) + 1$: position of the first character of phrase id.
- $last\_pos(id) = select_1(B, id)$: position of the trailing character of phrase id.

Recall from Section 4.2.1 that this bitmap also allows us to compute the length of the phrase as $length(id) = select_1(B, id + 1) - select_1(B, id)$.

5.2.6 Implementation Considerations

Here we explain some considerations we made when implementing our index.

- **Skips:** as the average value for the skips is usually very low and computing them from the text phrases is slow in practice, we considered storing the skips, for one or for both tries, using the *Directly Addressable Codes* (Section 2.4.1). Note that in this case we never access array *id* nor *rev_id* during the trie traversal, they are only accessed when checking and reporting the occurrences.
- **Binary Search:** instead of storing the trie we can do a binary search over the ids (rev_ids) of the suffix trie (reverse trie). For the suffix trie, we do not have explicitly the array of ids, but as shown in Remark 5.6 we can retrieve them using the range structure and the rev_ids array. This alternative modifies the complexity of searching for a prefix/suffix of P to $O(mH \log n')$ for LZ77 or $O((m+H) \log n')$ for LZ-End (actually, since we extract the phrases right-to-left, binary search on the reverse trie costs $O(m \log n')$ for LZ-End). Additionally, we could store explicitly the array of ids, instead of accessing them through the rev ids. Although this alternative increases the space usage of the index and does not improve the complexity, it gives an interesting trade-off in practice.

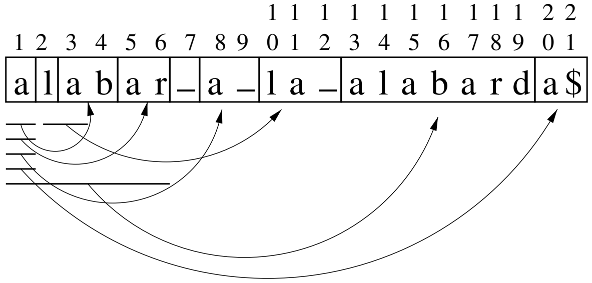
5.3 Secondary Occurrences

Secondary occurrences are found from the primary occurrences and, recursively, from other previously discovered secondary occurrences.

5.3.1 Basic Idea

The idea to find the secondary occurrences is to locate all sources (of the LZ parsing) covering the occurrence and then mapping their corresponding phrases to real text positions. To do this we use another bitmap, called *bitmap of sources* B_S. The bitmap is built by first writing in unary the amount of empty sources (ε) and then for each position of the text writing in unary how many sources start at that position. In this way each `1` corresponds to a source and a `0` represents the position where the sources (`1`s) immediately preceding it start. Figure 5.5 shows the sources and the corresponding phrases they generate (except the empty sources), and below the resulting bitmap. Since there are 3 empty sources the bitmap starts with `1110`, then are 5 sources starting at position 1, hence `111110` follows, then just one source starting at position 2, adding `10`, and finally one `0` for each remaining position.

Additionally, we need a permutation P_S connecting the `1`s in the bitmap B of phrases (recall Section 5.2.5) to the `1`s in the bitmap B_S of sources. The sources starting at a given position are sorted by increasing length, thus the last `1` before a `0` marks the longest source starting at that position. An example is given in Figure 5.6. This permutation replaces the array *source* of Section 4.2.1



$B_S = 1\,1\,1\,0\,1\,1\,1\,1\,1\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0$

Figure 5.5: Marking sources on bitmap B_S

For each occurrence found, we find the position *pos* of the `0` corresponding to its starting position in the bitmap of sources. Then we consider all the `1`s to the left

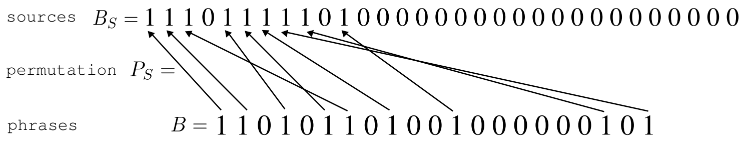

sources B_S = 1 1 1 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
permutation P_S =
phrases B = 1 1 0 1 0 1 1 0 1 0 0 1 0 0 0 0 0 1 0 1

Figure 5.6: Permutation connecting bitmap of phrases B (bottom) and bitmap of sources B_S (top)

of *pos*. We convert each source to its target phrase, compute its length and see if the source covers the occurrence. If so, we report it as a secondary occurrence and recursively generate all secondary occurrences from this new occurrence. In case the source does not cover the occurrence, we stop the process and continue processing the remaining occurrences. The algorithm is depicted in Figure 5.7.

```
secondaryOcc($start, len$)
$pos \leftarrow select_0(B_S, start + 1)$
$source\_id \leftarrow pos - start - 1$
while $source\_id > 0$ do
  $phrase\_id \leftarrow P_S^{-1}(source\_id)$
  $source\_start \leftarrow select_1(B_S, source\_id) - source\_id$
  if $source\_start + len(phrase\_id) \geq start + len$ then
    $occ\_pos \leftarrow first\_pos(source\_id) + start - source\_start$
    report $occ\_pos$
    secondaryOcc($occ\_pos, len$)
  else
    return
  $source\_id \leftarrow source\_id - 1$
```

Figure 5.7: Searching for secondary occurrences from $T[start, start{+}len]$ (preliminary version)

Example 5.8. Consider the only primary occurrence of the pattern `'la'` starting at position 2. We find the third `0` in the bitmap of sources at position 12. Then we consider all ones starting from position 11 to the left. The first `1` at position 11 maps to a phrase of length 2 that covers the occurrence, hence we report an occurrence at position 10. The second `1` maps to a phrase of length 6 that also covers the occurrence, thus we report another occurrence at position 15. The third `1` maps to a phrase of length 1, hence it does not cover the occurrence and we stop. We proceed recursively for the secondary occurrences found at position 10 and 15.

Remark 5.9. The method explained above is just introductory, as it does not work for general LZ77-like parsings. It only works for parsings in which no source strictly contains another source. Is it easy to see that if a source S_2 is strictly covered by another S_1 some secondary occurrences are lost. Let M be a match of the pattern sought and let M be between the rightmost positions of S_2 and S_1. Then, as S_2 is the first source to the left of M, we test if it covers M, stopping the process. However, S_1 does cover M and produces a secondary occurrence, which was not detected by the algorithm presented above.

Example 5.10. Let us start with the primary occurrence of the pattern `'ba'` starting at position 4. The first source to the left is `'la'`, at position 2 and of length 2, which does not cover the pattern. Hence, the algorithm explained above would stop, reporting no secondary occurrences. However, to the left of this source is the source `'alabar'` that does cover the pattern and generates the secondary occurrence starting at position 16.

5.3.2 Complete Solution

Kärkkäinen in his thesis [Kär99] proposes a method for converting the LZ77 parsing into one in which no source contains another. However, we decided not to use it as it increases excessively the number of phrases. Recall that our index will use space proportional to the number of phrases of the parsing, thus any increase in the number of phrases affects directly the final size of the index.

Another proposal of Kärkkäinen is to separate the sources by levels, so that within a level no source strictly contains another, and then apply the method explained in Section 5.3.1 within each level.

Definition 5.11. The *depth* of a source is defined as

$$depth(s) = \begin{cases} 0 & cover(s) = \emptyset \\ 1 + \max_{s' \in cover(s)} depth(s') & otherwise \end{cases},$$

where $cover(s)$ is the set of all sources containing the source s. Let S_1, S_2 be two sources starting at p_1, p_2 and of lengths l_1, l_2. S_1 is said to cover S_2 if $p_1 < p_2$ and $p_1 + l_1 \geq p_2 + l_2$. Note that, by definition, sources starting at the same position are not covered by each other. However, sources ending at the same position may cover each other. For $s = \varepsilon$ we define $depth(\varepsilon) = 0$.

Figure 5.8 shows the additional array storing the depths of each source. The four sources `'a'` and the source `'alabar'` have depth equal to 0 as all of them start at the same position. Source `'la'` has depth 1, as it is contained by the source `'alabar'`.

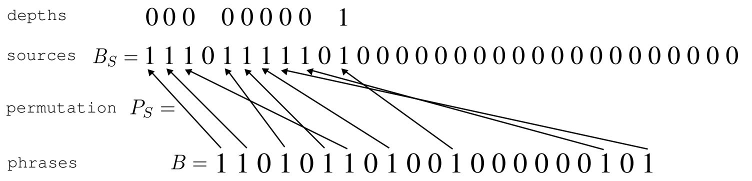

depths
0 0 0 0 0 0 0 0 1
sources $B_S = $ 1 1 1 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
permutation $P_S =$
phrases $B = $ 1 1 0 1 0 1 1 0 1 0 0 1 0 0 0 0 0 0 1 0 1

Figure 5.8: The depth of the sources for the string `alabar_a_la_alabarda$'

The process now is similar to the idea presented earlier; however, now when we find a source not covering the occurrence we look for its depth d and then consider to the left only sources with depth $d' < d$, as those at depth $\geq d$ are guaranteed not to contain the occurrence. This process works because in each level it holds that sources to the left will end earlier than the current source, because of the definition of depth. Moreover, sources at higher depths to the left will also end earlier as they are contained in a source of the current depth to the left.

Now the total running time to find all *occ* secondary occurrences given a seed occurrence is $\Omega(\frac{1}{\varepsilon} occ \cdot L)$ and $O(\frac{1}{\varepsilon} occ \cdot L + D)$, where ε is the parameter for computing the inverse permutation P_S^{-1} (Section 2.7), L is the time to find the next element to the left with depth less than a given value (an operation we consider next), and D is the maximum depth. The additional $O(D)$ cost is because in the worst case after finding the last occurrence we will be in a source of depth D, then move to a source of depth $D - 1$, that does not yield an occurrence, and so on up to a source of depth 1.

5.3.3 Prev-Less Data Structure

As explained above, we need to be able to, given a position *pos* in an array U and a value v, find the rightmost position p preceding *pos* for which it holds $U[p] \leq v$. We will call this query $prevLess(U, p, v)$.

To solve this query we will encode U (i.e., the array of depths of Section 5.3.2) using a wavelet tree (Section 2.6) supporting this additional operation. The algorithm descends according to the bits of value v. If the value v gets mapped to a `0` we recursively search in the left subtree. If the value v gets mapped to a `1` we recursively search in the right subtree. In this case, as the answer could be at the left side, we look for the rightmost `0` preceding *pos* in the bitmap of the wavelet tree node. Finding this `0` takes constant time using *rank* and *select*. Finally, we return the maximum of the value returned by the right subtree and the rightmost zero.

The pseudocode of the algorithm is presented in Figure 5.9. The algorithm receives as parameters a wavelet tree *tree*, a position *pos*, and a value v, and returns $prevLess(array(tree), pos, v)$, where $array(tree)$ are the values represented by the wavelet tree. The bitmap of the wavelet tree is denoted $tree.B$. Function *toBit* returns to which side the value goes, and its output depends on the level.

```
prevLess($tree, pos, v$)
//toBit depends on the level
if $toBit(v, tree) = 0$ then
  $lpos \leftarrow prevLess(tree.left, rank_0(tree.B, pos), v)$
  return $select_0(tree.B, lpos)$
else
  //rightmost zero
  $rm0 \leftarrow select_0(tree.B, rank_0(tree.B, pos))$
  $lpos \leftarrow prevLess(tree.right, rank_1(tree.B, pos), v)$
  return $\max\{rm0, select_1(tree.B, lpos)\}$
```

Figure 5.9: PrevLess algorithm

As the algorithm just performs constant-time operations at each level, its total running time is $L = O(\log D)$.

If, we label each source with its depth, and label the changes to the next text position with a $D + 1$, , we can get rid of the original bitmap of sources. Since the wavelet tree also supports *rank* and *select* queries we have the same functionality as the bitmap of sources, yet with the ability to answer *prevLess* queries. However, as in practice the bitmap of sources is very sparse, we preferred to use a δ-encoded bitmap to represent it and the wavelet tree for the depths.

Using this operation we can now modify algorithm *secondaryOcc* of Figure 5.7. We keep track of the maximum depth d for which there may be sources covering the occurrence. When a source does not cover the occurrence, we update the value of d. Using the operation *prevLess*, we move to the next candidate source. The final algorithm is presented in Figure 5.10.

5.4 Query Time

Combining all the steps gives us the total running time to find the occurrences.

- **Primary Occurrences:** the total time is $O(m(Find_m^{sst} + Find_m^{rev} + \log n' +$

```
**secondaryOcc**($start, len$)
$pos \leftarrow select_0(B_S, start+1)$
$source\_id \leftarrow pos - start - 1$
$//D$ is the maximum depth
$d \leftarrow D$
**while** $source\_id > 0$ **do**
  $phrase\_id \leftarrow P_S^{-1}(source\_id)$
  $source\_start \leftarrow select_1(B_S, source\_id) - source\_id$
  **if** $source\_start + len(phrase\_id) \geq start + len$ **then**
    $occ\_pos \leftarrow first\_pos(source\_id) + start - source\_start$
    **report** $occ\_pos$
    secondaryOcc($occ\_pos, len$)
  **else**
    $d \leftarrow depth[source\_id] - 1$
    **if** $d < 0$ **then**
      **return**
  $source\_id \leftarrow prevLess(depth\_tree, source\_id, d)$
```

Figure 5.10: Searching for secondary occurrences from $T[start, start + len]$

$Extract_m) + occ_1 \log n')$, where $Find_m^{sst}$ is the time to search for a subpattern of length m in the suffix trie, $Find_m^{rev}$ is the time to search for a subpattern of length m in the reverse trie, and occ_1 is the number of primary occurrences. Thus the time to count the occurrences in the range structure is $O(\log n')$, the total time to locate the primary occurrences in the range structure is $O(occ_1 \log n')$, and $Extract_m$ is the time to extract m characters to verify the PATRICIA searches. $Extract_m$, as said in Section 4.2.2, depends on the parsing and is $O(mH)$ for LZ77 and $O(m + H)$ for LZ-End in the worst case. As our experiments show later (Section 6.1), in practice the difference is not as drastic: LZ77 is about 3 times (for most texts) slower for long substrings and not much slower for short substrings.

The $Find$ times depend on the structures used:

- Tries: $O(Find_m^{sst}) = O(Find_m^{rev}) = O(m+Skips) = O(m+mH) = O(mH)$ (as the time to compute all skips is $O(mExtract_1)$ in the worst case).
- Tries+Skips: $O(Find_m^{sst}) = O(Find_m^{rev}) = O(m + Skips) = O(m)$ (as the skips are stored).
- Binary Search: $O(Find_m^{sst}) = O(\log n' \cdot Extract_m)$ if we store the ar-

ray of ids explicitly, otherwise the time is $O(Find_m^{sst}) = O(\log n'(\log n' + Extract_m))$. $O(Find_m^{rev}) = O(\log n' \cdot Extract_m)$ on LZ77, and $O(m \log n')$ on LZ-End (as the extraction takes constant time per extracted symbol in this case). Here we save the verification of PATRICIA trees but this has no effect on the total complexity.

With this the total time using tries is $O(m^2H + m\log n' + occ_1 \log n')$, independent of the parsing. When adding skips the time drops to $O(m^2{+}mH{+}m\log n'{+} occ_1 \log n')$ on LZ-End. When using, instead, binary searching, the time is $O(m(m + H)\log n' + occ_1 \log n')$ for LZ-End and $O(m^2H \log n' + occ_1 \log n')$ for LZ77 if we store the *id* array explicitly, otherwise the time increases to $O(m(\log n'{+}m{+}H)\log n'{+}occ_1 \log n')$ for LZ-End and $O(m(\log n'{+}mH)\log n'{+} occ_1 \log n')$ for LZ77.

- **Secondary Occurrences:** the total time is $O(\frac{1}{\varepsilon}occ(\log D + D))$, where D is the maximum depth and ε is the parameter for the representation of the permutation (Section 2.7). Recall from Sections 5.3.1 and 5.3.2 that the time to find the secondary occurrences from a seed is $O(\frac{1}{\varepsilon}occ \log D + D)$. However in this case we are recursively locating the secondary occurrences from all the occurrences found, and in a worst case we could pay $O(D)$ for each occurrence, not finding new ones.

Taking $\varepsilon = \frac{1}{\log n'}$ gives us a total time similar to the one given for the primary occurrences, except that $occ_1 \log n'$ changes to $occ \cdot D \log n'$.

To solve *exists* queries, we basically search for the first primary occurrence. Hence the total time is as given for the primary occurrences replacing $occ_1 = 0$ (the details can be seen in Table 5.1).

5.5 Construction

In this section we explain the construction algorithm of the proposed index. We propose a practical construction algorithm, with bounded space usage and decent times. (See table 5.2 for a reminder of the definitions of the variables.)

1. **Alphabet mapping:** as we work with standard texts that represent each symbol using 1 byte, we map the byte values to effective alphabet positions and vice versa.

2. **LZ parsing:** For LZ77 we use the algorithm CPS2 of Chen *et al.* [CPS08] and for LZ-End we use the algorithm described in Section 4.4. At this stage we generate three different files containing the trailing characters, the lengths of the sources, and the starting positions of the sources. Using suffix trees (Section 2.10), the LZ77 parsing can be computed in $O(n)$ time using $O(n)$ words of space, and the LZ-End parsing in $O(N)$ time using the same space. Additionally, the LZ77 parsing can be computed theoretically in $O(n \log n(\log^\epsilon n + o(\log \sigma)))$ time using $n(H_k(T) + 2) + o(n \log \sigma)$ bits and the LZ-End parsing in time $O(\log n(N \log^\epsilon n + n\, o(\log \sigma)))$ using $n(H_k(T) + 2) + n' \log n + o(n \log \sigma)$ bits (Section 4.4). The practical algorithms take total time $O(n \log n)$ for LZ77 and $O(N \log n)$ for LZ-End (recall Section 4.4). The space usage is around $5.7n$ bytes for LZ77 and $9n$ bytes for LZ-End, and this is the peak space usage for the self-index construction. In the index we only store explicitly the trailing characters using $n' \log \sigma$ bits.

3. **Bitmap of phrases:** this bitmap is easily computed from the array containing the lengths of the sources in time $O(n')$. It uses $n' \log \frac{n}{n'} + O(n' + \frac{n \log \log n}{\log n})$ bits (Section 2.5). In practice we use δ-encoded bitmaps, using $n' \log \frac{n}{n'} + O(n' \log \log n + \frac{n' \log n'}{s})$ bits, where s is the sampling step. All query times are then multiplied by $s + \log \frac{n'}{s}$. For the analysis we will assume $s = \log n'$.

4. **Suffix trie and reverse trie:** for constructing these trees, we decided to insert all indexed substrings in a PATRICIA trie. This is $O(n)$ time for the reverse trie, but it could be quadratic for the suffix trie (there are complex $O(n)$-time algorithms for building suffix tries [IT06]). In practice this does not happen and the running time is good, as the number of phrases n' generated by the parsing is relatively small. Of course we insert and store pointers to the text in the trie, rather than the whole strings. From the PATRICIA trie, we can extract the sorted ids, the skips and the DFUDS representation of the tree in time $O(n')$. Each tree will have at most $2n'$ nodes, hence they require $4n' + o(n')$ bits (Section 2.8) for the topology of the tree, plus $2n'$ label characters encoded using $2n' \log \sigma$ bits. Additionally, the rev_ids are stored using $n' \log n'$ bits.

5. **Range structure:** to build the range structure we need a permutation from the ids of the suffix tries to the ids of the reverse trie. This is done in $O(n')$ time, inverting the permutation of the ids and then traversing the rev_ids and assigning each to the corresponding id. Then, the range structure is built starting from the permutation in time $O(n' \log n')$. It uses $n' \log n' + O(n' \log \log n')$ bits (Section 2.6.1).

6. **Sources depths:** for computing the secondary occurrences related structures

we need to first compute the depth of each source. First we sort all sources by increasing starting position, breaking ties by decreasing length. Doing this we know that all parents of a source are to its left. We keep track of the rightmost source of depth d for each possible depth. Then for each source we binary search the rightmost sources and find the deepest source d that covers the current phrase. Afterward, we set the current source as the rightmost source of depth $d+1$. The running time of the algorithm is $O(n' \log n')$.

7. **Prev-Less Depth Structure:** this structure is constructed in $O(n' \log D)$ time as it is just a wavelet tree It uses $n' \log D + O(n' \log\log D)$ bits (Section 5.3.3).

8. **Source-Phrase Permutation:** it takes $O(n')$ time starting from the ids of the sorted sources. It is stored using $(1+\varepsilon)n' \log n' + O(n')$ bits (Section 2.7), and as we set $\varepsilon = \frac{1}{\log n'}$, the total space is $n' \log n' + O(n')$ bits.

9. **Bitmap of Sources:** it takes $O(n')$ time to build from the starting positions of the sorted sources. It uses $n' \log \frac{n}{n'} + O(n' + \frac{n \log\log n}{\log n})$ bits (Section 2.5). In practice we use δ-encoded bitmaps, so the same considerations as for the bitmap of phrases apply.

Adding up the space of all structures we get that the index requires $2n' \log n + n' \log n' + n' \log D + O(n' \log \sigma + \frac{n \log\log n}{\log n})$ bits of space, which in our practical implementation is $2n' \log n + n' \log n' + n' \log D + O(n' \log \sigma + n' \log\log n)$ bits plus the skips we store. Note that in the case of binary searching we do not use tries, yet the asymptotic space complexity is not reduced.

Note that our practical index space is fully proportional to n', depending on n only logarithmically.

For the construction time and space of the index we have given practical figures. We give now two trade-offs for the theoretical upper bounds. The first, Theory$_1$, uses the least possible construction time, and the second, Theory$_2$, the least possible construction space.

- Theory$_1$: The space gets dominated by the $O(n \log n)$ bits needed to build the parsing. All construction times are $O(n' \log n')$, except the parsing and creating the PATRICIA trees. Hence, the index is built in time $O(n + n' \log n')$ $(O(N + n' \log n')$ for LZ-End) using $O(n \log n)$ bits.

- Theory$_2$: In Section 4.4 we showed that the LZ77 parsing can be computed in $O(n \log n(\log^{\epsilon} n + o(\log \sigma)))$ time using $O(nH_k(T)) + o(n \log \sigma)$ bits and the LZ-

End parsing in time $O(\log n(N \log^{\epsilon} n + n\, o(\log \sigma)))$ using the same space. All data structures (except PATRICIA trees) are constructed as explained above, each structure requiring at most $O(n' \log n) = O(nH_k(T)) + o(n \log \sigma)$ (recall Lemma 4.13) and $O(n' \log n')$ time. In the following we show that we can build the PATRICIA trees in time $O(n' \log^{1+\epsilon} n)$ using $O(nH_k(T) + o(n \log \sigma))$ bits. The idea is similar to that presented by Claude and Navarro [CN10] enhanced with some ideas from Russo *et al.* [RNO08]

1) First we build the FM-index [FM05] of T in $O(n \log n \frac{\log \sigma}{\log \log n})$ time within $nH_k(T) + o(n \log \sigma)$ bits of space. 2) Then, we build the Fully-Compressed Suffix Tree (FCST) [RNO08], that supports all tree operations in $O(\log^{1+\varepsilon} n)$ time. For building the FCST, we simulate a traversal of the suffix tree starting from the root using Weiner links [Wei73], which are simulated using the LF mapping (see Section 2.12) over the FM-Index. During the traversal we mark all nodes that are at depth multiple of δ in the implicit tree defined by the Weiner links, where δ is the space/time trade-off parameter of the FCST (which takes $o(n \log \sigma)$ bits of space for $\delta = \log^{1+\epsilon} n$). These nodes are stored in a simple array using $o(n \log \sigma)$ bits with all the information required by the FCST construction algorithm. The running time of this process is $O(n \frac{\log \sigma}{\log \log n})$ and the space is $o(n \log \sigma)$ bits. 3) We use a dynamic balanced tree to mark some of the nodes of the FCST; these will be the nodes of our PATRICIA tree. For each phrase starting position we convert it using A^{-1} to an FM-Index position, and then *selectLeaf* (which gives the i-th leaf) converts it to a position in the FCST. We mark in the balanced tree the *preorder* of the FCST node, as well as the phrase id. Then, we traverse the balanced tree from left to right computing $LCA(x_i, x_{i+1})$ (where x_i is the current node and x_{i+1} is the node to the right, and LCA is their lowest common ancestor in the FCST) and inserting the value in the tree. To build the PATRICIA tree, we traverse the balanced tree again from left to right, creating the PATRICIA nodes, generating the parentheses representation and the labels of the edges. Using the operation *letter* of the FCST (which gives any letter of the path leading to a node) we retrieve the label. Using the FCST we determine the topology of the tree (we keep the current PATRICIA path in a stack; add closing parentheses and pop the stack until the top of the stack is an ancestor of the new node; and then we add the opening parenthesis for the current node and push it to the stack). This step runs in $O(n' \log^{1+\epsilon} n)$ time and within $O(n' \log n) = O(nH_k(T)) + o(n \log \sigma)$ (recall Lemma 4.13) bits of space.

Hence, the total time is $O(n \log n \frac{\log \sigma}{\log \log n} + n' \log^{1+\varepsilon} n)$ and the total space is

$O(nH_k(T)) + o(n \log \sigma)$. The process for constructing the reverse trie is almost the same, but now we do not consider the whole suffixes, because they are limited by the phrase length. Given $A^{-1}(pos)$, we use the operation $LAQ_S(d)$ of the FCST (which retrieves the ancestor with string depth d) to find which node we need to mark.

As the time for constructing the PATRICIA trees gets dominated by the parsing algorithm the total space required is $O(nH_k(T)) + o(n \log \sigma)$ and the total running time is $O(n \log n(\log^\epsilon n + o(\log \sigma)))$ for LZ77 and $O(\log n(N \log^\epsilon n + n\, o(\log \sigma)))$ for LZ-End.

5.6 Summary

We have presented a self-index that given a text of length n, parsed into n' different phrases by a Lempel-Ziv like parsing, uses space proportional to that of the compressed text, i.e., $O(n' \log n) + o(n)$ bits. Table 5.1 summarizes the space and time of the operations over the index and Table 5.2 summarizes all the parameters of the index. In practice, due to our sparse bitmap representation, the $o(n)$ bits disappear from the space but the times of *extract*, *exists* and *locate* are multiplied by $O(\log n')$.

	Tries	Binary Search
Construction Time	Theory$_1$: $O(n + n' \log n')$ for LZ77 and $O(N + n' \log n')$ for LZ-End Theory$_2$: $O(n \log n(\log^{\epsilon} n + o(\log \sigma)))$ for LZ77 and $O(\log n(N \log^{\epsilon} n + n\, o(\log \sigma)))$ for LZ-End Practice: $O(n \log n)$ for LZ77, $O(N \log n)$ for LZ-End.	
Construction Space	Theory$_1$: $O(n \log n)$ bits Theory$_2$: $O(nH_k(T)) + o(n \log \sigma)$ Practice: LZ77 $\approx 6n$ bytes, LZ-End $\approx 9n$ bytes	
Index Space	Theory: $2n' \log n + n' \log n' + n' \log D + O(n' \log \sigma + \frac{n \log \log n}{\log n})$ bits Practice: $2n' \log n + n' \log n' + n' \log D + O(n' \log \sigma + n' \log \log n)$ bits	
Extract Time	LZ77: $O(mH)$, LZ-End: $O(m + H)$	
Exists Time	$O(m^2H + m \log n')$ With skips and LZ-End: $O(m^2 + mH + m \log n)$	Using $n' \log n'$ additional bits: LZ77: $O(m^2H \log n')$, LZ-End: $O(m(m+H) \log n')$. Otherwise: LZ77: $O(m(\log n' + mH) \log n')$, LZ-End: $O(m(\log n' + m + H) \log n')$
Locate Time	$O(m^2H + m \log n' + occ \cdot D \log n')$ With skips and LZ-End: $O(m^2 + mH + m \log n' + occ \cdot D \log n')$	Using $n' \log n'$ additional bits: LZ77: $O(m^2H \log n' + occ \cdot D \log n')$, LZ-End: $O(m(m + H) \log n' + occ \cdot D \log n')$. Otherwise: LZ77: $O(m(\log n' + mH) \log n' + occ \cdot D \log n')$, LZ-End: $O(m(\log n' + m + H) \log n' + occ \cdot D \log n')$

Table 5.1: Summary table of LZ77-Index. Adding skips requires at most $4n' \log n$ more bits, but far less in practice. In practice times are multiplied by $O(\log n')$.

Parameter	Description	Defined in
σ	size of the alphabet	Section 2.1
n	length of the text	Section 2.1
n'	length of the LZ parsing	Definitions 2.30 and 4.2
m	length of the pattern	Section 2.2
s	sampling step of δ-encoded bitmap	Section 2.5.2
ε	parameter of the permutation	Section 2.7
D	maximum depth of the sources	Section 5.3.2
H	height of the LZ parsing	Definition 4.5
N	total text retraversed in the LZ-End parsing	Section 4.4

Table 5.2: Summary table of parameters of LZ77-Index

Chapter 6

Experimental Evaluation

6.1 Experimental Setup

In our tests we compared the proposed index against RLCSA [NM07]. We did not test the SLP Index [CN09] because we could not make it run consistently in our collections, yet some comparison results can be inferred from their experimental evaluation [CFMPN10], as we do in Section 6.2.

We used in our experiments the LZ77 and the LZ-End parsings. For the LZ indexes we used the following variants (defined in Section 5.2.6), ordered by decreasing space requirement. In all variants we stored the skips of the trees using DAC (Section 2.4.1), because not using them lead to results worse than using binary search (our slowest variant).

1. Suffix trie and reverse trie (original proposal).

2. Binary search on ids with the explicit ids and reverse trie.

3. Binary search on reverse ids and suffix trie.

4. Binary search on ids with explicit ids and binary search on reverse ids.

5. Binary search on ids with implicit ids and binary search on reverse ids.

Recall from Section 5.2.6 that the array of the ids is not stored in the index, only the array of reverse ids. Thus, if we want to binary search over ids we have two alternatives: (1) spend $n' \log n'$ additional bits to store explicitly the array of ids, or

(2) using Remark 5.6 to access the array implicitly by paying $O(\log n')$ access time. The index variants with *explicit ids* refer to the alternative (1) and the ones with *implicit ids* refer to alternative (2). The alternative using the suffix trie do not need to access the id array, but rather the reverse ids array, which is always maintained in explicit form

Remark 6.1. The reader may note that the results concerning the alternative

- Binary search on ids with implicit ids and reverse trie

is not present. We omit the empirical results of this alternative as the compression ratio is about the same obtained using alternative number 3 and the performance of *locate* is noticeably worse. Remember that accessing the implicit array of ids takes time $O(\log n')$.

The parameters used for the data structure are as follows: $s = 16$ for the δ-codes bitmap (Section 2.5.2), $\varepsilon = 1/32$ for the permutation (Section 2.7) and sampling step $b = 20$ for the bitmaps of González *et al.* (Section 2.5.1). We used these parameter values as they are the typical ones used in experimentation, and additionally with these values our indexes achieve a good space/time trade-off.

For RLCSA we used sampling with steps 512, 256, 128 and 64. The index was built using a buffer of 100MiB.

All our experiments were conducted on a machine with two Intel Xeon CPU running at 2.00GHz with 16GiB main memory. The operating system is Ubuntu 8.04.4 LTS with Kernel 2.6.24-27-server. The compiler used was `g++` (`gcc` version 4.2.4) executed with the `-O3` optimization flag.

We present the results obtained for the following texts (the results of only one text from each collection are presented in this section, the remaining results are presented in Appendix A):

- Artificial: F_{41}, R_{13}, T_{29}.
- Pseudo-Real: DNA 0.1% (Scheme 1), Proteins 0.1% (Scheme 1), English 0.1% (Scheme 2), Sources 0.1% (Scheme 2).
- Real: Para, Cere, Influenza, Escherichia Coli, Coreutils, Kernel, Einstein (en), Einstein (de), World Leaders.

We restricted the experiments to the texts listed above since many of the texts produced similar results during preliminary experiments. For DNA and Wiki domains, we chose the largest texts. For the case of pseudo-real texts we kept the DNA, Proteins, English and Sources texts, as these kind of texts naturally form repetitive collections.

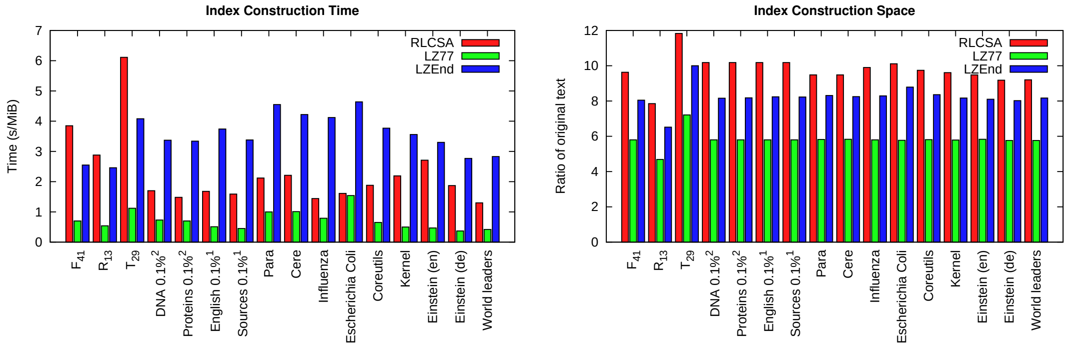
Two types of experiments were carried out. One, labeled "results (1)" in Figures 6.2-6.7 and A.1-A.26 , considers the time of the operations as a function of the pattern length $|P|$. The second, labeled "results (2)" in Figures 6.2-6.7 and A.1-A.26 , shows the space/time trade-off of the operations. Although we do not show a space/time tuning for LZ77/LZ-End index, the plots of figures labeled "results (2)" show a line formed by 5 points. These refer to the variants 1-5 described above. The results are presented and discussed in Section 6.1. The experiments conducted are the following:

- **Construction time and space:** we present the build time for each index as well as the peak memory usage. Results are given in Figure 6.1.
- **Compression Ratio:** we present the compression ratio for different self-indexes. We show alternatives 1 and 5 of our indexes, which are respectively the largest and smallest variants. For RLCSA we show the space achieved with a sampling step of 512, and without the samples, which is the lowest space reachable by that index. For the LZ78-index [ANS06] (Section 2.14.3) we used $\varepsilon = \frac{1}{128}$ as the sampling step of the permutation. Additionally, we show the compression ratio of ILZI [RO08] (Section 2.14.3). The results are shown in Table 6.1, where we also include p7zip as a baseline.
- **Structures Space:** we present the space usage of the different data structures used in our indexes. The results are given in Tables 6.4 and 6.5 as percentage of the size of the index.
- **Parsing Statistics:** we present the value of D and H, which affect the performance of our indexes. The results are displayed in Tables 6.2 and 6.3.
- **Extraction speed:** we extracted 10,000 substrings of length $2^i, i \in \{0, \ldots, 12\}$. We show only one line for the LZ77 and the LZ-End index, as all the variants have the same extraction speed. See the top-left plots of Figures 6.2-6.7 labeled "results (1)", which are representative of all the results (the rest are in Appendix A, in Figures A.1-A.26). We also show the space/time trade-off of the indexes for extracting a pattern of length 2^{12}. Extraction times per character stabilize at this length. See the top-left plot of Figures 6.2-6.7 and A.1-A.26 labeled "results (2)".

- **Search time:** we located 1,000 patterns of length 10, 15, and 20. We limited the number of occurrences reported to 30,000. See plots 2-4 in reading order of Figures 6.2-6.7 and A.1-A.26 labeled "results (2)". We also located patterns of increasing length from 5 to 40. In this case we only show the results for alternative 1 (original proposal) and 5 (minimum space) of both LZ77 and LZ-End. See top-right plot of Figures 6.2-6.7 and A.1-A.26 labeled "results (1)".

- **Locate time:** we located 1,000 patterns of length 2 and 4. This test highlights the time needed to find the occurrences in our indexes, as it dominates the time for traversing the tries. We limited the number of occurrences reported to 100,000. See plots 5-6 of Figures 6.2-6.7 and A.1-A.26 labeled "results (2)".

- **Exists Time:** we generated 20,000 patterns of lengths 5, 10, 20, 40 and 80; half of them were present in the text and the other half were a random concatenation of symbols of the text. For RLCSA we check the existence using a *count* query. For this reason, we only show one line for RLCSA, as *count* time is independent of the sampling size. The *exists* query of the LZ77 index is basically a search of a primary occurrence, and thus it illustrates the time for traversing the tries. We only show the results for alternative 1 (original proposal) of both LZ77 and LZ-End, since the other alternatives are orders of magnitude slower for this. See left-bottom plot of Figures 6.2-6.7 and A.1-A.26 labeled "results (1)" for existing patterns and right-bottom plot for non-existing patterns. We also show the space/time trade-off of these two queries for patterns of length 20, see plots 7-8 of Figures 6.2-6.7 and A.1-A.26 labeled "results (2)".



	LZ77 Index		LZ-End Index		RLCSA	
File	Time	Space	Time	Space	Time	Space
F_{41}	0.70	5.80	2.55	8.05	3.85	9.63
R_{13}	0.54	4.69	2.46	6.52	2.88	7.85
T_{29}	1.12	7.21	4.08	10.00	6.11	11.83
DNA 0.1% [2]	0.73	5.80	3.37	8.16	1.70	10.18
Proteins 0.1% [2]	0.70	5.80	3.34	8.18	1.48	10.18
English 0.1% [1]	0.51	5.80	3.74	8.24	1.68	10.18
Sources 0.1% [1]	0.45	5.80	3.38	8.23	1.59	10.18
Para	1.00	5.82	4.55	8.31	2.12	9.48
Cere	1.01	5.83	4.22	8.25	2.21	9.48
Influenza	0.79	5.80	4.12	8.29	1.44	9.90
Escherichia Coli	1.54	5.77	4.64	8.79	1.61	10.11
Coreutils	0.65	5.81	3.77	8.36	1.88	9.74
Kernel	0.50	5.79	3.56	8.17	2.19	9.61
Einstein (en)	0.47	5.83	3.30	8.10	2.71	9.48
Einstein (de)	0.37	5.76	2.77	8.02	1.87	9.18
World leaders	0.42	5.76	2.83	8.17	1.30	9.20

Figure 6.1: Construction time and space for the indexes. Times are in seconds per MiB and spaces are the ratio between construction space and text space.

Text	LZ78	ILZI	RLCSA	RLCSA$_{512}$	LZ77$_5$	LZ77$_1$	LZ-End$_5$	LZ-End$_1$	p7zip
F_{41}	2.43	0.59	4.73E-4	0.96	**3.28E-4**	6.04E-4	**3.75E-4**	5.75E-3	0.18
R_{13}	3.68	0.59	7.47E-4	0.94	**4.15E-4**	7.36E-4	**6.72E-4**	1.10E-3	0.36
T_{29}	4.18	0.76	6.03E-4	0.52	**3.80E-4**	6.93E-4	**5.95E-4**	1.02E-3	0.17
DNA 0.1% [2]	83.28	60.55	4.25	4.73	**2.12**	**3.47**	**4.21**	6.19	0.51
Proteins 0.1% [2]	82.94	71.67	3.80	4.30	**2.37**	**3.69**	4.69	6.71	0.59
English 0.1% [1]	90.71	69.42	4.52	5.00	**2.49**	**3.91**	7.32	10.62	0.55
Sources 0.1% [1]	82.82	58.40	3.77	4.25	**2.10**	**3.31**	6.83	10.13	0.44
Para	82.04	72.99	9.86	10.83	**5.36**	**8.45**	**9.20**	13.38	1.46
Cere	79.37	66.93	7.60	8.57	**3.74**	**5.94**	**6.16**	8.96	1.14
Influenza	46.88	36.19	3.84	4.77	4.55	7.49	9.20	13.89	1.35
Escherichia Coli	88.36	83.69	24.71	25.62	**18.82**	29.71	30.36	43.58	4.72
Coreutils	81.36	58.38	6.53	7.46	7.89	12.47	11.89	17.47	1.94
Kernel	85.94	60.19	3.78	4.71	**3.31**	5.26	5.12	7.50	0.81
Einstein (en)	33.26	13.28	0.23	1.20	**0.18**	0.30	0.32	0.48	0.07
Einstein (de)	48.99	21.61	0.48	1.39	**0.32**	0.54	0.59	0.89	0.11
World leaders	49.74	35.08	3.32	4.20	3.85	6.27	6.44	9.63	1.29

Table 6.1: Compression ratio (given in percentage of original file size) of different self-indexes. In bold are highlighted those LZ-based indexes outperforming the best compression achievable by RLCSA.

	LZ77 Index		LZ-End Index	
Text	Mean	D	Mean	D
F_{41}	0.00	0	0.10	1
R_{13}	0.35	2	1.47	4
T_{29}	0.51	2	1.58	3
DNA 0.1% [2]	13.45	27	11.43	26
Proteins 0.1% [2]	14.77	38	12.25	43
English 0.1% [1]	1.65	15	2.81	23
Sources 0.1% [1]	1.55	16	2.65	34
Para	2.76	24	3.31	21
Cere	3.53	37	4.32	21
Influenza	2.00	33	1.58	24
Escherichia Coli	2.15	13	2.40	15
Coreutils	2.51	32	2.29	40
Kernel	2.57	28	3.05	41
Einstein (en)	2.53	14	2.62	22
Einstein (de)	2.86	13	3.27	16
World leaders	2.81	46	2.63	29

Table 6.2: D value (i.e., maximum depth) and mean depth for the LZ indexes

	LZ77 Index		LZ-End Index	
Text	Mean	H	Mean	H
F_{41}	26.56	39	24.56	36
R_{13}	19.54	27	21.7	45
T_{29}	14.41	22	13.53	27
DNA 0.1% [2]	6.63	26	5.81	25
Proteins 0.1% [2]	4.77	19	4.86	22
English 0.1% [1]	30.36	96	9.67	43
Sources 0.1% [1]	31.06	98	10.08	64
Para	9.38	31	8.39	259
Cere	9.25	34	8.66	257
Influenza	11.93	46	11.65	80
Escherichia Coli	7.95	28	6.86	28
Coreutils	9.03	39	11.21	175
Kernel	12.36	51	9.45	46
Einstein (en)	176.28	1003	23.66	118
Einstein (de)	73.01	507	15.72	61
World leaders	11.31	41	14.77	103

Table 6.3: H value (i.e., maximum extraction cost) and mean extraction cost for the LZ indexes

	Trailing characters	B	Suffix trie	Suffix trie skips	Reverse trie	Reverse trie skips	*rev_ids*	Range	Depth	P_S	B_S
F_{41}	0.03	0.35	47.00	0.76	47.00	0.74	0.08	1.93	0.51	0.54	0.32
R_{13}	0.03	0.34	46.87	0.72	46.86	0.67	0.08	1.92	0.94	0.53	0.31
T_{29}	0.04	0.46	46.63	0.79	46.62	0.80	0.11	1.91	0.94	0.57	0.41
DNA 0.1% [2]	1.98	10.45	13.69	7.14	13.36	6.52	11.86	12.51	3.51	12.93	6.04
Proteins 0.1% [2]	3.48	9.69	12.94	6.94	12.09	5.64	12.51	13.19	4.10	13.64	5.78
English 0.1% [1]	4.50	9.59	13.15	5.99	13.40	5.59	11.56	12.19	2.76	12.60	8.67
Sources 0.1% [1]	4.36	9.92	13.42	5.94	13.43	5.36	11.22	11.84	3.09	12.23	9.18
Para	1.90	8.52	12.53	6.81	12.23	6.43	13.28	13.95	3.32	14.36	6.65
Cere	1.87	8.60	12.72	6.85	12.42	6.45	13.06	13.72	3.55	14.12	6.64
Influenza	2.34	9.35	13.08	7.82	12.71	7.26	11.70	12.30	3.46	12.68	7.31
Escherichia Coli	2.56	8.20	12.66	6.65	12.40	6.38	13.43	14.11	2.68	14.52	6.42
Coreutils	4.92	7.50	13.23	5.99	13.21	5.62	12.91	13.57	3.47	13.96	5.62
Kernel	5.09	7.47	13.41	6.01	13.25	5.72	12.71	13.37	3.37	13.78	5.82
Einstein (en)	5.19	8.23	15.06	6.59	14.45	5.95	11.02	11.70	2.88	12.06	6.86
Einstein (de)	4.55	7.93	16.80	6.42	17.52	5.78	9.75	10.55	3.08	10.77	6.78
World leaders	4.47	8.28	13.57	6.83	13.98	6.16	11.49	12.13	4.08	12.52	6.49

Table 6.4: Detailed space of LZ77 index structures. Values are in percentage of the total size.

	Trailing characters	B	Suffix trie	Suffix trie skips	Reverse trie	Reverse trie skips	*rev_ids*	Range	Depth	P_S	B_S
F_{41}	0.03	0.51	47.13	0.47	47.12	0.46	0.08	1.94	0.51	0.54	0.48
R_{13}	0.05	0.82	45.70	0.78	45.70	0.78	0.15	2.24	1.75	0.61	0.74
T_{29}	0.06	1.02	45.55	1.05	45.54	0.84	0.20	2.22	1.33	0.65	0.83
DNA 0.1% [2]	1.70	15.01	11.25	6.12	10.66	5.67	10.78	11.35	2.99	11.71	12.75
Proteins 0.1% [2]	2.91	14.38	10.70	5.62	10.17	5.20	11.07	11.65	3.55	12.03	12.71
English 0.1% [1]	3.90	14.23	11.58	5.43	10.73	4.67	10.58	11.13	2.78	11.49	13.48
Sources 0.1% [1]	3.76	13.93	12.55	5.33	11.53	4.48	10.22	10.75	3.10	11.10	13.24
Para	1.63	13.66	11.10	5.69	10.48	5.28	11.93	12.53	2.67	12.88	12.16
Cere	1.66	14.13	10.99	5.89	10.31	5.48	11.60	12.18	2.71	12.54	12.52
Influenza	2.09	13.99	11.41	6.54	11.03	6.28	10.44	10.97	2.75	11.32	13.18
Escherichia Coli	2.21	13.61	10.69	5.65	10.05	5.24	12.16	12.77	2.33	13.12	12.18
Coreutils	4.26	12.63	11.70	5.10	11.53	4.81	11.18	11.75	3.17	12.09	11.78
Kernel	4.40	12.84	11.63	5.11	11.42	4.85	11.01	11.57	3.30	11.93	11.92
Einstein (en)	4.47	13.82	12.73	5.69	12.10	5.05	9.49	10.05	2.92	10.38	13.28
Einstein (de)	3.94	13.69	14.00	5.60	13.38	4.92	8.99	9.65	2.80	9.89	13.11
World leaders	3.90	13.77	11.77	5.65	12.12	5.33	10.02	10.57	3.02	10.93	12.91

Table 6.5: Detailed space of LZ-End index structures. Values are in percentage of the total size.

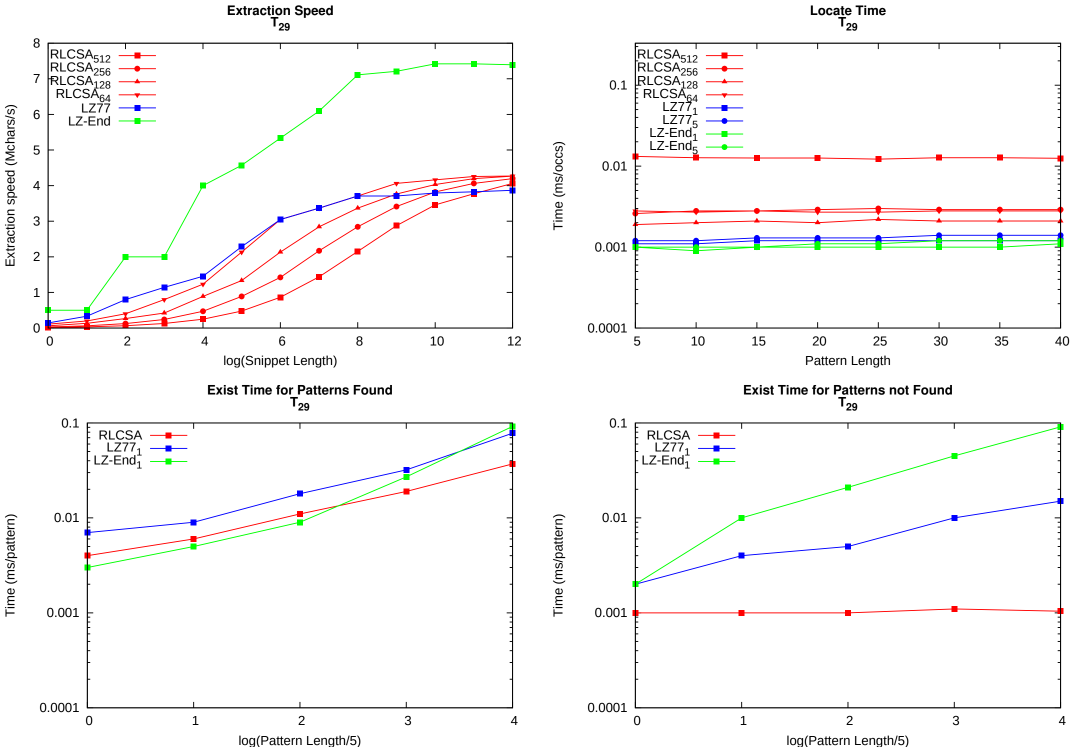


Figure 6.2: T_{29} results (1). Note the logscales.

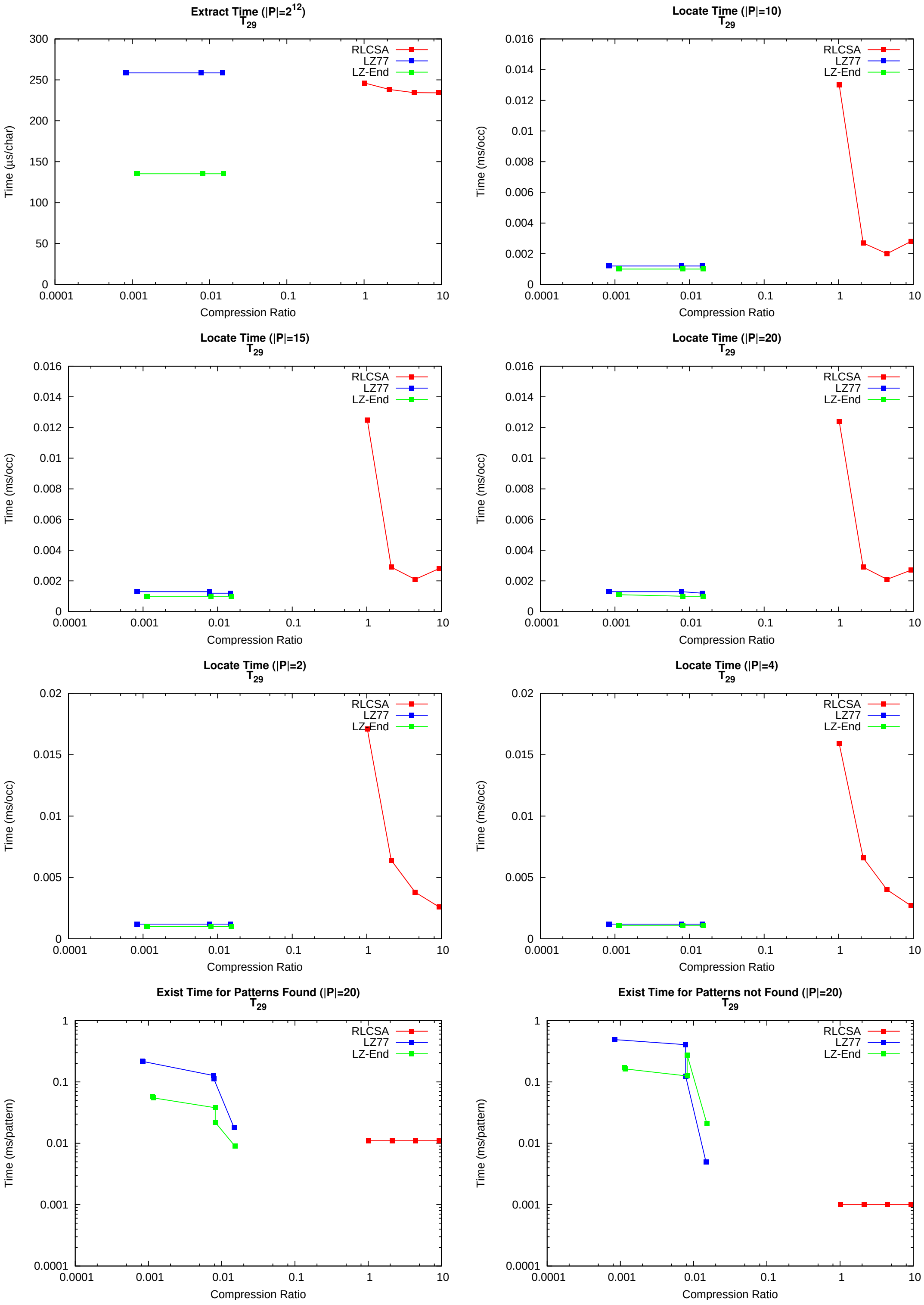


Figure 6.3: T_{29} results (2). Note the logscales.

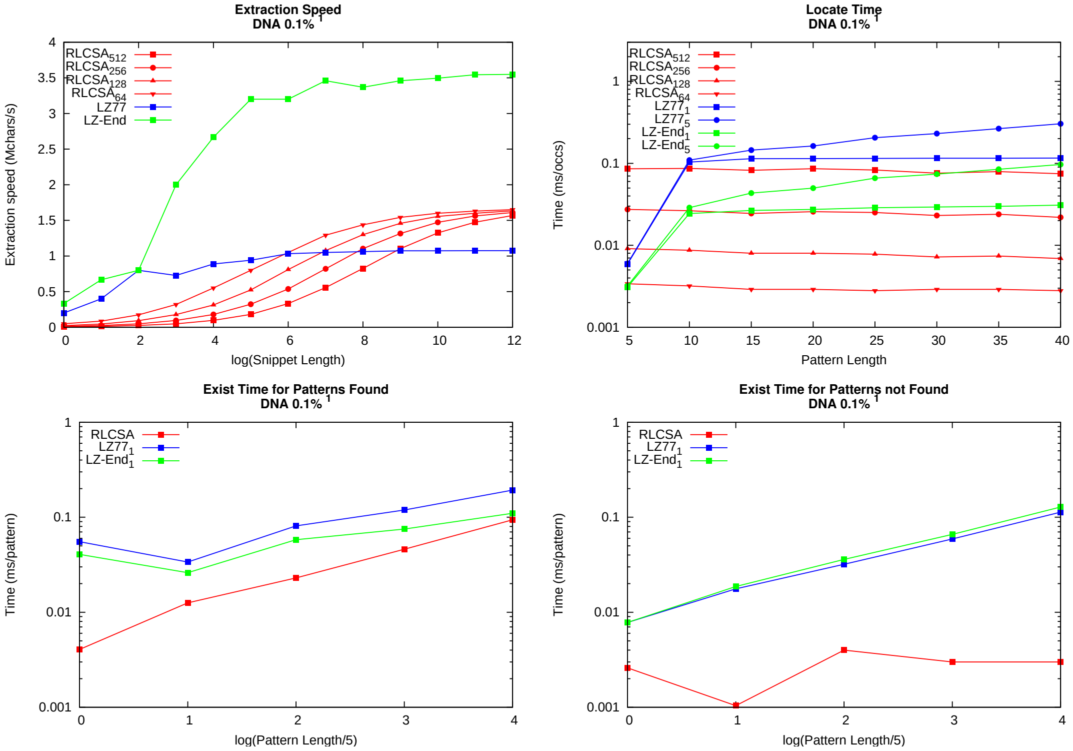


Figure 6.4: DNA 0.1% [1] results (1). Note the logscales.

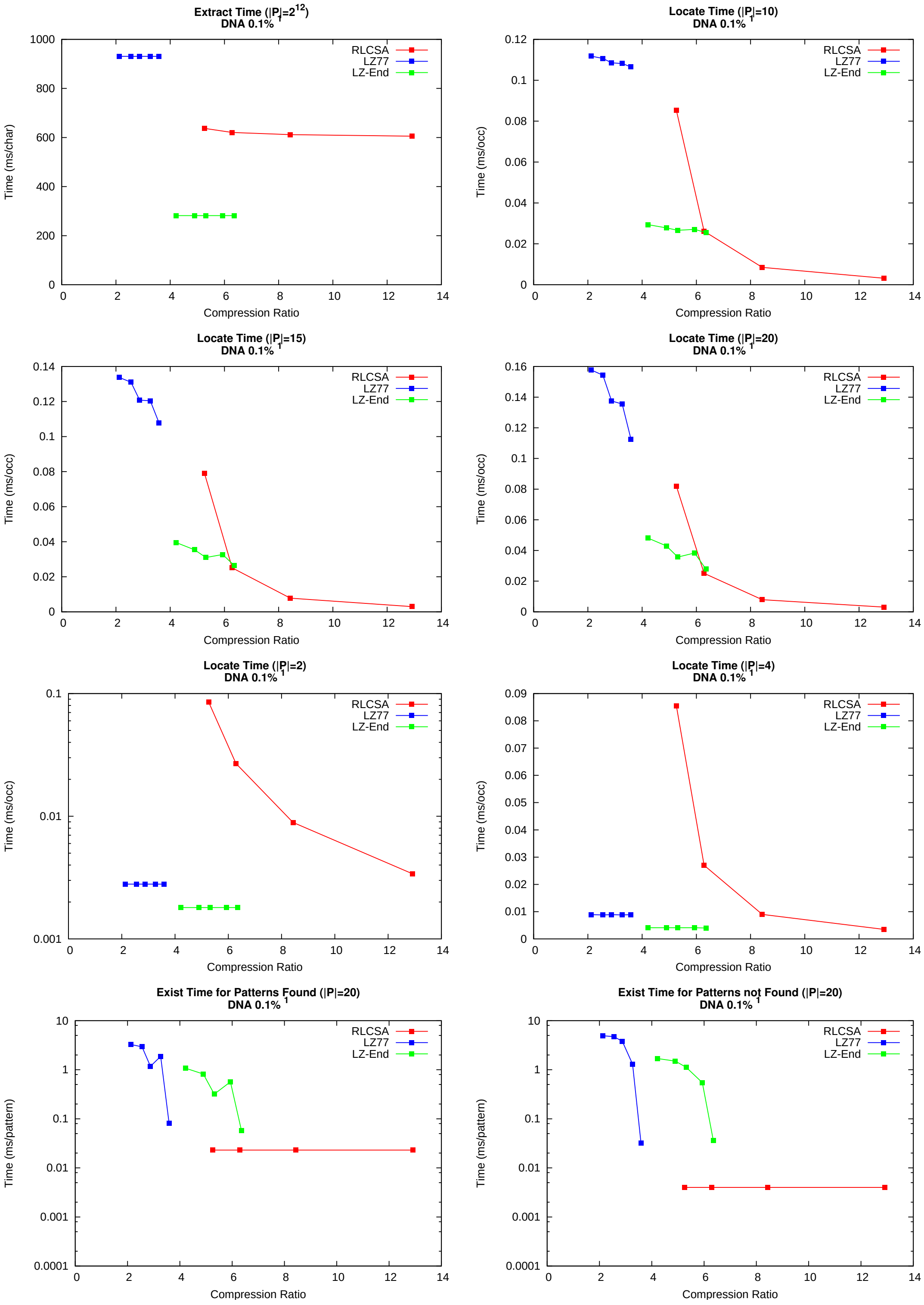


Figure 6.5: DNA 0.1% [1] results (2). Note the logscales.

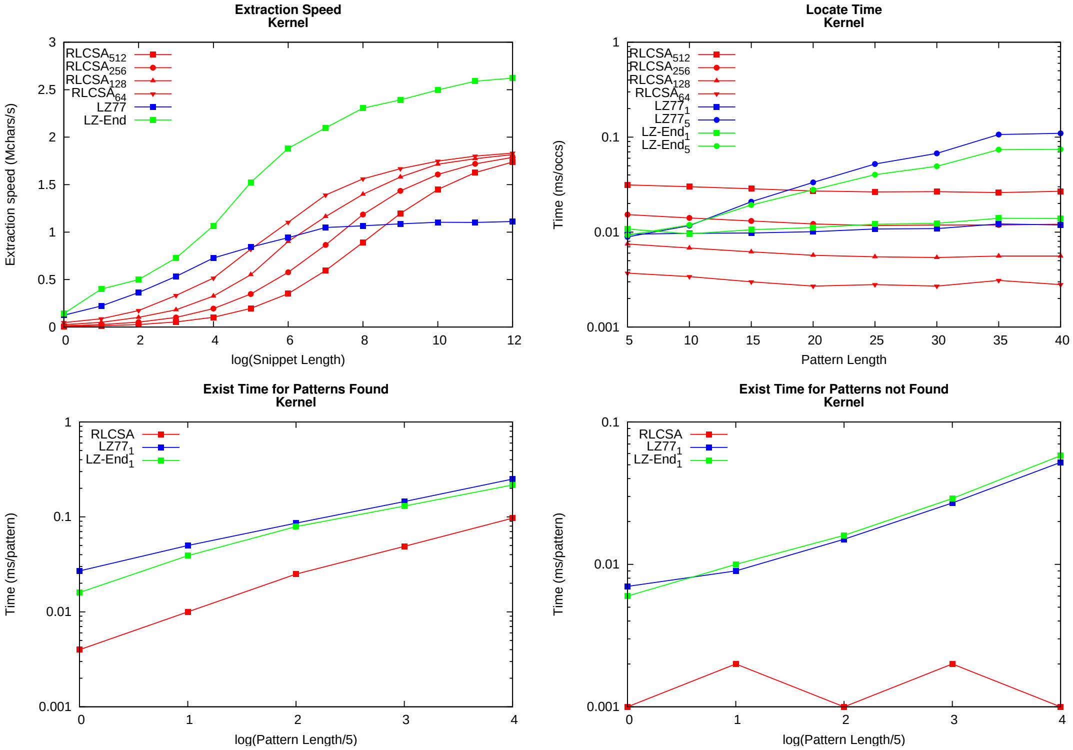


Figure 6.6: Kernel results (1). Note the logscales.

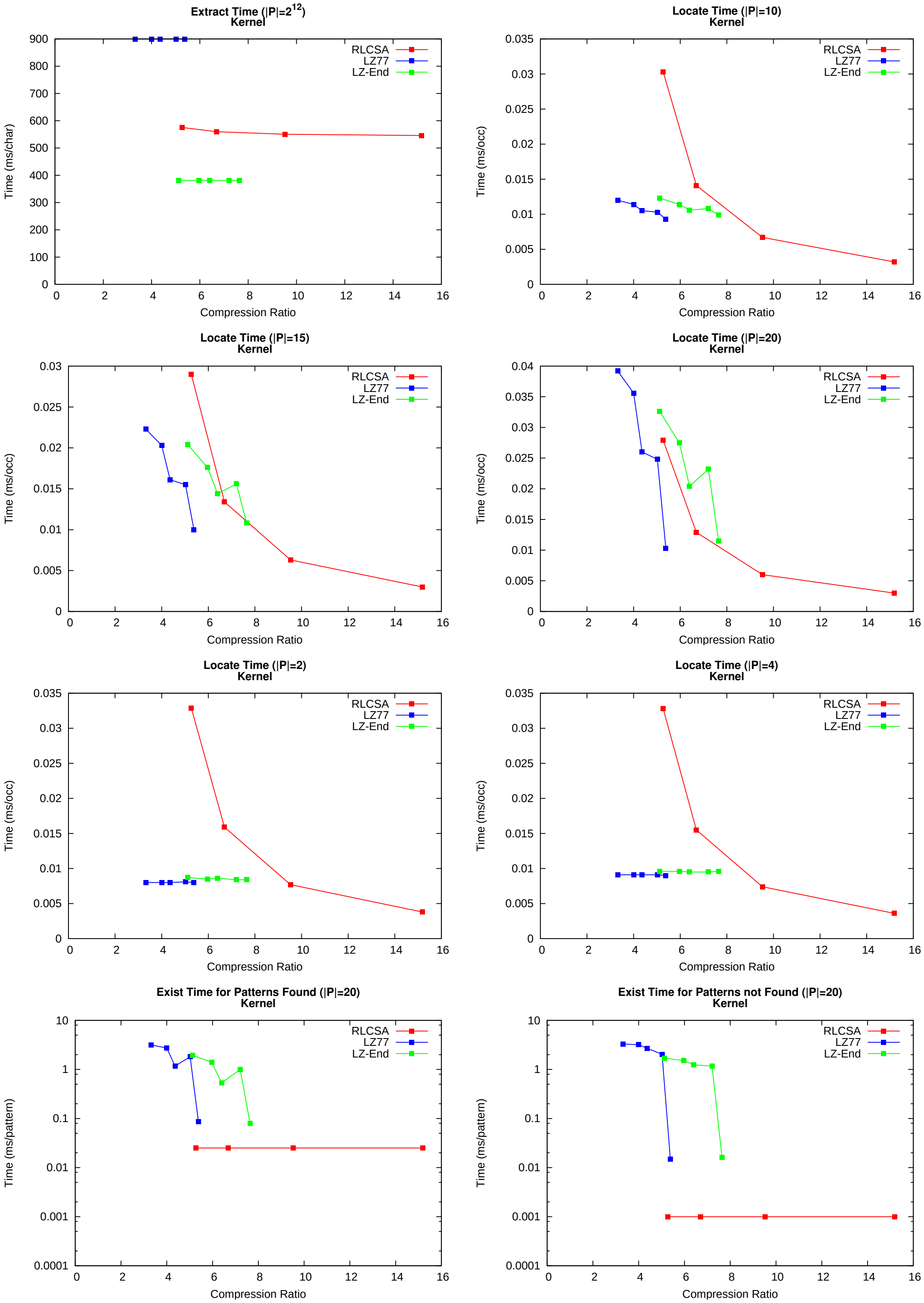


Figure 6.7: Kernel results (2). Note the logscales.

6.2 Analysis of the Results

It can be seen from the results presented above that our indexes are competitive with RLCSA and in most cases they show better space/time trade-offs.

Figure 6.1 shows that our indexes are built more efficiently than RLCSA. The space needed to build the LZ77 index is about 60% of that of RLCSA, and for the case of the LZ-End index the space is about 85% of that of RLCSA. The construction time for LZ77 is about 35% of that of RLCSA, yet for LZ-End it is about 185% of that of RLCSA (i.e., slower). Our space occupancy during construction is a great advantage against RLCSA as it allows us to build the index for larger texts using the same resources. We could reduce construction space further by sacrificing construction time, recall Section 4.4.

The compression ratio of our indexes is usually superior to that of RLCSA (see Table 6.1). When considering the lower bound of RLCSA, which only supports *count* and *exists* queries, our smallest index compresses better than RLCSA for all except texts influenza, coreutils and World Leaders. When considering RLCSA with a sampling step equal to 512, our compression is better for all except text coreutils. From now on we compare our indexes with the RLCSA with sampling step 512. The compression difference is more noticeable for artificial texts, where our compression is 100-1000 times better than RLCSA. For DNA collections Para and Cere our best compression space (always achieved with LZ77) is about 45% of RLCSA's, yet it raises to 70-80% for Influenza and Escherichia Coli. The space is also 80% on World Leaders. On kernel our space is 60% of RLCSA, but they are almost the same on Coreutils. On Wikipedia articles our space is 20-30% of that of RLCSA. LZ-End needs more space that LZ77, losing to RLCSA in Influenza, Escherichia Coli, Coreutils and World Leaders. For texts of pseudo-real collections the compression ratio of LZ77 is about 60% of that of RLCSA, and even the alternative using more space has better compression ratios than RLCSA. It can also be seen in Figures 6.2-6.7 and A.1-A.26 labeled "results (2)" that in most cases the space/time trade-off is in favor of LZ77/LZ-End. Our LZ77 indexes use 2.63-7.52 times the space of p7zip and our LZ-End indexes use 4.57-23.03 times the space, depending on the type of text and the number of structures we use (e.g., tries). Finally, the compression ratios of LZ78-based indexes are not competitive at all (note that ILZI's maximal parsing performs better than LZ78).

We expected to improve the compression of RLCSA for highly repetitive texts, since LZ77 is more powerful to detect repetitions (see Lemma 4.1 and Section 2.14.1). For artificial texts, the most repetitive ones, this situation is even more clear. For such texts, most of the space of the RLCSA corresponds to the space to represent the samplings, which is difficult to compress [MNSV10].

Tables 6.2 and 6.3 show that D is generally moderate (below 42), and that the greatest extraction cost is also moderate (at most 257 steps in LZ-End and at most 98 steps in LZ77 except for the texts of Wikipedia). When taking into account the mean values of the depth and extraction cost, the values decrease noticeably, being the average extraction cost below 25 steps for LZ-End and below 32 steps for LZ77 (except for the texts of Wikipedia).

Tables 6.4 and 6.5 show that the suffix trie and the reverse trie use more space than the rest of the data structures. Then we see that the range structure, the reverse ids and the permutation use roughly the same space (they require $n' \log n' + o(n' \log n')$ bits). For LZ-End we see that the space needed to represent both bitmaps increases noticeably, being even higher to that of the range structure, or the permutation. Finally, we see that the skips only use 11-13% and the wavelet tree of depths uses 2-4% of the total space. Additionally, for the artificial texts, we see that more than 90% of the space is used to represent the suffix trie and the reverse trie. This is because our implementation of DFUDS stores a boosting table of constant size 1kiB, and for these texts the number of phrases n' is less than 100, being the space of the table considerably larger than the space of the remaining data structures. We also note that our implementation stores the labels of the tree using 1 byte for each symbol.

The extract time of LZ-End index is better than RLCSA in all texts, being at least twice as fast, and up to 10 times faster for short passages. The LZ77 index extracts substrings of length up to 50 faster than RLCSA. When taking into account the space/time trade-offs (top-left plot of Figures 6.2-6.7 and A.1-A.26 labeled "results (2)") we see that our indexes improve RLCSA both in extraction time and space, dominating the curve defined by RLCSA, excepting the texts where LZ-End loses in space, in which case no one dominates. The extraction space/time trade-off is better than that of RLCSA, because since RLCSA cannot compress the sampling, it has to use a sparse sampling to be competitive in space.

The performance of locate queries is related to the pattern length. This is because in our indexes the locating cost is quadratic on the pattern length (yet, this is in the worst case; in practice many searches are abandoned earlier). It can be seen in plots 2-6 of Figures 6.2-6.7 and A.1-A.26 labeled "results (2)" that for patterns of length 2 or 4 all of our indexes are significantly faster than RLCSA. This is because our indexes are much faster to locate each occurrence, as this is the cost that dominates for short patterns. However, when we increase the length of the patterns, the increase in cost is noticeable for the alternatives using binary search, which are those using least space. However, the alternative number 1, using tries, shows a time basically independent of $|P|$ (except somewhat in DNA 0.1%[1] and Escherichia Coli). This is also seen in the top-right plot of Figures 6.2-6.7 and A.1-A.26 labeled "results (1)" . RLCSA time is almost insensitive to $|P|$, thus in several cases it becomes faster for longer patterns (which also have fewer occurrences, for reporting which RLCSA is slower).

By analyzing the performance results of SLPs [CFMPN10] it is clear that the compression ratio of SLPs (at least when using Re-Pair to create the grammar) is worse than that of RLCSA. For the case of DNA (Para, Cere and Influenza) the compression ratio is more than twice that of the LZ77 indexes. Furthermore, the locate time of SLPs is only comparable to RLCSA for small patterns ($|P| \leq 6$), in which case our indexes show a space/time trade-off much better than that of RLCSA.

Finally, we have that *exists* queries are solved consistently faster by RLCSA than by our indexes. Looking at plots 3 and 4 of Figures 6.2-6.7 and A.1-A.26 labeled "results (1)" we notice that our larger variants are comparable to RLCSA, although always slower, in the case of patterns present in the text. The difference widens in the case of non-existent patterns, as RLCSA improves more sharply. Moreover, in our indexes the time increases with the length, opposite to RLCSA where the time is practically constant when the pattern does not exist. In plots 7 and 8 of Figures 6.2-6.7 and A.1-A.26 labeled "results (2)" one can see the trade-off of *exists* queries. They show that binary search is not an alternative if we are interested in this type of queries. For the case of patterns present in the text, binary searching the queries takes about 10 times more, and for patterns not present in the text about 10-1000 times more, than the time needed using tries.

Chapter 7

Conclusions

We have presented a new full-text self-index based on the Lempel-Ziv parsing. This index is especially well suited for applications in which the text is highly repetitive and the user is interested in finding patterns in the text (*locate*) and accessing arbitrary substrings of the text (*extract*). Our indexes provide a much better space/time tradeoff than the previous ones for these operations.

The compression ratio of our indexes is more than 10 times better than previous indexes based on LZ78, which are shown to be inappropriate for very repetitive texts. Additionally, the compression ratio of our smallest index is, for almost all texts (13 out of 16)[1], better than the lower bound achievable using RLCSA [SVMN08], the best previous self-index for these texts. When compared to the smallest practical RLCSA, the compression of our index is better for all except one text, usually by a factor of 2 at least. Compared to pure LZ77 compression, our index takes usually 3-6 times the space achieved by p7zip.

We also introduced a new LZ-parsing called LZ-End, which is close to LZ77 in compression ratio and gives faster access to text substrings. The extraction speed when using LZ-End is always better than that of RLCSA, and the extraction speed of our LZ77-based index is also superior for small substrings. Our indexes are always better for locating the occurrences of short patterns (of length up to 10), and the results are mixed for longer ones. This is because our locating time is quadratic and depends also in the extraction speed, which shows different behaviors according to the text.

[1] We could not devise any characteristic that explains why the compression ratio of RLCSA for those texts is superior to our indexes.

The only operation for which RLCSA is consistently better than our indexes is for answering if a pattern is present in the text (*exists*), the difference being even more notorious for the case of non-existent patterns. Similarly, our indexes cannot count the number of occurrences without locating them all, whereas the RLCSA can do this very fast. Nevertheless, it has been argued [AN10] that these two queries are used in much more specific applications serving as a basis for complex tasks such as approximate pattern matching or text categorization, while extracting and locating are the most important for general applications.

An interesting goal for future research would be to reduce the m^2 factor of the *locate* query time to just m. This improvement would make our index even more attractive. This has been achieved for other LZ-based indexes [AN07, RO08], yet these have been built on LZ parsings that are too weak for very repetitive texts.

Another line of research would be to design new LZ-like compression schemes allowing fast decompression of random substrings of the LZ parsing. Note that our only trade-off related to extraction speed is the use of LZ-End instead of LZ77, and still LZ-End takes constant time per extracted symbol only in certain cases. In a recent work, Kuruppu *et al.* [KPZ10] use a single document as the dictionary for the LZ77 algorithm, storing that source document in plain form. This method is a heuristic and works fairly well enough only when the documents of the collection are not successive versions, as in collections of DNA. However, even the compression achieved for DNA collections (para and cere) is almost the same than the compression we achieved using our best LZ77 variant, yet we have a self-index and they are only able to extract text, although their extraction times are more than 100 times faster than ours. Nevertheless, this method is orthogonal to our index proposal and we could build our self-index on top of their compression scheme.

Another important line of research is to devise an LZ parsing algorithm that uses space proportional to that of the final compressed text. Currently, to build the LZ77 parsing one needs about six times the space of the original text. Although this space is lower than that of RLCSA, is still too much to handle very large text collections. Alternatively, a parsing algorithm working in secondary storage would also be useful to handle very large collections. We are aware that this is a more than challenging task, as the parsing process is strongly related with dynamic self-indexes for repetitive texts. That is, if we have a dynamic self-index (or at least an index able to insert strings at the end) we can easily produce the LZ parsing of the text. Hence, studying how we can build a dynamic LZ-based index is a natural research direction.

It would be also interesting to study if counting could be answered more efficiently, and if more meaningful operations like approximate pattern matching could be implemented, or if some operations of the suffix tree could be simulated on the index.

Another interesting research goal is to decrease the space factor, both in theory and in practice. Compared to a pure LZ77 compressor, the factor is 4 in theory and 3-6 in practice, as mentioned. Such a reduction has been achieved for Arroyuelo and Navarro's LZ-index, reducing the factor from 4 [Nav04] to $(2 + \epsilon)$ [ANS06] (see Section 2.14.3). This was possible because there was some redundancy between the components of the index. We could also reduce the factor by coding the bitmaps of the wavelet tree of depths in compressed form [RRR02] (see Section 2.5), since most depths are very low in practice and only some are high. However, the space improvement would not be too impressive, since the space of wavelet tree of depths is just 2-4% of the index size.

We have also presented a text corpus oriented to repetitive texts. The main goal of this corpus is to become a reference set in experimentation with this kind of texts. The corpus is publicly available at `http://pizzachili.dcc.uchile.cl/repcorpus.html`.

Finally, our implementation has been left public in the site `http://pizzachili.dcc.uchile.cl/indexes/LZ77-index`, to promote its use in real-world and research applications and to serve as a baseline for future developments in repetitive text indexing.

Bibliography

[ACNS10] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. Succinct trees in practice. In *Proc. 11th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 84–97. SIAM Press, 2010.

[AN] Diego Arroyuelo and Gonzalo Navarro. Space-efficient construction of Lempel-Ziv compressed text indexes. Manuscript.

[AN07] Diego Arroyuelo and Gonzalo Navarro. Smaller and faster Lempel-Ziv indices. In *Proc. 18th International Workshop on Combinatorial Algorithms (IWOCA)*, pages 11–20. College Publications, UK, 2007.

[AN10] Diego Arroyuelo and Gonzalo Navarro. Practical approaches to reduce the space requirement of Lempel-Ziv-based compressed text indices. *ACM Journal of Experimental Algorithmics (ACM JEA)*, 2010. To appear.

[ANS06] Diego Arroyuelo, Gonzalo Navarro, and Kunihiko Sadakane. Reducing the space requirement of LZ-index. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009, pages 319–330, 2006.

[ANS10] Diego Arroyuelo, Gonzalo Navarro, and Kunihiko Sadakane. Stronger Lempel-Ziv based compressed text indexing. *Algorithmica*, 2010. To appear.

[AS99] Jean-Paul Allouche and Jeffrey Shallit. The ubiquitous Prouhet-Thue-Morse sequence. In *Proc. 1st International Conference on Sequences and their Applications (SETA)*, pages 1–16. Springer-Verlag, 1999.

[B$^+$08] David R. Bentley et al. Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, 456(7218):53–59, 2008.

[Ban09] Mohammad Banikazemi. LZB: Data compression with bounded references. In *Proc. 19th Data Compression Conference (DCC)*, page 436. IEEE Computer Society, 2009. Poster.

[BDM$^+$05] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[BLN09] Nieves Brisaboa, Susana Ladra, and Gonzalo Navarro. Directly addressable variable-length codes. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5721, pages 122–130. Springer, 2009.

[BM77] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[BW94] Michael Burrows and David Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[CFMPN10] Francisco Claude, Antonio Fariña, Miguel Martínez-Prieto, and Gonzalo Navarro. Compressed q-gram indexing for highly repetitive biological sequences. In *Proc. 10th IEEE Conference on Bioinformatics and Bioengineering (BIBE)*, pages 86–91. IEEE Press, 2010.

[Cha88] Bernard Chazelle. Functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[CIT08] Maxime Crochemore, Lucian Ilie, and Liviu Tinta. Towards a solution to the ”runs” conjecture. In *Proc. 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 290–302. Springer-Verlag, 2008.

[Cla96] David Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, 1996.

[CLL$^+$05] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[CN09] Francisco Claude and Gonzalo Navarro. Self-indexed text compression using straight-line programs. In *Proc. 34th International Symposium* *on Mathematical Foundations of Computer Science (MFCS)*, LNCS 5734, pages 235–246. Springer, 2009.

[CN10] Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 2010. to appear.

[CPS08] Gang Chen, Simon J. Puglisi, and William F. Smyth. Lempel-Ziv factorization using less time & space. *Mathematics in Computer Science*, 1(4):605–623, June 2008.

[FG89] Edward R. Fiala and Daniel H. Greene. Data compression with finite windows. *Communications of the ACM*, 32(4):490–505, 1989.

[FH07] Johannes Fischer and Volker Heun. A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In *Proc. 1st International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE)*, volume 4614 of *LNCS 4614*, pages 459–470. Springer-Verlag, 2007.

[FM05] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[FMMN07] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):article 20, 2007.

[FSS03] Frantisek Franek, R.J. Simpson, and William F. Smyth. The maximum number of runs in a string. In *Proc. Australian Workshop on Combinatorial Algorithms (AWOCA)*, pages 26–35, 2003.

[GBYS92] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: Pat trees and pat arrays. In *Information Retrieval: Data Structures & Algorithms*, pages 66–82. Prentice Hall, 1992.

[GGMN05] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38. CTI Press and Ellinika Grammata, 2005.

[GGV03] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850. SIAM Press, 2003.

[GN08] Rodrigo González and Gonzalo Navarro. Rank/select on dynamic compressed sequences and applications. *Theoretical Computer Science*, 410:4414–4422, 2008.

[GV05] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal of Computing*, 35(2):378–407, 2005.

[Ham86] Richard Wesley Hamming. *Coding and Information Theory*. Prentice-Hall, 1986.

[IT06] Shunsuke Inenaga and Masayuki Takeda. On-line linear-time construction of word suffix trees. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 60–71. Springer-Verlag, 2006.

[Jac89] Guy Jacobson. Space-efficient static trees and graphs. In *Annual IEEE Symposium on Foundations of Computer Science*, pages 549–554. IEEE Computer Society, 1989.

[Kär99] Juha Kärkkäinen. *Repetition-Based Text Indexes*. PhD thesis, Department of Computer Science, Univeristy of Helsinki, Finland, November 1999.

[KK99] Roman Kolpakov and Gregory Kucherov. On maximal repetitions in words. *Journal of Discrete Algorithms*, 1:159–186, 1999.

[KM99] S. Rao Kosaraju and Giovanni Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.

[KMP77] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.

[KN10] Sebastian Kreft and Gonzalo Navarro. LZ77-like compression with fast random access. In *Proc. 20th Data Compression Conference (DCC)*, pages 239–248, 2010.

[KPZ10] Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 201–206, 2010.

[KS03] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 2719, pages 943–955, 2003.

[KU96a] Juha Kärkkäinen and Esko 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] Juha Kärkkäinen and Esko Ukkonen. Sparse suffix trees. In *Proc. 2nd Annual International Conference on Computing and Combinatorics (COCOON)*, pages 219–230. Springer-Verlag, 1996.

[LM00] N. Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, 2000.

[Lot02] M. Lothaire. *Algebraic Combinatorics on Words*. Cambridge University Press, 2002.

[LZ76] Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

[Mai89] Michael G. Main. Detecting leftmost maximal periodicities. *Discrete Applied Mathematics*, 25(1-2):145–153, 1989.

[Man01] Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 32(2):262–272, 1976.

[MKI$^+$08] Wataru Matsubara, Kazuhiko Kusano, Akira Ishino, Hideo Bannai, and Ayumi Shinohara. New lower bounds for the maximum number of runs in a string. In *Proc. Prague Stringology Conference (PSC)*, pages 140–145, 2008.

[MM93] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[MN07] Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007. Special issue on “The Burrows-Wheeler Transform and its Applications”.

[MNSV10] Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[Mor68] Donald R. Morrison. Patricia-practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[MR01] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[MRRR03] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations. In *Proc. 30th International Colloquium on Automata, Languages and Computation (ICALP)*, LNCS 2719, pages 345–356. Springer, 2003.

[Mun86] J. Ian Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log n)$ time. *Journal of Computer System Sciences*, 33(1):66–74, 1986.

[Nav04] Gonzalo Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms*, 2(1):87–114, 2004.

[Nav08] Gonzalo Navarro. Indexing LZ77: The next step in self-indexing. *Keynote talk at Third Workshop on Compression, Text, and Algorithms*, 2008.

[Nav09] Gonzalo Navarro. Implementing the LZ-index: Theory versus practice. *ACM Journal of Experimental Algorithmics (JEA)*, 13:article 2, 2009.

[NM07] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[OS07] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM Press, 2007.

[OS08] Daisuke Okanohara and Kunihiko Sadakane. An online algorithm for finding the longest previous factors. In *Proc. 16th Annual European Symposium on Algorithms (ESA)*, pages 696–707. Springer-Verlag, 2008.

[PST07] Simon J. Puglisi, William F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):4, 2007.

[PWZ92] Eli Plotnik, Marcelo Weinberger, and Jacob Ziv. Upper bounds on the probability of sequences emitted by finite-state sources and on the redundancy of the Lempel-Ziv algorithm. *IEEE Transactions on Information Theory*, 38(1):66–72, 1992.

[RNO08] Luís M. S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. Fully-compressed suffix trees. In *Proc. 8th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 4957, pages 362–373, 2008.

[RO08] Luís M. S. Russo and Arlindo L. Oliveira. A compressed self-index using a Ziv-Lempel dictionary. *Journal of Information Retrieval*, 5(3):501–513, 2008. Special issue SPIRE 2006.

[RRR02] Rajeev Raman, Venkatesh Raman, and S. Srinivasa 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. SIAM Press, 2002.

[Ryt03] Wojciech Rytter. Application of lempel–ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.

[Sad03] Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294 – 313, 2003.

[SS82] James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982.

[SVMN08] Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo 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] Esko Ukkonen. Constructing suffix trees on-line in linear time. *Algorithmica*, 14(3):249–260, 1995.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[Wel84] Terry A. Welch. A Technique for High-Performance Data Compression. *Computer*, 17(6):8–19, 1984.

[Wil91] Ross N. Williams. An extremely fast ziv-lempel data compression algorithm. In *Data Compression Conference*, pages 362–371, 1991.

[WZ99] Hugh E. Williams and Justin Zobel. Compressing integers for fast file access. *Computer Journal*, 42(3):193–201, 1999.

[ZdMNBY00] Nivio Ziviani, Edleno Silva de Moura, Gonzalo Navarro, and Ricardo Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, 2000.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[ZL78] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

Appendix A

Experimental Results

In this appendix we present the results of the experiments described in Section 6.1 for the remaining texts.

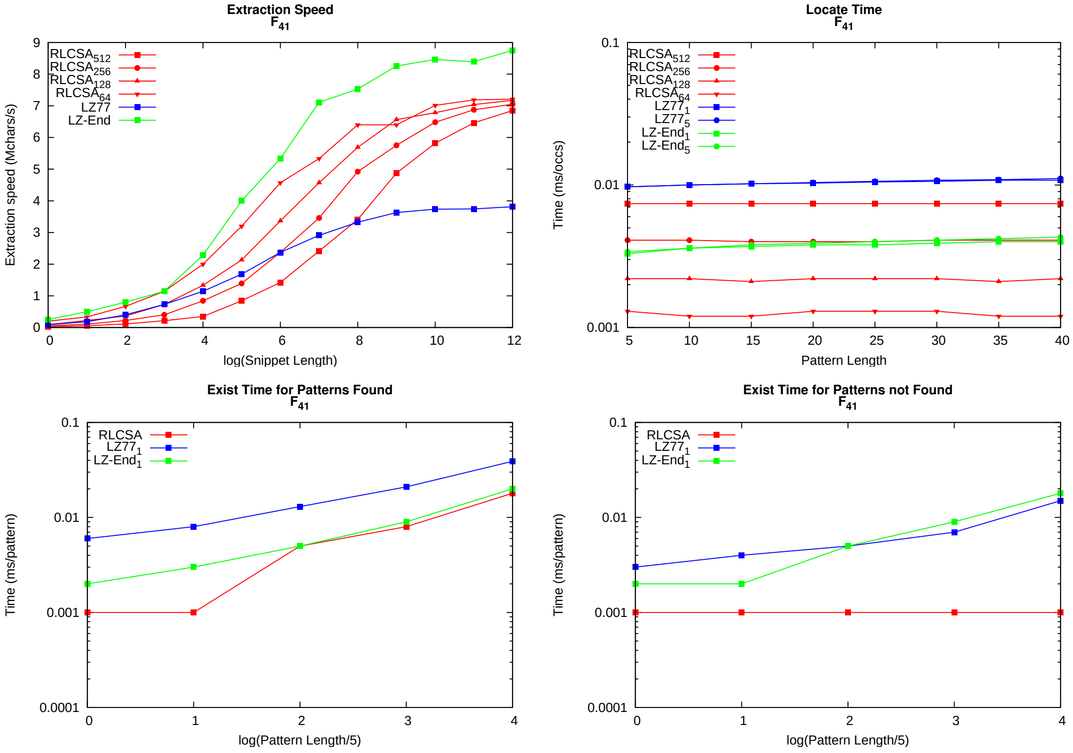


Figure A.1: F_{41} results (1). Note the logscales.

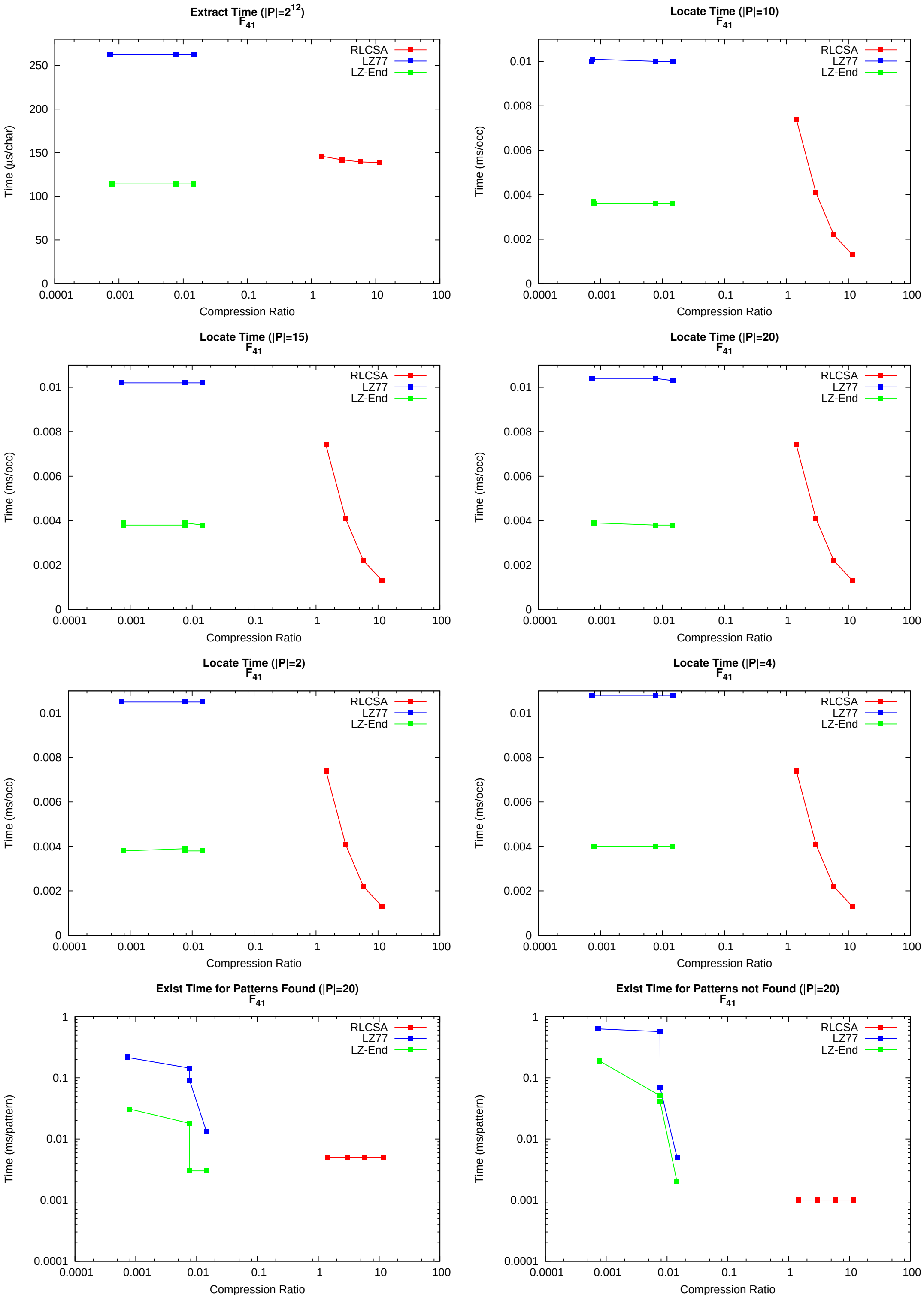


Figure A.2: F_{41} results (2). Note the logscales.

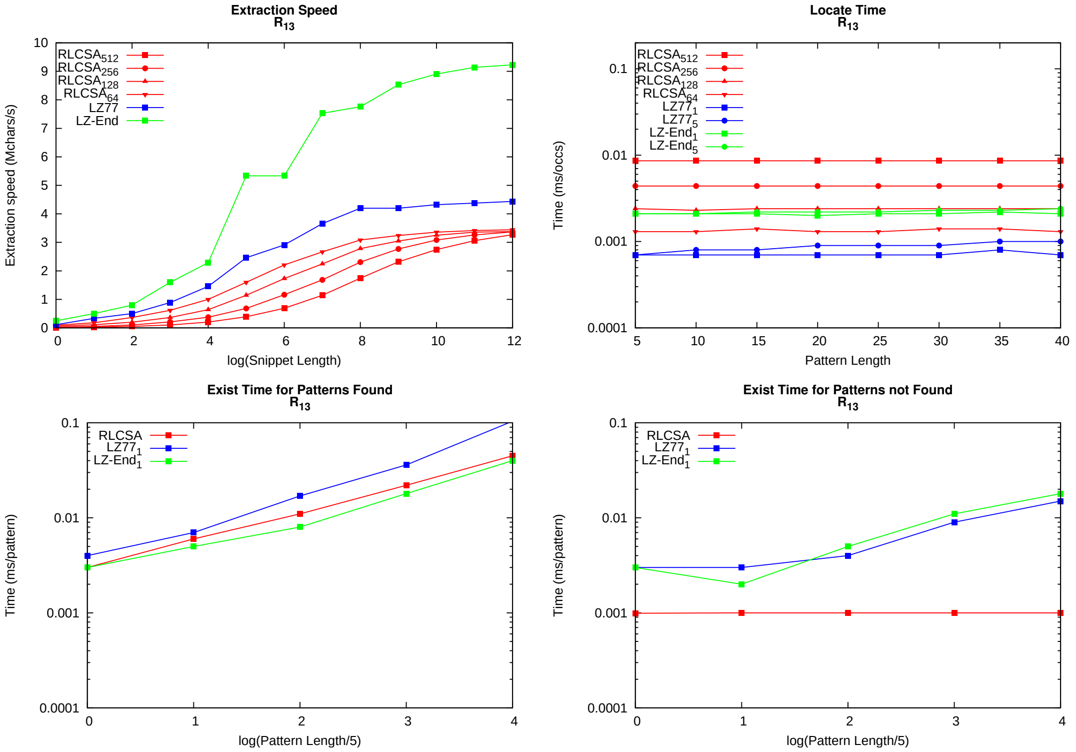


Figure A.3: R_{13} results (1). Note the logscales.

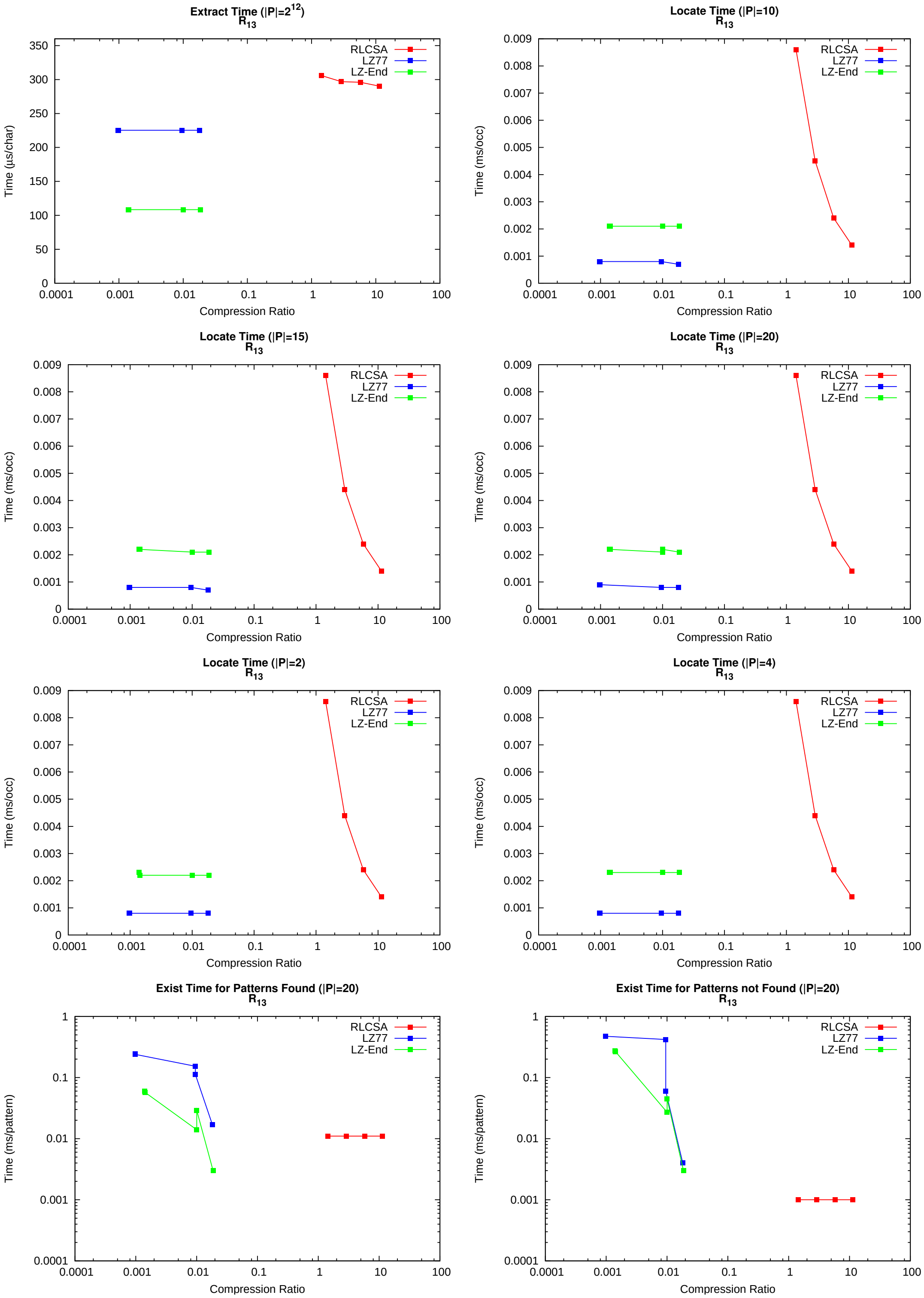


Figure A.4: R_{13} results (2). Note the logscales.

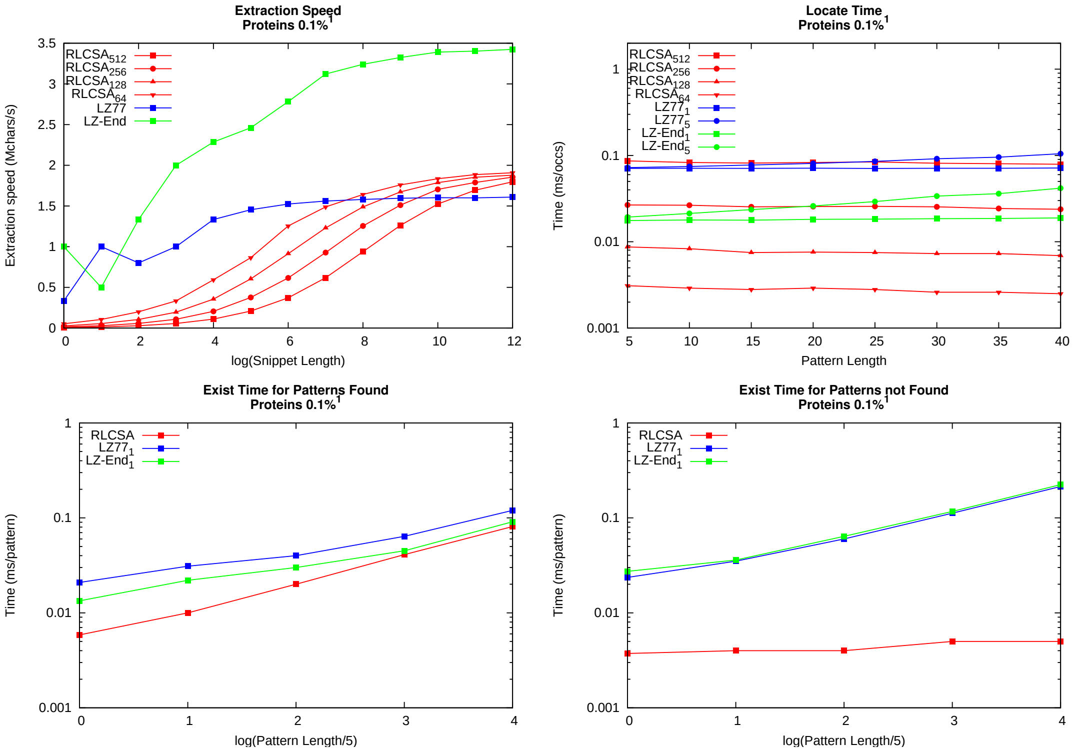


Figure A.5: Proteins 0.1% [1] results (1). Note the logscales.

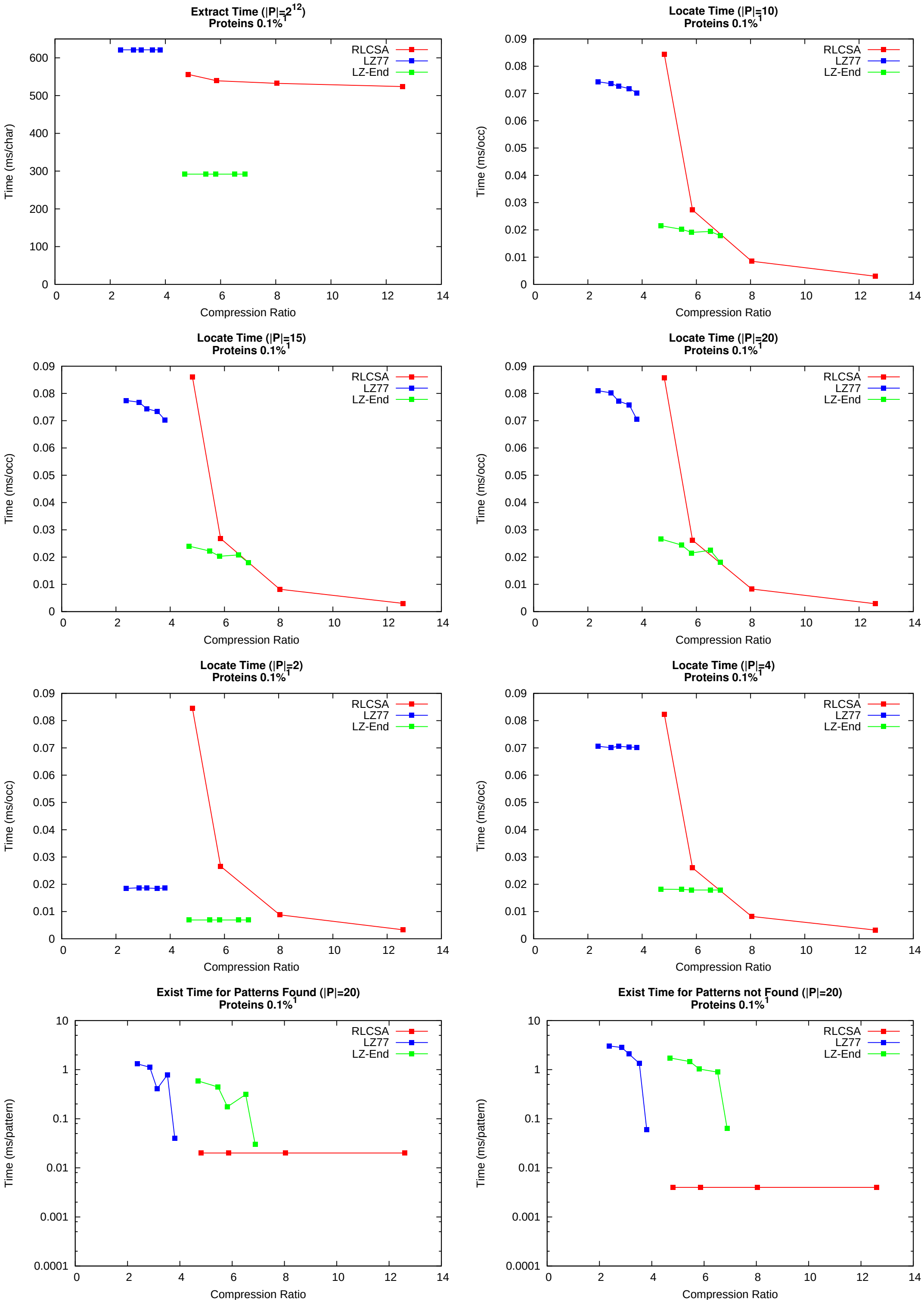


Figure A.6: Proteins 0.1% [1] results (2). Note the logscales.

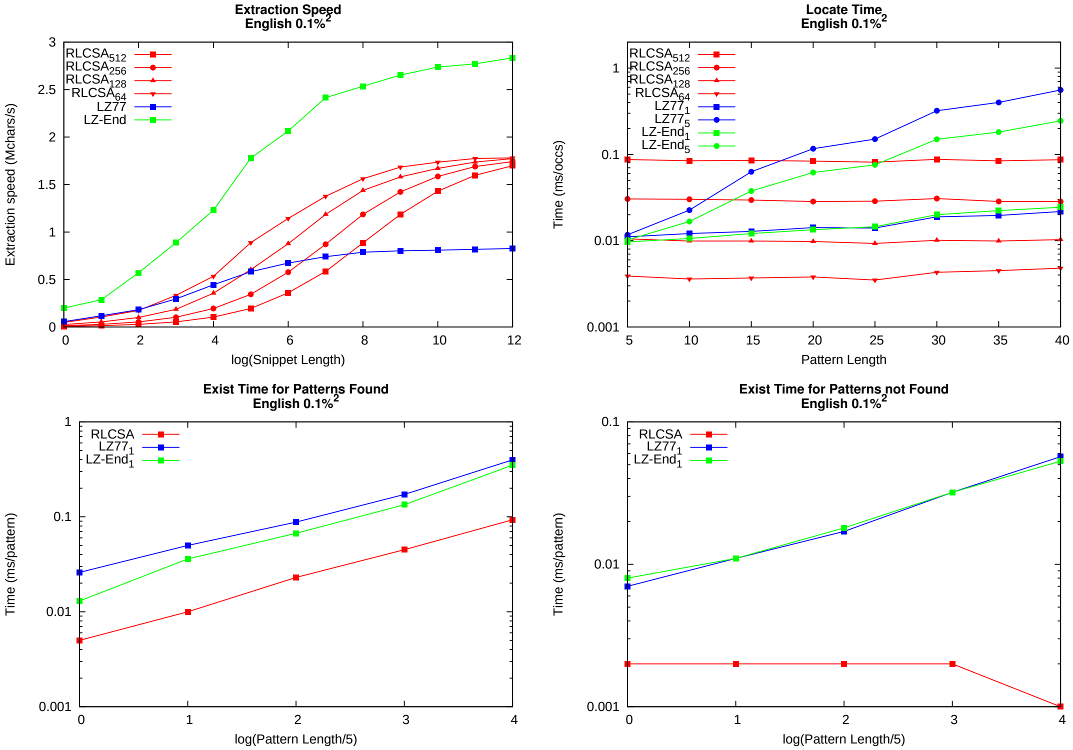


Figure A.7: English 0.1% ^{2} results (1). Note the logscales.

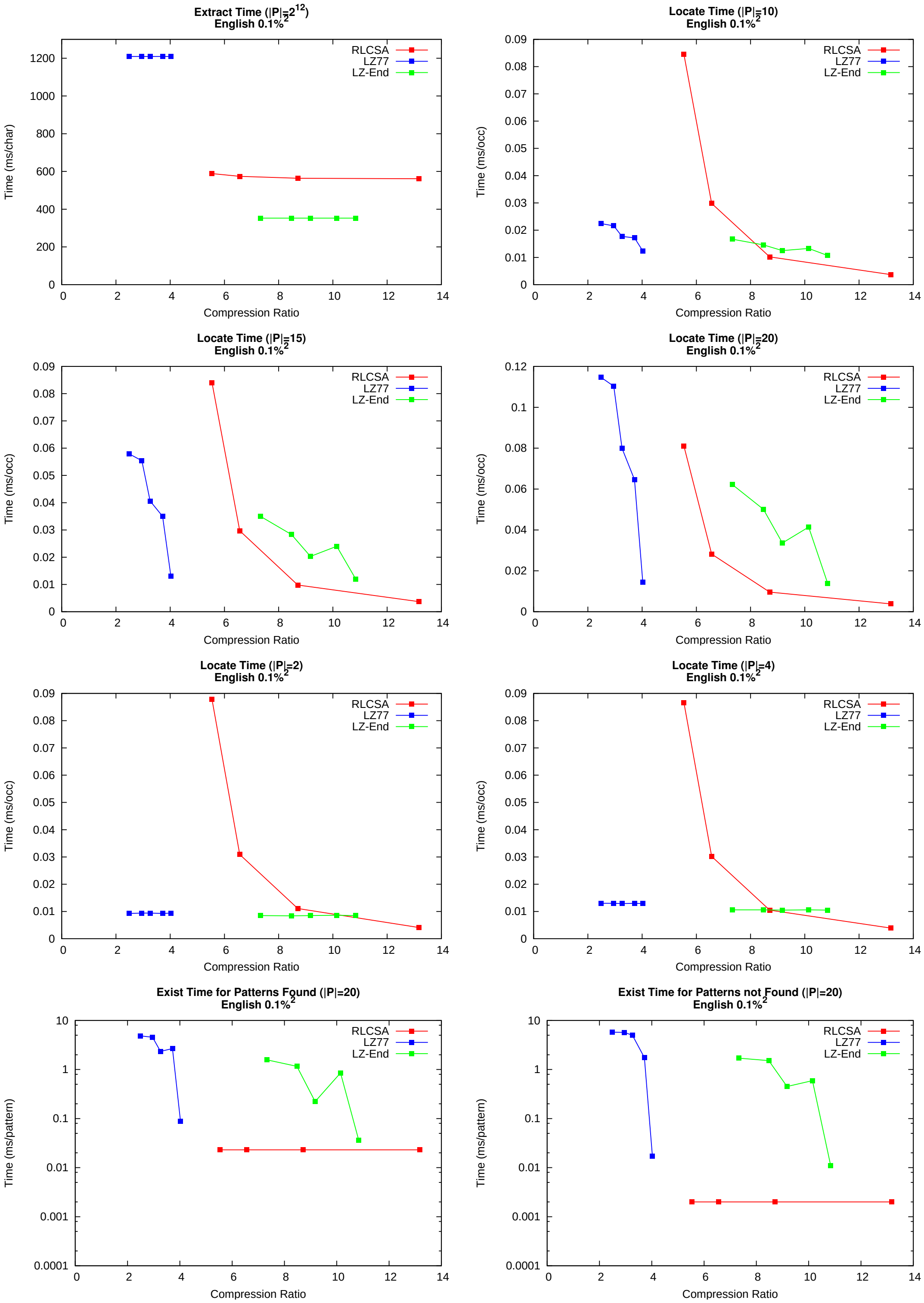


Figure A.8: English 0.1% ^{2} results (2). Note the logscales.

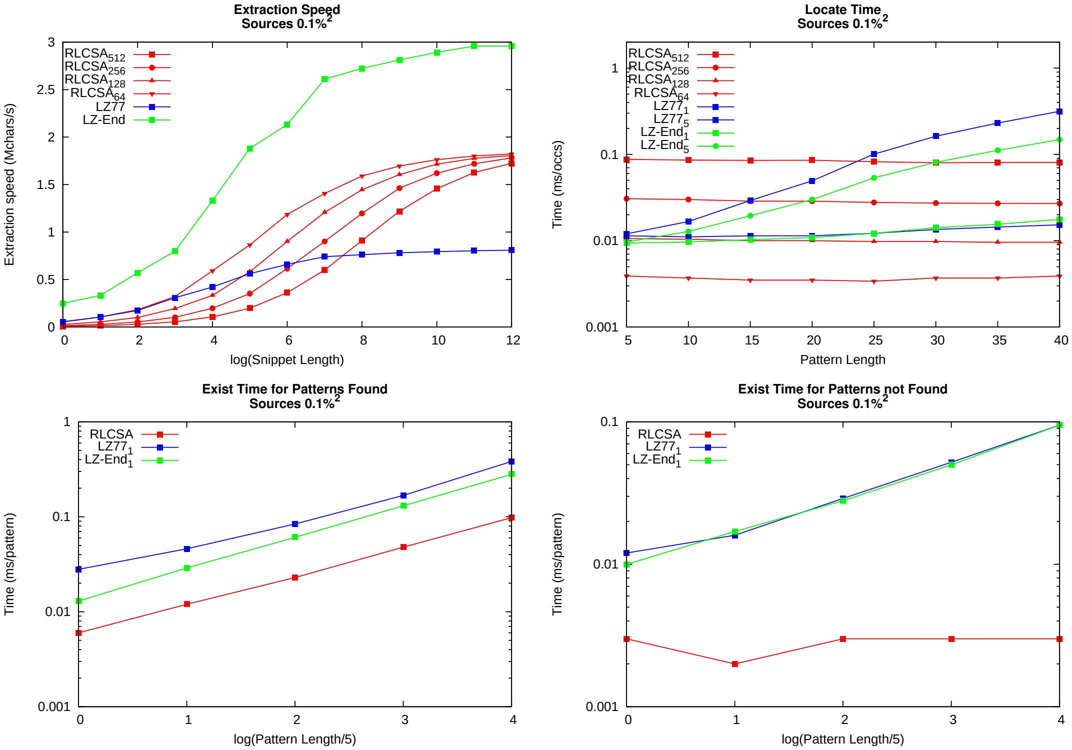


Figure A.9: Sources 0.1% ^{2} results (1). Note the logscales.

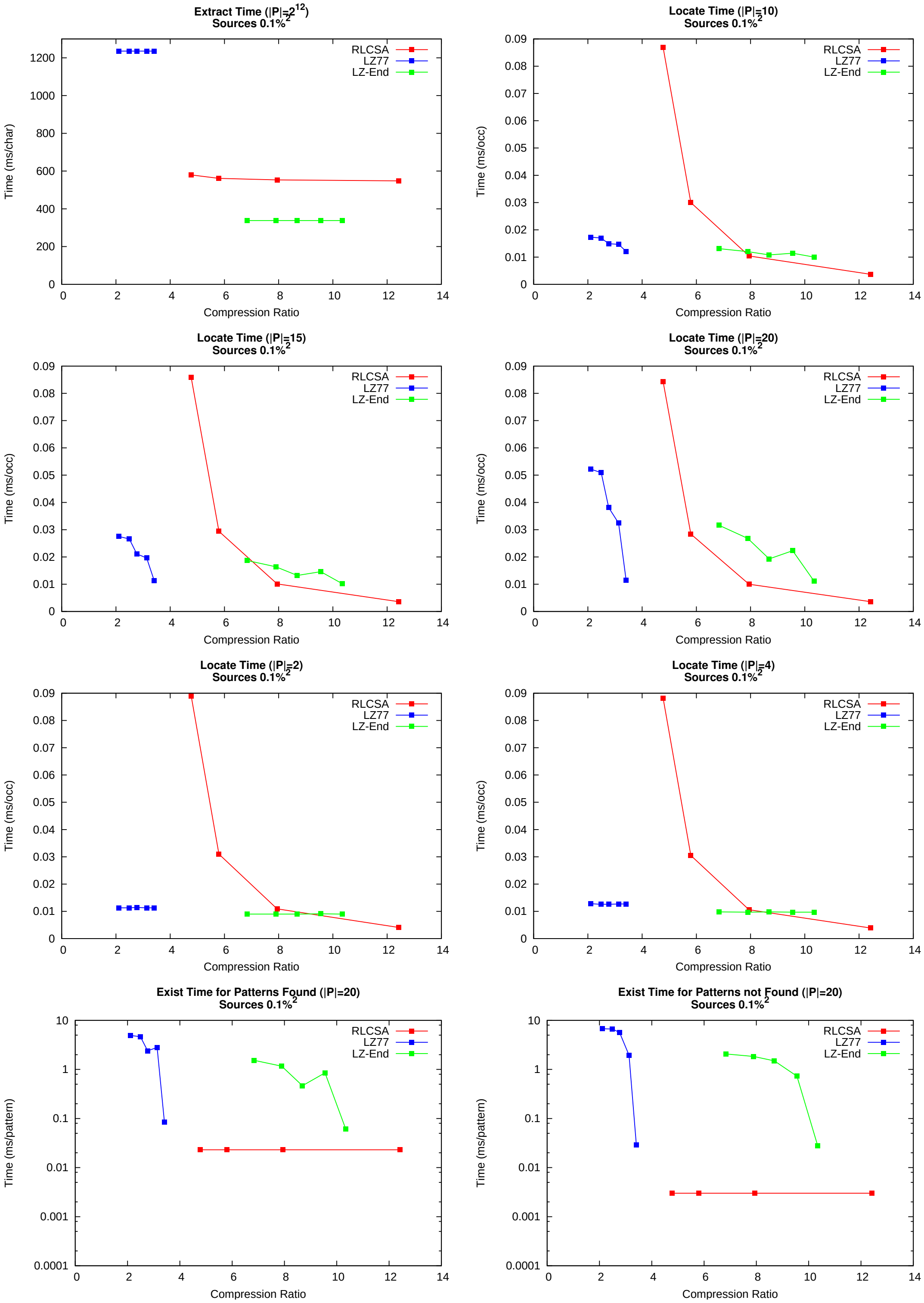


Figure A.10: Sources 0.1% ^{2} results (2). Note the logscales.

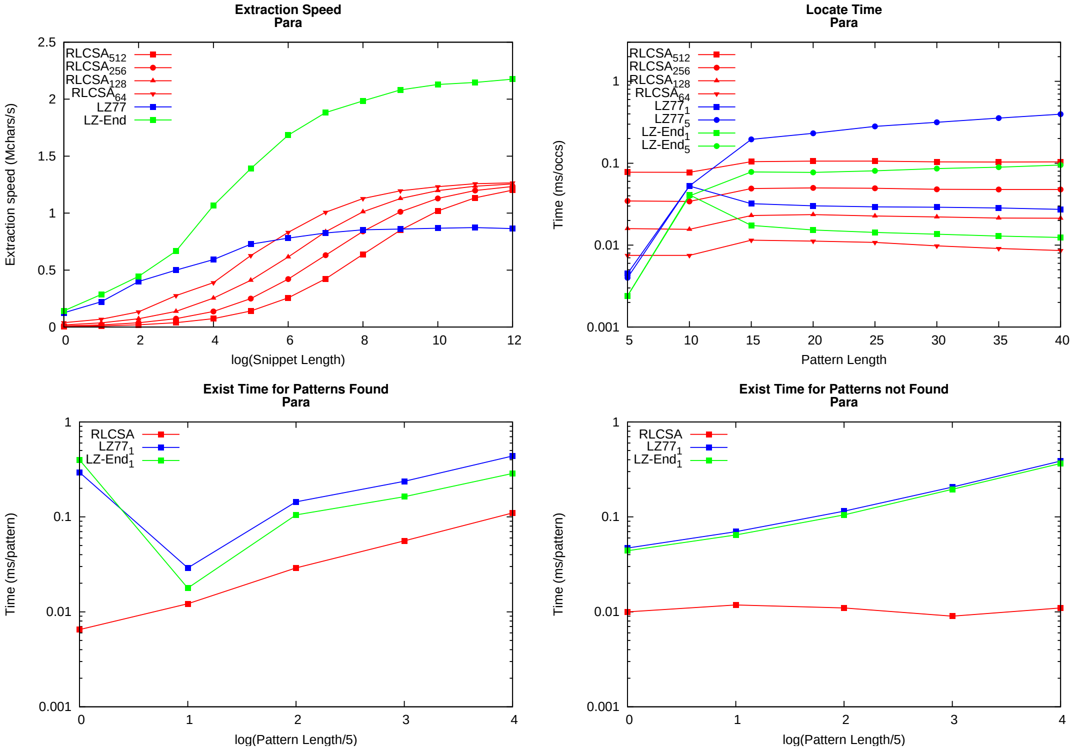


Figure A.11: Para results (1). Note the logscales.

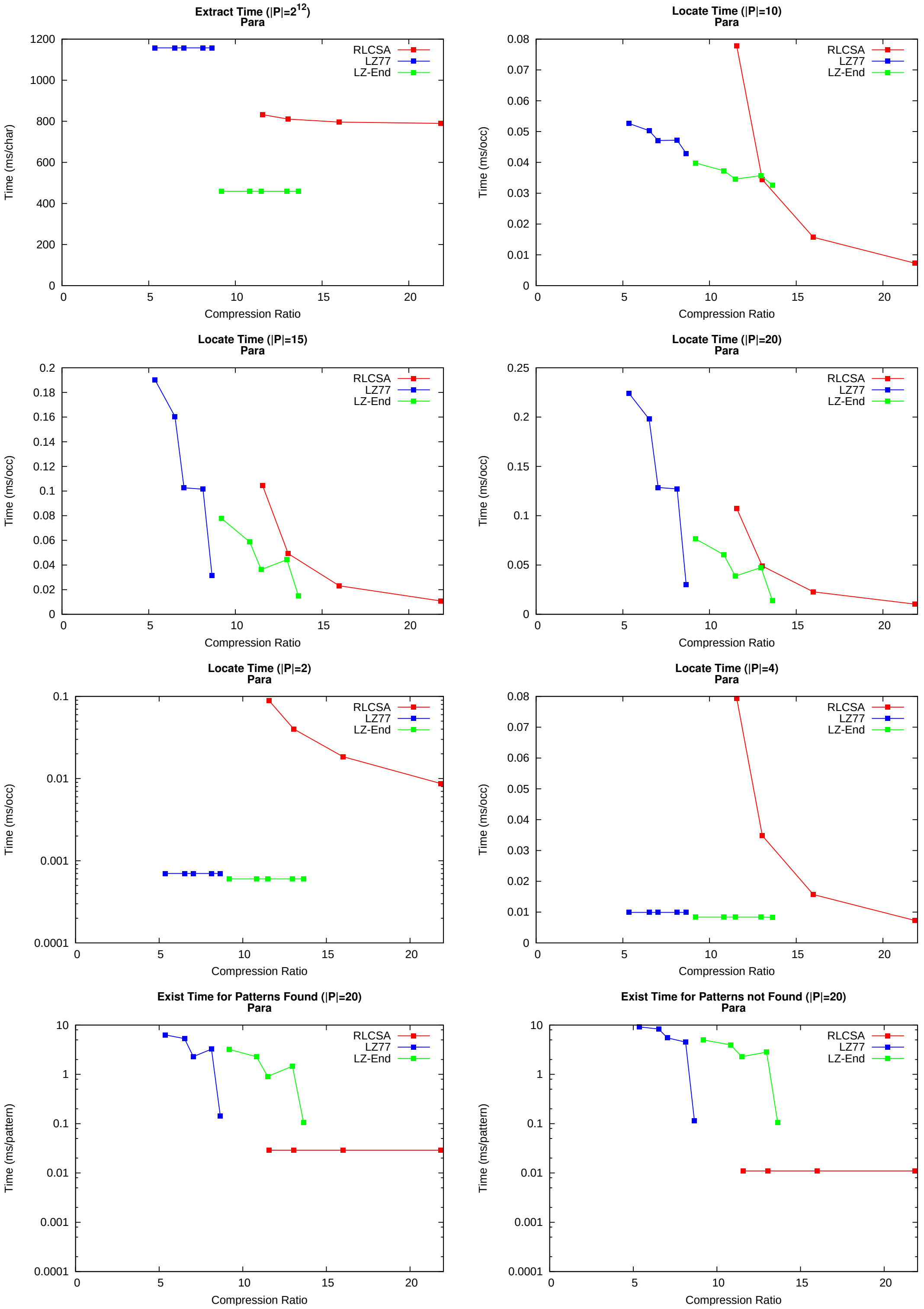


Figure A.12: Para results (2). Note the logscales.

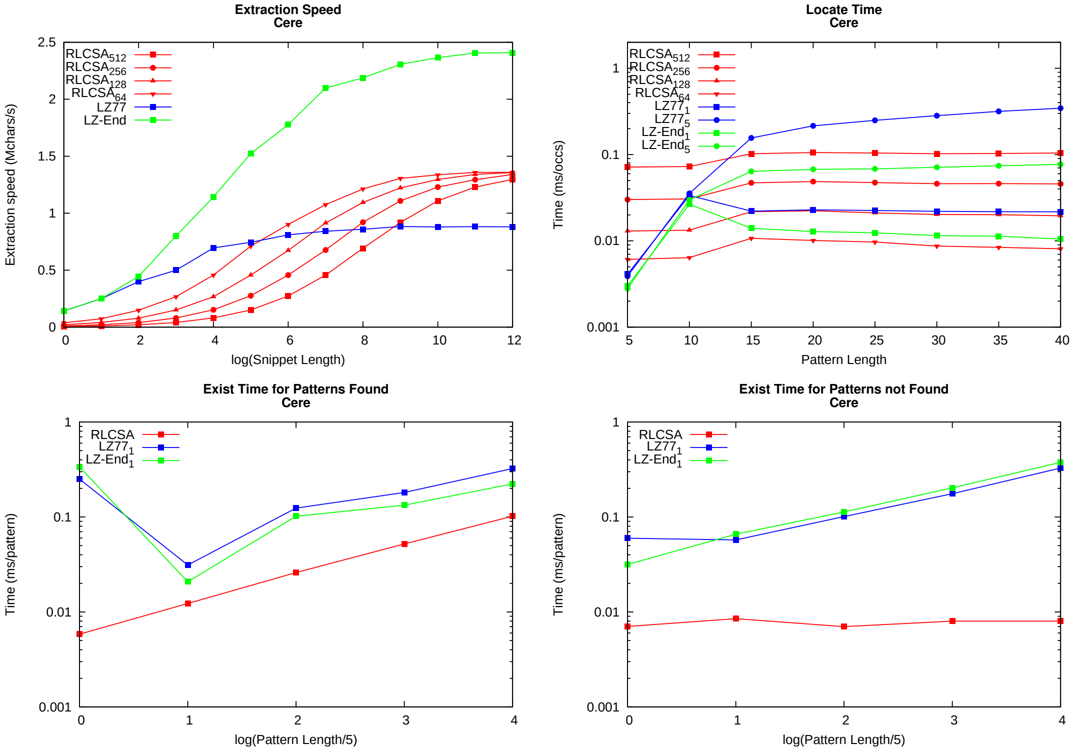
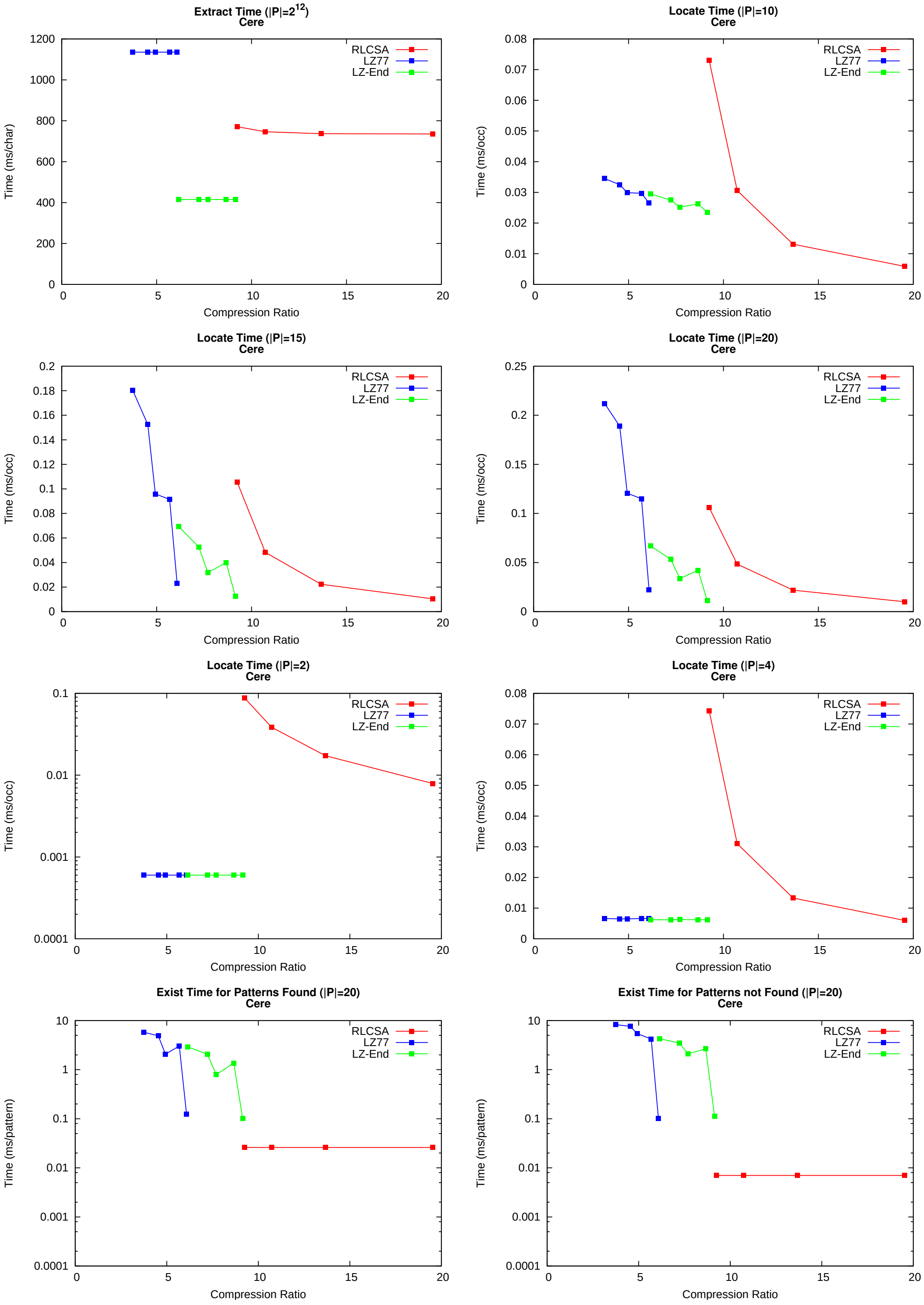


Figure A.13: Cere results (1). Note the logscales.


Extract Time (|P|=2^{12}) Cere

Locate Time (|P|=10) Cere

Locate Time (|P|=15) Cere

Locate Time (|P|=20) Cere

Locate Time (|P|=2) Cere

Locate Time (|P|=4) Cere

Exist Time for Patterns Found (|P|=20) Cere

Exist Time for Patterns not Found (|P|=20) Cere

Figure A.14: Cere results (2). Note the logscales.

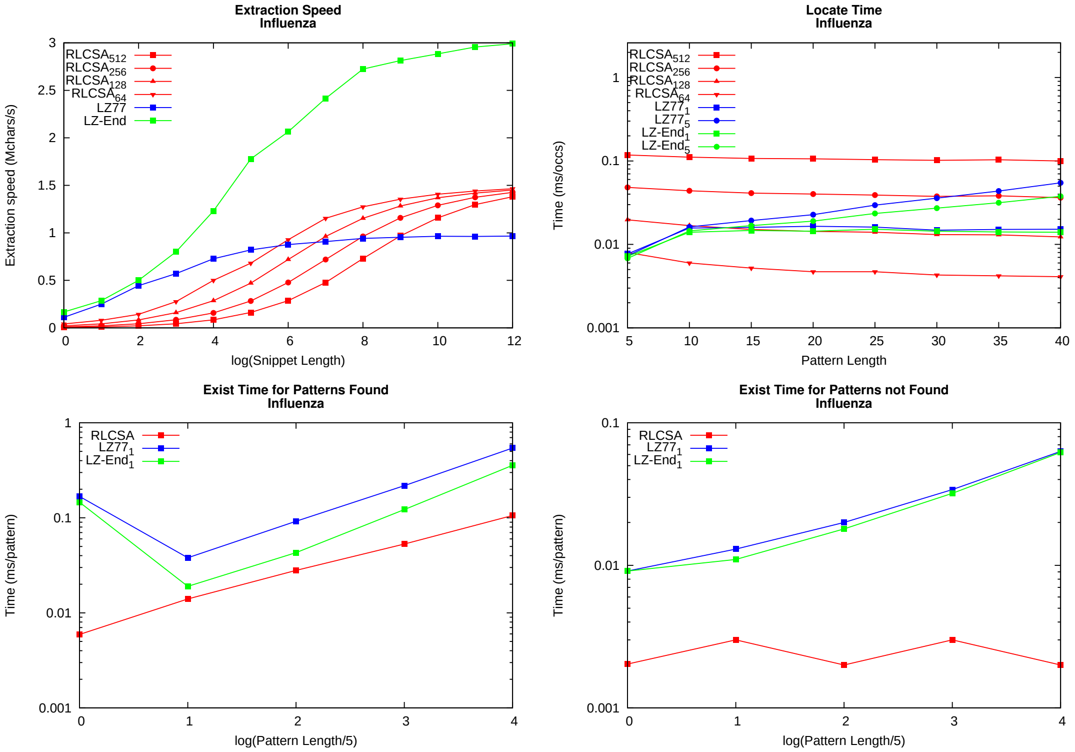


Figure A.15: Influenza results (1). Note the logscales.

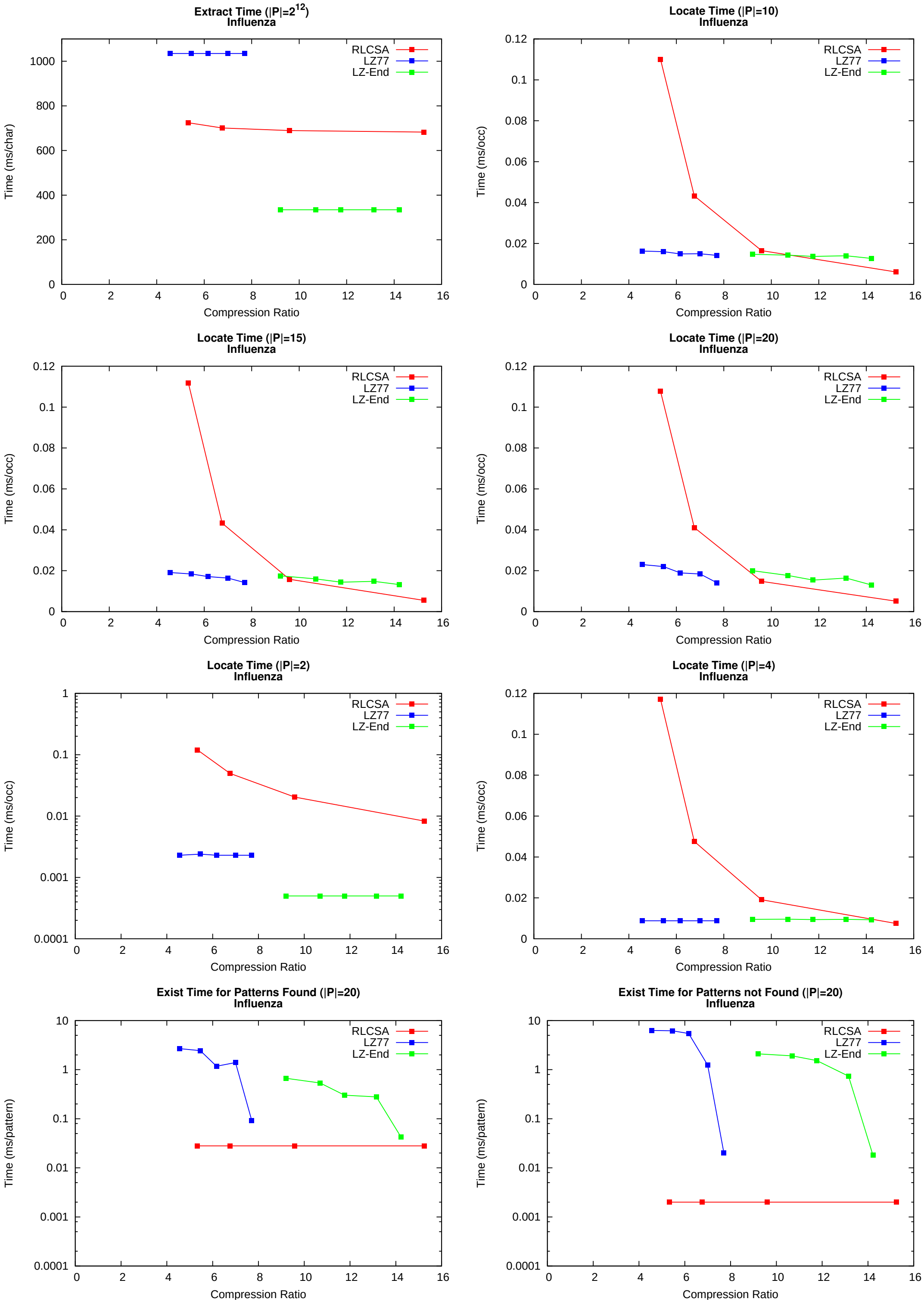


Figure A.16: Influenza results (2). Note the logscales.

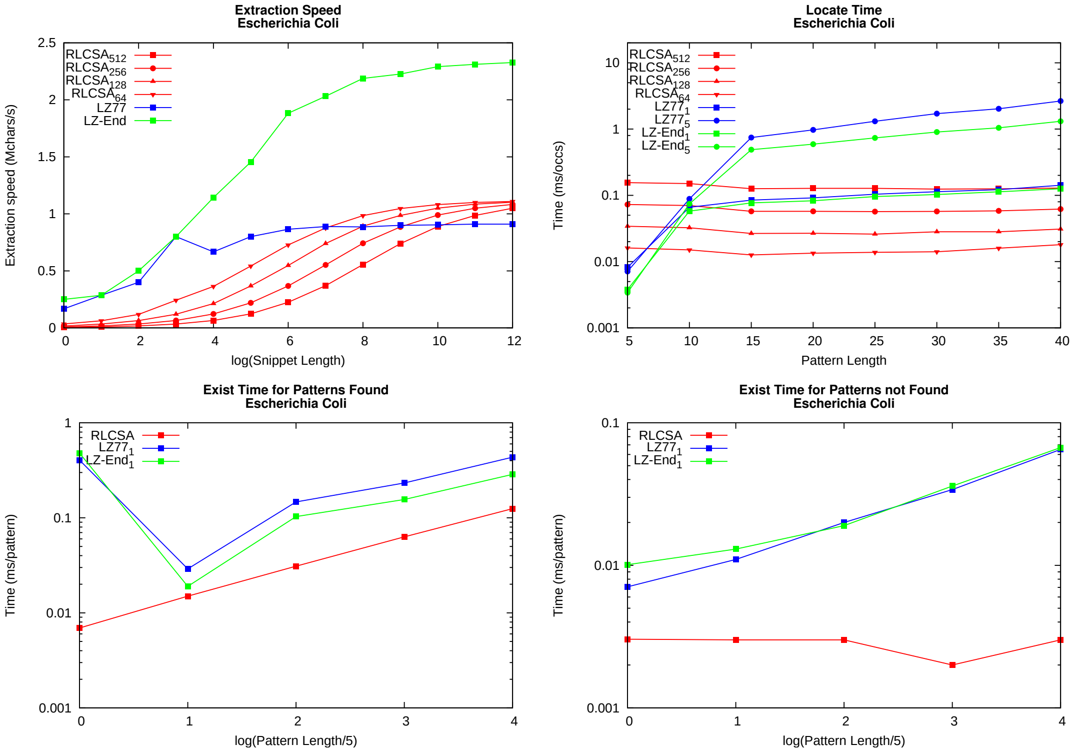


Figure A.17: Escherichia Coli results (1). Note the logscales.

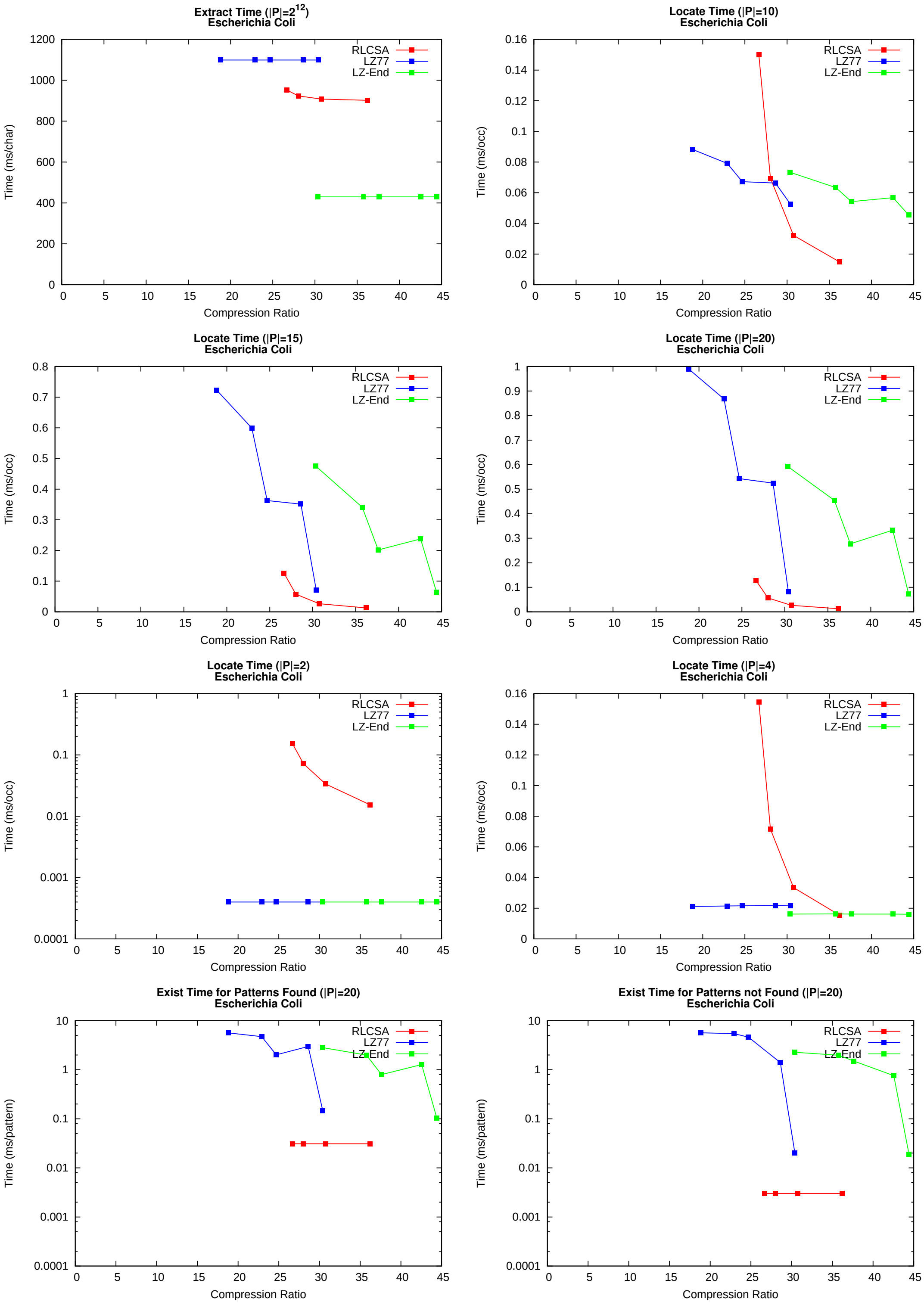


Figure A.18: Escherichia Coli results (2). Note the logscales.

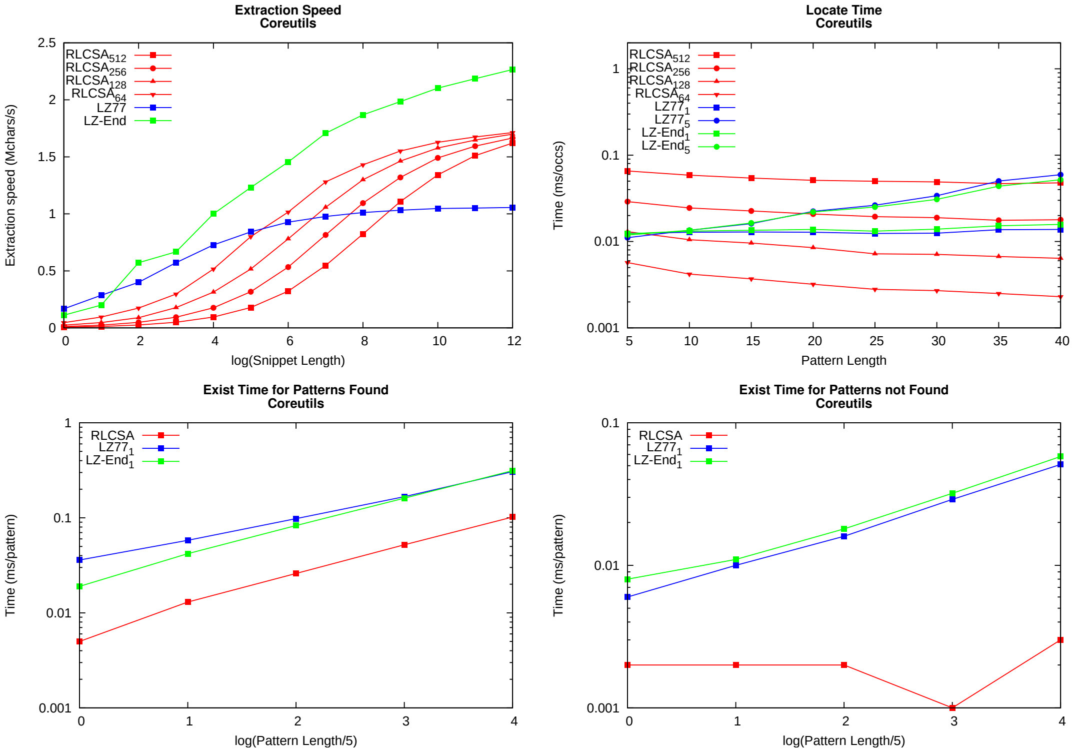


Figure A.19: Coreutils results (1). Note the logscales.

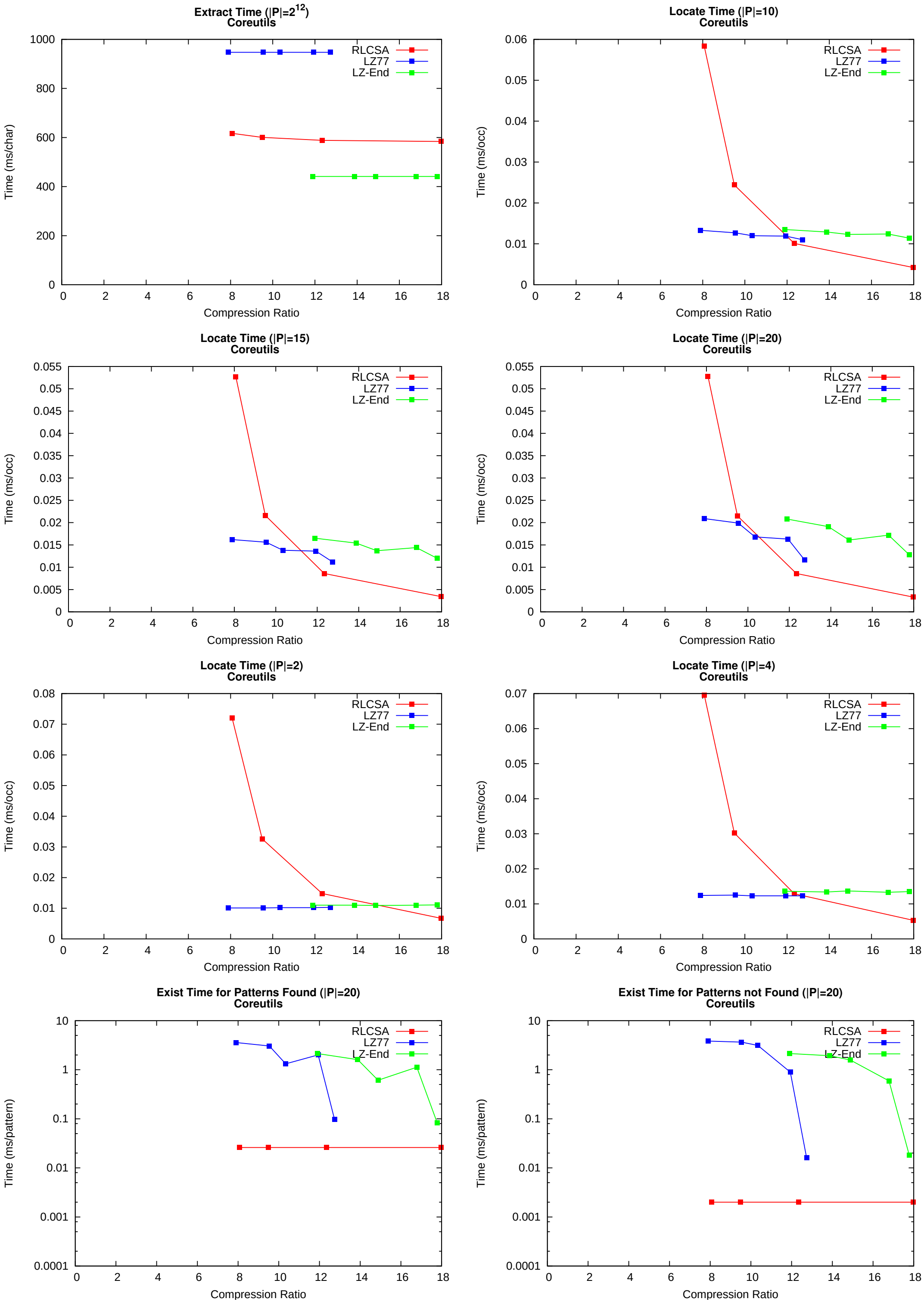


Figure A.20: Coreutils results (2). Note the logscales.

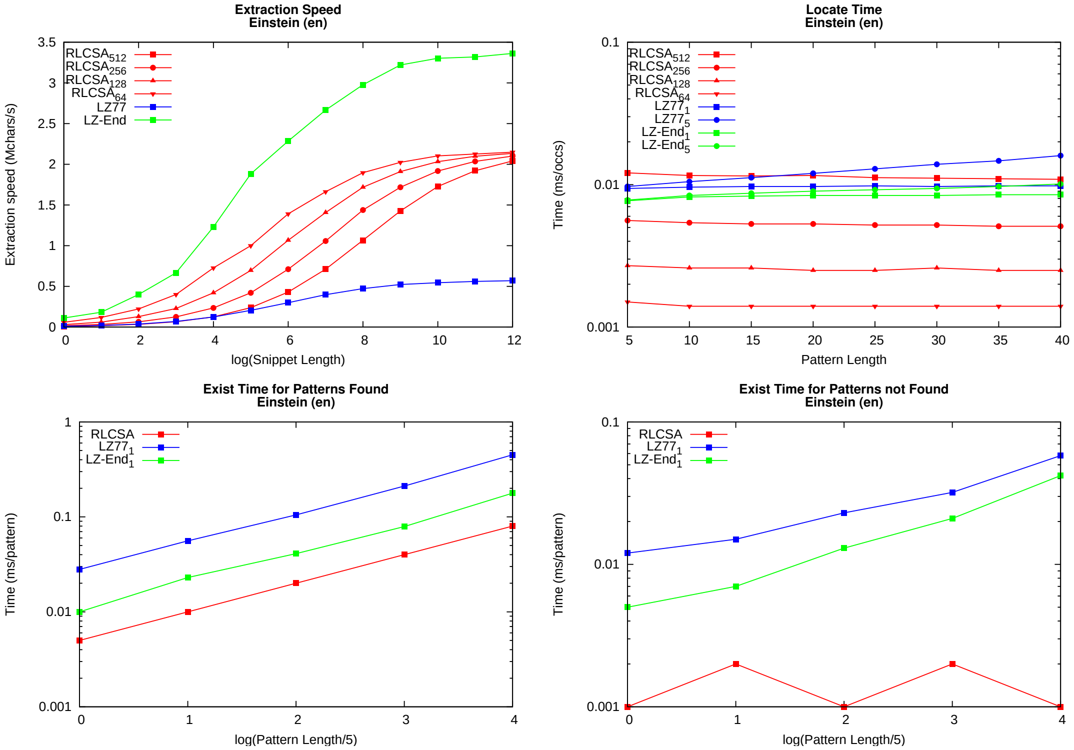


Figure A.21: Einstein (en) results (1). Note the logscales.

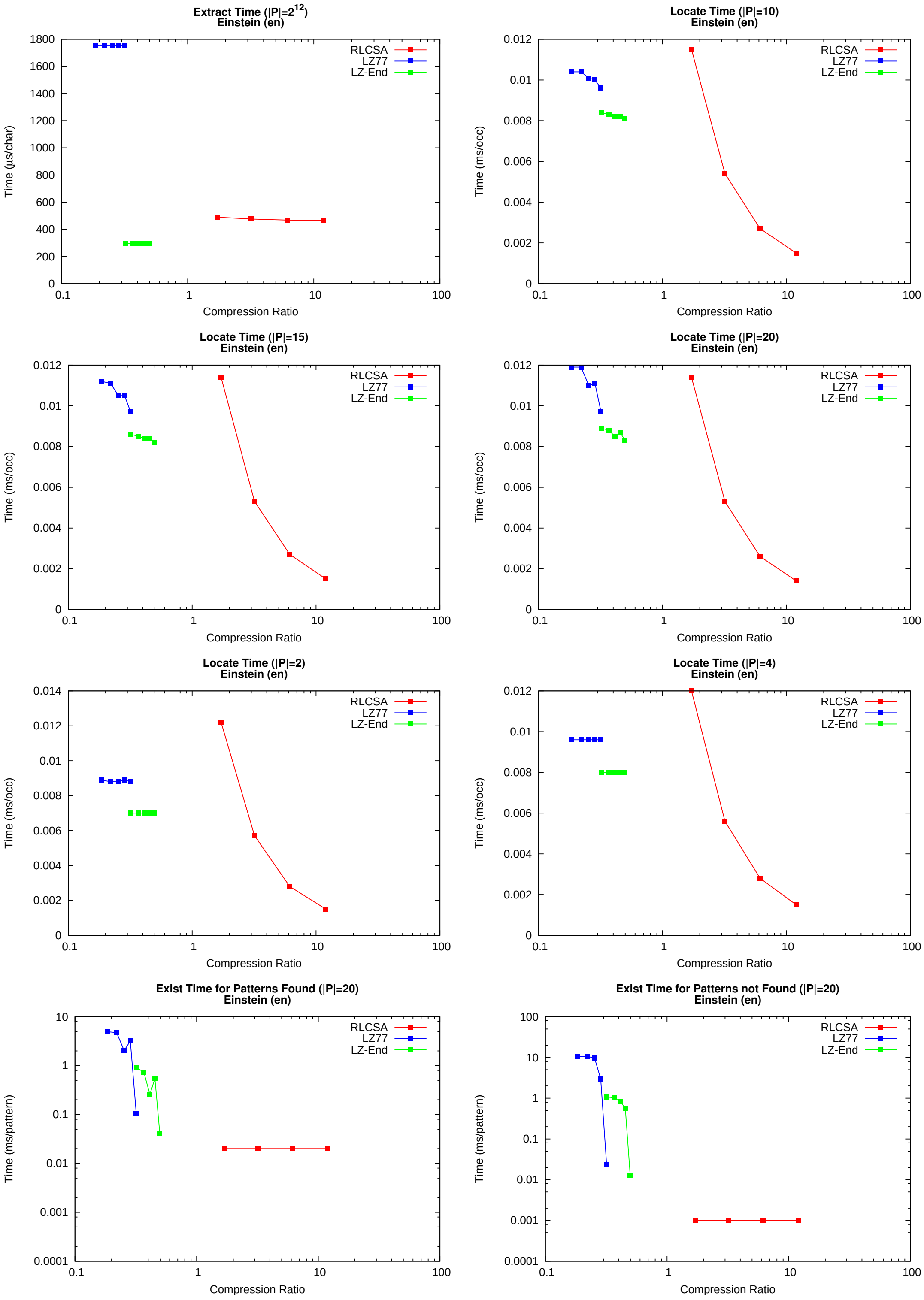


Figure A.22: Einstein (en) results (2). Note the logscales.

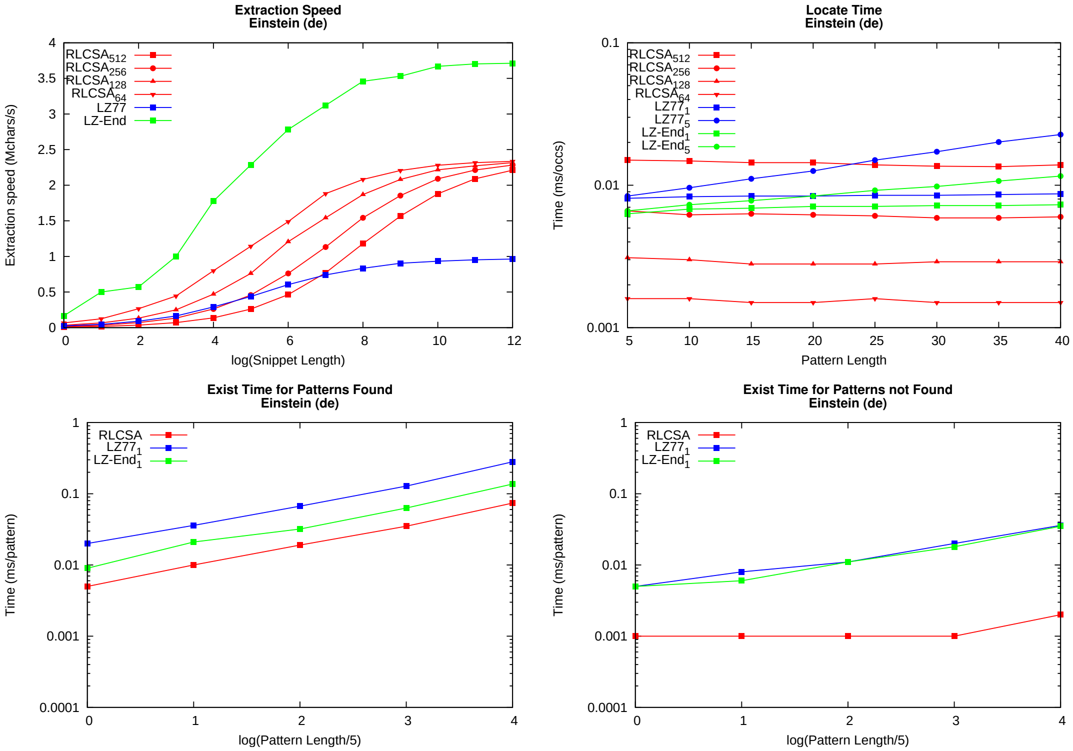


Figure A.23: Einstein (de) results (1). Note the logscales.

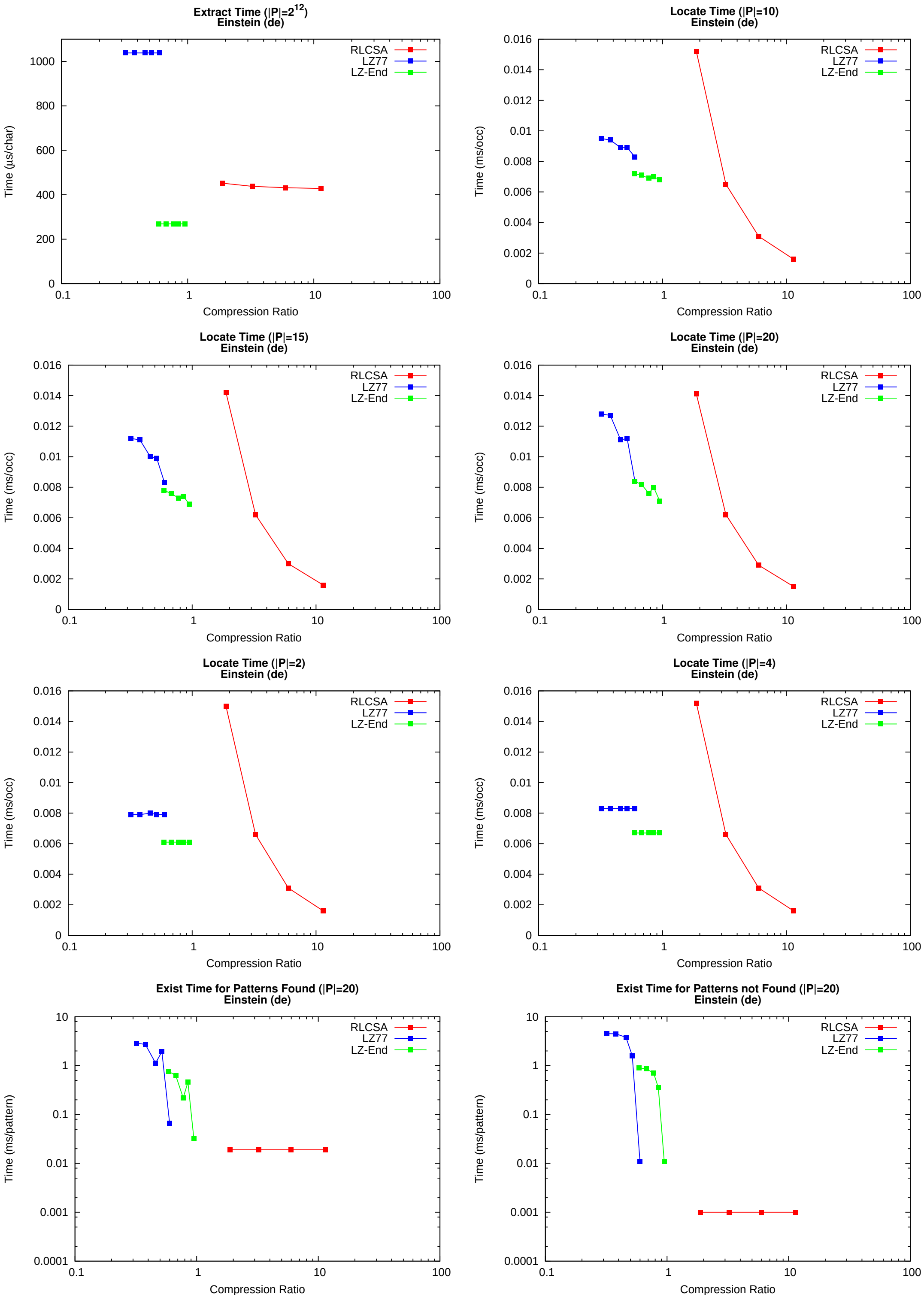


Figure A.24: Einstein (de) results (2). Note the logscales.

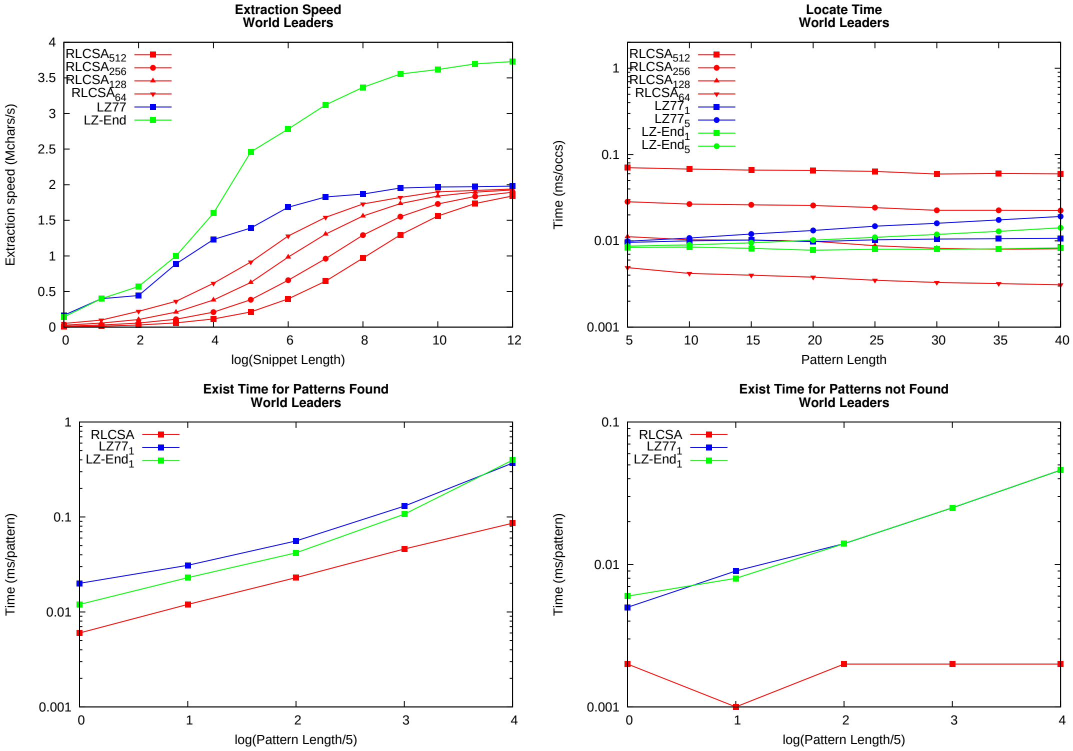


Figure A.25: World Leaders results (1). Note the logscales.

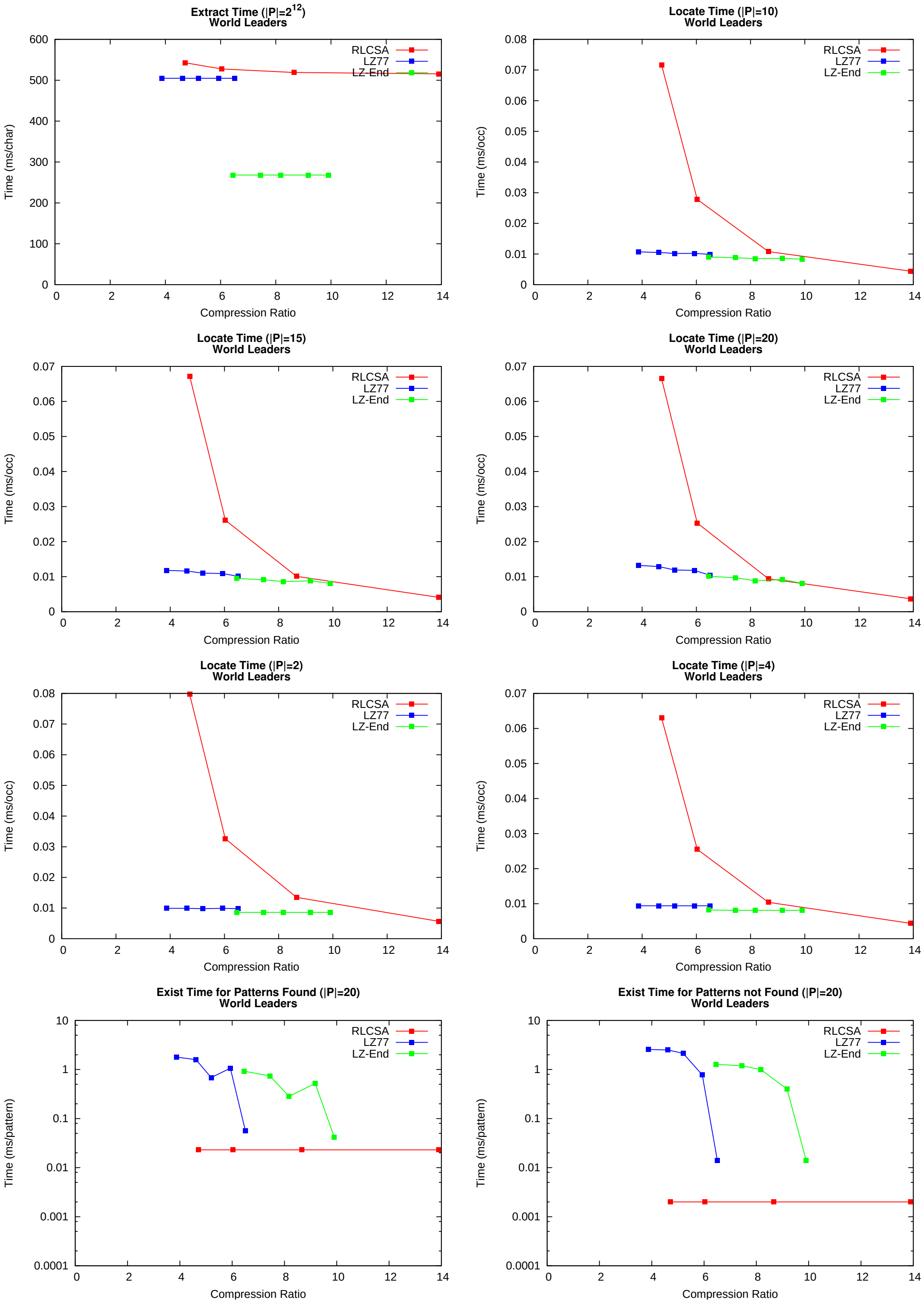


Figure A.26: World Leaders results (2). Note the logscales.