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

AUTOÍNDICES COMPRIMIDOS PARA TEXTO

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

RODRIGO RENATO GONZÁLEZ DEL BARRIO

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

MIEMBROS DE LA COMISIÓN:
JÉRÉMY BARBAY
BENJAMIN BUSTOS CÁRDENAS
PATRICIO POBLETE OLIVARES
KUNIHIKO SADAKANE

SANTIAGO DE CHILE
DICIEMBRE 2008

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

AUTOÍNDICES COMPRIMIDOS PARA TEXTO

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

RODRIGO RENATO GONZÁLEZ DEL BARRIO

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

MIEMBROS DE LA COMISIÓN:
JÉRÉMY BARBAY
BENJAMIN BUSTOS CÁRDENAS
PATRICIO POBLETE OLIVARES
KUNIHIKO SADAKANE

Esta tesis ha recibido el apoyo de MECESUP, Proyecto UCH 0109, Chile; del Núcleo Milenio Centro de Investigación de la Web, Proyecto P04-067-F, Mideplan, Chile; de Yahoo! Research Latin America; y del Instituto Milenio de Dinámica Celular y Biotecnología, Proyecto P05-001-F, Mideplan, Chile.

SANTIAGO DE CHILE
DICIEMBRE 2008

Resumen

Los autoíndices de texto comprimidos ofrecen una funcionalidad similar a la de los índices clásicos ocupando espacio proporcional al tamaño del texto comprimido, y además lo reemplazan, pues son capaces de reproducir cualquier subcadena del texto. Aunque un índice comprimido es más lento que su versión clásica, puede funcionar en memoria principal en casos en que un índice tradicional tendría que recurrir a la memoria secundaria, que es órdenes de magnitud más lenta. Por otra parte, los autoíndices de texto comprimidos actuales sufren de varias deficiencias, como la falta de practicidad, la lentitud para localizar un patrón y para extraer un texto, y la falta de mecanismos de construcción eficientes en espacio, de versiones en memoria secundaria o de capacidades para actualizar el índice. Esta tesis aporta soluciones para todos estos problemas.

Nuestra primera contribución es una estructura de datos para arreglos de bits, sencilla y eficiente, que permite las consultas de *rank* y *select*, y que se ha hecho muy popular por su practicidad. También se creó el sitio *Pizza&Chili*, que contiene una colección de textos y de índices comprimidos, y un estudio práctico que compara los índices más prometedores. Cabe destacar que este sitio se ha convertido en una referencia habitual en la comunidad.

Se desarrolló un nuevo índice de texto comprimido basado en regularidades del arreglo de sufijos, el cual permite localizar ocurrencias rápidamente, y aún es más pequeño que los índices clásicos. Esta estructura se basa en Re-Pair, un compresor que posee propiedades de localidad que no tienen los índices comprimidos clásicos.

Se desarrolló un codificador estadístico de secuencias, que permite el acceso directo a cualquier parte de la secuencia y logra una compresión de alto orden. Esta es una herramienta clave para lograr velocidad y localidad en la extracción de texto en un índice comprimido.

Aprovechando esta localidad en la localización y en la extracción, se presentó un nuevo índice para memoria secundaria, cuyo tiempo de acceso mejora gracias a la compresión, en lugar de empeorar como es lo normal en otros autoíndices. Este índice ofrece un compromiso muy competitivo entre espacio y tiempo.

Por último, se desarrolló una nueva estructura para secuencias dinámicas comprimidas, que permite *rank* y *select*. Esta estructura reúne las mejores características de trabajos previos, y se utilizó para obtener uno de los mejores resultados en índices de texto comprimidos dinámicos (es decir, que permiten actualizaciones al texto). También se aplicó en la construcción eficiente de índices de texto comprimidos.

University of Chile

Faculty of Physics and Mathematics
Graduate School

Compressed Full-Text Self-Indexes

by

Rodrigo González

Submitted to the University of Chile in fulfillment
of the thesis requirement to obtain the degree of
Ph.D. in Computer Science

Advisor	:	**Gonzalo Navarro**
Committee	:	Jérémy Barbay
	:	Benjamin Bustos
	:	Patricio Poblete
	:	Kunihiko Sadakane (External Professor, Kyushu University, Japan)

This work has been supported in part by MECESUP, Grant UCH 0109, Chile; Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile; Yahoo! Research Latin America; and Millennium Institute for Cell Dynamics and Biotechnology, Grant P05-001-F, Mideplan, Chile.

Departament of Computer Science - University of Chile
Santiago - Chile
December 2008

Abstract

Compressed full-text self-indexing is a recent trend that builds on the discovery that traditional text indexes like suffix trees and suffix arrays can be compacted to take space proportional to the compressed text size, and moreover be able to reproduce any text substring. Therefore self-indexes replace the text, take space close to that of the compressed text, and in addition provide indexed search into it. Although a compressed index is slower than its uncompressed version, it can run in main memory in cases where a traditional index would have to resort to the (orders of magnitude slower) secondary memory. In those situations a compressed index is extremely attractive. On the other hand, existing compressed text indexes suffer from several weaknesses, such as lack of practicality, slowness in locating a pattern and in extracting text, and absence of space-efficient construction, secondary memory versions or update capabilities. This thesis contributes in all of these problems.

Our first contribution is a simple and efficient data structure for uncompressed bitmaps supporting *rank* and *select* queries, that has become very popular for its practicality. We also introduce the *Pizza&Chili* site, which contains a testbed for experimenting on compressed text indexes, including a standardized collection of texts and indexes, and a practical survey that covers and compares the most promising self-indexes. The site has also become a usual reference for practitioners.

We also develop a novel compressed text index based on well-known regularity properties of suffix arrays, which permits locating the occurrences very fast, while still being significantly smaller than classical indexes. The scheme is based on Re-Pair, a dictionary-based compressor, and enjoys locality properties that are absent in classical compressed indexes.

We also develop a statistical encoding of sequences, which permits direct access to any part of the sequence while achieving high-order compression. This is a key point to achieve speed and locality when extracting text on a compressed index.

We profit from this locating and extracting locality to introduce a new index for secondary memory, whose access time improves thanks to compression, instead of worsening as is the norm in other self-indexes. The index offers a very competitive space/time tradeoff.

Finally, we develop a novel structure supporting dynamic compressed sequences with *rank* and *select* capabilities. This result brings together the best features of previous works, and is used to obtain one of the best dynamic compressed text indexes (i.e., supporting updates to the text). We also apply it to the space-efficient construction of text indexes, obtaining one of the best results up to date.

There are other important byproducts. We study the relationship between the entropy of a text, its wavelet tree, and its Burrows-Wheeler transform. We develop a new technique to reduce the dictionary of rules of the Re-Pair algorithm. We generalize the well-known partial sums problem to handle a collection of somehow "synchronized" sequences, creating a new data structure to solve it. We also obtain new algorithms to construct suffix arrays and the Burrows-Wheeler transform of a text.

Contents

1 Introduction **1**

1.1 Weaknesses of Compressed Text Indexes 2

1.2 Contributions of the Thesis 3

1.3 Thesis Organization 5

2 Basic Concepts **7**

2.1 Entropy, Modeling and Coding 7

2.2 Statistical Encoding 9

2.3 Variable-Length Integer Encoding 10

2.4 Rank and Select Queries 11

2.5 Searchable Partial Sums with Indels 14

2.6 Classical Full-Text Indexes 15

2.7 Backward Search 16

2.8 The Burrows-Wheeler Transform 16

2.9 Compressed Text Indexes 17

2.9.1 The FM-index Family 18

2.9.2 The Compressed Suffix Array (CSA) 19

2.9.3 The Lempel-Ziv Index 21

2.10 Re-Pair 22

3 Compressed Text Indexes: From Theory to Practice **24**

3.1 Practical Binary Rank and Select Queries 25

3.1.1 Rank Queries 25

3.1.2 Select Queries 30

3.1.3 SelectNext Queries 35

3.1.4 Discussion 36

3.2 Implementing Indexes 39

3.2.1 Implementing the FM-index 39

3.2.2 The Alphabet-Friendly FM-index 40

3.3 The Pizza&Chili Site 42

3.3.1 Indexes 43

3.3.2 Texts 44

3.4 Experimental Results 45
3.4.1 Construction 47
3.4.2 Counting 48
3.4.3 Locating 49
3.4.4 Extracting 51
3.5 Discussion and Open Challenges 53

4 Locally Compressed Suffix Arrays 55
4.1 Locally Compressed Suffix Array (LCSA) 56
4.1.1 Basic LCSA Idea 57
4.1.2 Compression using Ψ 58
4.1.3 Stronger Compression based on Ψ 59
4.1.4 Compressing the Dictionary 61
4.2 Analysis of Compression Ratio 64
4.3 Towards a Text Index 66
4.3.1 A Smaller Classical Index 66
4.3.2 A Compressed Self-Index 67
4.4 Experimental Results 67

5 Statistical Encoding of Sequences 76
5.1 A New Entropy-Bounded Data Structure 77
5.1.1 Data Structures for Substring Decoding 77
5.1.2 Substring Decoding Algorithm 79
5.1.3 Space Requirement 79
5.2 Supporting Appends 80
5.2.1 Data Structures 81
5.2.2 Substring Decoding Algorithm 81
5.2.3 Construction Time 81
5.2.4 Space Requirement 81
5.3 Application to Full-Text Indexing 82
5.3.1 The Burrows-Wheeler Transform 82
5.3.2 The Wavelet Tree 83

6 A Compressed Text Index on Secondary Memory 85
6.1 An Entropy-Compressed Rank Dictionary on Secondary Memory 87
6.2 An Entropy-Bounded Data Structure for Secondary Memory 87
6.3 A Compressed Secondary Memory Structure 89
6.3.1 Counting 89
6.3.2 Locating 93
6.3.3 Extracting 93
6.4 Experiments 94
6.5 LCSA Construction in Secondary Memory 95

6.5.1 Compressing the Differential Suffix Array 98
6.5.2 Compressing the Dictionary 102

7 Rank/Select on Dynamic Compressed Sequences 105
7.1 Collection of Searchable Partial Sums with Indels 106
7.2 Uncompressed Dynamic Rank-Select Structures 110
7.3 Compressed Dynamic Rank-Select Structures 114
7.4 Applications 116

8 Conclusions 118
8.1 Contribution of this Thesis 118
8.1.1 Compressed Text Indexes: From Theory to Practice 118
8.1.2 Locally Compressed Suffix Arrays 119
8.1.3 Statistical Encoding of Sequences 120
8.1.4 A Compressed Text Index on Secondary Memory 120
8.1.5 Rank/Select on Dynamic Compressed Sequences 121
8.2 Further Work 121

Appendix:

A API for Text Indexes 123

Bibliography 125

List of Figures

2.1 Algorithm to count using an FM-index. 18
2.2 Algorithm to locate using an FM-index. 19
2.3 Algorithm to count using a CSA. 20

3.1 Comparison of different popcount methods to solve *rank*. 28
3.2 Comparison of different approaches to solve *rank*. 29
3.3 Comparison of alternatives to solve *select* by binary search. 31
3.4 Comparison of different space overheads for *select* based on binary search. . 32
3.5 Pseudocode for $select(B, \ell)$ 34
3.6 Comparison of Clark's select and two binary searches. 35
3.7 Pseudocode for $selectNext(B, i)$ 36
3.8 Comparison of different alternatives to solve $selectNext$. 37
3.9 Space-time tradeoffs for locating occurrences. 50
3.10 Space-time tradeoffs for extracting text symbols. 52

4.1 Algorithm to compress A' using Ψ in $O(n)$ time. 60
4.2 Algorithm to compress the dictionary R and to update C in $O(n)$ time. . . . 63
4.3 Compression achieved per pass using RPΨ_0SP. 68
4.4 Simulating a suffix array to binary search and locate the occurrences. 72
4.5 Time to extract a portion of the suffix array. 74

6.1 Block propagation over the wavelet tree. 90
6.2 Algorithm to obtain the number of occurrences inside a disk block. 91
6.3 Compression ratio achieved. 94
6.4 Counting cost vs. space requirement 96
6.5 Locating cost vs. space requirement 97
6.6 Operations generated by a replacement. 101

List of Tables

1.1 Contribution per research problem 5

3.1 General statistics for our indexed texts. 45
3.2 Ideal compressibility of our indexed texts. 46
3.3 Real compressibility of our indexed texts. 46
3.4 Parameters used for the different indexes in our experiments. 47
3.5 Time and peak of main memory usage required to build the various indexes. 47
3.6 Experiments on the counting of pattern occurrences. 48
3.7 Number of searched patterns and total number of located occurrences. 49
3.8 *Locate* time required by plain SA 51
3.9 The most promising indexes. 53

4.1 Compression ratio obtained using different values of s for RPΨSP. 69
4.2 Compression obtained using different values of δ for RPΨSP. 69
4.3 Index size and build time using Re-Pair and its Ψ-based approximations. . . 70
4.4 *Locate* time required by the LCSA and the LZ-index. 75

6.1 Different sizes and times obtained to answer *rank*. 88
6.2 Different sizes and times obtained to answer *count*. 93
6.3 Message types and meanings used by the secondary memory construction. . 100

Chapter 1

Introduction

The digital information we process every day has been increasing sharply over time. Although the storage space required for these data is rarely a problem by itself given the availability of cheap massive storage (e.g., disk), the access time to this storage has not improved much for decades. For example, currently an access to disk can be up to one million times slower than to main memory. This huge difference in access time has motivated the research in *compressed data structures*, whose size is ideally bounded by the entropy of the original data, since they permit one to maintain more information in a faster memory (e.g., main memory) and leave as few data as possible on the slower one (e.g., disk). The use of compressed data structures in main memory may dramatically improve the execution times compared to their uncompressed counterparts when handling large data repositories. Several compressed data structures nowadays exist for the representation of sequences [Mun96, RRR02, GMR06, BGMR06, GHSV07b, MN08], trees [MR97, BDMR99, GRRR04, FLMM05, BHMR07], graphs [MR97, CN07], permutations and functions [MRRR03, MR04], binary relations [BGMR06, BHMR07], texts [GGV03, Sad03, MN04, FM05, GV06, MN05, NM07, GHSV07a, FMMN07, MN08], discrete grids [MN06b], partial sums [HSS03b], and many others.

One of the cases where compressed data structures have been particularly successful is that of managing large text collections. This is essential in applications like bio-informatics, computational linguistics, multimedia databases, search engines, and text processing and retrieval in general. *Compressed text indexing* copes with the problem of giving indexed access to those large text collections without using up too much space. The current trend in compressed indexing is *full-text compressed self-indexes* [NM07]. Such a self-index (for short) *replaces* the text by providing fast access to arbitrary text substrings, and in addition gives indexed access to the text by supporting fast search for the occurrences of arbitrary patterns. These indexes take little space, usually from 30% to 150% of the text size (note that this includes the text). This is to be compared with classical indexes such as suffix trees [Wei73] and suffix arrays [MM93], which require at the very least 10 [Kur99] and 4 times, respectively, the space of the text, plus the text itself. In theoretical terms, to index a text $T = t_1 \dots t_n$ over an alphabet of size σ, the best self-indexes require $nH_k + o(n \log \sigma)$ bits for any $k \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$, where $H_k \leq \log \sigma$ is the k-th order empirical entropy of T [Man01, NM07][1]. Just the uncompressed text alone would need $n \log \sigma$ bits, and classical indexes require $O(n \log n)$ bits on top of it.

The basic search functionality of self-indexes is given via three operations. The first is, given a pattern $P = p_1 \dots p_m$, to *count* the number of times P occurs in T. The second is to *locate* those occurrences, that is, list their positions in T. The third is to *extract* an arbitrary text portion. Current self-indexes achieve a counting performance that is comparable in practice with that of classical indexes. In theoretical terms, for the best self-indexes the complexity is $O(m(1 + \frac{\log \sigma}{\log \log n}))$ [FMMN07] and even $O(1 + \frac{m}{\log_\sigma n})$ [GGV03], compared to $O(m \log \sigma)$ or $O(m)$ of suffix trees and $O(m \log n)$ or $O(m + \log n)$ of suffix arrays. For locating and extracting, however, they are far behind. This is discussed next, together with several other limitations.

1.1 Weaknesses of Compressed Text Indexes

Despite the great success of self-indexes, which have almost reached their theoretical limits both in counting time and index space, much more is necessary in order to have practical and useful structures. By the time this thesis started the weakest points of this technology were the following:

I **Practicality.** Many self-index structure proposals bear importance only at a theoretical level, being in practice utterly inefficient. Theoretical analysis is not always sufficient to predict practical performance. In particular, in this area, theory and practice were diverging dangerously. For example, one could find $o(n)$-size structures whose real size was $2^{(\log \log n)^4} = o(n)$, but indeed larger than n for $n < 2^{65536}$. Most theoretical proposals do involve interesting ideas, but cannot be implemented verbatim. This motivated us to explore new approaches to implementations that could yield smaller, faster and simpler indexes. A good practical survey, including implementations and practical comparisons, would be a contribution to the state of the art.

II **Locating and extracting.** Locating is in most self-indexes many times slower than in their classical counterparts. While classical indexes pay $O(occ)$ time to locate the occ occurrences, self-indexes pay $O(occ \log^\epsilon n)$, where ϵ can in theory be any constant larger than zero but is in practice larger than 1. Moreover, in a self-index it is essential to be able to extract part of the text, because the text itself is discarded once we construct the index. An important disadvantage of self-indexes is their slowness in extracting the

[1]In this thesis log stands for $\log_2$.

text, a consequence of their non-local access patterns to the data. For example, the original FM-index (Section 2.9.1) takes $O(\sigma \log^{1+\epsilon} n)$ time to locate an occurrence and $O(\sigma \left(l + \log^{1+\epsilon} n\right))$ to display a text substring of length l, for any constant $\epsilon > 0$. Another example, the CSA (Section 2.9.2) takes $O(\log^{\epsilon} n)$ time to locate an occurrence and $O(l+\log^{\epsilon} n)$ to display a text substring of length l, but the mechanism is rather complex and an $O(\log^{1+\epsilon} n)$ solution was implemented instead. The term $\log^{1+\epsilon} n$, together with the lack of access locality, makes the locating and extracting time hundreds to thousands times times slower than in classical indexes. The only implemented self-index which has more local accesses and faster *locate* is the LZ-index [Nav04], yet its counting time is not competitive. This issue claimed for alternative research lines aiming at novel solutions.

III **Construction.** Constructing these compressed structures using limited working memory is not an easy task. This is an important point, because no matter how compressed these structures are, if they cannot be built within the available memory, they will not be useful. For example, a solution exists for the CSA [LSSY02], but it is slow and requires more space than the final index. For the FM-index, they outline a solution that requires first to construct the CSA. For the LZ-index, in [AN05] they presented an efficient method that needs space close to that of the final index, yet that is not too compressed.

IV **Secondary memory.** Despite the chances of fitting a compressed index in main memory are higher, there will be applications where the text is large enough to force the use of secondary memory. The memory access patterns of self-indexes are highly non-local, which makes their potential secondary-memory versions rather unpromising. For example, a compressed self-index in secondary memory is proposed in [MNS04]. Yet, it has not been implemented and its complexity is still not satisfactory. This makes up a very interesting area of research.

V **Dynamism.** The problem of maintaining a collection of texts upon insertions and deletions arises in many applications. Some solutions are presented, for example, in [FM00, CHL04]. Let n be the total length of the collection. In [FM00] the complexity to search for a pattern $P = p_1 \ldots p_m$ is $O(m \log^3 n + occ \log n)$. Adding a text T to the collection takes $O(|T| \log n)$ amortized time and deleting a text T takes $O(|T| \log^2 n)$ amortized time. In [CHL04] they propose a index of size $O(\sigma n)$ bits (note this is much more than the $n \log \sigma$ bits used by the text). In this index, the search cost is $O(m \log n + occ \log^2 n)$ time and the cost to insert or delete a text T is $O\left(|T| \sigma \log n\right)$ time. The space can be reduced to $O(n \log \sigma)$, but deletions costs raise by an $O(\log n)$ factor.

1.2 Contributions of the Thesis

This thesis contributes in all the weak points outlined above, both in theory and in practice. Our specific contributions are as follows:

1. Design, implementation and experimental evaluation of data structures for uncompressed bitmaps supporting *rank* and *select* queries. Those are essential data structures for all compressed indexes and hence this contribution was our first step to obtain practical compresseded indexes (point I). This work has been published in the *4th Workshop on Efficient and Experimental Algorithms (WEA'05)* [GGMN05].

2. Implementation and experimental evaluation of most relevant compressed text indexes. We present these results in our *Pizza&Chili* site (`http://pizzachili.dcc.uchile.cl`), which also contains a testbed for experiments, including a variety of implemented self-indexes and text collections. This contribution permits the first thorough practical comparison of several compressed indexes (point I). An experimental survey related to this work has been accepted in *ACM Journal of Experimental Algorithmics (JEA)*.

3. Design, analysis, implementation and experimental evaluation of a new type of compressed suffix array called locally compressed suffix array. This contribution solves the problem of locating (point II) by proposing a scheme which maintains the locality of reference in a compressed suffix array, thus allowing for much faster *locate*. We also present an efficient way of constructing it on secondary memory (point III). This work has been published in the *18th Annual Symposium on Combinatorial Pattern Matching (CPM'07)* [GN07b], and a journal version has been submitted to *ACM Transactions on Algorithms (TALG)*.

4. Design and analysis of statistical encoding of sequences permitting direct access. This contribution solves the problem of extracting any subpart of a text efficiently (point II) thanks to local access. This work has been published in the *17th Annual Symposium on Combinatorial Pattern Matching (CPM'06)* [GN06].

5. Design, analysis, implementation and experimental evaluation of a novel compressed text index on secondary memory. This brings contributions 3 and 4 to secondary memory while retaining access locality, and yields a new competitive data structure for point IV. This work has been published in the *18th International Workshop on Combinatorial Algorithms (IWOCA'07)* [GN07a]. A journal version, comprising also the results of contribution 4, has been accepted in a special issue of the *Journal of Combinatorial Mathematics & Combinatorial Computing*, devoted to the best *IWOCA'07* papers.

6. Design and analysis of a novel structure supporting dynamic compressed sequences with *rank* and *select* capabilities. This contribution improves previous works on compressed index construction and on dynamic compressed indexes, points III and V respectively. This has been published in the *8th Latin American Symposium on Theoretical Informatics (LATIN'08)* [GN08]. A journal version of this paper has been accepted in a special issue of *Theoretical Computer Science (TCS)*.

Our contributions enrich the compressed text indexing area, approaching the problem from different points, and are a step forward towards the solution of the many problems outlined in Section 1.1. Table 1.1 depicts this relation in detail.

Research Problem \ Contribution	1	2	3	4	5	6
I	✓	✓				
II			✓	✓		
III			✓			✓
IV					✓	
V						✓

Table 1.1: Contribution per research problem.

We have also obtained several byproducts along this thesis. The most prominent follow:

- From contribution 3: We developed a new technique to reduce the dictionary of rules of the Re-Pair algorithm.
- From contribution 4: We obtain some relationships between the entropy of a text, its wavelet tree, and its Burrows-Wheeler transform.
- From contribution 5: We generalize our sequence representation with direct access to secondary memory. We also present a new data structure for bitmaps supporting *rank* queries in secondary memory.
- From contribution 6: We generalize the well-known partial sums problem to handle a collection of somehow "synchronized" sequences, creating a new data structure to solve it. We also obtain new solutions to build suffix arrays and to compute the Burrows-Wheeler transform in reduced space.

1.3 Thesis Organization

This document is divided into eight chapters, as follows:

In chapter *Basic Concepts* we introduce the background necessary to follow the document. This includes a number of concepts from a wide set of areas.

In chapter *Compressed Text Indexes: From Theory to Practice,* we present a practical data structure for bitmaps supporting *rank* and *select*, which is an essential block in the implementation of our compressed text indexes, and use it to improve several existing self-index implementations. We also present the first implementation of the compressed text index that provides one of the best theoretical space/time trade-offs [FMMN07]. We introduce the Pizza&Chili site, devoted to the implementation of compressed text indexes. This site contains several collections of texts, and establishes an API for new implementations. We present a series of experiments based on the contents of our Pizza&Chili site.

In chapter *Locally Compressed Suffix Arrays,* we present a suffix array compression technique that preserves locality of reference, therefore permitting fast locating of patterns. We show how to use it standalone, as a reduced-space classical index, as well as plugged into a compressed text index.

In chapter *Statistical Encoding of Succinct Data Structures,* we present a scheme to convert any sequence of symbols into a compressed representation that permits recovering any substring of the original sequence. We apply it to compressed text indexing.

In chapter *A Compressed Text Index on Secondary Memory,* we make up a new index by adapting three substructures to secondary memory: one based on the FM-index, another based on the locally compressed suffix array, and the last one based in our statistically encoded structure. The last two structures perform better due to their locality of access.

In chapter *Rank/Select on Dynamic Compressed Sequences,* we present a structure to handle dynamic compressed sequences with *rank* and *select* capabilities. We also present several byproducts of independent interest applicable to partial sums, text indexes, suffix arrays, and the Burrows-Wheeler transform.

In chapter *Conclusions,* we review our results with a more global perspective, and give further research directions.

Chapter 2

Basic Concepts

Let us introduce some notation. In this thesis log stands for $\log_2$, unless otherwise stated. We will ignore most floors and ceilings for simplicity in our descriptions. We will refer to *sequences or strings* in several ways as $S = S[1,\ell] = S_{1,\ell} = s_1 s_2 \ldots s_\ell$. These sequences will be over an alphabet Σ of size σ, unless otherwise stated. By $S[i,j] = S_{i,j} = s_i s_{i+1} \ldots s_j$ we will denote *substrings* of S, which will be called *prefixes* if $i = 1$ or *suffixes* if $j = \ell$. The *length* of a string will be written $|S| = |S_{1,\ell}| = \ell$, and the reverse of a string will be written $S^r = s_\ell s_{\ell-1} \ldots s_1$.

Given two integers a, b; the operation a div b returns the integer division of a/b and the operation a mod b returns the reminder of a/b.

We will call *succinct data structure* to a data structure that provides some functionalities using a space proportional to that of the uncompressed data. We also call *compressed data structure* to a data structure whose size can be written as a term proportional to the entropy (see next section) of the original data, plus a term that is sublinear on the size of the original data. For example, if a sequence uses $n \log \sigma$ bits of space, the suffix array of that sequence is not succinct because it uses $O(n \log n)$ bits of space. A data structure that uses $O(n \log \sigma)$ bits is succinct, and one that uses $O(nH_0(S)) + o(n \log \sigma)$ bits is compressed.

2.1 Entropy, Modeling and Coding

We use the *empirical entropy* as our compressibility measure for strings. The empirical entropy resembles the entropy defined in the probabilistic setting (for example, when the input comes from a Markov source [Ash65]). However, the empirical entropy is defined for any individual string and can be used to measure the performance of compression algorithms without any assumption on the input [Man01].

Given a sequence $S[1, n]$ over an alphabet Σ of size σ, the empirical k-th order entropy is defined using that of zero-order. This is defined as

$$H_0(S) = -\sum_{a\in\Sigma} \frac{n_S^a}{n} \log \frac{n_S^a}{n} \tag{2.1}$$

with n_S^a the number of occurrences of symbol a in sequence S.[1] This definition extends to $k > 0$ as follows. Let Σ^k be the set of all sequences of length k over Σ. For any string $w \in \Sigma^k$, called a *context* of size k, let w_S be the string consisting of the concatenation of characters following w in S. Then, the k-th order empirical entropy of S is

$$H_k(S) = \frac{1}{n} \sum_{w\in\Sigma^k} |w_S| H_0(w_S) . \tag{2.2}$$

The k-th order empirical entropy captures the dependence of symbols upon their context. For $k \geq 0$, $nH_k(S)$ provides a lower bound to the output of any compressor that considers a context of size k to encode every symbol of S. Note that the uncompressed representation of S takes $n \log \sigma$ bits, and that $0 \leq H_k(S) \leq H_{k-1}(S) \leq \ldots \leq H_1(S) \leq H_0(S) \leq \log \sigma$.

Statistical compression is carried out via the interaction of two actors. A *modeler* estimates the probability of each symbol in S, $\pi_1, \pi_2, \ldots, \pi_n$. An *encoder* generates codes for each symbol based on those probabilities. It is well known that the ideal (i.e., shortest) encoding gives length $\pi_i \log \frac{1}{\pi_i}$ to symbol s_i [Sha48]. Hence $\sum_{i=1}^{n} \pi_i \log \frac{1}{\pi_i}$ is a lower bound (independent of the modeler) to how good compression can be.

There are two main methods to model an input, adaptive and semi-static modeling. The main difference is how they initialize and maintain the model (i.e., the probabilities).

Adaptive modeling works as follows:

1. Initialize all the probabilities with the same value.
2. While there are more symbols to encode:
   (a) Encode the next symbol using the model.
   (b) Update the model based on this last symbol.

Semi-static modeling, instead, proceeds as follows:

1. Initialize the probabilities by doing a first pass over all the symbols to be encoded.
2. While there are more symbols to encode:

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

(a) Encode the next symbol using the model.

The decoding follows the same idea in both cases. Note that, while a semi-static modeler poses the disadvantage of having to read the input twice and having to transmit (or store) the model separately, it permits in principle to access the data at arbitrary point without reading the input from the beginning.

In particular, a semi-static k-th order modeler will actually estimate $\pi_i \approx Pr(s_i|s_{i-k} \dots s_{i-1})$ using the formula $\pi_i = \frac{n_{w_S}^{s_i}}{|w_S|}$, where $w = s_{i-k} \dots s_{i-1}$. It follows, by grouping all the terms with the same w in the summation [Man01, GGV03], that

$$\sum_{i=k+1}^{n} \log \frac{1}{\pi_i} \quad = \quad nH_k(S). \tag{2.3}$$

2.2 Statistical Encoding

Given a k-th order modeler as described in Section 2.1, which yield the probabilities $\pi_1, \pi_2, \dots, \pi_n$ for the successive symbols in S, we will encode S trying to use $\pi_i \log \frac{1}{\pi_i}$ bits for s_i. If we reach exactly $\pi_i \log \frac{1}{\pi_i}$ bits, then the overall number of bits produced will be $nH_k(S) + O(k \log n)$, according to Eq. (2.3) of Section 2.1.

Different encoders provide different approximations to the ideal $\pi_i \log \frac{1}{\pi_i}$ bits. The simplest encoder is probably Huffman coding [Huf52], while the best one, from the point of view of the number of bits generated, is Arithmetic coding [BCW90].

Given a statistical encoder E and a semi-static modeler over sequence $S[1, n]$ yielding probabilities $\pi_1, \pi_2, \dots, \pi_n$, we call $E(S)$ the bitwise output of E for those probabilities, and $|E(S)|$ its bit length. We call $f_k(E, S) = |E(S)| - (\sum_{1 \leq i \leq n} \pi_i \log \frac{1}{\pi_i})$ the extra space in bits needed to encode S using E, on top of the entropy of the model. For example, the wasted space of Huffman encoding is bounded by 1 bit per symbol, and thus $f_k(\text{Huffman}, S) < |S|$ (tighter bounds exist but are not relevant for this thesis [BCW90]). On the other hand, Arithmetic encoding approaches $\pi_i \log \frac{1}{\pi_i}$ as closely as desired, requiring only at most two extra bits to terminate the whole sequence [BCW90, Section 5.2.6 and 5.4.1]. Thus $f_k(\text{Arithmetic}, S) \leq 2$. Again, we can relate the model entropy of $\pi_1, \pi_2, \dots, \pi_n$ with the empirical entropy of S using Eq. (2.3), achieving that, say, Arithmetic coding encodes S using at most $nH_k(S) + O(k \log n) + 2$ bits.

The idea of the Huffman coding [Huf52] is to use fewer bits to encode symbols that appear more frequently. The Huffman encoding can be represented by a binary tree, where each leaf contains a symbol and its probability, and each internal node contains the accumulated probability of all its leaves. To know the coding of a symbol, we traverse the tree until we reach its leaf, writing down a 0 when we go to a left child and a 1 otherwise. One way to obtain this Huffman tree is to use a priority queue (the node with lowest probability has the highest priority). The algorithm is as follows:

1. We create a leaf for each symbol and insert them into the priority queue.
2. We remove two nodes from the priority queue and make them children of a new node that has a probability equal to the sum of the two nodes.
3. We insert this new node into the priority queue
4. If the priority queue has more than two nodes we go to step 2 again, otherwise we go to step 5.
5. The unique remaining node is the root of our Huffman tree.

Arithmetic coding essentially expresses S using a number in $[0, 1)$ which lies within a range of size $\Pi = \pi_1 \times \pi_2 \times \cdots \times \pi_n$. We need $-\log \Pi = -\sum \log \pi_i$ bits to distinguish a number within that range (plus two extra bits for technical reasons). Thus each new symbol s_i, which appears within its context $n\pi_i$ times, requires $\log \frac{1}{\pi_i}$ bits to be encoded. This totalizes $n \sum \pi_i \log \frac{1}{\pi_i} + 2$ bits.

There are some limitations to the near-optimality achieved by Arithmetic coding in practice [BCW90]. One is that many bits are required to manipulate Π, which can be cumbersome. This is mainly alleviated by emitting the most significant bits of the final number as soon as they are known, and thus scaling the remainder of the number again to the range $[0, 1)$ (that is, dropping the emitted bits from our number). Still, some symbols with very low probability may require many bits. To simplify matters, fixed precision arithmetic is used to approximate the real values, and this introduces a very small (yet linear) inefficiency in the coding.

Another limitation applies to adaptive modeling, where some kind of aging technique is used to let the model forget symbols that have appeared many positions away in the sequence. This does not apply when using semi-static encoding.

2.3 Variable-Length Integer Encoding

A variable-length integer encoding is used in applications where it is needed to represent a sequence of integers in compact form [BCW90, Appendix A]. We introduce two variable-length integer encodings, which are relevant for this thesis.

Given a positive integer x and its binary representation X, where X starts with the most significant 1, the *γ-encoding* of x is $\gamma(x) = 0^{|X|-1}X$, where $0^{|X|-1}$ stands for a sequence of $|X| - 1$ zeros. For example, $\gamma(13) = \gamma(1101) = 0^3 1101 = 0001101$.

Let $y = x - 2^{|X|-1}$, then the *δ-encoding* of x is $\delta(x) = \gamma(|X|)Y'$, where Y' is the binary representation of y using $|X| - 1$ bits. Note that y is x without its most significant bit. For example, $\delta(13 = 2^3 + 5) = \delta(1101) = \gamma(4)101 = 00100101$.

The γ-encoding works well for sequences where small integers are much more frequent than large integers. For bigger integers, the δ-encoding may be more compact. The length, in bits, of these encodings are:

- $|\gamma(x)| = 2\lceil\log(x + 1)\rceil - 1 = O(\log x)$.
- $|\delta(x)| = |\gamma(|X|)| + |X| - 1 = (2\lceil\log(\lfloor\log x\rfloor + 2)\rceil - 1) + (\lceil\log(x + 1)\rceil - 1) = \log x + O(\log\log x)$.

To decode a γ-encoding we count the 0s until we reach the first 1. Let Z be the number of 0s read. Then we read $Z + 1$ bits including the 1 just visited; this number is the integer encoded. This can be done in constant time by using a precomputed table Γ, where $\Gamma(X)$ gives the position of the first 1 in X.

To decode a δ-encoding, we first decode a γ-encoded integer; we call this interger N. Then the result is 2^{N-1} plus the binary representation formed by the next $N - 1$ bits. Again this can be done in constant time.

2.4 Rank and Select Queries

Operations *rank* and *select* on sequences have a great impact on many other data structures, especially on those aimed at compressed text indexing, but also on the space-efficient representation of trees, graphs, permutations and functions, to name a few. In this section, we present different variants of *rank* and *select* problems.

Rank/select over Binary Sequences. Let $B_{1,n}$ be a binary sequence. In this case, $rank_1(B, i)$ gives the number of 1-bits in $B[1, i]$ and $select_1(B, i)$ gives the position of the i-th 1 in B. Similarly, we can define $rank_0$ and $select_0$. Note that, $rank_0$ can be easily obtained from $rank_1$, but $select_0$ is not directly related. By default, we will refer with *rank* and *select* to their 1s version.

Both *rank* and *select* can be computed in constant time using $o(n)$ bits of space in addition to B [Mun96], or $nH_0(B) + o(n)$ bits [RRR02]. In both cases, the $o(n)$ term is $\Theta(n \log\log n/ \log n)$. From the compressed representation one can easily retrieve $B[i] = rank(B, i) - rank(B, i - 1)$ in constant time, so the compressed representation *replaces* B and in addition gives rank/select functionality on it.

The solution using $n + o(n)$ bits is detailed in Section 3.1.1.1. We briefly survey here the compressed one. The binary sequence $B_{1,n}$ is divided into *blocks*, each representing $\lfloor\frac{1}{2}\log n\rfloor$

bits. We represent each block using a (c, o)-pair encoding. The c part is of fixed width and tells how many 1's are there in the block, whereas the o part is of variable width and gives the identifier of the block among those sharing the same c component. Each c component uses at most $\log\log n$ bits; while the o components use at most $\frac{1}{2}\log n$ bits each, and overall add up to $nH_0(B) + O(n/\log n)$ bits. The c components, together with several accumulators for partial *rank* solutions and directories to access the o sequence, add up to $O(n \log\log n/\log n) = o(n)$ bits of extra space.

Let s be the number of one-bits in $B_{1,n}$. Then $nH_0(B) = s \log \frac{n}{s} + O(s)$, and thus the $o(n)$ terms above are too large if s is far from $n/2$. Existing lower bounds [Mil05, Gol07] show that constant-time *rank* and *select* can only be achieved with $\Omega(n \log\log n/\log n)$ extra bits on top of B (if B is represented verbatim). In Chapter 6, we will have $s << n$, so we are also interested in techniques with less overhead over the entropy, even if not of constant-time. One such *rank* dictionary [GHSV07a] encodes the gaps between successive 1's in B using δ-encoding (Section 2.3) and adds some data to support a binary-search-based *rank* and *select*. It requires $s(\log \frac{n}{s} + \frac{\log n}{\log s} + 2\log\log \frac{n}{s}) + O(\log n)$ bits of space and supports *rank* and *select* in $O(\log s)$ time. This structure is called *BSGAP* (binary searchable gap encoding). Recently [GGG$^+$07], constant-time *rank* and *select* has been achieved using $nH_0(B) + O(\frac{n \log\log n}{\log^2 n})$ bits, but the practicality of this approach has not yet been established .

Rank/select over General Sequences. Given a sequence $S[1, n]$ over an alphabet of size σ, one aims at a (hopefully compressed) representation efficiently supporting the following operations:

- $access(S, i)$ returns the symbol $S[i]$.
- $rank_c(S, i)$ returns the number of times symbol c appears in the prefix $S[1, i]$.
- $select_c(S, i)$ returns the position of the i-th c in S.

The first structure providing support for *rank* and *select* on a sequence of symbols was the *wavelet tree* [GGV03]. The wavelet tree is a perfect binary tree of height $\Theta(\log \sigma)$, built on the alphabet symbols, such that the root represents the whole alphabet and each leaf represents a distinct alphabet symbol. If a node v represents alphabet symbols in the range $\Sigma^v = [i, j]$, then its left child v_l represents $\Sigma^{v_l} = [i, \frac{i+j}{2}]$ and its right child v_r represents $\Sigma^{v_r} = [\frac{i+j}{2} + 1, j]$. We associate to each internal node v the subsequence S^v of S formed by the symbols in Σ^v. Sequence S^v is not really stored at the node, but it is replaced by a bit sequence B^v such that $B^v[i] = 0$ if $S^v[i]$ is a symbol whose leaf resides in the left subtree of v. Otherwise, $B^v[i]$ is set to 1. Leaves are actually not represented.

The power of the wavelet tree is to reduce *rank* and *select* operations over general alphabets to *rank* and *select* operations over a binary alphabet, so that the *rank*/*select*-machinery above can be used in each wavelet-tree node. Precisely, let us answer the query

$rank_c(S,i)$. We start from the root v of the wavelet tree (with associated vector B^v), and check which subtree encloses the queried symbol c. If c descends into the right subtree, we set $i \leftarrow rank_1(B^v, i)$ and move to the right child of v. Similarly, if c belongs to the left subtree, we set $i \leftarrow rank_0(B^v, i)$ and go to the left child of v. We repeat this until we reach the leaf that represents c, where the current i value is the answer to $rank_c(S,i)$. If the binary *ranks* take $O(1)$ time, the overall $rank_c$ operation takes $O(\log \sigma)$ time.

Now, let us answer the query $select_c(S,i)$. We start from the parent v of the leaf representing c. Say that c is represented in v by bit b, that is, $b = 0$ if the leaf for c is the left child of v, and $b = 1$ if it is v's right child. Then we calculate $i' = select_b(B^v, i)$. This value indicates the corresponding position within v. We repeat this query over the parent of v, $select_{b'}(parent(v), i')$, where b' now is the value associated to node v in $parent(v)$, i.e, 0 if v is the left child of its parent and 1 otherwise. We repeat this process until we reach the root node, and the position within the root node is our answer. If a binary *select* takes $O(1)$ time, then the overall $select_c$ operation takes $O(\log \sigma)$ time.

We note that the wavelet tree can replace S as well: to obtain $S[i]$, we start from the root v of the wavelet tree. If $B^v[i] = 0$, then we set $i \leftarrow rank_0(B^v, i)$ and go to the left child. Similarly, if $B^v[i] = 1$, then we set $i \leftarrow rank_1(B^v, i)$ and go to the right child. We repeat this process until we reach a leaf, where the symbol associated to the leaf is the answer. Again, this takes $O(\log \sigma)$ time.

The wavelet tree has similar space occupancy than the original sequence, as it requires $n \log \sigma \ (1 + o(1))$ bits of space. A practical way to reduce the space occupancy to about the zero-order entropy of S is to replace the balanced tree structure by the Huffman tree of S [GGV03] (see Section 2.2). Now we have to follow the binary Huffman code of a symbol to find its place in the tree. It follows that the total number of bits required by such a tree is at most $n(H_0(S) + 1) + o(n \log \sigma)$ and the average time taken by *rank*, *select* and *access* operations is $O(H_0(S) + 1)$ if the query position is chosen uniformly over the sequence S. This structure is the key tool in our implementation of SSA and AF-index (Section 3.4). Another way to achieve zero-order compression is to use a balanced wavelet tree where each B^v is compressed using the structure presented in [RRR02]; this yields $nH_0(S) + o(n \log \sigma)$ bits of space and worst case time $O(\log \sigma)$.

Rank/select over Dynamic Sequences. Chan et al. [CHL04] considered dynamic capabilities for the sequences, by including insert/delete operations. We focus on this problem in Chapter 7, which we now define formally.

The *Dynamic Sequence with Indels* problem consists in maintaining a sequence $S = s_1 s_2 \ldots s_n$ of symbols over an alphabet Σ of size σ, supporting the queries $access(S,i)$, $rank_c(S,i)$, and $select_c(S,i)$ as defined previously, as well as the operations:

- $insert_c(S,i)$ inserts symbol c between $S[i-1]$ and $S[i]$.
- $delete(S,i)$ deletes $S[i]$ from S.

Chan et al. presented a structure for binary sequences taking $O(n)$ bits of space and performing all the operations in $O(\log n)$ time. Blanford and Blelloch [BB04] improved the space to $O(nH_0)$, and finally Mäkinen and Navarro [MN06a, MN08] achieved $nH_0(S) + o(n)$ bits of space, still solving all the operations in $O(\log n)$ time. This is achieved with a binary tree that stores $\Theta(\log^2 n)$ bits at the leaves, and at internal nodes stores summary *rank*/*select* information on the subtrees.

The solution is easily extended to handle sequences. A wavelet tree using dynamic bitmaps yields a dynamic sequence representation that takes $nH_0(S) + o(n \log \sigma)$ bits and solves all the operations in time $O(\log n \log \sigma)$ [MN06a].

Recently, Lee and Park [LP07] managed to improve the time complexities of this solution. They show that the $O(\log n)$ time complexities can be achieved for alphabets of size up to $\sigma = O(\log n)$, but only in an amortized sense. They combine this tool with a multiary wavelet tree to achieve $O(\log n(1 + \frac{\log \sigma}{\log \log n}))$ amortized time.

The key to the success of Lee and Park is a clever detachment of two roles of tree leaves that are entangled in Mäkinen and Navarro's solution [MN08]: In the latter, the leaves are the memory allocation unit (that is, whole leaves are allocated or freed), and also the information summarization unit (that is, the tree maintains information up to leaf granularity, and the rest has to be collected by sequentially scanning a leaf). In Lee and Park's solution [LP07], leaves are the information summarization unit, but handle an internal linked list with smaller memory allocation units. This permits moving symbols to manage the space upon insertions/deletions within a leaf, without having to update summarization information for the data moved. This was the main bottleneck that prevented the use of larger alphabets in $O(\log n)$ time in Mäkinen and Navarro's method [MN08].

2.5 Searchable Partial Sums with Indels

The *Searchable Partial Sums with Indels (SPSI)* problem [HSS03b] consists in maintaining a sequence S of nonnegative integers $s_1, \ldots, s_n$, each one of $k = O(\log n)$ bits, supporting the following queries and operations:

- $sum(S, i)$ is $\sum_{l=1}^{i} s_l$.
- $search(S, y)$ is the smallest i' such that $sum(S, i') \geq y$.
- $update(S, i, x)$ updates s_i to $s_i + x$ (x can be negative as long as the result is not).
- $insert(S, i, x)$ inserts a new integer x between s_{i-1} and s_i.
- $delete(S, i)$ deletes s_i from the sequence.

It is possible to solve the SPSI problem using $kn + o(kn)$ bits of space and $O(\log n)$ time per operation, in a way similar to that used for dynamic *rank* and *select* on sequences [MN08], by maintaining a balanced binary tree holding $\Theta(\log^2 n)$ bits in the leaves. In Section 7.1, we generalize the SPSI problem to handle a collection of somehow "synchronized" sequences.

2.6 Classical Full-Text Indexes

Many different indexing data structures have been proposed in the literature for text searching, most notably suffix trees and suffix arrays.

The *suffix tree* [Gus97] of a text $T_{1,n}$ is a trie (or digital tree) built on all the n suffixes $T_{i,n}$ of T, where unary paths are compressed to ensure $O(n)$ words of space. The suffix tree has n leaves, each corresponding to a suffix of T, and each internal suffix tree node corresponds to a unique substring of T that appears more than once. The suffix tree can count the *occ* occurrences of pattern $P_{1,m}$ in time $O(m)$, independent of n and *occ*, by descending in the tree according to the successive symbols of $P_{1,m}$ (each node should store the number of leaves that descend from it). Afterwards, it can locate the occurrences in optimal $O(occ)$ time by traversing the subtree of the node arrived at counting. The suffix tree, however, uses much more space than the text itself. In theoretical terms, it uses $\Theta(n \log n)$ bits whereas the text needs $n \log \sigma$ bits. In practice, a suffix tree requires from 10 to 20 times the text size [Kur99].

The *suffix array* [MM93] is a compact version of the suffix tree. It still requires $\Theta(n \log n)$ bits, but the constant is smaller: 4 times the text size in practice. The suffix array $A[1, n]$ of a text $T_{1,n}$ contains all the starting positions of the suffixes of T listed in lexicographical order, that is, $T_{A[1],n} < T_{A[2],n} < \ldots < T_{A[n],n}$. A can be obtained by traversing the leaves of the suffix tree, or it can be built directly by naive or sophisticated ad-hoc sorting methods [PST07].

Any substring of T is the prefix of a text suffix, thus finding all the occurrences of P is equivalent to finding all the text suffixes that start with P. Those form a lexicographical interval in A, which can be binary searched in $O(m \log n)$ time, as each comparison in the binary search requires examining up to m symbols of the pattern and of a text suffix. The time can be boosted to $O(m + \log n)$, by using an auxiliary structure that doubles the space requirement of the suffix array [MM93], or even to $O(m + \log \sigma)$ by adding some further data structures (called *suffix trays* [CKL06]). Once the interval $A[sp, ep]$ containing all the text suffixes starting with P has been identified, counting is solved as $occ = ep - sp + 1$, and the occurrences are located at $A[sp]$, $A[sp + 1], \ldots A[ep]$.

For technical convenience, we will assume that the last text character is $t_n = \$$, a special end-marker symbol that belongs to Σ but does not appear elsewhere in T nor P, and that is lexicographically smaller than any other symbol in Σ.

2.7 Backward Search

In the previous section, we described the classical binary search method over suffix arrays. Here we review an alternative approach which has been proposed in [FM00], hereafter named *backward search*. For any $i = m, m-1, \ldots, 1$, this search algorithm keeps the interval $A[sp_i, ep_i]$ storing all text suffixes which are prefixed by $P_{i,m}$. This is done via two main steps:

Initial step. We have $i = m$, so that it suffices to access a precomputed table that stores the pair $\langle sp_m, ep_m \rangle$ for all possible symbols $p_m \in \Sigma$.

Inductive step. Let us assume we have computed the interval $A[sp_{i+1}, ep_{i+1}]$, whose suffixes are prefixed by $P_{i+1,m}$. The present step determines the next interval $A[sp_i, ep_i]$ for $P_{i,m}$ from the previous interval and the next pattern symbol p_i. The implementation is not obvious, and leads to different realizations of backward searching in several compressed indexes, with various time performances (see Section 2.9).

The backward-search algorithm is executed by decreasing i until either an empty interval is found (i.e. $sp_i > ep_i$), or $A[sp_1, ep_1]$ contains all pattern occurrences. In the former case, no pattern occurrences are found; in the latter case, the algorithm has found $occ = ep_1 - sp_1 + 1$ pattern occurrences.

2.8 The Burrows-Wheeler Transform

The *Burrows-Wheeler Transform (BWT)* [BW94] is a key tool in designing compressed full-text indexes. It is a reversible permutation of T, which has the nice property of putting together symbols followed by the same context. This ensures that the permuted T offers better compression opportunities: a locally adaptive zero-order compressor is able to achieve on this string the k-th order entropy of T (recall Eq. (2.2))[2]. The BWT works as follows:

1. Create a conceptual matrix M, whose rows are cyclic shifts of T.
2. Sort the matrix rows lexicographically.
3. Define the last column of M as the BWT of T, and call it T^{bwt}.

[2]Here the k-th order entropy considers the context of size k that follows each symbol, but the difference with the original k-th order entropy where the contexts precede the symbols (Section 2.1) is asymptotically negligible [FM05].

There is a close relationship between matrix M and the suffix array A of text T, because when we lexicographically sort the rows, we are essentially sorting the suffixes of T (recall indeed that $t_n = \$$ is smaller than any other alphabet symbol). Specifically, $A[i]$ points to the suffix of T which prefixes the i-th row of M. Hence, another way to compute T^{bwt} is to concatenate the symbols that precede each suffix of T in the order listed by A, that is, $T^{bwt} = t_{A[1]-1}\, t_{A[2]-1} \ldots t_{A[n]-1}$, where we assume that $t_0 = t_n$.

Given the way matrix M has been built, all columns of M are permutations of T. So the first and last column of M are indeed one permutation of the other. The question is how to map symbols in the last column T^{bwt} to symbols in the first column. It follows [BW94] that occurrences of equal symbols preserve their relative order in the last and the first columns of M (as in both cases they are ordered by the suffix that follows them in T). Thus, the j-th occurrence of a symbol c within T^{bwt} corresponds to the j-th occurrence of c in the first column. If $c = T^{bwt}[i]$, then we have that $j = rank_c(T^{bwt}, i)$ in the last column; whereas in the first column, where the symbols are sorted alphabetically, the j-th occurrence of c is at position $C[c] + j$, where $C[c]$ counts the number of occurrences in T of symbols smaller than c. By plugging one formula in the other we derive the so called *Last-to-First column mapping* (or, LF-mapping): $LF(i) = C[c] + rank_c(T^{bwt}, i)$. We talk about LF-mapping because the symbol $c = T^{bwt}[i]$ is located in the first column of M at position $LF(i)$.

The LF-mapping allows one to navigate T backwards: If $t_k = T^{bwt}[i]$, then $t_{k-1} = T^{bwt}[LF(i)]$ because row $LF(i)$ of M starts with t_k and thus ends with t_{k-1}. As a result, we can reconstruct T backwards by starting at the first row, equal to $\$T$, and repeatedly applying LF for n steps.

2.9 Compressed Text Indexes

Compressed text indexes provide an efficient, reduced space, alternative to classical indexes. They have undergone significant development in the last years, so that we count now in the literature many solutions that offer a plethora of space-time tradeoffs [NM07]. In theoretical terms, the most compressed index [FMMN07] achieves $nH_k(T) + o(n \log \sigma)$ bits of space, and for any fixed $\epsilon > 0$, requires $O(m(1 + \frac{\log \sigma}{\log \log n}))$ counting time, $O(\log^{1+\epsilon} n)$ time per located occurrence, and $O(\ell(1 + \frac{\log \sigma}{\log \log n}) + \log^{1+\epsilon} n)$ time to extract a substring of T of length ℓ.[3] This shows that whenever $T[1, n]$ is compressible it can be indexed into smaller space than its plain form and still offer search capabilities in efficient form.

In the following, we present the most competitive compressed indexes for which there is an implementation we are aware of. We will review the FM-index family, which builds on the BWT and backward searching; Sadakane's Compressed Suffix Array (CSA), which is based

[3]These locating and extracting complexities are better than those reported in [FMMN07], and can be obtained by setting their sampling step to $\frac{\log^{1+\epsilon} n}{\log \sigma} \cdot \log \log n$.

on compressing the suffix array via a so-called Ψ function that captures text regularities; and the LZ-index, which is based on Lempel-Ziv compression. All of them are self-indexes since they include the indexed text, which therefore may be discarded.

A self-index built on a text T supports at least the following queries:

- *count*(P): counts the number of occurrences of pattern P in T.
- *locate*(P): locates the positions of all those *occ* occurrences of P.
- *extract*(l, r): extracts the substring $T_{l,r}$ of T, with $1 \leq l, r \leq n$.

2.9.1 The FM-index Family

The *FM-index* is composed of a compressed representation of T^{bwt} plus auxiliary structures for efficiently computing *rank* queries on it. The main idea [FM00, FM05] is to obtain a text index from the BWT and then use backward searching for identifying the pattern occurrences (Sections 2.7 and 2.8). Several variants of this algorithmic scheme exist [FM01, FM05, MN05, GNP$^+$06, FMMN07] which induce several time/space tradeoffs for the counting, locating, and extracting operations.

Counting. The counting procedure takes a pattern P and obtains the interval $A[sp, ep]$ of text suffixes prefixed by it (or, which is equivalent, the interval of rows of the matrix M prefixed by P, see Section 2.8). Figure 2.1 gives the pseudocode to compute *sp* and *ep*.

```
**Algorithm** FM-count($P_{1,m}$)
$i \leftarrow m,\ sp \leftarrow 1,\ ep \leftarrow n$;
**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** "no occurrences of $P$" **else return** $\langle sp, ep \rangle$;
```

Figure 2.1: Algorithm to get the interval $A[sp, ep]$ of text suffixes prefixed by P, using an FM-index.

The algorithm is correct: Let $[sp_{i+1}, ep_{i+1}]$ be the range of rows in M that start with $P_{i+1,m}$, and we wish to know which of those rows are preceded by p_i. These correspond precisely to the occurrences of p_i in $T^{bwt}[sp_{i+1}, ep_{i+1}]$. Those occurrences, mapped to the first column of M, form a (contiguous) range that is computed with a rationale similar to that for $LF(\cdot)$ in Section 2.8, and thus via just two *rank* operations.

Locating. Algorithm FM-locate in Figure 2.2 obtains the position of the suffix that prefixes the i-th row of M. The basic idea is to logically mark a suitable set of rows of M, and keep their positions in T (that is, we store the corresponding A values). Then, FM-locate(i) scans backwards the text T using the LF-mapping until a marked row i' is found, and then it reports $A[i'] + t$, where t is the number of backward steps used to find such i'. This is because we logically move one position backwards in T each time we apply LF. To compute the position of all occurrences of a pattern P, it is thus enough to call FM-locate(i) for every $sp \leq i \leq ep$.

```
**Algorithm** FM-locate($i$)
$i' \leftarrow i, t \leftarrow 0$;
**while** $A[i']$ is not explicitly stored **do**
    $i' \leftarrow LF(i')$;
    $t \leftarrow t + 1$;
**return** $A[i']\ + t$;
```

Figure 2.2: Algorithm to obtain $A[i]$ using an FM-index.

The sampling rate of M's rows, hereafter denoted by s_A, is a crucial parameter that trades space for query time. Most FM-index implementations mark all the $A[i]$ that are a multiple of s_A, via a bitmap $B[1, n]$. All the marked $A[i]$s are stored contiguously in suffix array order, so that if $B[i] = 1$ then one finds the corresponding $A[i]$ at position $rank_1(B, i)$ in that contiguous storage. This guarantees that at most s_A LF-steps are necessary for locating the text position of any occurrence. The extra space is $\frac{n \log n}{s_A} + n + o(n)$ bits.

A way to avoid the need of bitmap B is to choose a symbol c having some suitable frequency in T, and then store $A[i]$ if $T^{bwt}[i] = c$ [FM01]. Then, the position of $A[i]$ in the contiguous storage is $rank_c(T^{bwt}, i)$, so no extra space is needed other than T^{bwt}. In exchange, there is no guarantee of finding a marked cell after a given number of steps.

Extracting. The same text sampling mechanism used for locating permits extracting text substrings. Given s_A, we store the positions i such that $A[i]$ is a multiple of s_A, now in text order (previously we followed the A-driven order). To extract $T_{l,r}$, we start from the first sample that follows the area of interest, that is, sample number $d = \lceil (r + 1)/s_A \rceil$. From it, we obtain backwards the desired text with the same mechanism for inverting the BWT (see Section 2.8), here starting with the value i stored for the d-th sample. We need at most $s_A + r - l + 1$ applications of LF.

2.9.2 The Compressed Suffix Array (CSA)

The *compressed suffix array (CSA)* was not originally a self-index, and required $O(n \log \sigma)$ bits of space [GV00, GV06]. Sadakane [Sad00, Sad02, Sad03] then proposed a variant which is a self-index and achieves high-order compression.

The CSA represents the suffix array $A[1,n]$ by a sequence of numbers $\Psi(i)$, such that $A[\Psi(i)] = A[i]+1$. It follows [Sad03] that Ψ is piecewise monotone increasing in the areas of A where the suffixes start with the same symbol. In addition, if T is compressible there are long *runs* where $\Psi(i+1) = \Psi(i)+1$, and these runs can be mapped one-to-one to the equal-letter runs in T^{bwt} [NM07]. These properties permit a compact representation of Ψ and its fast access. Essentially, we compute the differences $\Psi(i) - \Psi(i-1)$, run-length encode the long runs of 1's occurring over those differences, and for the rest use an encoding favoring small numbers (see Section 2.3). Absolute samples are stored at regular intervals to permit the efficient decoding of any $\Psi(i)$. The sampling rate (hereafter denoted by s_Ψ) gives a space/time tradeoff for accessing and storing Ψ. In [Sad03], it is shown that the index requires $O(nH_0(T) + n\log\log\sigma)$ bits of space. The analysis has been then improved in [NM07] to $nH_k(T) + O(n\log\log\sigma)$ for any $k \le \alpha\log_\sigma n$ and constant $0 < \alpha < 1$.

Counting. The original CSA [Sad00, Sad03] used the classical binary searching to count the number of pattern occurrences in T. The actual implementation, proposed in [Sad02], uses backward searching (Section 2.7): Ψ is used to obtain $\langle sp_i, ep_i\rangle$ from $\langle sp_{i+1}, ep_{i+1}\rangle$ in $O(\log n)$ time, for a total of $O(m\log n)$ counting time. Precisely, $A[sp_i, ep_i]$ is the range of suffixes $A[j]$ that start with p_i and such that $A[j]+1$ $(= A[\Psi(j)])$ starts with $P_{i+1,m}$. The former is equivalent to the condition $[sp_i, ep_i] \subseteq [C[p_i]+1, C[p_i+1]]$. The latter is equivalent to saying that $sp_{i+1} \le \Psi(j) \le ep_{i+1}$. Since $\Psi(i)$ is monotonically increasing in the range $C[p_i] < j \le C[p_i+1]$ (since the first characters of suffixes in $A[sp_i, ep_i]$ are the same), we can binary search this interval to find the range $[sp_i, ep_i]$. Figure 2.3 shows the pseudocode for counting using the CSA.

```
**Algorithm** CSA-count($P_{1,m}$)
$i \leftarrow m,\ sp \leftarrow 1,\ ep \leftarrow n$;
**while** $((sp \le ep)$ **and**$(i \ge 1))$ **do**
    $c \leftarrow p_i$;
    $\langle sp, ep\rangle \leftarrow \langle \min, \max\rangle\ \{j \in [C[c]+1, C[c+1]], \Psi(j) \in [sp, ep]\}$;
    $i \leftarrow i-1$;
**if** $(ep < sp)$ **then return** "no occurrences of $P$" **else return** $\langle sp, ep\rangle$;
```

Figure 2.3: Algorithm to get the interval $A[sp, ep]$ prefixed by P, using the CSA. The $\langle\min, \max\rangle$ interval is obtained via binary search.

Locating. Locating is similar to the FM-index, in that the text is sampled at regular intervals of size s_A. However, instead of using the LF-mapping to traverse the text backwards, this time we use Ψ to traverse the text forward, given that $A[\Psi(i)] = A[i]+1$. This points out an interesting duality between the FM-index and the CSA. Yet, there is a fundamental difference: function $LF(\cdot)$ is implicitly stored and calculated on the fly over T^{bwt}, while function $\Psi(\cdot)$ is explicitly stored. The way these functions are calculated/stored makes the CSA a better alternative for large alphabets.

Extracting. Given C and Ψ, we can obtain $T_{A[i],n}$ symbolwise from i, as follows. The first symbol of the suffix pointed to by $A[i]$, namely $t_{A[i]}$, is the character c such that $C[c] < i \leq C[c+1]$, because all the suffixes $A[C[c]+1], \ldots, A[C[c+1]]$ start with symbol c. Now, to obtain the next symbol, $t_{A[i]+1}$, we compute $i' = \Psi(i)$ and use the same procedure above to obtain $t_{A[i']} = t_{A[i]+1}$, and so on. The binary search in C can be avoided by representing it as a bit vector $D[1,n]$ such that $D[C[c]] = 1$, thus $c = rank_1(D,i)$.

Now, given a text substring $T_{l,r}$ to extract, we must first find the i such that $l = A[i]$ and then we can apply the procedure above. Again, we sample the text at regular intervals by storing the i values such that $A[i]$ is a multiple of s_A. To extract $T_{l,r}$ we actually extract $T_{\lfloor l/s_A \rfloor \cdot s_A, r}$, so as to start from the preceding sampled position. This takes $s_A + r - l + 1$ applications of Ψ.

Another index based on a similar idea is Mäkinen's Compact Suffix Array (Mak-CSA) [Mäk03]. In a suffix array A, a *run* is an interval that appears in another place of the suffix array shifted by 1 (these runs correspond exactly to the runs in Ψ). For example, a run of length $l+1$ that start in j is such that $A[j+s] = A[i+s]+1$ for $0 \leq s \leq l$. If the suffix array can be partitioned into n_r runs, then the Mak-CSA stores n_r blocks. Each block stores the partition where the interval repeats, the offset from the origin of that partition, the length of the repetition, and the explicit value of the first element of the interval. Any run can then be recovered by (recursively) expanding the definition and shifting the explicit values. This index uses $2nH_k \log n + O(n \log\log n)$ bits of space plus the text, and answers *count* in $O(m \log n + \log^2 n / \log\log n)$ time, *locate* in $O(\log^2 n/(\log\log n)^2)$ time, and *extract* of l symbols in $O(l)$ time (the text is available).

2.9.3 The Lempel-Ziv Index

The *Lempel-Ziv index (LZ-index)* is a compressed self-index based on a Lempel-Ziv partitioning of the text. There are several members of this family [Nav04, ANS06, FM05], yet we focus on the version described in [Nav04, ANS06] and available in the *Pizza&Chili* site. This index uses the LZ78 parsing [ZL78] to generate a partitioning of $T_{1,n}$ into n' *phrases*, $T = Z_1, \ldots, Z_{n'}$. These phrases are all different, and each phrase Z_i is formed by appending a single symbol to a previous phrase Z_j, $j < i$ (except for a virtual empty phrase Z_0). Since it holds $Z_i = Z_j \cdot c$, for some $j < i$ and $c \in \Sigma$, the set is prefix-closed. We can then build a trie on these phrases, called LZ78-trie, which consists of n' nodes, one per phrase.

The original LZ-index [Nav04] is formed by (1) the LZ78 trie; (2) a trie formed with the reverse phrases Z_i^r, called the reverse trie; (3) a mapping from phrase identifiers i to the LZ78 trie node that represents Z_i; and (4) a similar mapping to Z_i^r in the reverse phrases. The tree shapes in (1) and (2) are represented using parentheses and the encoding proposed in [MR97] so that they take $O(n')$ bits and constant time to support various tree navigation operations. Yet, we must also store the phrase identifier in each trie node, which accounts for the bulk of the space for the tries. Overall, we have $4n' \log n'$ bits of space, which can be bounded by $4nH_k(T) + o(n \log \sigma)$ for $k = o(\log_\sigma n)$ [NM07]. This can be reduced to $(2+\epsilon)nH_k(T) + o(n \log \sigma)$ [ANS06, ANS08] by noticing that the mapping (3) is essentially the inverse permutation of the sequence of phrase identifiers in (1), and similarly (4) with (2). It is possible to represent a permutation and its inverse using $(1+\epsilon)n' \log n'$ bits of space and access the inverse permutation in $O(1/\epsilon)$ time [MRRR03].

An occurrence of P in T can be found according to one of the following situations:

1. P lies within a phrase Z_i. Unless the occurrence is a suffix of Z_i, since $Z_i = Z_j \cdot c$, P also appears within Z_j, which is the parent of Z_i in the LZ78 trie. A search for P^r in the reverse trie finds all the phrases that have P as a suffix. Then the node mapping permits, from the phrase identifiers stored in the reverse trie, to reach their corresponding LZ78 nodes. All the subtrees of those nodes are occurrences.

2. P spans two consecutive phrases. This means that, for some j, $P_{1,j}$ is a suffix of some Z_i and $P_{j+1,m}$ is a prefix of Z_{i+1}. For each j, we search for $P^r_{1,j}$ in the reverse trie and $P_{j+1,m}$ in the LZ78 trie, choosing the smaller subtree of the two nodes we arrived at. If we choose the descendants of the reverse trie node for $P^r_{1,j}$, then for each phrase identifier i that descends from the node, we check whether $i+1$ descends from the node that corresponds to $P_{j+1,m}$ in the LZ78 trie. This can be done in constant time by comparing preorder numbers.

3. P spans three or more phrases. This implies that some phrase is completely contained in P, and since all phrases are different, there are only $O(m^2)$ different phrases to check, one per substring of P. Those are essentially verified one by one.

Notice that the LZ-index carries out counting and locating simultaneously, which renders it not competitive for counting alone. Extracting text is done by traversing the LZ78 paths upwards from the desired phrases, and then using mapping (3) to continue with the previous or next phrases. The LZ-index is very competitive for locating and extracting, in exchange for using more space than other compressed indexes.

2.10 Re-Pair

Re-Pair [LM00] is a dictionary-based compressor that permits fast local decompression using only the dictionary. It consists of repeatedly finding the most frequent pair of symbols in a sequence and replacing it with a new symbol; this is repeated until every pair appears only once. Given a text $T = t_1 \ldots t_n$ over an alphabet Σ of size σ, the Re-Pair compression algorithm is as follows:

1. Identify the most frequent pair ab in T.
2. Create a new symbol s, larger than all existing symbols in T, and add rule $s \rightarrow ab$ to a dictionary R.
3. Replace every occurrence of ab in T by s.[4]
4. Iterate until every pair has frequency 1.

The result of the compression algorithm is the dictionary of rules R plus the sequence C of (original and new) symbols into which T has been compressed. Note that R can be easily stored as a vector of pairs, so that rule $s \rightarrow ab$ is represented by $R[s - \sigma] = a : b$.

Any portion of C can be easily decompressed in optimal time and fast in practice. To decompress $C[i]$, we first check if $C[i] \leq \sigma$. If it is, then it is an original symbol of T and we are done. Otherwise, we obtain both symbols from $R[C[i] - \sigma]$, and expand them recursively (they can in turn be original or created symbols, and so on). We reproduce u characters of T in $O(u)$ time, and the accesses pattern is local if R is small.

Since R grows by 2 integers (a, b) for every new pair, we could stop creating pairs when the most frequent one appears only twice. R can be further reduced by preempting the process earlier, which trades its size for overall compression ratio.

[4]If $a = b$ it might be impossible to replace all occurrences, e.g. aa in aaa. But, in such case, one can at least replace each other occurrence in a row.

Chapter 3

Compressed Text Indexes: From Theory to Practice

A compressed full-text self-index represents a text in a compressed form and still answers queries efficiently. This represents a breakthrough over the text indexing techniques of the previous decade, whose indexes required several times the size of the text. Although it is relatively new, this technology has matured up to a point where theoretical research is giving way to practical developments. Nonetheless, this deserves significant programming skills, a deep engineering effort, and a strong algorithmic background to dig into the research results. Up to the time of this thesis, only isolated implementations and focused comparisons of compressed indexes had been reported, and they missed a common API, which prevented their re-use or deploy within other applications.

The goal of this chapter is to introduce the *Pizza&Chili* site, which offers tuned implementations and a standardized API for the most successful compressed full-text self-indexes, together with effective testbeds and scripts for their automatic validation and test. We have extensively tested these codes with the aim of demonstrating the practical relevance of this novel and exciting technology.

At the beginning of this thesis, we noted that one of the essential building blocks of many compressed data structures, namely the data structure to perform *rank* and *select* operations over a bit array, had not been carefully studied in practice. Section 3.1 shows some results in this respect, which suggest that in many practical cases simpler solutions are better in terms of time and extra space. These results are a basic building block for almost all the compressed text indexes presented in our *Pizza&Chili* site. Section 3.2 describes how the main practical indexes implement the basic theoretical ideas, in particular including the first implementation of the alphabet-friendly FM-index [FMMN07]. Section 3.3 presents the *Pizza&Chili* site, and next Section 3.4 comments on a large suite of experiments aimed at comparing the most successful implementations of the compressed indexes present in this site.

Finally, Section 3.5 discusses the open challenges derived from this experimental evaluation, which have driven the development of the rest of this thesis.

The *Pizza&Chili* site and the experiments were a joint work with Paolo Ferragina and Rossano Venturini, University of Pisa, Italy.

3.1 Practical Binary Rank and Select Queries

Given a binary sequence $B[1, n]$, in Section 2.4 we saw that operation $rank(B, i)$ is the number of 1's in $B[1 \ldots i]$ and $select(B, j)$ is a kind of inverse of $rank()$, that is, the position of the j-th bit set in B. The first results on this problem achieving constant time on *rank* and *select* [Mun96, Cla96] used $n + o(n)$ bits. In those schemes, n bits are used by B itself and $o(n)$ additional bits are needed by the data structures used to answer *rank* and *select* queries. Further refinements [Pag99, RRR02] achieved constant time on the same queries by using $nH_0(B) + o(n)$ bits overall, where $H_0(B)$ is the zero-order entropy of B (Section 2.1). In this case, a further nontrivial query that is solved is to determine $B[i]$ given i.

It is not clear at all how efficient those solutions are in practice, nor how irrelevant is the $o(n)$ extra space in practical cases. In this section, we will focus on the most promising solutions among those that use $n + o(n)$ bits.

In Section 3.1.1, we analyze the structures used to answer *rank*, beginning with the classical one, to follow with the analysis of several tune-ups. In Section 3.1.2, we study structures used to answer *select* queries, considering first *select* techniques based on *rank* structures, to follow with a structure that answers *select* in constant time. In Section 3.1.3, we study a particular case of *select* dubbed *selectNext*.

All of our experiments in this section ran on an AMD Athlon of 1.5 GHz, 1 GB of RAM, 256 KB L2 cache, running Linux. We use the `GNU gcc` compiler for C++ with full optimizations. We measure user times.

3.1.1 Rank Queries

3.1.1.1 The Constant-Time Classical Solution

The constant-time solution for *rank* is relatively simple [Jac89, Mun96, Cla96]. We divide the bit array into *blocks* of length $b = \lfloor \log(n)/2 \rfloor$. Consecutive blocks are grouped into *superblocks* of length $s = b \cdot \lfloor \log n \rfloor$.

For each superblock j, $0 \leq j \leq \lfloor n/s \rfloor$ we store a number $R_s[j] = rank(B, j \cdot s)$. Array R_s needs overall $n/b = O(n/\log n)$ bits since each $R_s[j]$ value needs $\log n$ bits and there are $n/s = n/(b \log n)$ entries.

For each block k of superblock $j = k \text{ div } \lfloor \log n \rfloor$, $0 \leq k \leq \lfloor n/b \rfloor$ we store a number $R_b[k] = rank(B, k \cdot b) - rank(B, j \cdot s)$. Array R_b needs $(n/b) \log s = O(n \log \log n / \log n)$ bits since there are n/b blocks overall and each $R_b[k]$ value needs $\log s = O(\log \log n)$ bits, as it represents the number of bits set inside a superblock.

Finally, for every possible bit stream S of length b and for every position i inside S, we precompute $R_p[S, i] = rank(S, i)$. This requires $O(2^b \cdot b \cdot \log b) = O(\sqrt{n} \log n \log \log n)$ bits.

The above structures need $O(n/ \log n + n \log \log n / \log n + \sqrt{n} \log n \log \log n) = o(n)$ bits. They permit computing *rank* in constant time as follows:

$$\begin{aligned} rank(B, i) \quad = \quad & R_s[i \text{ div } s] + R_b[i \text{ div } b] + \\ & R_p[B[(i \text{ div } b) \cdot b + 1, (i \text{ div } b) \cdot b + b], i \bmod b] \end{aligned}$$

This structure can be implemented with little effort and promises to work fast. Yet, consider its extra space, for example, for $n = 2^{30}$ bits. R_s poses a space overhead of 6.67%, R_b of 60%, and R_p of 0.18%. Overall, the $o(n)$ additional space is 66.85% of n, which is not so negligible.

3.1.1.2 Resorting to Popcounting

The term *popcount* (population count) refers to counting how many bits are set in a bit array. We note that table R_p can be replaced by popcounting, as $R_p[S, i] = popcount(S \,\&\, 1^i)$, where "&" is the bitwise *and* and 1^i is a sequence of i 1's (obtained for example as $2^i - 1$). This permits us removing the second argument of R_p, which makes the table smaller. In terms of time, we perform an extra *and* operation in exchange for either a multiplication or an indirection to handle the second argument. The change is clearly convenient.

Popcounting can be implemented by several means, from bit manipulation in a single computer register to table lookup. Probably the best example of the first family, which computes $popcount(x)$, is the following

```
bx = x - ((x>>1) & 0x77777777) - ((x>>2) & 0x33333333) - ((x>>3) & 0x11111111)
popcount = ((bx + (bx>>4)) & 0x0F0F0F0F) % 0xFF
```

where a computer word of $w = 32$ bits is assumed. However, we have found that the implementation of `GNU g++` is about twice as fast:

```
popc = { 0, 1, 1, 2, 1, 2, 2, 3, 1, ... }
popcount = popc[x & 0xFF] + popc[(x >> 8) & 0xFF]
         + popc[(x >> 16) & 0xFF] + popc[x >> 24]
```

where popc is a precomputed popcount table indexed by bytes.

Yet, this table lookup solution is only one choice among several alternatives. The width of the argument of the precomputed table has been fixed at 8 bits and b has been fixed at 32 bits, hence requiring 4 table accesses. In a more general setup, we can choose $b = \log(n)/k$ and the width of the table argument to be $\log(n)/(rk)$, for integer constants r and k. Thus, the number of table accesses to compute *popcount* is r and the space overhead for table R_b is $k \log\log n/\log n$. What prevents us to choose minimal r and k is the size of table popc, which is $n^{\frac{1}{rk}} \log\log n$. Hence we need $rk > 1$, which yields a space/time trade-off.

In practice, b should be a multiple of 8 because the solutions to *popcount* work at least by chunks of whole bytes. With the setting $s = b \cdot \lfloor \log n \rfloor$, and considering the range $2^{16} < n \leq 2^{32}$ to illustrate, the overall extra space (not counting R_p) is 112.5% with $b = 8$, 62.5% with $b = 16$, 45.83% with $b = 24$ and 34.38% with $b = 32$.

We have tried the reasonable (k, r) combinations for $b = 16$ and $b = 32$: (1) $b = 32$ and a 16KB popc table needing 2 accesses for *popcount*, (2) $b = 16$ and a 16KB popc table needing 1 access for *popcount*, (3) $b = 16$ and a 256-byte popc table needing 2 accesses for *popcount*, and (4) $b = 32$ and a 256-byte popc table needing 4 accesses for *popcount*. Other choices require too much space or too many table accesses. We have also excluded $b = 8$ because its space overhead is too high and $b = 24$ because it requires non-aligned memory accesses (see later).

Figure 3.1 shows execution times for $n = 2^{12}$ to $n = 2^{30}$ bits. For each size, we randomly generate 200 arrays and average the times of 1,000,000 *rank* queries over each. We compare the four alternatives above (labeled (1)-(4)) as well as the mentioned method that does not use tables (label "no tables"). As it can be seen, the combination (4), that is, $b = 32$ making 4 accesses to a table of 256 entries, is the fastest in most cases, and when it is not, the difference is negligible. The alternative without tables is clearly inferior due to the many operations.

Up to now, we have stuck to $s = b \cdot \lfloor \log n \rfloor$. However, it is preferable to read word-aligned numbers than numbers that occupy other number of bits such as $\log n$, which can cross word boundaries and force reading two words from memory. In particular, we have considered the alternative $s = 2^8$, which permits storing R_b elements as bytes. The space overhead of R_b is thus only 25% with $b = 32$ (and 50% for $b = 16$), and accesses to R_b are byte-aligned. The price for such a small s is that R_s gets larger. For example, for $n = 2^{20}$ it is 7.81%, but the sum is still inferior to the 34.38% obtained with the basic scheme $s = b \cdot \lfloor \log n \rfloor$. Actually, for little more space, we could store R_s values as full 32-bit integers (or 16-bit if $\log n \leq 16$). The overhead factor due to R_s becomes now 32/256 (or 16/256), which is at most 12.5%. Overall, the space overhead is 37.5%, close to that of the non-aligned version. Figure 3.1 shows that this alternative is much faster than any other, and it will be our choice for popcount-based methods.

Note that up to $n = 2^{20}$ bits, the original bit array together with the additional

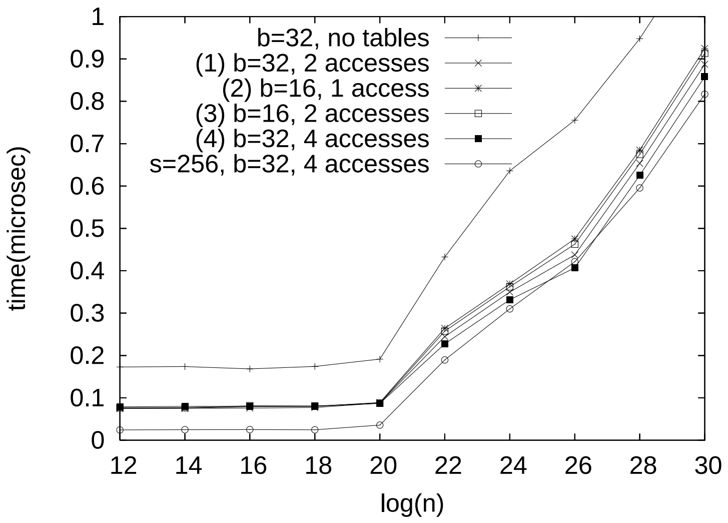


Figure 3.1: Comparison of different popcount methods to solve *rank*.

structures need at most 176 KB with $b = 32$, and 208 KB with $b = 16$. Thus our 256 KB cache accommodates the whole structure. However, for $n = 2^{22}$, we need 512 KB just for the bit array. Thus the cache hit ratio decreases as n grows.

3.1.1.3 Using a Single Level Plus Sequential Scan

At this point, we still follow the classical scheme in the sense that we have two levels of blocks, R_s and R_b. This forces us to make two memory accesses in addition to accessing the bit array block. We consider now the alternative of using the same space to have a single level of blocks, R_s, with one entry each $s = 32 \cdot k$ bits, and using a single 32-bit integer to store the ranks. To answer a $rank(B, i)$ query, we would first find the latest R_s entry that precedes i, and then sequentially scan the array, popcounting in chunks of $w = 32$ bits, until reaching the desired position. The procedure looks as follows:

$$\begin{aligned} rank(B,i) &= R_s[i \text{ div } s] \\ &+ \sum_{j=(((i \text{ div } s)\cdot s) \text{ div } w)+1,(i \text{ div } w)-1} popcount(B[j \cdot w + 1 \ldots j \cdot w + w]) \\ &+ popcount(B[(i \text{ div } w) \cdot w + 1, (i \text{ div } w) \cdot w + w] \ \& \ 1^{i \bmod w}) \end{aligned}$$

Note that the sequential scan accesses at most k memory words, and the space overhead is $1/k$. Thus, we have a space/time trade-off. For example, with $k = 3$ we have approximately the same space overhead as in our preferred two-level version.

Figure 3.2 compares the execution time of different trade-offs against the classical and the best previous alternatives. For the variant of using only one level of blocks, we have considered extra spaces of 5%, 10%, 25%, 33% (close to the space of our best two-level alternative), and 50%.

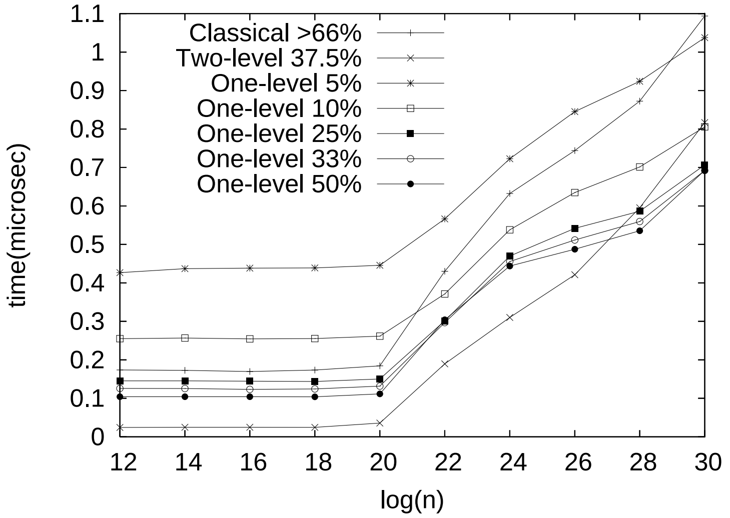


Figure 3.2: Comparison of different approaches to solve *rank*: classical solution, popcounting with two levels of blocks, and popcounting with one level of blocks.

We can see that the implementation of the classical solution, presented in 3.1.1.1, is far from competitive: It wastes the most space and is among the slowest. Our two-level popcount alternative is usually the fastest by far, showing that the use of two levels of blocks plus an access to the bit array is normally better than using the same space (and even more) for a single level of blocks. Yet, note that the situation is reversed for large n. The reason is the locality of reference of the one-level versions: They perform one access to R_s and then a few accesses to the bit array (on average, 1 access with 50% overhead, 1.5 accesses with 33% overhead and 2 accesses with 25% overhead). Those last accesses are close to each other, thus from the second on they are surely cache hits. On the other hand, the two-level version performs three accesses (R_s, R_b, and the bit array) with no locality among them (access to R_p is surely a cache hit). When the cache hit ratio decreases significantly, those three nonlocal accesses become worse than the two nonlocal accesses (plus some local ones) of the one-level versions.

Thus, which is the best choice among one and two levels depends on the application. Two levels is usually better, but for large n one can use even less space and be faster. Yet, there is no fixed concept of what is “large”, as other data structures may compete for the cache and thus the real limit can be lower than in our experiments, where only the *rank* structures are present.

3.1.2 Select Queries

3.1.2.1 Binary Searching with Rank

A simple, yet $O(\log n)$ time, solution to $select(B, j)$, is to binary search in B the position i such that $rank(B, i) = j$ and $rank(B, i-1) = j-1$. Hence, the same structures used to compute $rank(B, i)$ in constant time can be used to compute $select(B, j)$ in $O(\log n)$ time.

More efficient than using $rank(B, i)$ as a black box in this scheme is to take advantage of its layered structure, so as to first binary search for the proper superblock using R_s, then binary search that superblock for the proper block using R_b, and finally binary search for the position inside the correct block.

For the search within the superblock of s bits, there are three alternatives: ($2a$) binary search using R_b, ($2b$) sequential search using R_b (since there are only a few blocks inside a superblock), and ($2c$) sequential search using popcount. The latter alternative consists of simply counting the number of bits set inside the superblock, and has the advantage of not needing array R_b at all. For the search in the last block of b bits, binary search makes little sense because *popcount* proceeds anyway bytewise, so we have considered two alternatives: ($3a$) bytewise search using popcount plus bitwise search in the final byte, and ($3b$) sequential bitwise search of the b bits.

In the case of *select*, the density of the bit array may be significant. We have generated bit arrays of densities (fraction of bits set) from 0.001 to 1. For each density we randomly generated 50 different arrays of each size. For each array, we average the times of 400,000 *select* queries.

The results in this subsection are almost independent of the density of bits set in B, hence we show in Figure 3.3 only the case of density 0.4. We first compare alternatives $(2a, 3b)$, $(2b, 3b)$ and $(2c, 3b)$. Then, as $(2c, 3b)$ turns out to be the fastest, we consider also $(2c, 3a)$, which is consistently the best. We have also plotted the basic binary search (not level-wise) to show that it is much slower than any other. In this experiment, we have used $b = 32$ and $s = b \cdot \lfloor \log n \rfloor$. Note that the best alternative only requires space for R_s, as all the rest is solved with sequential scanning.

From Figure 3.3 we conclude that using a single level is preferable for *select*. Now, to speed up the access to R_s we consider to use 32-bit integers (or 16-bits when $\log n \leq 16$) instead of $\log n$ bits. Moreover, we can choose any sampling step of the form $s = k \cdot b = k \cdot 32$ so that the sequential scan accesses at most k blocks and we pay $\frac{1}{s} \cdot 32 = 1/k$ overhead.

Figure 3.4 compares different space overheads, from 5% to 50%. We also include the case $(2c, 3a)$, which is the best version where $s = b \cdot \lfloor \log n \rfloor$ and R_s stores $\log n$ bits per entry instead of 32 bits; the overhead of this version is $\frac{1}{32 \log n} \cdot \log n = 1/32$. It can be seen that these word-aligned alternatives are generally faster than using exactly $\log n$ bits for R_s. Moreover, there is a clear cache effect as n grows. For small n, higher space overheads yield

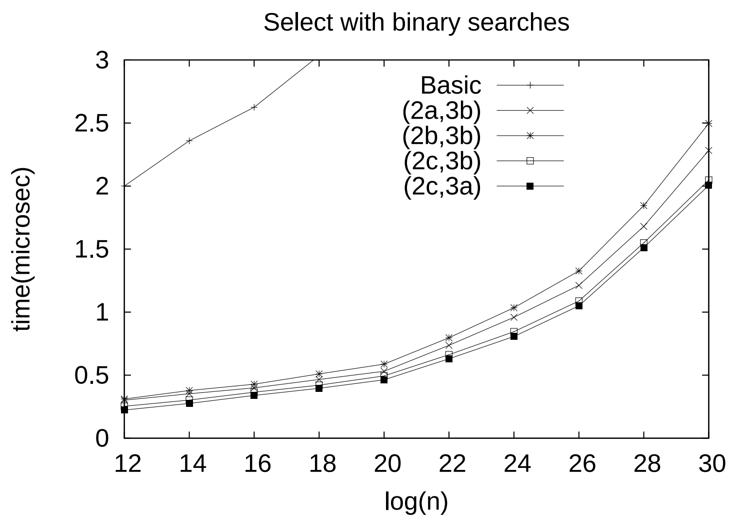


Figure 3.3: Comparison of alternatives to solve *select* by binary search.

better times as expected, albeit the difference is not large because the binary search on R_s is a significant factor that smoothes the differences in the sequential search. For larger n, the price of the cache misses during the binary search in R_s is the dominant factor, thus lower overheads take much less time because their R_s arrays are smaller and their cache hit ratios are higher. The sequential search, on the other hand, is not so important because only the first access may be non-local, all the following ones are surely cache hits. Actually, the variant of $(2c, 3a)$ with, $\frac{100}{32}\% = 3.125\%$ overhead is finally the fastest, albeit it stores non-aligned R_s values.

The best alternative is the one that balances the number of cache misses during binary search on R_s with those occurring in the sequential search on the bit array. It is interesting, however, that a good solution for *select* requires little space. Still, note that *rank* can be made one order of magnitude faster by using more space (see Figure 3.2).

3.1.2.2 The Constant-Time Solution

The constant-time solution to $select(B, j)$ is significantly more complex than for $rank(B, i)$. We focus on Clark's version [Cla96], as the previous solution [Jac89] is $O(\log\log n)$ time. Clark's structure requires $\frac{3n}{\lceil\log\log n\rceil} + O(\sqrt{n}\log n \log\log n)$ bits of extra space. The idea is to use a three-level directory tree to store information of the bits set in B.

The first directory, $d1$, stores the position of each $(\lceil\log n\rceil\lceil\log\log n\rceil)$-th bit set. Each of these entries requires $\lceil\log n\rceil$ bits, totalizing $\left\lfloor\frac{n}{\lceil\log\log n\rceil}\right\rfloor$ bits for $d1$. The entries of $d1$ are

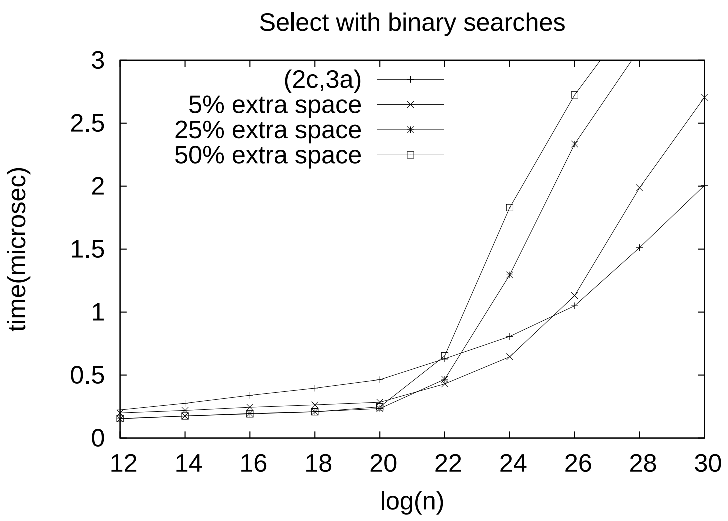


Figure 3.4: Comparison of different space overheads for *select* based on binary search.

defined as $d1_0 = 0$ and

$$d1_k \;=\; select(B, \lceil \log n \rceil \lceil \log \log n \rceil \cdot k),\ 1 \le k \le \left\lfloor \frac{n}{\lceil \log n \rceil \lceil \log \log n \rceil} \right\rfloor.$$

In the second directory, $d2$, let us define $r_k = d1_k - d1_{k-1}$, $1 \le k \le \left\lfloor \frac{n}{\lceil \log n \rceil \lceil \log \log n \rceil} \right\rfloor$, that is, the number of bits in B between two successive bits represented in $d1$. We state that the representation for each such range should occupy $\left\lfloor \frac{r_k}{\lceil \log \log n \rceil} \right\rfloor$ bits, so that added over all ranges we occupy $\left\lfloor \frac{n}{\lceil \log \log n \rceil} \right\rfloor$ bits for $d2$. It is easy to find the data of the k-th range in $d2$, as the sum of the lengths of all previous ranges is $r_1 + r_2 + \ldots r_{k-1} = d1_{k-1}$, thus the k-th range starts at position $1 + \left\lfloor \frac{d_{k-1}}{\lceil \log \log n \rceil} \right\rfloor$.

Let us see how to reach the bound $\left\lfloor \frac{r_k}{\lceil \log \log n \rceil} \right\rfloor$ bits per range. There are $\lceil \log n \rceil \lceil \log \log n \rceil$ bits set in each range. Explicitly storing all those positions requires $\lceil \log n \rceil^2 \lceil \log \log n \rceil$ bits. There are two cases. If

$$\left\lfloor \frac{r_k}{\lceil \log \log n \rceil} \right\rfloor \ge \lceil \log n \rceil^2 \lceil \log \log n \rceil \tag{3.1}$$

then we have enough space to store explicitly all the positions of bits set. Thus, for these entries k, we store all

$$d2_{k,j} \;=\; select(B[d1_{k-1} + 1 \ldots d1_k], j),\ 1 \le j \le \lceil \log n \rceil \lceil \log \log n \rceil.$$

Otherwise, we have

$$r_k < (\lceil \log n \rceil \lceil \log \log n \rceil)^2 \tag{3.2}$$

and thus we subdivide the range, in the same way as for $d1$, except that now we store the position of every ($\lceil \log r_k \rceil \lceil \log \log n \rceil$)-th bit set. This position is stored as an offset inside the range, thus we need $\lceil \log r_k \rceil$ bits per entry. As there are at most $\left\lfloor \frac{r_k}{\lceil \log r_k \rceil \lceil \log \log n \rceil} \right\rfloor$ entries, we use at most $\left\lfloor \frac{r_k}{\lceil \log \log n \rceil} \right\rfloor$ bits per range. Thus, $d2_{k,0} = 0$ and we store all

$$d2_{k,j} \quad = \quad \mathit{select}(B[d1_{k-1} + 1 \ldots d1_k], \lceil \log r_k \rceil \lceil \log \log n \rceil \cdot j),\ 1 \leq j \leq \left\lfloor \frac{r_k}{\lceil \log r_k \rceil \lceil \log \log n \rceil} \right\rfloor .$$

Finally, for this second case, we use a third directory, $d3$. Let us define $r'_{k,j} = d2_{k,j} - d2_{k,j-1}$, that is, the number of bits in B between two consecutive entries of $d2_k$. In this range, there are at most $\lceil \log r_k \rceil \lceil \log \log n \rceil$ bits set. In order to explicitly store those positions, we need $\lceil \log r'_{k,j} \rceil \lceil \log r_k \rceil \lceil \log \log n \rceil$ bits. Again, we require to use $\left\lfloor \frac{r'_{k,j}}{\lceil \log \log n \rceil} \right\rfloor$ bits per subrange, so that after adding over j we do not surpass the $\left\lfloor \frac{r_k}{\lceil \log \log n \rceil} \right\rfloor$ bits limit per range. We have again two cases. If

$$\left\lfloor \frac{r'_{k,j}}{\lceil \log \log n \rceil} \right\rfloor \geq \lceil \log r'_{k,j} \rceil \lceil \log r_k \rceil \lceil \log \log n \rceil \tag{3.3}$$

then we have enough space to store them explicitly. Thus for this k, j, we store all

$$d3_{k,j,i} \quad = \quad \mathit{select}(B[d1_{k-1} + d2_{k,j-1} + 1 \ldots d1_{k-1} + d2_{k,j}], i),\ 1 \leq i \leq \lceil \log r_k \rceil \lceil \log \log n \rceil .$$

For the remaining case we have

$$r'_{k,j} < \lceil \log r'_{k,j} \rceil \lceil \log r_k \rceil \lceil \log \log n \rceil^2 \tag{3.4}$$

and together with Eq. (3.2) it is easy to derive $r'_{k,j} < 16 \lceil \log \log n \rceil^4$, which is asymptotically smaller than $\lceil \log n \rceil$. Therefore, we set up a table containing, for each possible sequence of $\lceil \log(n)/2 \rceil$ bits, the number of bits set in the sequence and the position of each of them. The space required by this table is $O(\sqrt{n} \log n \log \log n)$. With a constant number of lookups into this table, we solve *select* for a subrange of length $r'_{k,j}$: take successive chunks of $\lceil \log n \rceil$ bits and track how many bits set have we seen, until finding the final answer in the chunk where the argument of *select* is reached or exceeded.

Thus, the overall space we use is $\frac{3n}{\lceil \log \log n \rceil} + O(\sqrt{n} \log n \log \log n)$ bits. For example, for $n = 2^{30}$, the overhead is 60%. Figure 3.5 shows the pseudocode for $\mathit{select}(B, \ell)$.

Figure 3.6 shows the execution times for our implementation of Clark's *select* (we show different lines for different densities of the bit arrays). We note that, although the time is $O(1)$, there are significant differences as we change n or the density of the arrays (albeit of course those differences are bounded by a constant). Note, for example, that for density 0.001 the search is extremely fast, as there is only one superblock and all the answers are explicitly stored in $d2$.

```
Algorithm $select(B, \ell)$
    $k \leftarrow \ell$ div $\lceil \log n \rceil \lceil \log \log n \rceil$
    $j \leftarrow \ell$ mod $\lceil \log n \rceil \lceil \log \log n \rceil$
    if $j = 0$ return $d1_k$
    $r_k \leftarrow d1_k - d1_{k-1}$
    if $r_k \geq (\lceil \log n \rceil \lceil \log \log n \rceil)^2$ return $d1_k + d2_{k,j}$
    $j' \leftarrow j$ div $\lceil \log r_k \rceil \lceil \log \log n \rceil$
    $i \leftarrow j$ mod $\lceil \log r_k \rceil \lceil \log \log n \rceil$
    if $i = 0$ return $d1_k + d2_{k,j'}$
    $r'_{k,j'} \leftarrow d2_{k,j'} - d2_{k,j'-1}$
    if $r'_{k,j'} \geq \lceil \log r'_{k,j'} \rceil \lceil \log r_k \rceil \lceil \log \log n \rceil^2$ return $d1_k + d2_{k,j'} + d3_{k,j',i}$
    use the lookup table to find the position $p$ of the $i$-th bit set in $B[d1_k + d2_{k,j'} + 1, n]$
    return $d1_k + d2_{k,j'} + p$
```

Figure 3.5: Pseudocode to calculate $select(B, \ell)$

Density 0.01 is the strangest case, as times grow and then decrease with n. The reason is that, for fixed density d, we have that the expectation of r_k is $\frac{1}{d} \log n \log \log n$, and thus the probability of storing $d2$ values explicitly decreases as n grows (see Eq. (3.1)). The same happens to $d3$ in Eq. (3.3). When n is large enough, however, there is a beneficial effect related to the final sequential scanning, which is constant but quite large. That is, the number of steps of the final sequential scan is $2r'_{k,j}/\log n$, where the expectation of $r'_{k,j}$ is $\frac{1}{d} \log \log n (\log \log n + \log \log \log n + \log \frac{1}{d})$, and therefore this heavy part of the computation decreases with n.

For higher densities, the last sequential scan is not that heavy and thus there is no such a significant decrease of the cost for large n. As n grows the probability of storing explicit values in $d2$ and $d3$ decreases. This effect does show up, although the effect is milder as the density increases. This is why, once all directory levels are used, times decrease as the density increases.

The plot also shows the time for our binary search versions using 5% and 50% space overhead. For very low densities (up to 0.005 and sometimes 0.01), Clark's implementation is superior. However, we note that for such low densities, the *select* problem is trivially solved by explicitly storing all the positions of all the bits set (that is, precomputing all answers), at a space overhead that is only 32% for density 0.01. Hence this case is not very interesting. For higher densities, our binary search versions are superior up to $n = 2^{22}$ or 2^{26} bits, depending on the space overhead we chose (and hence on how fast we want to be for small n). After some point, however, the $O(\log n)$ nature of the binary search solution shows up, and the constant-time solution of Clark finally takes over. We remark that Clark's implementation imposes a space overhead of 60% at least. Moreover, Clark's *select* is totally oriented to answer queries over 1's, if we want $select_0$ we must replicate all the structure. The case of binary search is easily adapted to answer $select_0$ (remember that $rank_0(B, i) = i - rank_1(B, i)$).

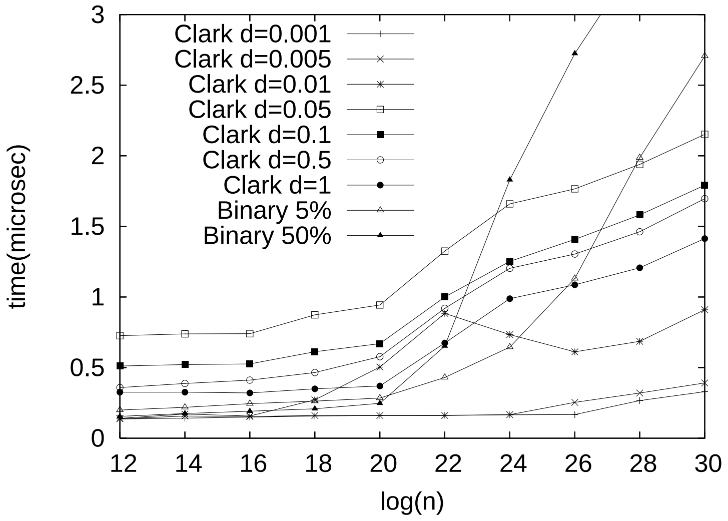


Figure 3.6: Comparison of Clark's select on different densities and two binary searches based implementations using different space overheads.

3.1.3 SelectNext Queries

It turns out that several applications require a restricted version of *select*, namely $select(B, i+1)$ given $j = select(B, i)+1$. We call this operation $selectNext(B, j)$, which gives the position of the first bit set in $B[j \ldots n]$, or $n + 1$ if no such bit set exists.

Of course one choice is to exploit the equivalence $selectNext(B, j) = select(B, 1 + rank(B, j - 1))$ and solve the query using the traditional *rank* and *select*. We propose now a simpler solution, specific for *selectNext*.

We divide B as for *rank*, into blocks and superblocks of sizes $b = \lfloor \frac{1}{2} \log n \rfloor$ and $s = b \cdot \lfloor \log n \rfloor$, respectively. For each superblock j, $1 \le j \le \lfloor n/s \rfloor$ we store a number $N_s[j] = selectNext(B, j \cdot s + 1)$. Array N_s needs overall $O(n/\log n)$ bits since each $N_s[j]$ value needs $O(\log n)$ bits.

For each block k of superblock $j = k$ div $\lfloor \log n \rfloor$, $1 \le k \le \lfloor n/b \rfloor$, we store a number $N_b[k] = selectNext(B[j \cdot s + 1 \ldots (j + 1) \cdot s], k \cdot b - j \cdot s + 1)$. Array N_b needs overall $O(n \log\log n/\log n)$ bits since each $N_b[k]$ value needs $O(\log\log n)$ bits, as it represents a position inside a superblock of length $O(\log^2 n)$.

Finally, for every possible bit stream S of length b and for every position i inside S, we precompute $N_p[S, i] = selectNext(S, i)$. This requires $O(2^b \cdot b \cdot \log b) = O(\sqrt{n} \log n \log\log n)$ bits. Note that $N_p[S, i] = b + 1$ if $S[i \ldots b]$ contains all zeros.

As before, the structures require $O(n \log\log n/\log n)$ bits, with exactly the same overhead as for *rank*. With them, we can answer $selectNext(B, i)$ in $O(1)$ time as as shown in Figure 3.7.

```
**Algorithm** $selectNext(B, i)$
$i_b \leftarrow (i \text{ div } b) \cdot b$
$pos \leftarrow N_p[B[i_b + 1 \ldots i_b + b], i - i_b + 1]$
if $pos \leq b$ return $i_b + pos$ // there is a bit set in $B[i \ldots i_b + b - 1]$
// at this point $selectNext(B, i) = selectNext(B, i_b + b)$, so find the
// answer corresponding to the beginning of the next block
$pos \leftarrow N_b[(i \text{ div } b) + 1]$
if $pos \leq s$ return $((i_b + b) \text{ div } s) \cdot s + pos$
// at this point there are all zeros in $B[i \ldots i_s + s - 1]$,
// where $i_s = (i \text{ div } s) \cdot s$, so $selectNext(B, i) = selectNext(B, i_s + s)$
return $N_s[(i \text{ div } s) + 1]$
```

Figure 3.7: Pseudocode to calculate $selectNext(B, i)$

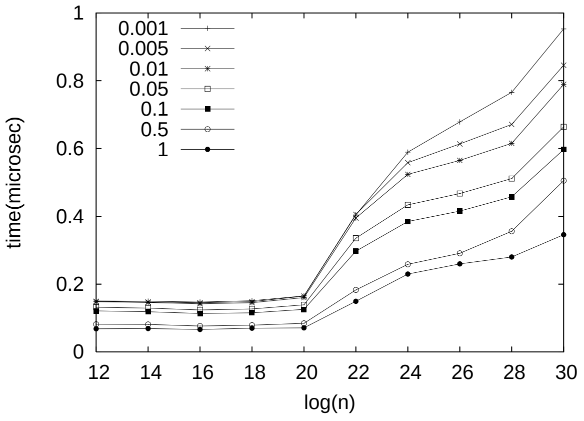
A much simpler alternative solution to $selectNext(B, j)$ is to sequentially scan all the bits in $B[j \ldots n]$ until finding a bit set. We search word by word rather than bit by bit (that is, we scan B from i onwards until finding the first word different from zero). When we finally find a nonzero word, we use a precomputed table that, for every byte, tells the position of the first bit set. Then, in at most four access to the table, we find the position of the first bit set in the word where the sequential scanning stopped. We have considered other alternatives such as (1) using two accesses to a table of 2^{16} entries for the last step, and (2) going by chunks of 16 bits and performing one single access to a table of 2^{16} entries for the last step, but these were slightly worse.

Figure 3.8 compares both solutions, for different densities. As it can be seen, the brute force solution (sequential scan) not only requires much less extra space but also is consistently faster, even with densities as low as 1/1000 (that is, where we have to scan 500 bits on average to find the answer). Note that the constant-time solution also worsens for lower densities, because it is more probable to require more accesses (1, 2 or 3 table accesses). Finally, we point out that the solutions for *selectNext* are significantly faster than any solution for general *select* and hence for *rank-select* implementation of *selectNext*.

3.1.4 Discussion

We have obtained in this section several important practical results on *rank*, *select* and *selectNext* on binary sequences.

- For *rank*, it is better to replace the table access of the third level by popcounting, yet the other two levels have to be maintained to obtain competitive times. It is also beneficial to word-align the numbers, even at the cost of slightly more space. The space overhead of our two-level version is 37.5%, half of the theoretical solution in practice.



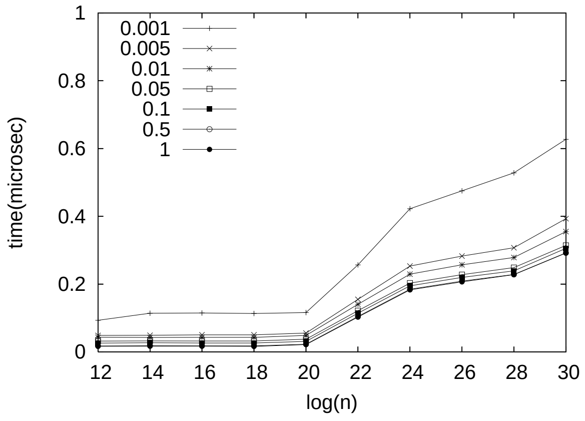


Figure 3.8: Comparison of different alternatives to solve *selectNext*, with different bit densities. On the left, the constant-time solution, on the right, sequential scanning.

A one-level version requiring even less space is competitive for large bit arrays due to cache effects. In our machine, *rank* needs 0.02 to 0.7 microseconds as n moves from 2^{20} to 2^{30}.

- For *select*, the constant-time solution by Clark requires at least 60% extra space in practice, while a simple $O(\log n)$ binary search based on *rank* succeeds with very little space overhead (5%, for example). The binary search is actually faster until the bit array gets very long and the constant time of Clark takes over. This comparison holds for sufficiently dense bit sequences, yet for sparser ones even more naive solutions are the best. In our machine, *select* requires 0.2 microseconds for $n \leq 2^{20}$. For $n = 2^{30}$, it requires 0.4 to 2.2 depending on the density.

- For *selectNext*, even with very sparse bit sequences (density of bits set $d = 1/1000$), a brute-force solution based on $O(1/d)$-time sequential scanning and almost no extra space is faster than a constant-time and $O(n \log\log n / \log n)$ extra space solution we design in Section 3.1.3. In our machine, *selectNext* takes 0.02 to 0.4 microseconds as n goes from 2^{20} to 2^{30}, if $d > 1/1000$. For the case $d = 1/1000$, the times are 0.1 to 0.6 microseconds.

- New developments in the field of microprocessors could improve our practical performance, by not relying on the tables used to answer *popcount* and *selectNext*. In Phenom microprocessors (AMD) it was introduced support of SSE4a, which contains two new interesting instructions: *POPCNT*, Population count (count number of bits set to 1); and *LZCNT*, Leading Zero Count, which can be used in conjunction with a bit masking operation to support *selectNext*. Also, in Nehalem microprocessors it will be introduced support of SSE4.2, which contains the *POPCNT* instruction.

This research has not only interest by itself, but it also has great impact over the text indexing structures that use *rank* and *select*. In the next section we use these new structures in our practical compressed text indexes.

3.2 Implementing Indexes

Over the course of this thesis, we implemented various indexes of the FM-index family. In this section, we show how to achieve practical implementations. In particular, we present the only existing implementation of the so-called *Alphabet-Friendly FM-index*, which provides the best current theoretical space/time guarantees. We are not going into the implementation details of the compressed suffix array (CSA) or the Lempel-Ziv index (LZ-index), as those were already implemented. However, those indexes are explained in Sections 2.9.2 and 2.9.3.

3.2.1 Implementing the FM-index

As can be seen in Section 2.9.1, all the query complexities in the FM-index are governed by the time required to obtain $C[c]$, $T^{bwt}[i]$, and $rank_c(T^{bwt}, i)$ (all of them implicit in LF as well). While C is a small table of $\sigma \log n$ bits, the other two are problematic. Counting requires up to $2m$ calls to $rank_c$, locating requires s_A calls to $rank_c$ and T^{bwt}, and extracting ℓ symbols requires $s_A + \ell$ calls to $rank_c$ and T^{bwt}. In what follows, we briefly comment on the solutions adopted to implement those basic operations.

The original FM-index implementation (*FM-index* [FM01]) compresses T^{bwt} by splitting it into blocks and using independent zero-order compression on each block. Values of $rank_c$ are precomputed for all block beginnings, and the rest of the occurrences of c from the beginning of the block to any position i are obtained by sequentially decompressing the block. The same traversal finds $T^{bwt}[i]$. This is very space-effective: It approaches in practice the k-th order entropy because the partition into blocks takes advantage of the local compressibility of T^{bwt}. On the other hand, the time to decompress the block makes computation of $rank_c$ relatively expensive. For locating, this implementation marks the BWT positions where some chosen symbol c occurs, as explained in Section 2.9.1.

A very simple and effective alternative to represent T^{bwt} has been proposed with the *Succinct Suffix Array* (*SSA*) [FMMN07, MN05]. It uses a Huffman-shaped wavelet tree (Section 2.4), plus the marking of one out of s_A text positions for locating and extracting. The space is $n(H_0(T)+1)+o(n \log \sigma)$ bits, and the average time to determine $rank_c(T^{bwt}, i)$ and $T^{bwt}[i]$ is $O(H_0(T)+1)$. The space bound is not much appealing because of the zero-order compression, but the relative simplicity of this index makes it rather fast in practice. In particular, it is an excellent option for DNA text, where the k-th order compression is not much better than the zero-th order one, and the small alphabet makes $H_0(T) \leq \log \sigma$ small also in absolute terms.

For performance reasons, we want that length of the Huffman code for any symbol not to exceed 32 bits (our word size). This way, we need fewer operations to manage the codes, which are used in all the operations that access the wavelet tree. To obtain all codes shorter or equal to 32 bits, we can use an optimal construction as proposed in [ML00], but we prefer to use a simpler approach, that loses at most 0.2% of compression for $n \leq 10^9$. (In our text collections, the code lengths go from 12 to 27 bits, so this rarely occurs in practice.) Let π_1 be the probability of the least frequent symbol. An upper bound to the maximum length of a code is $\min\{\lfloor -\log_\phi \pi_1 \rfloor, \sigma - 1\}$, where $\phi = (1+\sqrt{5})/2$ [Bur93]. Thus, if $\pi_1 \geq \pi = \phi^{-32} \geq 0.206 \cdot 10^{-6}$, then no Huffman code is longer than 32 bits. So we just need to increase the frequency of the symbols with probability lower than π, by virtually adding more of these symbols at the end of the text. Now we construct the Huffman code for this extended text, and apply it to encode the original text. The result is shorter than applying the code to the virtual text (as more symbols are to be encoded with the same code), and this in turn is shorter than applying the original code to the virtual text (as the original code is not optimal for the virtual text). Hence we analyze the latter to upper bound the extra space incurred. We lose at most $(\sigma - 32)\frac{n}{\phi^{32}} \log_\phi n$ bits of space, where $\sigma - 32$ is an upper bound to the number symbols with Huffman code greater than 32, $\frac{n}{\phi^{32}}$ is an upper bound to the number of symbols that we need to add to obtain the probability π, and $\log_\phi n$ ($\pi_1 \geq 1/n$) is an upper bound to the length in bits for any Huffman code. For example, for a text of size 200MB, our scheme implies that any symbol must appear at least 44 times and we lose at most 0.2% of compression.

The *Run-Length FM-index* (*RLFM*) [MN05] was introduced to achieve k-th order compression by applying *run-length compression* to T^{bwt} prior to building a wavelet tree on it. The BWT generates long runs of identical symbols on compressible texts [MN05], which makes the RLFM an interesting alternative in practice. The price is that the mappings from the original to the run-length compressed positions slow down the query operations somewhat, in comparison to the SSA. This index uses $nH_k \log\sigma + 2n + o(n\log\sigma)$ bits of space, and answers *count* in $O(m(1+\frac{\log\sigma}{\log\log n}))$ time, *locate* in $O(\log^{1+\epsilon} n)$ time and *extract* of l symbols in $O(l(1+\frac{\log\sigma}{\log\log n}) + \log^{1+\epsilon} n)$ time. These complexities are for $\sigma = o(n)$, $k \leq \alpha \log_\sigma n$, and constants $0 < \alpha < 1$ and $\epsilon > 0$. The implementation of the RLFM uses a Huffman-shaped wavelet tree over the first symbols of the runs.

3.2.2 The Alphabet-Friendly FM-index

The *Alphabet-Friendly FM-index* (*AF-index*) [FMMN07] resorts to the definition of k-th order entropy in Eq. (2.2) of Section 2.1, by encoding each substring w_S up to its zero-order entropy. Since all the w_S are contiguous in T^{bwt} (regardless of which k value we are considering), it suffices to split T^{bwt} into blocks given by the k-th order contexts, for any desired k, and to use a Huffman-shaped wavelet tree (see Section 2.4) to represent each such block. In addition, we need all $rank_c$ values precomputed for every block beginning, as the local wavelet trees can only answer $rank_c$ within their blocks. In total, this achieves $nH_k(T) + o(n\log\sigma)$ bits, for moderate $k \leq \alpha \log_\sigma n$ and constant $0 < \alpha < 1$.

Actually, the AF-index does better, by splitting T^{bwt} in an *optimal way*, thus guaranteeing that the space bound above holds simultaneously for every k. This is done by resorting to the idea of *compression boosting* [FM04, GS03, FGMS05]. The compression booster finds the optimal partitioning of T^{bwt} into t nonempty blocks, $s_1, \ldots, s_t$, assuming that each block s_j will be represented using $|s_j|H_0(s_j) + f(|s_j|)$ bits of space, where $f(\cdot)$ is a nondecreasing concave function supplied as a parameter. Given that the partition is optimal, it can be shown that the resulting space is upper bounded by $nH_k + \sigma^k f(n/\sigma^k)$ bits *simultaneously for every* k. That is, the index is not built for any specific k.

The AF-index represents each block s_j by means of a Huffman-shaped wavelet tree wt_j, which will take at most $|s_j|(H_0(s_j)+1) + \sigma \log n$ bits. The last term accounts for the storage of the Huffman code. In addition, for each block j we store an array $C_j[c]$, which tells the $rank_c$ values up to block j. This accounts for other $\sigma \log n$ bits per block. Finally, we need a bitmap $R[1, n]$ indicating the starting positions of the t blocks in T^{bwt}. Overall, the formula giving the excess of storage over the entropy for block j is $f(|s_j|) = 2|s_j| + 2\sigma \log n$.

To carry out any operation at position i, we start by computing the block where position i lies, $j = rank_1(R, i)$, and the starting position of that block, $i' = select_1(R, j)$. Hence $T^{bwt}[i] = s_j[i'']$, where $i'' = i - i' + 1$ is the offset of i within block j. Then, the different operations are carried out as follows.

- For counting, we use the algorithm of Figure 2.1. In this case, we have $rank_c(T^{bwt}, i) = C_j[c] + rank_c(s_j, i'')$, where the latter is computed using the wavelet tree wt_j of s_j.
- For locating, we use the algorithm of Figure 2.2. In this case, we have $c = T^{bwt}[i] = s_j[i'']$. To compute $s_j[i'']$, we also use the wavelet tree wt_j of s_j.
- For extracting, we proceed similarly as for locating, as explained in Section 2.9.1.

As a final twist, R is actually stored using $2\sqrt{nt}$ rather than n bits. We cut R into $\sqrt{nt}$ chunks of length $\sqrt{n/t}$. There are at most t chunks which are not all zeros. Concatenating them all requires only $\sqrt{nt}$ bits. A second bitmap of length $\sqrt{nt}$ indicates whether each chunk is all-zero or not. It is easy to translate *rank* and *select* operations into this representation. A refinement of this method was later called *recrank* in the literature [OS07].

Using binary wavelet trees, the AF-index complexities are $O(m \log \sigma)$ for counting, $O(\log^{1+\epsilon} n)$ for locating, and $O(\ell \log \sigma + \log^{1+\epsilon} n)$ for extracting ℓ symbols.

A stronger version of wavelet trees are *multiary wavelet trees* [FMMN07], which achieve the same space but reduce all $\log \sigma$ to $1 + \frac{\log \sigma}{\log \log n}$ in the time complexities. The trick is to make the tree ρ-ary for some $\rho = O(\log^\alpha n)$ and constant $0 < \alpha < 1$, so that its height is reduced. Now the tree does not store a bitmap per level, but rather a sequence over an alphabet of size ρ. They show how to do *rank*/*select* on those sequences in constant time for such a small ρ.

Using multiary wavelet trees, we achieve in theory the complexities claimed in Section 2.9. In practice, however, using $\rho > 2$ yields much higher space occupancy, due to the extra space needed to represent some partial sums for the ρ symbols. This also applies to the practical implementation of the SSA and the RLFM. In all cases, we opted for (binary) Huffman-shaped wavelet trees.

3.3 The Pizza&Chili Site

The *Pizza&Chili* site has two mirrors: one in Chile (`http://pizzachili.dcc.uchile.cl`) and one in Italy (`http://pizzachili.di.unipi.it`).[1] Its goal is to push towards the technology transfer of this algorithmic development. In order to achieve this goal, the *Pizza&Chili* site offers publicly available and highly tuned implementations of various compressed indexes. The implementations follow a suitable C/C++ API of functions which should allow any programmer to easily plug the provided compressed indexes within his/her own software. The site also offers a collection of texts for experimenting with and validating the compressed indexes. In detail, it offers three kinds of material:

- A set of compressed indexes which are able to support the search functionalities of classical full-text indexes (e.g., substring searches), but require succinct space occupancy and offering, in addition, some text access operations that make them useful within text retrieval and data mining software systems.
- A set of text collections of various types and sizes useful to test experimentally the available (or new) compressed indexes. The text collections have been selected to form a representative sample of different applications where indexed text searching might be useful. The size of these texts is large enough to stress the impact of data compression over memory usage and CPU performance. The goal of experimenting with this testbed is to conclude whether compressed indexing is beneficial over uncompressed indexing approaches, like suffix trees and suffix arrays. And, in case it is beneficial, which compressed index is preferable according to the various application scenarios represented by the testbed.
- Additional material useful to experiment with compressed indexes, such as scripts for their automatic validation and efficiency test over the available text collections.

The *Pizza&Chili* site hopes to mimic the success and impact of other initiatives, such as *data-compression.info* and the *Calgary* and *Canterbury* corpora, just to cite a few. Actually, the *Pizza&Chili* site is a mix, as it offers both software and testbeds. Several people have already contributed to make this site work and, hopefully, many more will contribute to turn it into a reference for all researchers and software developers interested in experimenting and

[1] Up to now, we have counted, in the Chilean version of the site, accesses from more than 700 different IPs and more than 200 registered users (registration is optional).

developing the compressed indexing technology. The API we propose (see Appendix A) is thus intended to ease the deployment of this technology in real software systems, and to provide a reference for any researcher who wishes to contribute to the *Pizza&Chili* repository with his/her new compressed index.

3.3.1 Indexes

The *Pizza&Chili* site provides several index implementations, all adhering to a common API. All indexes, except CSA and LZ-index, are built through the deep-shallow algorithm of Manzini and Ferragina [MF04] which constructs the suffix array using little extra space and fast in practice. Unless otherwise stated they were implemented by ourselves.

- The Suffix Array [MM93] is a plain implementation of the classical index (see Section 2.6), using either $n \log n$ bits of space or simply n computer integers, depending on the version.
- The SSA [FMMN07, MN05] uses a Huffman-based wavelet tree over the string T^{bwt} (Section 2.9.1). It achieves zero-order entropy space with little extra overhead and striking simplicity. It was implemented in cooperation with Veli Mäkinen, University of Helsinki, Finland.
- The AF-index [FMMN07] combines compression boosting [FGMS05] with the above wavelet tree data structure (Section 3.2.2). It achieves high-order compression, at the cost of being more complex than SSA.
- The RLFM [MN05] is an improvement over the SSA (Section 2.9.1), which exploits the equal-letter runs of the BWT to achieve k-th order compression, and in addition uses a Huffman-shaped wavelet tree. It is slightly larger than the AF-index. It was implemented in cooperation with Veli Mäkinen.
- The FMI-2 is an engineered implementation of the original FM-index [FM01], where a different sampling strategy is designed in order to improve the performance of the locating operation. It was implemented by Paolo Ferragina and Rossano Venturini.
- The CSA [Sad03, Sad02] is the variant using backward search (Section 2.9.2). It achieves high-order compression and is robust for large alphabets. It was implemented by Kunihiko Sadakane and then adapted by us to adhere the API of the *Pizza&Chili* site. To construct the suffix array, it uses the *qsufsort* library by Jesper Larsson and Kunihiko Sadakane [LS99].
- The LZ-index [Nav04, ANS06, AN08] is a compressed index based on LZ78 compression (Section 2.9.3), implemented by Diego Arroyuelo and Gonzalo Navarro. It achieves high-order compression, yet with relatively large constants. It is slow for counting but very competitive for locating and extracting.

These implementations support any byte-based alphabet of size up to 255 symbols: one symbol is automatically reserved by the indexes as the terminator "$".

3.3.2 Texts

We have chosen the texts forming the *Pizza&Chili* collection by following three basic considerations. First, we wished to cover a representative set of application areas where the problem of full-text indexing might be relevant, and for each of them we selected texts freely available on the Web. Second, we aimed at having one file per text type to avoid unreadable tables of many results. Third, we have chosen the size of the texts to be large enough to make indexing relevant and compression apparent. These are the current collections provided in the repository:

- `dna` (DNA sequences). This file contains bare DNA sequences without descriptions, separated by `newline`, obtained from files available at the Gutenberg Project site (`http://www.gutenberg.org/`): namely, from *01hgp10* to *21hgp10*, plus *0xhgp10* and *0yhgp10*. Each of the four DNA bases is coded as an uppercase letter A,G,C,T, and there are a few occurrences of other special symbols.
- `english` (English texts). This file is the concatenation of English texts selected from the collections *etext02—etext05*, obtained from the Gutenberg Project site (`http://www.gutenberg.org/`). We deleted the headers related to the project so as to leave just the real text.
- `pitches` (MIDI pitch values). This file is a sequence of pitch values (bytes whose values are in the range 0-127, plus a few extra special values) obtained from a myriad of MIDI files freely available on the Internet. The MIDI files were converted into the IRP format by using the `semex` tool by Kjell Lemstrom [LP00]. This is a human-readable tuple format, where the 5th column is the pitch value. The pitch values were coded in one byte each and concatenated all together.
- `proteins` (protein sequences). This file contains bare protein sequences without descriptions, separated by `newline`. It was downloaded from the Swissprot database at `ftp://ftp.ebi.ac.uk/pub/databases/swissprot/`. Each of the 20 amino acids is coded as an uppercase letter.
- `sources` (source program code). This file is formed by C/Java source codes obtained by concatenating all the *.c*, *.h*, *.C* and *.java* files of the *linux-2.6.11.6* (`http://ftp.kernel.org/`) and *gcc-4.0.0* (`http://ftp.gnu.org/`) distributions.
- `xml` (structured text). This file is in XML format and provides bibliographic information on major computer science journals and proceedings. It was downloaded from the DBLP archive at `http://dblp.uni-trier.de/`.

Text	Size (MB)	Alphabet size	Inv. match prob.
`dna`	200	16	3.86
`english`	200	225	15.12
`pitches`	50	133	40.07
`proteins`	200	25	16.90
`sources`	200	230	24.81
`xml`	200	96	28.65

Table 3.1: General statistics for our indexed texts.

For the experiments, we have limited the short file `pitches` to its initial 50 MB, whereas all the other long files have been cut down to their initial 200 MB. We show now some statistics on those files. These statistics and the tools used to compute them are also available at the *Pizza&Chili* site.

Table 3.1 summarizes some general characteristics of the selected files. The last column, inverse match probability, is the reciprocal of the probability of matching between two randomly chosen text symbols. This may be considered as a measure of the *effective* alphabet size — indeed, on a uniformly distributed text, it would be precisely the alphabet size.

Table 3.2 provides some information about the compressibility of the texts by reporting the value of H_k for $0 \leq k \leq 4$, measured as number of bits per input symbol. As a comparison on the *real* compressibility of these texts, Table 3.3 shows the performance of three well-known compressors (sources available in the site): gzip (Lempel-Ziv-based compressor), bzip2 (BWT-based compressor), and PPMDi (k-th order modeling compressor). Notice that, as k grows, the value of H_k decreases but the size of the *dictionary* of length-k contexts grows significantly, eventually approaching the size of the text to be compressed. It is interesting to note in Table 3.3 that the compression ratios achievable by the tested compressors may be superior to H_4, because they use (explicitly or implicitly) longer contexts. For example, typical ratios achieved by PPMDi are around H_5 or H_6.

3.4 Experimental Results

In this section, we report the experimental results concerning a subset of the compressed indexes available at the *Pizza&Chili* site. All the experiments were executed on a 2.6 GHz Pentium 4, with 1.5 GB of main memory, and running Fedora Linux. The searching and building algorithms for all compressed indexes were coded in C/C++ and compiled with gcc or g++ version 4.0.2.

We restricted our experiments to the following indexes: Succinct Suffix Array (SSA, version SSA_v2 in *Pizza&Chili*), Alphabet-Friendly FM-index (AF-index, version AF-

Text	$\log \sigma$	H_0	1st order		2nd order		3rd order		4th order	
			H_1	#	H_2	#	H_3	#	H_4	#
`dna`	4.000	1.974	1.930	16	1.920	152	1.916	683	1.910	$2,222$
`english`	7.814	4.525	3.620	225	2.948	$10,829$	2.422	$102,666$	2.063	$589,230$
`pitches`	7.055	5.633	4.734	133	4.139	$10,946$	3.457	$345,078$	2.334	$3,845,792$
`proteins`	4.644	4.201	4.178	25	4.156	607	4.066	$11,607$	3.826	$224,132$
`sources`	7.845	5.465	4.077	230	3.102	$9,525$	2.337	$253,831$	1.852	$1,719,387$
`xml`	6.585	5.257	3.480	96	2.170	$7,049$	1.434	$141,736$	1.045	$907,678$

Table 3.2: Ideal compressibility of our indexed texts. For every k-th order model, with $0 \leq k \leq 4$, we report the number of distinct contexts of length k, and the empirical entropy H_k, measured as number of bits per input symbol.

Text	H_4	gzip	bzip2	PPMDi
`dna`	1.910	2.162	2.076	1.943
`english`	2.063	3.011	2.246	1.957
`pitches`	2.334	2.448	2.890	2.439
`proteins`	3.826	3.721	3.584	3.276
`sources`	1.852	1.790	1.493	1.016
`xml`	1.045	1.369	0.908	0.745

Table 3.3: Real compressibility of our indexed texts, as achieved by the best-known compressors: gzip (option `-9`), bzip2 (option `-9`), and PPMDi (option `-l 9`).

index_`v2` in *Pizza&Chili*), Compressed Suffix Array (CSA in *Pizza&Chili*), and LZ-index (LZ-index, version LZ-index`-4` in *Pizza&Chili*), because they are the best representatives of the three classes of compressed indexes we discussed in Section 2.9. This small number will provide us with a succinct, yet significant, picture of the performance of all known compressed indexes [NM07].

Further algorithmic engineering of the indexes in *Pizza&Chili* could possibly change the results shown below. However, we believe that the overall conclusions drawn from our experiments should not change significantly, unless new algorithmic ideas are devised for them. Indeed, the experimental results have two goals: to quantify the space and time performance of compressed indexes over real datasets, and to motivate further algorithmic research by highlighting the limitations of the present indexes and their implementations. The latter point is discussed in Section 3.5.

Index	*count*	*locate / extract*
AF-index	–	$s_A = \{4, 16, 32, 64, 128, 256\}$
CSA	$s_\Psi = \{128\}$	$s_A = \{4, 16, 32, 64, 128, 256\}; s_\Psi = \{128\}$
LZ-index	$\epsilon = \{\frac{1}{4}\}$	$\epsilon = \{1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{10}, \frac{1}{20}\}$
SSA	–	$s_A = \{4, 16, 32, 64, 128, 256\}$

Table 3.4: Parameters used for the different indexes in our experiments. The cases of multiple values correspond to space/time tradeoff curves.

Index	Build Time (sec)	Main Memory Usage (MB)
AF-index	772	$1,751$
CSA	423	$1,801$
LZ-index	198	$1,037$
SSA	217	$1,251$

Table 3.5: Time and peak of main memory usage required to build the various indexes over the 200 MB file `english`. The indexes are built using the default value for the *locate* tradeoff (that is, $s_A = 64$ for AF-index and SSA; $s_A = 64$ and $s_\Psi = 128$ for CSA; and $\epsilon = \frac{1}{4}$ for the LZ-index).

3.4.1 Construction

Table 3.4 shows the parameters used to construct the indexes in our experiments. Table 3.5 shows construction time and space for one collection, namely `english`, as all the others give roughly similar results. The bulk of the time of SSA and CSA is that of suffix array construction (prior to its compression). The times differ because different suffix array construction algorithms are used (see Section 3.3.1). The AF-index takes much more time because it needs to run the compression boosting algorithm over the suffix array [FGMS05]. The LZ-index spends most of the time in parsing the text and creating the LZ78 and reverse tries. In all cases construction times are practical, 1–4 sec/MB with our machine.

The memory usage might be problematic, as it is 5–9 times the text size. Albeit the final index is small, one needs much memory to build it first[2]. This is a problem of compressed indexes, which is attracting a lot of practical and theoretical research [LSSY02, AN05, HSS03a, MN06a]. In Section 7.4, we obtain a new theoretical solution to build FM-index and the suffix array in compressed space.

We remark that the indexes allow different space/time tradeoffs. The SSA and AF-index have a sampling rate parameter s_A that trades locating and extracting time for space. More precisely, they need $O(s_A)$ accesses to the wavelet tree for locating, and $O(s_A + r - l + 1)$

[2]In particular, this limited us to indexing up to 200 MB of text in our machine.

	SSA		AF-index		CSA		LZ-index		plain SA	
Text	Time	Space	Time	Space	Time	Space	Time	Space	Time	Space
`dna`	**0.956**	**0.29**	1.914	**0.28**	5.220	0.46	43.896	0.93	0.542	5
`english`	**2.147**	0.60	2.694	**0.42**	4.758	**0.44**	68.774	1.27	0.512	5
`pitches`	**2.195**	0.74	2.921	**0.66**	3.423	**0.63**	55.314	1.95	0.363	5
`proteins`	**1.905**	**0.56**	3.082	**0.56**	6.477	0.67	47.030	1.81	0.479	5
`sources`	**2.635**	0.72	2.946	0.49	4.345	**0.38**	162.444	1.27	0.499	5
`xml`	2.764	0.69	**2.256**	0.34	4.321	**0.29**	306.711	0.71	0.605	5

Table 3.6: Experiments on the counting of pattern occurrences. Time is measured in microseconds per pattern symbol. The space usage is expressed as a fraction of the original text size. We put in boldface those results that lie within 10% of the best space/time tradeoffs (excluding plain suffix array).

accesses to extract $T_{l,r}$, in exchange for $\frac{n \log n}{s_A}$ additional bits of space. We can remove those structures if we are only interested in counting.

The CSA has two space/time tradeoffs. A first one, s_Ψ, governs the access time to Ψ, which is $O(s_\Psi)$ in exchange for $\frac{n \log n}{s_\Psi}$ bits of space required by the samples. The second, s_A, affects locating and extracting time just as above. For pure counting, we can remove the sampling related to s_A, whereas for locating the best is to use the default value (given by Sadakane) of $s_\Psi = 128$. The best choice for extracting is less clear, as it depends on the length of the substring to extract.

Finally, the LZ-index has one parameter ϵ which trades counting/locating time per space occupancy: The cost per candidate occurrence is multiplied by $\frac{1}{\epsilon}$, and the additional space is $2\epsilon n H_k(T)$ bits. No structure can be removed in the case of counting, but space can be halved if the *extract* operation is the only one needed (just remove the reverse trie).

3.4.2 Counting

We searched for $50,000$ patterns of length $m = 20$, randomly chosen from the indexed texts. The average counting time was then divided by m to display counting time per symbol. This is appropriate because the counting time of the indexes is linear in m, and 20 is sufficiently large to blur small constant overheads. The exception is the LZ-index, whose counting time is superlinear in m, and not competitive at all for this task.

Table 3.6 shows the results on this test. The space of the SSA, AF-index, and CSA does not include what is necessary for locating and extracting. We can see that, as expected, the AF-index is always smaller than the SSA, yet they are rather close on `dna` and `proteins` (where the zero-order entropy is not much larger than higher-order entropies). The space usages of the AF-index and the CSA are similar and usually the best, albeit the CSA

Text	# patterns	# occurrences
`dna`	10	$2,491,410$
`english`	100	$2,969,876$
`pitches`	200	$2,117,347$
`proteins`	$3,500$	$2,259,125$
`sources`	50	$2,130,626$
`xml`	20	$2,831,462$

Table 3.7: Number of searched patterns of length 5 and total number of located occurrences.

predictably loses in counting time on smaller alphabets (`dna`, `proteins`), due to its $O(m \log n)$ rather than $O(m \log \sigma)$ complexity. The CSA takes advantage of larger alphabets with good high-order entropies (`sources`, `xml`), a combination where the AF-index, despite of its name, profits less. Note that the space performance of the CSA on those texts confirms that its space occupancy is related to the high-order entropy.

With respect to time, the SSA is usually the fastest thanks to its simplicity. Sometimes the AF-index gets close and it is actually faster on `xml`. The CSA is rarely competitive for counting, and the LZ-index is well out of bounds for this experiment. Notice that the plain suffix array (last column in Table 3.6) is 2–6 times faster than any compressed index, but its space occupancy can be up to 18 times larger.

3.4.3 Locating

We locate sufficient random patterns of length 5 to obtain a total of 2–3 million occurrences per text (see Table 3.7). This way we are able to evaluate the average cost of a single locate operation, by making the impact of the counting cost negligible. Figure 3.9 reports the time/space tradeoffs achieved by the different indexes for the *locate* operation.

We remark that the implemented indexes include the sampling mechanism for *locate* and *extract* as a single module, and therefore the space for both operations is included in these plots. Therefore, the space could be reduced if we only wished to locate. However, as extracting snippets of pattern occurrences is an essential functionality of a self-index, we consider that the space for efficient extraction should always be included.[3]

The comparison shows that usually CSA can achieve the best results with minimum space, except on `dna` where the SSA performs better as expected (given its query time complexity, see before), and on `proteins` for which all indexes perform the same. The CSA is also the most attractive alternative if we fix that the space of the index should be equal to

[3]Of course, we could have a sparser sampling for extraction, but we did not want to complicate the evaluation more than necessary.

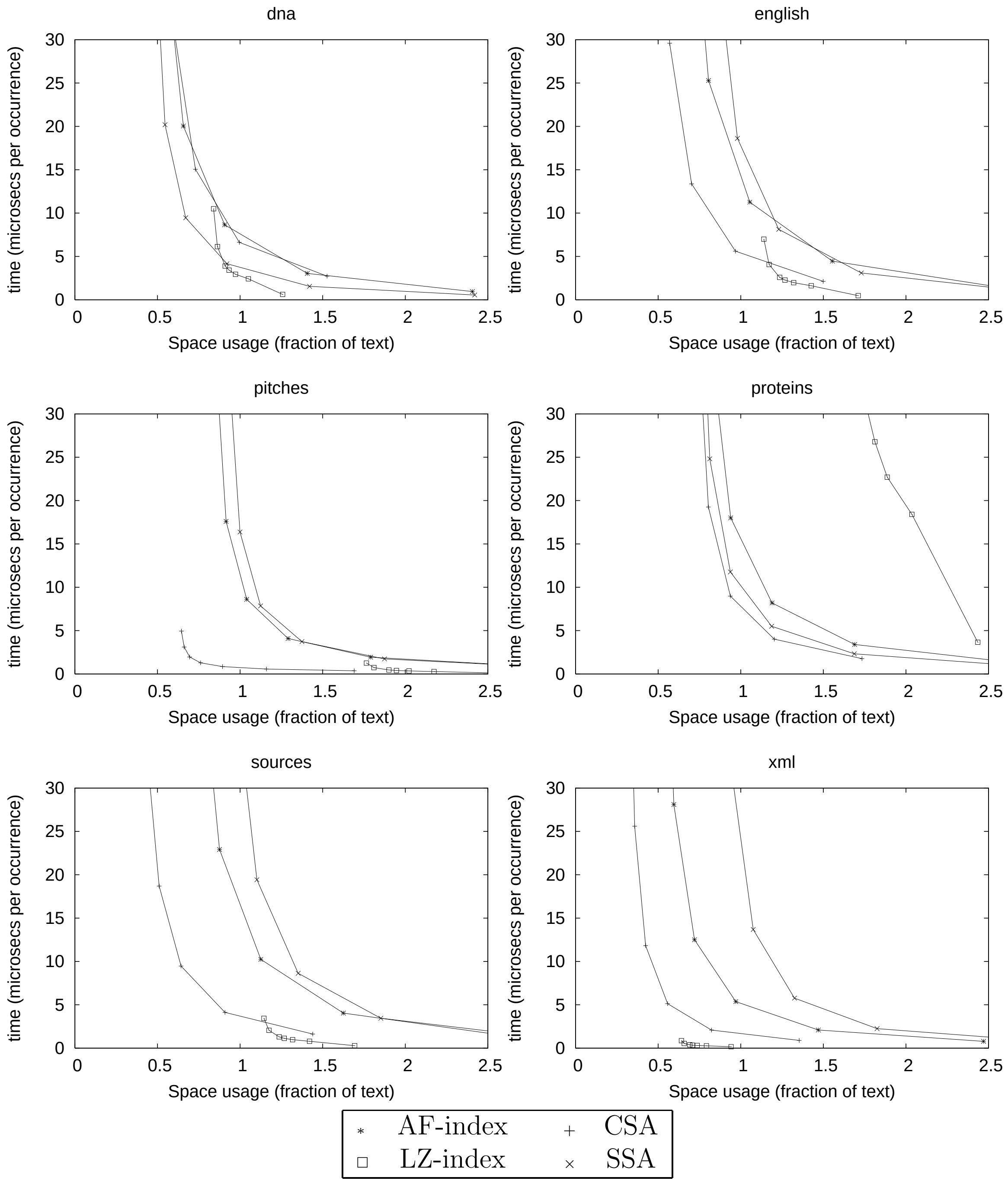


Figure 3.9: Space-time tradeoffs for locating occurrences of patterns of length 5.

	dna	english	pitches	proteins	sources	xml
plain SA	0.005	0.005	0.006	0.007	0.007	0.006

Table 3.8: *Locate* time required by plain SA in microseconds per occurrence, with $m = 5$. We recall that this implementation requires 5 bytes per indexed symbol.

that of the text (recall that it includes the text), being the exceptions `dna` and `xml`, where the LZ-index is superior.

The LZ-index can be much faster than the others if one is willing to pay for some extra space. The exceptions are `pitches`, where the CSA is superior, and `proteins`, where the LZ-index performs poorly. This may be caused by the large number of patterns that were searched to collect the 2–3 million occurrences (see Table 3.7), as the counting is expensive on the LZ-index.

Table 3.8 shows the locating time required by an implementation of the classical suffix array: it is between 100 and 1000 times faster than any compressed index, but always 5 times larger than the indexed text. Unlike counting, where compressed indexes are comparable in time with classical ones, locating is much slower on compressed indexes. This comes from the fact that each *locate* operation (except on the LZ-index) requires to perform several random memory accesses, depending on the sampling step. In contrast, all the occurrences are contiguous in a classical suffix array. As a result, the compressed indexes are currently very efficient in case of selective queries, but traditional indexes become more effective when locating many occurrences. In Chapter 4, we present a compressed index that is much faster for locating, and more comparable with plain suffix array performance.

3.4.4 Extracting

We extracted substrings of length 512 from random text positions, for a total of 5 MB of extracted text. Figure 3.10 reports the time/space tradeoffs achieved by the tested indexes. We still include both space to *locate* and *extract*, but we note that the sampling step affects only the time to reach the text segment to extract from the closest sample, and afterwards the time is independent of the sampling. We chose length 512 to smooth out the effect of this sampling.

The comparison shows that, for extraction purposes, the CSA is better for `sources` and `xml`, whereas the SSA is better on `dna` and `proteins`. On `english` and `pitches` both are rather similar, albeit the CSA is able to operate on reduced space. On the other hand, the LZ-index is much faster than the others on `xml`, `english` and `sources`, if one is willing to pay for some additional space.[4]

[4]Actually, the LZ-index is not plotted for `pitches` and `proteins` because it needs more than 1.5 times the text size.

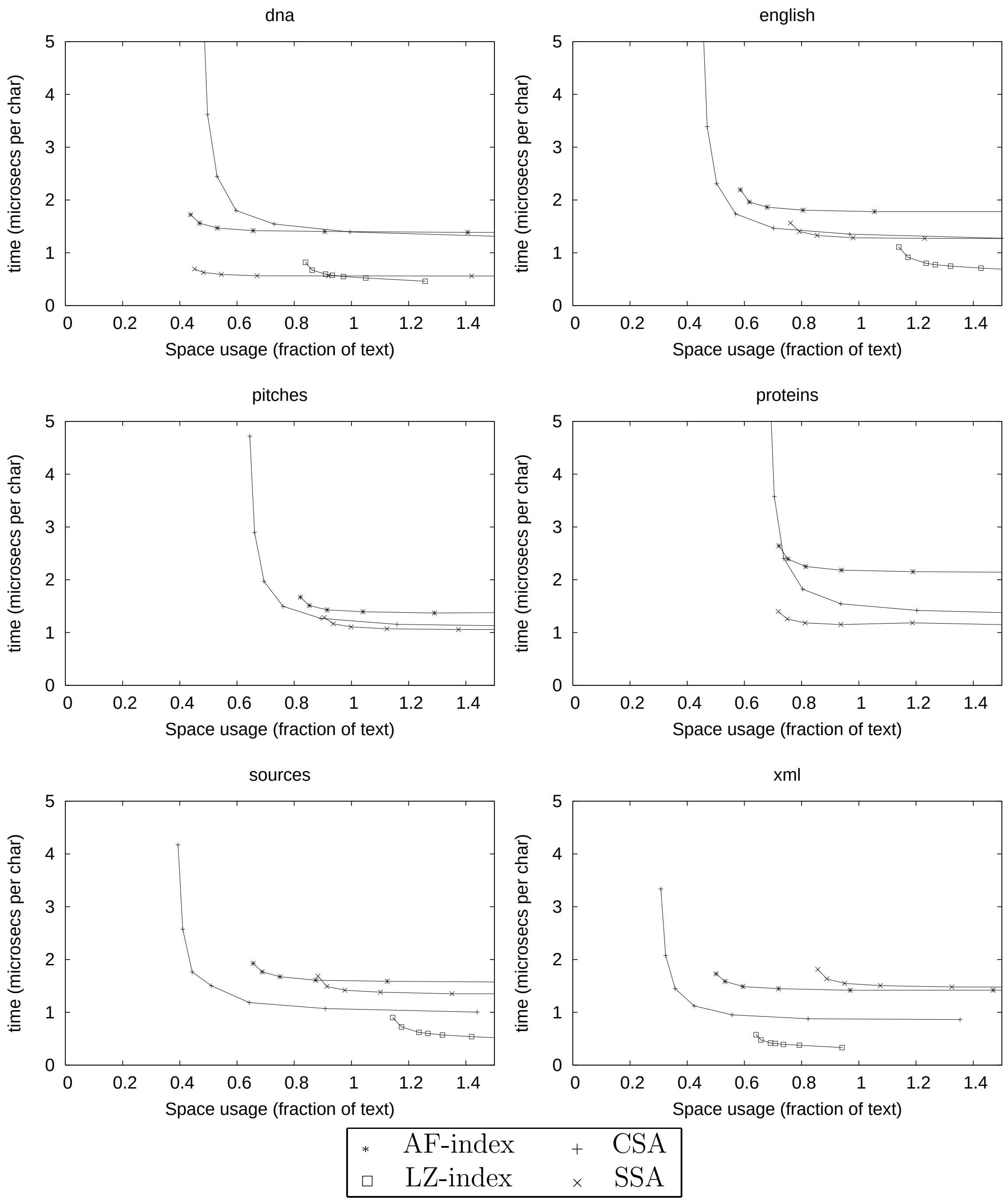


Figure 3.10: Space-time tradeoffs for extracting text symbols.

It is difficult to compare these times with those of a classical index, because the latter has the text readily available. Nevertheless, we note that the times are not bad: using the same space as the text (and some times up to half the space) for all the functionalities implemented, the compressed indexes are able to extract around 1 MB/sec, from arbitrary positions. This shows that self-indexes are appealing as compressed-storage schemes with the support of random accesses for snippet extraction. To achieve better times one should improve the access locality of text extraction. This is what we pursue in Chapter 5.

3.5 Discussion and Open Challenges

In this chapter, we have addressed the novel technology of compressed text indexing from a practical viewpoint. We have explained the main principles used by those indexes in practice, and presented the *Pizza&Chili* site, where implementations and testbeds are readily available for use. Finally, we have presented experiments that demonstrate the practical relevance of this emerging technology. Table 3.9 summarizes our experimental results by showing the most promising compressed index(es) depending on the text type and task.

	count	*locate*	*extract*
`dna`	SSA	LZ-index / SSA	SSA
`english`	SSA / AF-index	CSA / LZ-index	CSA / LZ-index
`pitches`	AF-index/ SSA	CSA	CSA
`proteins`	SSA	SSA	SSA
`sources`	CSA / AF-index	CSA / LZ-index	CSA / LZ-index
`xml`	AF-index	CSA / LZ-index	CSA / LZ-index

Table 3.9: The most promising indexes given the size and time they obtain for each operation/text.

For counting the best indexes are SSA and AF-index. This stems from the fact that they achieve very good zero- or high-order compression of the indexed text, while their average counting complexity is $O(m(H_0(T) + 1))$. The SSA has the advantage of a simpler search mechanism, but the AF-index is superior for texts with small high-order entropy (i.e., `xml`, `sources`, `english`). The CSA usually loses because of its $O(m \log n)$ counting complexity.

For locating and extracting, which are LF-computation intensive, the AF-index is hardly better than the simpler SSA because the benefit of a denser sampling does not compensate for the presence of many wavelet trees. The SSA wins for small-alphabet data, like `dna` and `proteins`. Conversely, for all other high-order compressible texts the CSA takes over the other approaches. We also notice that the LZ-index is a very competitive choice when extra space is allowed and the texts are highly compressible.

The ultimate moral is that there is not a clear winner for all text collections. Nonetheless, our results provide an upper bound on what these compressed indexes can achieve in practice:

*Count***.** We can compress the text within 30%–50% of its original size, and search for 20,000–50,000 patterns of 20 chars each within a second.

*Locate***.** We can compress the text within 40%–80% of its original size, and locate about 100,000 pattern occurrences per second.

*Extract***.** We can compress the text within 40%–80% of its original size, and decompress its symbols at a rate of about 1 MB/second.

The above figures are from one (*count*) to three (*locate*) orders of magnitudes slower than what one can achieve with a plain suffix array, at the benefit of using up to 18 times less space. This slowdown is due to the fact that search operations in compressed indexes access the memory in a non-local way thus eliciting many cache/IO misses, with a consequent degradation of the overall time performance. We hope that this site will spread their use in any software that needs to process, store and mine text collections of any size.

On the other hand, the experimental results clearly point out some challenges not yet fulfilled. The following chapters are devoted to addressing these challenges. The most important are (a more extensive description of them can be found in Section 1.1):

- **Locating and extracting.** A disadvantage of the compressed text indexes presented until now, is their slowness (compared to the uncompressed ones) in locating patterns and in extracting any part of the text. This is a direct consequence of their non-local access. In Chapters 4 and 5, we address the locating and extracting problem, respectively.
- **Secondary memory.** The memory access patterns of compressed text indexes are highly non-local, which makes their potential secondary-memory versions rather unpromising. In Chapter 6, we present a compressed text index on secondary memory, as well as an algorithm to construct it on secondary memory.
- **Dynamism and compressed construction.** In Chapter 7, we present a novel structure supporting dynamic compressed sequences with *rank* and *select* capabilities. This improves the current best solutions on dynamic compressed text indexes and on compressed text index construction.

Chapter 4

Locally Compressed Suffix Arrays

In this chapter, we present a suffix array compression technique that builds on well-known regularity properties that show up in suffix arrays, when the text they index is compressible (more precisely, we refer to the runs in the Ψ function of the suffix array, or which is the same, the number of equal-letter runs in the Burrows-Wheeler transform of the text, Section 2.9.2). This regularity has been exploited in several ways in the past [Mäk03, GV06, GGV03, Sad03, MN05], but we present a completely novel technique to take advantage of it. We represent the suffix array using differential encoding, which converts the regularities into true repetitions. Those repetitions are then factored out using Re-Pair [LM00], a compression technique that builds a dictionary of phrases and permits fast *local* decompression using only the dictionary (whose size one can limit at will, at the expense of losing some compression). We then introduce novel techniques to further compress the Re-Pair dictionary, which can be of independent interest. We also use specific suffix array properties to obtain a much faster compression method that loses just up to 1% of compression ratio.

Our so-called *locally compressed suffix array (LCSA)* is shown to reduce the suffix array to 20–70% of its original size, depending on the compressibility of various text types. This reduced index can still extract any portion of the suffix array very fast by adding a small set of sampled absolute values. By using the deep connection with function Ψ, we prove that the size of the result is $O(H_k \log \frac{1}{H_k}\ n \log n) + o(n)$ bits[1] for any $k \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$. Note that this reduced suffix array is not yet a self-index as it cannot reproduce the text.

This structure can be used in two ways. One way is to attach it to a self-index able of counting, which in this process identifies as well the segment of the (virtual) suffix array where the occurrences lie. We can then locate the occurrences by decompressing that segment

[1]This result is meaningful for $H_k = o(1)$. The general result is $O(N(1 + \log \frac{n}{N}) \log n)$ bits, where N is the number of runs in Ψ. Acording to [MN05], $N \leq \min(n, nH_k + \sigma^k)$ for any k.

using our structure. The result is a self-index that needs 1–3 times the text size (that is, considerably larger than current self-indexes but also much smaller than classical indexes) and whose counting and locating times are competitive with those of classical indexes, far better for locating than current self-indexes. In theoretical terms, assuming for example the use of an alphabet-friendly FM-index [FMMN07] for counting, our index needs $O(H_k \log \frac{1}{H_k} n \log n + n)$ bits of space, counts in time $O(m(1 + \frac{\log \sigma}{\log \log n}))$ and locates the *occ* occurrences of P in time $O(occ + \log n)$. In practice, even letting classical self-indexes use the same amount of space to speed up their locating time, they are much slower than our LCSA for reporting more than a few occurrences (2–10).

A second and simpler way to use the structure is, together with the plain text, as a replacement of the classical suffix array. In this case, we must not only use it for locating the occurrences but also for binary searching. The binary search is done over the samples first and then we decompress the area between two consecutive samples to finish the search. This yields a very practical alternative requiring 0.8–2.4 times the text size (as opposed to 4) plus the text, in exchange for being just 2–28 times slower for locating.

Achieving compressed indexes able of counting in competitive time was the first important breakthrough in this area. We believe this chapter is a first important step towards compressed indexes with practical locating times. This is up to date the major concern for adopting compressed indexes in practical applications (recall Section 3.5).

In Section 4.1, we describe our LCSA. In Section 4.2, we analyze its compression ratio, relating it to the compressibility of the text. In Section 4.3, we show how to use the LCSA as part of various indexing schemes. Finally, in Section 4.4, we carry out several experimental tunings and comparisons with alternative compressed and classical indexes.

4.1 A Locally Compressed Suffix Array (LCSA)

Suffix arrays turn out to be compressible whenever T is. The k-th order empirical entropy of T (Section 2.1), shows up in its suffix array A in the form of large segments $A[i, i+\ell]$ that appear elsewhere in $A[j, j+\ell]$ with all the values shifted by one position, $A[j+s] = A[i+s]+1$ for $0 \leq s \leq \ell$. Actually, one can partition A into *runs* of maximal segments that appear repeated (shifted by 1) elsewhere, and the number of such runs is at most $nH_k + \sigma^k$ for any k [MN05, NM07].

This property has been used several times in the past to compress A. Mäkinen's Compact Suffix Array (CSA) [Mäk03] replaces runs with pointers to their definition (copy) elsewhere in A, so that the run can be recovered by (recursively) expanding the definition and shifting the values (see Section 2.9.2). Mäkinen and Navarro [MN05] use the connection with FM-indexes (runs in A are related to equal-letter runs in the Burrows-Wheeler transform of T, basic building block of FM-indexes) and run-length compression (see Section 3.2.1).

Yet, the most successful technique to take advantage of those regularities has been the definition of function $\Psi(i) = A^{-1}[A[i]+1]$ (or $A^{-1}[1]$ if $A[i] = n$). It can be seen that $\Psi(i) = \Psi(i-1)+1$ within runs of A, and therefore a differential encoding of Ψ is highly compressible [GGV03, Sad03], recall Section 2.9.2.

4.1.1 Basic LCSA Idea

We present a completely different method to compress A. We first represent A in differential form A':

Definition 4.1. *Let $A[1,n]$ be an array of integers. Then we define $A'[1,n]$ as follows: $A'[1] = A[1]$ and $A'[i] = A[i] - A[i-1]$ for all $1 < i \leq n$.*

The next simple lemma shows that runs of A become true repetitions in A'.

Lemma 4.1. *Consider a run of A of the form $A[j+s] = A[i+s]+1$ for $0 \leq s \leq \ell$. Then $A'[j+s] = A'[i+s]$ for $1 \leq s \leq \ell$.*

Proof. $A'[j+s] = A[j+s] - A[j+s-1] = (A[i+s]+1) - (A[i+s-1]+1) = A[i+s] - A[i+s-1] = A'[i+s]$. □

We can now exploit those repetitions using any classical compression method. In particular, we seek a method that allows fast local decompression of A'. We resort to Re-Pair [LM00], a dictionary-based compression method (Section 2.10).

Apart from R and C as described in Section 2.10, we need a few more structures to recover arbitrary positions of A:

- An array S such that $S[i] = A[i \cdot l]$, that is, a sampling of absolute values of A at regular intervals l.
- A bitmap $L[1,n]$, marking the positions where each symbol of C (which could represent several symbols of A') starts in A'.
- $o(n)$ further bits to answer *rank* queries on L in constant time (Section 2.4).

With these structures, the algorithm to retrieve $A[i,j]$ is as follows:

1. Check if there is a multiple of l in $[i,j]$, extending i to the left or j to the right to include such a multiple if necessary.

2. Use the mechanism to decompress one symbol in C (described in Section 2.10) to obtain $A'[i,j]$, by expanding $C[rank(L,i), rank(L,j)]$. We expand from right to left, so the first symbol may be not fully expanded.

3. Use any absolute sample of A included in $S[\lfloor i/l \rfloor, \lfloor j/l \rfloor]$ to obtain, using the differences in $A'[i,j]$, the values $A[i,j]$.

4. Return the values in the original requested interval $[i,j]$.

The overall time complexity of this decompression is the output size plus what we have expanded the interval to include a multiple of l (i.e., $O(l)$) and to ensure an integral number of symbols in C. The latter can be controlled by limiting the length of the uncompressed version of the symbols we create.

4.1.2 Compression using Ψ

A weak point of using Re-Pair is its compression speed and space usage. Re-Pair can be implemented in $O(n)$ time, but this needs too much space [LM00]. Instead of using the original Re-Pair, we opt for a technique that needs less space and usually runs in $O(n \log n)$ time. This technique is not too interesting (and we do not describe it) except as a control value to test our more important contribution: We introduce a fast approximate technique specialized to compressing suffix arrays A. We show that Ψ (which is easily built in $O(n)$ time from A) can be used to obtain a much faster compression algorithm, which in practice compresses almost as much as the original Re-Pair.

Recall that $\Psi(i)$ tells where in A is the value $A[i]+1$. The idea is that, if $A[i,i+\ell]$ is a run such that $A[j+s] = A[i+s]+1$ for $0 \leq s \leq \ell$ (and thus $A'[j+s] = A'[i+s]$ for $1 \leq s \leq \ell$), then $\Psi(i+s) = j+s$ for $0 \leq s \leq \ell$. Thus, by following permutation Ψ we have a good chance of finding repeated pairs in A'. The basic idea is to choose the pairs while following permutation Ψ, cycling several times over A', until no further replacements can be done. This does not guarantee to choose the same pairs of the original Re-Pair, but we expect them to be sufficiently good.

Data Structures. To compress using Ψ we need only two arrays and one bitmap.

- An array $D[1,n]$, which initially stores the suffix array of text T in differential form, $D[i] = A'[i]$ $\forall i$. At the end, we compact the valid values of D to obtain C.

- An array $P[1,n]$, which initially stores the values of function Ψ of text T, $P[i] = \Psi(i)$ $\forall i$.

- The bitmap $L[1,n]$, where $L[i]=1$ indicates that $D[i]$ is a valid value. In the beginning $L[i]=1\ \forall i$. At the end, L can be preprocessed for *rank* queries and is ready for querying.

When we replace a pair with a new symbol, array D becomes sparse. A way to find the next valid symbol in constant time is as follows: If a valid symbol $D[i]$ is followed by an invalid symbol $D[i+1]$ (that is, $L[i,i+1]=10$), then $D[i+1]$ can be used to store the distance $i'-i$ to the next valid symbol $D[i']$ (we use i' with this meaning, for any i, in the next algorithm description). This permits obtaining any pair of the sparse D in constant time.

In practice, it turns out to be faster to calculate $i' = \mathit{selectNext}(L,i)$, which returns the position of the first 1 in $L[i+1,n]$, by scanning the bitmap word-wise. In Section 3.1.3, we study different approaches to solve *selectNext*.

Algorithm. We make a number of *passes* over D. Each pass starts at $i=1$ (where value $A'[1]=A[1]=n$ will not be replaced by Re-Pair as it is unique). For each i visited along the pass, we see if $D[i]D[i'] = D[P[i]]D[P[i]']$. If this does not hold, we move on to $i \leftarrow P[i]$ and iterate. If, instead, equality holds, we start a chain of replacements: We add a new pair $s \rightarrow D[i]D[i']$ to R, make the replacements at i and $P[i]$ (invalidating $i+1$ and $P[i]+1$), and move on to $i \leftarrow P[i]$, continuing the replacements until the pair changes. In this process, when a position $P[j]$ becomes invalid, we set $P[j] \leftarrow P[P[j]]$, so that the position is skipped in the next pass. When the pair finally changes, that is, $D[i]D[i'] \neq D[P[i]]D[P[i]']$, we restart the process with $i \leftarrow P[i]$, looking again for a new pair to create. We keep running passes over D (using P) as long as we replace at least $\alpha n'$ pairs in a pass, where $0<\alpha<1$ is a constant and n' is the number of valid elements in D in the previous pass. Figure 4.1 shows a more detailed pseudocode.

Cost. Let n_i be the number of elements in the i-th pass, then $n_{i+1} \leq (1-\alpha)n_i$. Since $n_0 = n$, it holds $n_i \leq (1-\alpha)^i n$. The i-th pass costs $O(n_i)$ time. Let k be the number of passes doing more than $\alpha n'$ replacements. So the total cost is at most

$$\sum_{i=0}^{k}(1-\alpha)^i n + (1-\alpha)^k n \ \leq\ n\sum_{i\geq 0}(1-\alpha)^i + (1-\alpha)^k n \ \leq\ \left(1+\frac{1}{\alpha}\right)n \ =\ O(n).$$

Thus our algorithm achieves linear time while requiring only the space for D (overwritten on A' and finally leaving there C), for P (overwritten on Ψ), and for L (which is also needed in the final structure). It is also simple and fast in practice (see Section 4.4).

```
**Algorithm** Compress($D$, $P$, $\alpha$)
    $s \leftarrow n$, $R \leftarrow \emptyset$
    **for** $i \leftarrow 1 \ldots n$ **do** $L[i] \leftarrow 1$
    $n' \leftarrow n$, $rep \leftarrow 0$
    **do** $n' \leftarrow n' - rep$
        $rep \leftarrow 0$
        $j \leftarrow 1$, $j' \leftarrow selectNext(L, j)$
        **do**
            **do** $i \leftarrow j$, $i' \leftarrow j'$
                **while** $L[P[i]] = 0$ **do** $P[i] \leftarrow P[P[i]]$
                $j \leftarrow P[i]$, $j' \leftarrow selectNext(L, j)$
            **while** $j \neq 1$ **and** $D[i]D[i'] \neq D[j]D[j']$
            **if** $j \neq 1$ **then**
                $ab \leftarrow D[i]D[i']$, $s \leftarrow s + 1$, $R \leftarrow R \cup \{s \rightarrow ab\}$
                $D[i] \leftarrow s$, $L[i'] \leftarrow 0$, $rep \leftarrow rep + 1$
                **do** $D[j] \leftarrow s$, $L[j'] \leftarrow 0$, $rep \leftarrow rep + 1$
                    $i \leftarrow j$, $i' \leftarrow j'$
                    **while** $L[P[i]] = 0$ **do** $P[i] \leftarrow P[P[i]]$
                    $j \leftarrow P[i]$, $j' \leftarrow selectNext(L, j)$
                **while** $j \neq 1$ **and** $ab = D[j]D[j']$
        **while** $j \neq 1$
    **while** $(rep > \alpha n')$
    $j \leftarrow 1$
    **for** $i \leftarrow 1 \ldots n$ **do**
        **if** $L[i] = 1$ **then** $D[j] \leftarrow D[i]$, $j \leftarrow j + 1$
    **return** $(C[1, n'] = D,\ R,\ L)$
```

Figure 4.1: Algorithm to compress $D = A'$ using $P = \Psi$ in $O(n)$ time.

4.1.3 Stronger Compression based on Ψ

The only advantage of using the original Re-Pair is that it yields better compression and enforces the property that each new rule in the dictionary removes no more pairs than the previous rule. The latter comes from the fact that the pairs in Re-Pair are replaced in decreasing order of frequency. This prevents less frequent pairs to break longer chains of replacements. We now modify the algorithm that uses Ψ to obtain compression ratios as close to Re-Pair's as desired, at the expense of $O(n \log n)$ complexity (multiplied by a constant that increases as the compression ratio improves). The key idea is to replace longer chains first.

Algorithm. The algorithm is as follows:

- We make one pass searching for the longest chain of equal pairs obtained by following Ψ, let f be its length.
- We apply the previous algorithm, yet we only replace the chains of length at least $t_0 = \delta \cdot f$, where $0 < \delta < 1$ is a constant.
- Again we apply the previous algorithm using $t_1 = \delta \cdot t_0$ then $t_2 = \delta \cdot t_1$ and so on, until $t_i \leq \gamma$. At this point we decrement t_i one by one until we reach $t_i = 1$. Here γ is another parameter.

Cost. We already know that the total cost of all passes that replace more than $(1-\alpha)n'$ elements adds up to $O(n)$. The number of passes where we replace less than $(1-\alpha)n'$ pairs, on the other hand, is at most $\log_\delta f + \gamma$. This is, $\log_\delta f$ for the part where $t_{(\cdot)}$ decreases by a δ fraction, plus γ for the part where $t_{(\cdot)}$ decreases one by one. Thus the total cost is at most:

$$\frac{1}{\alpha}\, n + (\log_\delta f + \gamma)\, n.$$

Now, if we choose a constant s, $\alpha = 1/(s \cdot \log n)$, and $\gamma = \log n$, the total time is $O(n \log n)$. By choosing other values of s, δ and γ we can obtain better complexities, but it worsens the compression quality. Within $O(n \log n)$ complexity, we can improve the compression ratio by tuning δ and s.

4.1.4 Compressing the Dictionary

We now develop a technique to reduce the dictionary of rules R without affecting C. This can be of independent interest for Re-Pair in general. We note that the dictionary compression methods in the original Re-Pair article [LM00] achieve much more compression. The advantage of our scheme is that we can decompress parts of the text without decompressing the dictionary. This permits handling larger dictionaries in main memory.

A first observation is that, if we have a rule $s \rightarrow ab$ and s is only mentioned in another rule $s' \rightarrow sc$, then we could perfectly remove rule $s \rightarrow ab$ and rewrite $s' \rightarrow abc$. This gives a net gain of one integer, but now we have rules of varying length. This is easy to manage, but we prefer to go further. We develop a technique that permits *eliminating every rule definition that is used within R, once or more, and gain one integer for each such rule eliminated.* The key idea is to write down explicitly the binary tree formed by expanding the definitions (by doing a preorder traversal and writing 1 for internal nodes and 0 for leaves), so that not only the largest symbol (tree root) can be referenced later, but also any subtree.

For example, assume the rules $R = \{s \rightarrow ab,\ t \rightarrow sc,\ u \rightarrow ts\}$, and $C = tub$. We could first represent the rules by the bitmap R_B = `100100100` (where s corresponds to position 1, t to 4, and u to 7) and the sequence $R_S = ab1c41$ (we are using letters for the original symbols of A', and bitmap positions as the identifiers of created symbols). We express C as $47b$. To expand, say, 4, we go to position 4 in R_B and compute $rank_0(R_B, 4) = 2$. Thus the corresponding symbols in R_S start at position $2 + 1 = 3$. We extract one new symbol from R_S for each new zero we traverse in R_B, and stop when the number of zeros traversed exceeds the number of ones (this means we have completed the subtree traversal). This way we obtain the definition $1c$ for symbol 4.

More generally, R can be seen as $R = \{s_1 \rightarrow a_1b_1, s_2 \rightarrow a_2b_2, \ldots, s_\nu \rightarrow a_\nu b_\nu\}$, where indeed $s_\nu = n + \nu$ (as $n = A'[1] = A[1]$ is the maximum value in A'). Thus, we write down R_B, R_S and the new C as follows (note that positions in R_B are written in R_S shifted by n to distinguish them from the original symbols):

- $R_B = (100)^\nu$.
- $R_S = a_1b_1a_2b_2 \ldots a_\nu b_\nu = r_1r_2r_3 \ldots r_{2\nu}$, except that if $r_i > n$ we set it to $r_i = n + 1 + 3(r_i - n - 1)$, so that they point to the 1's in R_B.
- $C = c_1c_2 \ldots c_{n'}$ undergoes the same transformation: if $c_i > n$, we set it to $c_i = n + 1 + 3(c_i - n - 1)$.

Let us now reduce the dictionary, in our example, by expanding the definition of s within t (even if s is used elsewhere). The new bitmap is R_B = `11000100` (where $t = 1$, $s = 2$, and $u = 6$), the sequence is $R_S = abc12$, and $C = 16b$. We can now remove the definition of t by expanding it within u. This produces the new bitmap R_B = `1110000` (where $u = 1$, $t = 2$, $s = 3$), the sequence $R_S = abc3$ and $C = 21b$. Further reduction is not possible because u's definition is only used from C.[2] At the cost of storing at most $2|R|$ bits (for R_B), we can reduce R by one integer for each definition that is used at least once within R.

[2]It is tempting to replace u in C, as it appears only once, but our example is artificial: A symbol that is not mentioned in R must appear at least twice in C.

The reduction can be easily implemented in linear time, avoiding the successive renamings of our example, as follows: We first check for each rule if it is used within R, marking this in a bitmap U. Then, we traverse R and only write down (the bits of R_B and the sequence R_S for) the entries that are not used within R. We recursively expand those entries, appending the resulting tree structure to R_B and leaf identifiers to R_S. Whenever we find a created symbol that does not yet have an identifier, we give it as identifier the current position in R_B and recursively expand it. We store these new identifiers in an array NV. Otherwise, the expansion finishes and we write down a leaf (a "0") in R_B and the identifier in R_S. Then we rewrite C using the renamed identifiers. Figure 4.2 shows detailed pseudocode.

```
**Algorithm** Compress_Dictionary($R = \{s_1 \rightarrow a_1b_1, \ldots, s_\nu \rightarrow a_\nu b_\nu\}$, $C = c_1 \ldots c_{n'}$)
    **for** $i \leftarrow 1 \ldots \nu$ **do** $U[i] \leftarrow 0$
    **for** $i \leftarrow 1 \ldots \nu$ **do**
        **if** $a_i > n$ **then** $U[a_i - n] \leftarrow 1$
        **if** $b_i > n$ **then** $U[b_i - n] \leftarrow 1$
    **for** $i \leftarrow 1 \ldots \nu$ **do** $NV[i] \leftarrow 0$
    $j \leftarrow 1$, $R_B \leftarrow \langle\rangle$, $R_S \leftarrow \langle\rangle$
    $LR_B \leftarrow 0$ // length in bits of bitmap $R_B$
    **for** $j \leftarrow 1 \ldots \nu$ **do**
        **if** $U[j] = 0$ **then** Expand_Rule($j$)
    **for** $i \leftarrow 1 \ldots n'$ **do**
        **if** $c_i > n$ **then** $c_i \leftarrow NV[c_i - n] + n$
**return** $(R_B, R_S, C)$

**Algorithm** Expand_Rule($j$)
    $R_B \leftarrow R_B : 1$, $LR_B \leftarrow LR_B + 1$
    $NV[j] \leftarrow LR_B$
    **if** $a_j \leq n$ **then**
        $R_S \leftarrow R_S : a_j$, $R_B \leftarrow R_B : 0$, $LR_B \leftarrow LR_B + 1$
    **else if** $NV[a_j - n] > 0$ **then**
        $R_S \leftarrow R_S : NV[a_j - n] + n$, $R_B \leftarrow R_B : 0$, $LR_B \leftarrow LR_B + 1$
    **else** Expand_Rule($a_j - n$)
    **if** $b_j \leq n$ **then**
        $R_S \leftarrow R_S : b_j$, $R_B \leftarrow R_B : 0$, $LR_B \leftarrow LR_B + 1$
    **else if** $NV[b_j - n] > 0$ **then**
        $R_S \leftarrow R_S : NV[b_j - n] + n$, $R_B \leftarrow R_B : 0$, $LR_B \leftarrow LR_B + 1$
    **else** Expand_Rule($b_j - n$)
```

Figure 4.2: Algorithm to compress the dictionary R and update C in $O(n)$ time. R_B, R_S, NV, and LR_B act as global variables. "$\langle\rangle$" is the empty sequence and ":" the concatenation operator.

We can further compress the dictionary, if we take into account that a rule only uses previous rules or original symbols. That is, the i-th rule can only point to elements with representation of length $\lceil \log_2 i \rceil$ bits. With a simple arithmetic computation we can directly access any rule.

Another way to further compress the dictionary, yet with a time penalty, is as follows: Instead of using the position i of a rule inside bitmap R_B, use $j = rank_1(R_B, i)$. Given that j, we find the position in R_B where the rule starts with $i = select_1(R_B, j)$. We gain at least 1 bit per rule in the dictionary and in the text.

4.2 Analysis of Compression Ratio

We analyze the compression ratios of our data structures, first for the exact and then for our approximate method based on Ψ.

Let N be the number of runs in Ψ. As shown in [MN05, NM07], $N \leq \min(n, nH_k + \sigma^k)$ for any $k \geq 0$. Except for the first cell of each run, we have that $A'[i] = A'[\Psi(i)]$ within the run. Thus, we cut off the first cell of each run, to obtain up to $2N$ runs in A'. Every pair $A'[i]A'[i+1]$ contained in such runs must be equal to $A'[\Psi(i)]A'[\Psi(i)+1]$, thus the only pairs of cells $A'[i]A'[i+1]$ that are not equal to the "next" pair are those where i is the last cell of its run. This shows that there are at most $2N$ different pairs in A', as a traversal following Ψ permutation will change pairs only $2N$ times.

Exact Re-Pair. Since there are at most $2N$ different pairs, the most frequent pair appears at least $\frac{n}{2N}$ times. Because of overlaps, it could be that only each other occurrence can be replaced, thus the total number of replacements in the first iteration is at least βn, for $\beta = \frac{1}{4N}$.

After we choose and replace the most frequent pair, we end up with at most $n(1-\beta)$ integers in A'. The number of runs has not varied, because a replacement cannot split a run. Thus, the same argument shows that the second time we end up with at most $n(1-\beta)^2$ symbols, and so on.

After M iterations, the length of C is at most $n(1-\beta)^M$ and the length of R is $2M$. Assume for a while $N/n \leq 1/4$, thus $\beta n \geq 1$. The total size is optimized for $M^* = \frac{\ln n + \ln\ln \frac{1}{1-\beta} - \ln 2}{\ln \frac{1}{1-\beta}}$, where it is $\frac{2(\ln n + \ln\ln \frac{1}{1-\beta} - \ln 2 + 1)}{\ln \frac{1}{1-\beta}}$. (Re-Pair shortens the total file size in each new iteration, so the final result cannot be worse than that after M^* iterations.) Since $\ln \frac{1}{1-\beta} = \ln \frac{4N}{4N-1} = \frac{1}{4N}(1 + O(\frac{1}{N}))$, we have $M^* = \ln \frac{n}{4N} + O(1)$ and the total size is $8N \ln \frac{n}{4N} + O(N)$ integers. Since $N \leq H_k n + \sigma^k$, if we stick to $k \leq \alpha \log_\sigma n$ for any constant $0 < \alpha < 1$, it holds $\sigma^k = O(n^\alpha)$ and $N/n \leq H_k + O(n^{\alpha-1})$. If $H_k + O(n^{\alpha-1}) < 1/e$ the space formula is increasing in N, thus the total space is $O((nH_k + O(n^\alpha)) \log \frac{1}{H_k + O(n^{\alpha-1})} + nH_k + O(n^\alpha)) = O(nH_k \log \frac{1}{H_k + O(n^{\alpha-1})} + n^\alpha \log n)$ integers.

As $h \log \frac{1}{h+x} = h \log \frac{1}{h} + h \log \frac{1}{1+x/h} = h \log \frac{1}{h} + O(x)$, the space is $O(nH_k \log \frac{1}{H_k} + n^\alpha \log n)$ integers. This is $O(H_k \log \frac{1}{H_k}\ n \log n) + o(n)$ bits, as even after the M^* replacements the numbers need $\Theta(\log n)$ bits. The result is interesting only for $N = o(n)$ and $H_k = o(1)$, in which case all of our conditions on N and H_k hold. Yet, note that the results hold anyway for larger H_k and N.

Theorem 4.2.1. *After running exact Re-Pair, our structure represents A' using R and C in $O(N(1+\log \frac{n}{N}))$ integers, where N is the number of runs in Ψ. This is $O(H_k \log \frac{1}{H_k}\ n \log n) + o(n)$ bits, for any $k \le \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$, if $H_k = o(1)$, and $O(n \log n)$ bits otherwise.*

As a comparison, Mäkinen's Compact Suffix Array [Mäk03] needs $O(H_k n \log n)$ bits of space [NM07], which is always better as a function of H_k. Yet, both tend to the same space as H_k goes to zero. Other self-indexes are usually smaller.

Approximate Re-Pair. We now show that the simplified replacement methods of Sections 4.1.2 and 4.1.3 reach the same asymptotic space complexity. Assume as before that N and H_k are sufficiently small.

Just as for the exact method, the traversal using Ψ will create up to $2N$ pairs per pass. Assume for simplicity that, as we find each new pair in the traversal using Ψ, we always replace the pair, even if this involves creating it in R for just one occurrence in C (this is never better than the real algorithm). Thus, we try to make all the $|A'|$ replacements, but we may fail because replacements overlap. That is, assume we have *abcd* and first replace $s \to bc$. In the new sequence *asd* we cannot make a replacement $s' \to ab$ nor $s' \to cd$. Indeed, in the best case we can carry out $\lfloor |A'|/2 \rfloor$ replacements, whereas in the worst case this is only $\lfloor |A'|/3 \rfloor$ (when we first choose all multiples of 3 as initial pair positions).

This shows that, in the first pass over Ψ, we add up to $4N$ integers to R and remove at least $n/3$ integers from A'. For the next pass, the key point is that the number of runs is still limited by $2N$: If we had that $A'[i] = A'[\Psi(i)]$, the fact stays valid after we replace both $A'[i]$ and $A'[\Psi(i)]$ by a new symbol (cells $A'[i+1]$ and $A'[\Psi(i)+1]$ are invalid for the next pass). Therefore, we can analyze the process using recurrence $S(n) = 4N + S(2n/3)$. If we repeat the process i times and then call C the remaining cells of A', we get $S(n) = 4Ni + (2/3)^i n$, which is optimized for $i^* = \log_{3/2}(n/4N) - 1$ iterations, where we get $S(n) = O(N \log \frac{n}{N})$ integers. Even after adding $O(Ni^*)$ new symbols these integers need $\Theta(\log n)$ bits.

Stronger approximate Re-Pair. This analysis is similar to that of exact Re-Pair. The relevant invariant, which is easy to check from the description of Section 4.1.3, is as follows: *The approximate algorithm always replaces a pair that appears at least $\delta \cdot f$ times, being f the frequency of the currently most frequent pair.* In this sense, the algorithm acts as a δ-approximation.

In exact Re-Pair, we first replace the most frequent pair, which appears at least $\frac{n}{2N}$ times. In this case, we first replace a pair that appears at least $\frac{\delta n}{2N}$ times. This gives us a total number of replacements in the first iteration of at least $\beta' n$, where $\beta' = \delta\beta = \frac{\delta}{4N}$. The same occurs at each stage of the algorithm. Applying the same arguments of the analysis of exact Re-Pair with this new β', and the fact that $0 < \delta < 1$ is a constant, we obtain the same result as in Theorem 4.2.1.

Theorem 4.1. *The process of replacing pairs following permutation Ψ, in either of its variants, achieves a data structure that fits in $O(N \log \frac{n}{N})$ integers, where N is the number of runs in Ψ. This is $O(H_k \log \frac{1}{H_k}\, n \log n) + o(n)$ bits, for any $k \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$.*

4.3 Towards a Text Index

As explained at the beginning of the chapter, the LCSA is not by itself a text index. We explore now some alternatives to upgrade it to a full-text index.

4.3.1 A Smaller Classical Index

A simple and practical alternative is to use our LCSA just like the classical suffix array, that is, not only for locating but also for searching, keeping the text in uncompressed form as well. This is not really a compressed index, but a practical alternative to a classical index.

The binary search of the interval that corresponds to P will start over the absolute samples of our LCSA. Only when we have identified the interval between consecutive samples of A where the binary search must continue, we decompress the whole interval and finish the binary search. If the two binary searches finish in different intervals, we will also need to decompress the intervals in between for locating all the occurrences. For displaying, the text is at hand.

The cost of this search is $O(m \log n)$ plus the time needed to decompress the portion of A between two absolute samples. We can easily force the compressor to make sure that no symbol in C spans the limit between two such intervals, so that the complexity of this decompression can be controlled with the sampling rate l. For example, $l = O(\log n)$ guarantees a total search time of $O(m \log n + occ)$, just as the suffix array version that requires 4 times the text size (plus text).

Theorem 4.2. *There exists a full-text index for text T of length n over an alphabet of size σ and k-th order entropy H_k, which requires $O(H_k \log \frac{1}{H_k}\, n \log n + n \log^{1-\epsilon} n)$ bits of space in addition to T, for any constant $0 \leq \epsilon \leq 1$, any $k \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$. It can count the occurrences of a pattern of length m in time $O(m \log n)$ and locate its occ occurrences in time $O(occ + \log^\epsilon n)$.*

The theorem is obtained by considering that C and R use $O(H_k \log \frac{1}{H_k}\ n \log n + O(n^{\alpha} \log^2 n)$ bits, to which we must add $O((n/l) \log n)$ bits for the absolute samples in A', and the extra cost to limit the formation of symbols that represent very long sequences. If we limit such symbol lengths to l as well, we have an overhead of $O((n/l) \log n)$ bits, as this can be regarded as inserting spurious symbols every l positions in A' to prevent the formation of longer symbols. By choosing $l = \log^{\epsilon} n$ we have $O(H_k \log \frac{1}{H_k}\ n \log n + n \log^{1-\epsilon} n)$ bits of space. The time is $O(m \log n + \log^{\epsilon} n)$ for counting, and $O(occ + \log^{\epsilon} n)$ for locating the occurrences.

4.3.2 A Compressed Self-Index

Another choice to achieve a full-text index is to plug our LCSA to any of the many self-indexes able of giving the suffix array range of the occurrences of P within little space [FM05, FMMN07, Sad03, GGV03]. Given the area $[sp, ep]$ where the occurrences lie in A, locating the occurrences boils down to decompressing $A[sp, ep]$ from our LCSA structure.

To fix ideas, consider the alphabet-friendly FM-index [FMMN07]. It takes $nH_k + o(n \log \sigma)$ bits of space for any $k \leq \alpha \log_{\sigma} n$ and constant $0 < \alpha < 1$, and can count in time $O(m(1 + \frac{\log \sigma}{\log \log n}))$. Our additional structure dominates the space complexity, requiring $O(H_k \log \frac{1}{H_k}\ n \log n + n \log^{1-\epsilon} n)$ bits.

Extracting substrings can be done with the same FM-index, but the time to display ℓ text characters is, using $n \log^{1-\epsilon} n$ additional bits of space, $O((\ell + \log^{\epsilon} n)(1 + \frac{\log \sigma}{\log \log n}))$. By using instead the structure proposed in Chapter 5, we have other $nH_k + o(n \log \sigma)$ bits of space for $k = o(\log_{\sigma} n)$ (this space is asymptotically negligible) and can extract the characters in optimal time $O(1 + \frac{\ell}{\log_{\sigma} n})$.

Theorem 4.3. *There exists a self-index for text T of length n over an alphabet of size σ and k-th order entropy H_k, which requires $O(H_k \log \frac{1}{H_k}\ n \log n + n \log^{1-\epsilon} n) + o(n \log \sigma)$ bits of space, for any $0 \leq \epsilon \leq 1$. It can count the occurrences of a pattern of length m in time $O(m(1 + \frac{\log \sigma}{\log \log n}))$ and locate its occ occurrences in time $O(occ + \log^{\epsilon} n)$. For $k = o(\log_{\sigma} n)$ it can display any text substring of length ℓ in time $O(1 + \frac{\ell}{\log_{\sigma} n})$. For larger $k \leq \alpha \log_{\sigma} n$, for any constant $0 < \alpha < 1$, this time becomes $O((\ell + \log^{\epsilon} n)(1 + \frac{\log \sigma}{\log \log n}))$.*

4.4 Experimental Results

We present three series of experiments in this section. The first one regards compression performance, the second regards the use of our technique as a classical index using reduced space, and the third regards the use as a plug-in for boosting the locating performance of a compressed self-index.

We use text collections obtained from the *Pizza&Chili* site. We use all the text types available, with sizes of 50MB and 100MB for our experiments. The experiments were run on a Pentium IV, 2.0 GHz with 2GB RAM using Linux with kernel 2.4.31 and `GNU g++` with -O3 optimization.

Compression performance. In Section 4.1.2, we mentioned that the compression time of exact Re-Pair would be an issue, and gave two approximate methods based on Ψ which should be faster.

In this section, we call RP our implementation of the exact Re-Pair compression algorithm[3], RPΨ_0 the Ψ-based approximation that runs in $O(n)$ time (Section 4.1.2), and RPΨ the Ψ-based approximation that runs in $O(n \log n)$ time (Section 4.1.3). We also include the methods RPSP, RPΨ_0SP, and RPΨSP. These forbid a rule to cross a 256-cell boundary. In all cases, we take absolute A' samples each 64 positions.

In practice, RPΨ_0 and RPΨ_0SP are very fast in comparison with the rest of the methods, yet compress less. Considering this, we could relax the stopping condition with the aim of better balancing these factors. In Figure 4.3 we show space reduction achieved from one pass to the next, using RPΨ_0SP. For example, in the case of `xml` (the one needing the most passes), we remove 45% of the file in the first pass, and 40% of that in the second, but after 8 passes the compression gain is less than 0.01%. Considering this, we force at least 8 passes after reaching less than $\alpha n'$ replacements per pass. We use $\alpha = 1/4$, which gives good results.

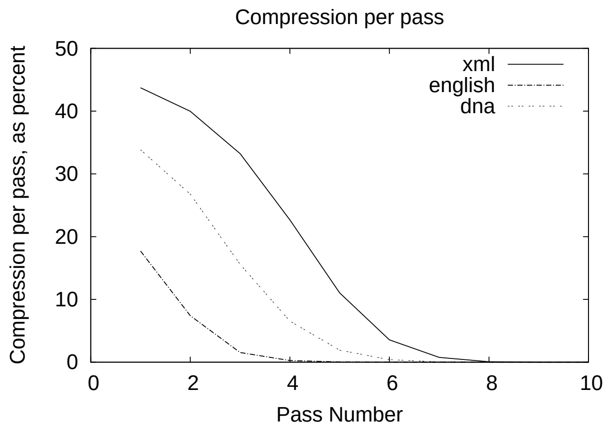


Figure 4.3: Compression achieved per pass using RPΨ_0SP. We use files of size 100MB.

To tune the parameters of the approximate variant RPΨSP, we test different values on two small files (`english` and `xml`, truncated to 50MB). We show, among several we carried out, the following experiments, as they best reflect the choice of parameters. Table 4.1 shows that the compression gain for increasing s loses importance for $s > 8$. Table 4.2, on the other

[3]Those we could find freely available did not work properly.

hand, shows that increasing δ does not give any gain on `english`, yet it slightly improves compression ratio on `xml`. A fair choice of parameters for RPΨSP, which we use for the rest of the experiments, is $s = 8$, $\delta = 3/4$ and $\gamma = \log n$.

s	Compr. ratio `xml`	Compr. ratio `english`	Compr. time (sec) `xml`	Compr. time (sec) `english`
1	26.03%	55.47%	766	1,374
2	25.95%	55.35%	905	1,475
4	25.91%	55.32%	1,080	1,621
8	25.89%	55.30%	1,241	1,837
16	25.88%	55.29%	1,344	2,028
32	25.87%	55.28%	1,519	2,425
64	25.87%	55.28%	1,645	2,801
128	25.87%	55.28%	1,717	3,213

Table 4.1: Compression ratio obtained using different values of s for the approximation RPΨSP. In this case, we use $\delta = 1/2$. The percentage is computed by comparing with the $4n$ bytes required by a standard suffix array implementation.

δ	Compr. ratio `xml`	Compr. ratio `english`	Compr. time (sec) `xml`	Compr. time (sec) `english`
1/2	25.89%	55.30%	1,241	1,837
3/4	25.81%	55.29%	1,573	2,159
7/8	25.74%	55.29%	2,091	2,786
15/16	25.68%	55.29%	2,835	4,184
31/32	25.60%	55.29%	4,185	7,150

Table 4.2: Compression obtained using different values of δ using approximation RPΨSP. In this case, we use $s = 8$.

Table 4.3 shows that the compression ratio varies widely. On `xml` data, we achieve 23.5% compression (the reduced suffix array is smaller than the text!), whereas compression is extremely poor on `dna`. In many text types of interest, we slash the suffix array to around half of its size. Below the name of each collection we wrote the percentage H_3/H_0, which gives an idea of the compressibility of the collection independent of its alphabet size (e.g., it is very easy to compress `dna` to 25% because there are mainly 4 symbols but one chooses to spend a byte for each symbol in the uncompressed text, otherwise `dna` is almost incompressible). The measure turns out to be an excellent predictor of the compression, except for `proteins` where we are closer to H_5/H_0.

We exclude `dna` to state the following numbers, because of its poor compression ratio. The approximation RPΨ_0 runs up to 180 times faster and just loses 3.3%–17.8% in compression ratio compared to RP. The approximation RPΨ runs up to 25 times faster and just loses up to 3.5% in compression ratio. RP runs at 25 to 1000 sec/MB, RPΨ_0 runs at 5 to 10 sec/MB and RPΨ runs at 31 to 56 sec/MB. Suffix array construction is the same in all the methods and takes around 100 seconds overall in all cases. Thus, most of the indexing time shown in Table 4.3 is spent by the compression methods.

Coll. size	Method	Index size	Compr.	Compr.	Expected	Dict.	Compr. 2%
(MB),H_3/H_0		(MB)	ratio	time (s)	decompr.	compr.	in RAM
`xml`,100,	RP	93.56	23.39%	29,800	6,936.54	57.22%	34.08%
26.28%	RPSP	99.52	24.88%	25,472	134.91	58.29%	35.84%
	RPΨ_0	103.06	25.76%	625	7,570.49	57.46%	91.42%
	RPΨ_0SP	116.13	29.03%	651	83.79	58.68%	91.59%
	RPΨ	94.26	23.56%	3,547	6,948.85	57.18%	36.15%
	RPΨSP	102.90	25.73%	3,598	95.83	58.23%	40.63%
`dna`,100,	RP	336.53	84.13%	8,511	5.01	79.49%	92.45%
97.02%	RPSP	337.11	84.28%	8,402	4.25	79.72%	92.56%
	RPΨ_0	342.52	85.63%	931	4.73	78.20%	99.17%
	RPΨ_0SP	343.11	85.78%	899	4.04	78.44%	99.18%
	RPΨ	336.37	84.09%	4,279	5.03	79.19%	93.19%
	RPΨSP	336.96	84.24%	4,260	4.28	79.41%	93.30%
`english`,100,	RP	227.59	56.90%	87,285	238.31	59.27%	88.15%
53.05%	RPSP	230.04	57.51%	86,273	30.37	59.71%	88.33%
	RPΨ_0	249.03	62.26%	974	202.79	59.70%	98.56%
	RPΨ_0SP	252.08	63.02%	944	26.83	60.19%	98.56%
	RPΨ	227.74	56.94%	4,621	215.12	59.17%	88.92%
	RPΨSP	230.26	57.56%	4,600	29.99	59.60%	89.12%
`pitches`,50,	RP	116.58	58.29%	11,454	33.96	69.51%	67.41%
61.37%	RPSP	117.61	58.81%	11,067	17.00	70.08%	67.78%
	RPΨ_0	126.56	63.28%	279	26.38	66.86%	97.32%
	RPΨ_0SP	127.83	63.91%	535	14.21	67.23%	97.34%
	RPΨ	117.98	58.99%	1,618	28.89	68.67%	70.17%
	RPΨSP	119.18	59.59%	1,807	15.66	69.00%	71.03%
`proteins`,100,	RP	284.61	71.15%	2,642	58.98	79.72%	75.63%
97.21%	RPSP	285.94	71.48%	2,732	13.87	80.09%	76.00%
	RPΨ_0	294.08	73.52%	1,045	52.52	75.16%	92.13%
	RPΨ_0SP	296.24	74.06%	1,032	10.79	75.03%	92.30%
	RPΨ	285.94	71.49%	5,719	58.46	78.98%	76.77%
	RPΨSP	287.73	71.93%	5,764	11.92	78.80%	77.31%
`sources`,100,	RP	154.88	38.72%	107,371	2,041.88	57.63%	65.16%
40.74%	RPSP	159.34	39.83%	103,292	60.48	58.33%	65.85%
	RPΨ_0	181.38	45.34%	684	1,778.79	58.09%	96.67%
	RPΨ_0SP	187.52	46.88%	677	50.97	58.80%	96.70%
	RPΨ	156.18	39.04%	4,380	1,928.86	57.48%	68.00%
	RPΨSP	161.21	40.30%	4,778	56.89	58.13%	69.15%

Table 4.3: Index size and build time using Re-Pair and its Ψ-based approximations. We also include versions with rules up to length 256 (SP extension). Compression ratio compares with the $4n$ bytes needed by a plain suffix array representation.

Other statistics are also available in Table 4.3. In column 6 we measure the average length of a cell of C if we choose uniformly in A (longer cells are in addition more likely to be chosen for decompression). Those numbers explain the times obtained for the next series of experiments. Note that they are related to compressibility, but not as much as one could expect. Rather, the numbers obey to a more detailed structure of the suffix array: they are higher when the compression is not uniform across the array. In every case, we can limit the maximum length of a C cell. The SP variants show how this impacts compression ratio and decompression speed. We can see that their compression ratio is almost the same, worsening at most by 6.37% (RP), 12.68% (RPΨ_0), or 9.17% (RPΨ) on `dna` (and much less on others).

In column 7 we show the compression ratio achieved on the dictionary part using the technique of Section 4.1.4, charging it the bitmap introduced as well. It can be seen that the technique is rather effective, approaching in some cases the limit of 50% to its possible effectiveness. (We remark that the compression ratios of previous columns do account for the dictionary space and all the necessary structures to operate.)

The last column shows how much compression we would achieve if the structures that must reside on RAM were limited to 2% of the original suffix array size. We still obtain attractive compression performance on texts like `xml`, `sources` and `pitches` (recall that on secondary memory the compression ratio translates almost directly to decompression performance). As expected, RPΨ_0 does a much poorer job here, as it does not choose the best pairs early, but RPΨ achieves almost the same performance as RP.

A classical reduced index. We test RPΨSP (from now on LCSA) as a replacement of the suffix array, that is, adding it the text and using it for binary searching, as explained in Section 4.3.1. We compare it with a plain suffix array (SA) and Mäkinen's Compact Suffix Array (Mak-CSA [Mäk03]), as the latter operates similarly, recall Section 2.9.2.

Figure 4.4 shows the result. Mak-CSA offers space-time tradeoffs, whereas those of our index (sample rate for absolute values) did not significantly affect the time. Our structure stands out as a relevant space/time tradeoff, especially when locating many occurrences (i.e., on short patterns). In particular, LCSA is usually noticeably faster than Mak-CSA for the same space, yet the latter is able of using less space (at least on `english`). Compared to a plain suffix array, LCSA requires 0.9–2.4 times the text size (as opposed to 4) plus the text, at the price of being 2–28 times slower for locating. Compared to current state of the art in compressed indexing (Section 3.4.3), this slowdown is rather modest.

A plugin for self-indexes. Section 4.3.2 considers using our reduced suffix array as a plugin to provide fast *locate* on existing self-indexes. In this experiment, we plug our structure to the counting structures of the alphabet-friendly FM-index (AF-index [FMMN07]), and compare the result against the original AF-index, Sadakane's CSA [Sad03] and the SSA [FMMN07, MN05], all from *Pizza&Chili*. We increased the sampling rate of the locating

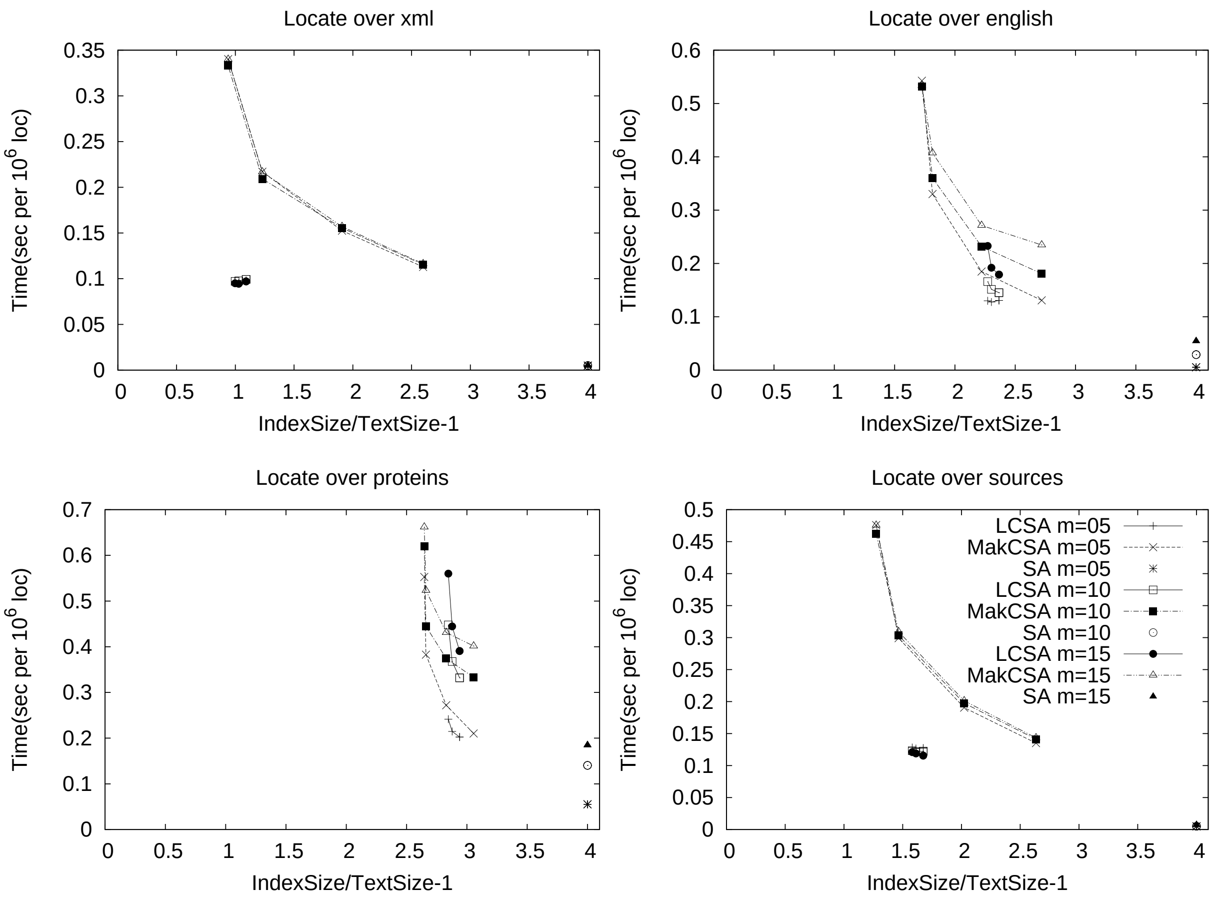


Figure 4.4: Simulating a classical suffix array to binary search and locate the occurrences. Each file is of size 100MB.

structures of AF-index, CSA and SSA, to match the size of our LCSA. Figure 4.5 and Table 4.4 show the results. We only show four texts, as the others yield similar conclusions.

The experiment in Figure 4.5 consists in choosing random ranges of the suffix array and obtaining the values in them . This simulates a locating query where we can control the amount of occurrences to locate. Our reduced suffix array has a constant time overhead (which is related to column 6 in Table 4.3 and to the sample rate of absolute values) and from then on the cost per located cell is very low. As a consequence, it crosses sooner or later all the other indexes. For example, it becomes the fastest on `xml` after locating 2 occurrences, and after 8 occurrences it becomes the fastest on `proteins`. Particularly on `xml`, this success owes to the fact that our LCSA uses the RPΨSP variant (cutting phrases at length 256 as explained). If instead we used RPΨ to gain a little further compression, the result would be very inefficient due to the long phrases that need to be decompressed. In the case of `xml`, RPΨ becomes the fastest only after locating 3,800 occurrences, not 2. In other texts where compression is not so good, there is not much difference between RPΨ and RPΨSP (both in time and space).

Table 4.4 shows the *locate* time required by a classical suffix array, our LCSA plugged to the SSA, and the LZ-index. The times are in seconds per 10^6 occurrences for different texts and pattern lengths. In our LCSA, we forbid a rule to cross a 256-cell boundary and take absolute A' samples each 64 positions. For the SSA, we store no samples for *locate*, but we sample each 1024 positions for *extract*. We use the fastest LZ-index for our experiments (`LZ-Index` in *Pizza&Chili* site). We show the values for the suffix array as a reference. Our index is 1.4–2 times larger than the LZ-index, but for short patterns ($m = 5$) we are 5–50 times faster and for long patterns ($m = 50$, where the time for counting is more important) we are 14–27 times faster.

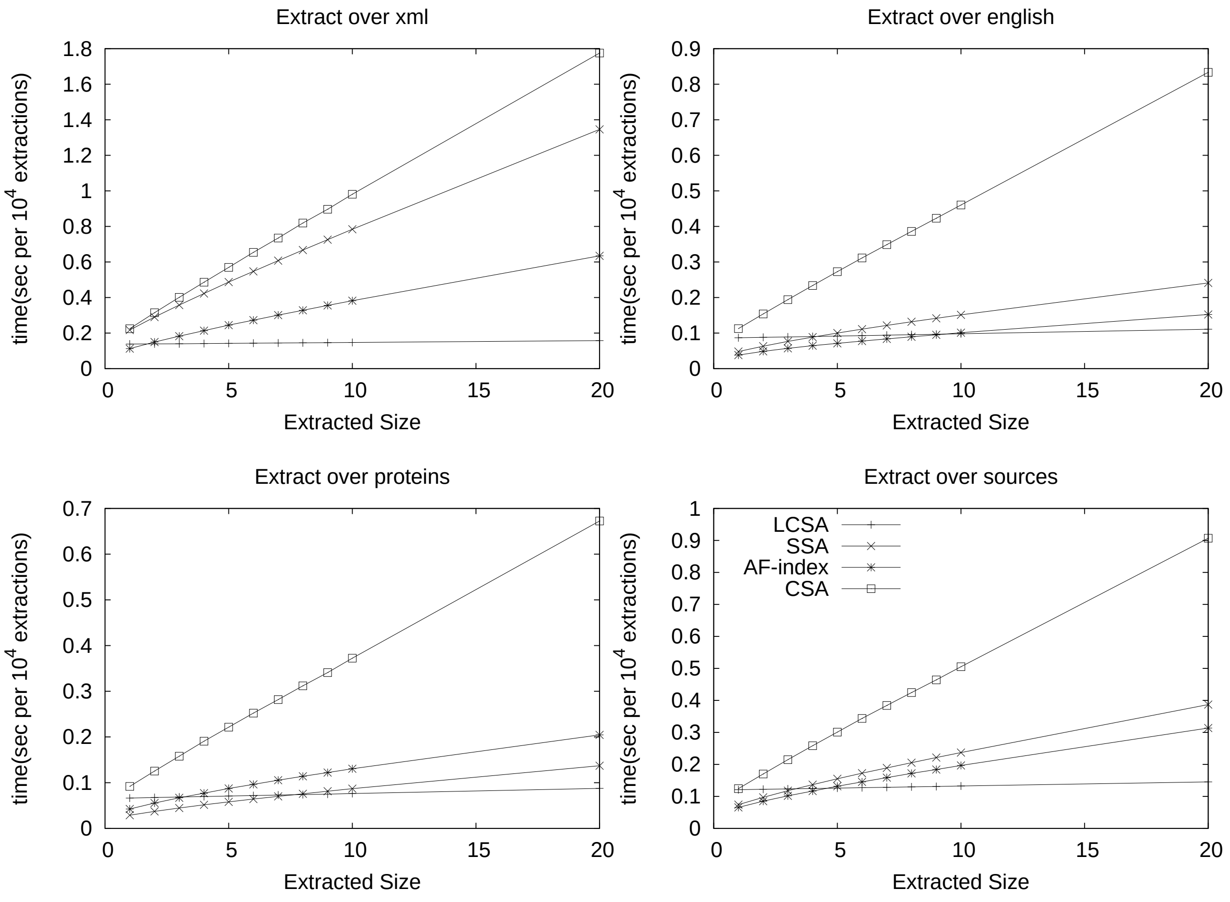


Figure 4.5: Time to extract a portion of the suffix array. Each file is of size 100MB.

Text		Suffix Array	LCSA	LZ-index
`xml` 100 MB	Index size	5.0000	1.8504	0.9529
	Time for m=5	0.0137	0.0922	0.6647
	Time for m=15	0.0143	0.0926	0.6911
	Time for m=50	0.0664	0.6681	9.7823
`english` 100 MB	Index size	5.0000	3.0368	1.7395
	Time for m=5	0.0145	0.1213	1.1075
	Time for m=15	0.0807	0.3433	5.7775
	Time for m=50	0.9226	9.3267	250.5900
`proteins` 100 MB	Index size	5.0000	3.5630	2.4889
	Time for m=5	0.0750	0.2054	10.7664
	Time for m=15	0.2312	0.8743	11.1488
	Time for m=50	0.4584	4.2309	75.6887
`sources` 100 MB	Index size	5.0000	2.4736	1.7328
	Time for m=5	0.0144	0.1242	0.6684
	Time for m=15	0.0158	0.1189	0.7371
	Time for m=50	0.0589	0.6311	10.0147

Table 4.4: *Locate* time required by the classical suffix array, the LCSA and the LZ-index, in seconds per 10^6 occurrences, with different texts and pattern lengths.

Chapter 5

Statistical Encoding of Sequences

Several *succinct* data structures are built over a sequence of symbols $S[1,n] = s_1 s_2 \ldots s_n$, from an alphabet Σ of size σ, and require only $o(|S|) = o(n \log \sigma)$ additional bits in addition to S itself (S requires $n \log \sigma$ bits). A more ambitious goal is a *compressed* data structure, which takes overall space proportional to the compressed size of S (plus sublinear terms) and is still able to recover any substring of S and manipulate the data structure.

A recent result by Sadakane and Grossi [SG06b] gives a tool to convert *any* succinct data structure on sequences into a compressed data structure. More precisely, they show that S can be encoded using $nH_k(S) + O(\frac{n}{\log_\sigma n}(k \log \sigma + \log \log n))$ bits of space[1]. Their structure permits retrieving any substring of S of $\Theta(\log_\sigma n)$ symbols in constant time. Under the RAM model of computation this is equivalent to having S in explicit form.

In particular, for sufficiently small $k = o(\log_\sigma n)$, the space is $H_k(S) + o(n \log \sigma)$. Any succinct data structure that requires $o(n \log \sigma)$ bits in addition to S can thus be replaced by a compressed data structure requiring $nH_k(S) + o(n \log \sigma)$ bits overall, where any access to S is replaced by an access to the novel structure. Their scheme is based on Ziv-Lempel encoding.

In this chapter, we show how the same result can be achieved by much simpler means. We present an alternative scheme based on semi-static k-th order modeling plus statistical encoding, just as a normal semi-static statistical compressor would process S (Sections 2.1 and 2.2). By adding some extra structures, we are able of retrieving any substring of S of $\Theta(\log_\sigma n)$ symbols in constant time. Although any statistical encoder works, we obtain the best results (matching exactly those of [SG06b]) using Arithmetic encoding [BCW90]. Furthermore, we show that we can append symbols to S without changing the asymptotic

[1]The term $k \log \sigma$ appears as k in [SG06b], but this is a mistake [SG06a]. The reason is that they take from [KM99] an extra space of the form $\Theta(kt + t)$ as stated in Lemma 2.3, whereas the proof in Theorem A.4 gives a term of the form $kt \log \sigma + \Theta(t)$.

space complexity, in constant amortized time per symbol. We call our scheme EBDS (entropy-bounded data structure).

In addition, we study the applicability of this technique to full-text self-indexes. Compressed self-indexes replace a text $T[1,n]$ by a structure requiring $O(nH_0(T))$ or $O(nH_k(T))$ bits of space. In order to provide efficient pattern matching over T, many of those structures [FM05, MN05, FMMN07] achieve space proportional to $nH_k(T)$ by first applying the Burrows-Wheeler Transform [BW94] over T, $S[1,n] = T^{bwt}$ (Section 2.8), and then struggling to represent S in efficient form, recall Section 2.9.1. An additional structure of $o(|S|)$ bits gives the necessary functionality to implement the search. One could thus apply the new structure over S, so that the overall structure requires $nH_k(S) + o(|S|)$ bits. Yet, the relation between $H_k(S)$ and $H_k(T)$ remains unknown. In this chapter, we move a step forward by proving a positive result: $H_1(S) \leq H_k(T)\log\sigma + o(1)$ for small $k = o(\log_\sigma n)$. Thus we can, for example, achieve essentially the same result of the Run-Length FM-Index [MN05] (Section 3.2.1) just by using the new structure on S, without the involved techniques they use.

Several indexes, however, compress $S = T^{bwt}$ by means of a *wavelet tree* [GGV03] on S, $wt(S)$. This is a balanced tree storing several binary sequences (Section 2.4). Each such sequence B can be represented using $|B|H_0(B)$ bits of space. If we call $nH_0(wt(S))$ the overall resulting space, it turns out that $nH_0(wt(S)) = nH_0(S)$. A natural idea advocated in [SG06b] is to use a k-th order representation for the binary sequences B, yielding space $nH_k(wt(S))$. Thus the question about the relationship between $H_k(wt(S))$ and $H_k(S)$ is raised. In this section, we exhibit examples where either is larger than the other. In particular, we show that when moving from $wt(S)$ to S, the k-th order entropy may grow at least by a factor of $\Theta(\log k)$.

5.1 A New Entropy-Bounded Data Structure

Given a sequence $S[1,n]$ over an alphabet $\Sigma = \{a_1, \ldots, a_\sigma\}$ of size σ, we encode S into a compressed data structure S' within entropy bounds. To perform all the original operations over S under the RAM model, it is enough to allow extracting any aligned block of $b = \lfloor \frac{1}{2}\log_\sigma n \rfloor$ consecutive symbols of S, using S', in constant time.

5.1.1 Data Structures for Substring Decoding

We describe our data structure to represent S in essentially $nH_k(S)$ bits, and to permit the access of any aligned substring of size $b = \lfloor \frac{1}{2}\log_\sigma n \rfloor$ in constant time. This structure is built using any statistical encoder E as described in Section 2.2.

Structure. We divide S into blocks of length $b = \lfloor \frac{1}{2}\log_\sigma n \rfloor$ symbols. Each block will be represented using at most $b' = \lfloor \frac{1}{2}\log n \rfloor$ bits (and hopefully less). We define the following sequences indexed by block number $i = 0, \ldots, \lfloor n/b \rfloor$:

- $S_i = S[bi+1, b(i+1)]$ is the sequence of symbols forming the i-th block of S.
- $C_i = S[bi-k+1, bi]$ is the sequence of symbols forming the k-th order context of the i-th block (a dummy value is used for C_0).
- $E_i = E(S_i)$ is the encoded sequence for the i-th block of S, initializing the k-th order modeler with context C_i.
- $\ell_i = |E_i|$ is the size in bits of E_i.
- $\tilde{E}_i = \left\{ \begin{array}{ll} S_i & \text{if } \ell_i > b' \\ E_i & \text{otherwise} \end{array} \right.$, is the shortest sequence among E_i and S_i.
- $\tilde{\ell}_i = |\tilde{E}_i| = \min(b', \ell_i)$ is the size in bits of $\tilde{E}_i$.

The idea behind $\tilde{E}_i$ is to ensure that no encoded block is longer than b' bits (which could happen if a block contains many infrequent symbols). These special blocks are encoded explicitly.

Our compressed representation EBDS of S stores the following information:

- $W[0, \lfloor n/b \rfloor]$: A bit array such that
$$W[i] = \left\{ \begin{array}{l} 0 \text{ if } \ell_i > b' \\ 1 \text{ otherwise} \end{array} \right. ,$$
with the additional $o(n/b)$ bits to support *rank* queries over W in constant time [Mun96].
- $C[1, rank(W, \lfloor n/b \rfloor)]$: $C[rank(W,i)] = C_i$, that is, the k-th order context for the i-th block of S if $\ell_i \leq b'$, with $1 \leq i \leq \lfloor n/b \rfloor$.
- $U = \tilde{E}_0 \tilde{E}_1 \ldots \tilde{E}_{\lfloor n/b \rfloor}$: A bit sequence obtained by concatenating all the variable-length $\tilde{E}_i$.
- $T : \Sigma^k \times 2^{b'} \longrightarrow 2^b$: A table defined as $T[\alpha, \beta] = \gamma$, where α is any context of size k, β represents any encoded block of at most b' bits, and γ represents the decoded form of β, truncated to the first b symbols (as less than the b' bits will be usually necessary to obtain the b symbols of the block).

- Where each $\tilde{E}_i$ starts within U. We group together every $c = \lceil \log n \rceil$ consecutive blocks to form superblocks of size $\Theta(\log^2 n)$ and store two tables:
  - $R_g[0, \lfloor n/(bc) \rfloor]$ contains the absolute position of each superblock.
  - $R_l[0, \lfloor n/b \rfloor]$ contains the relative position of each block with respect to the beginning of its superblock.

5.1.2 Substring Decoding Algorithm

We want to retrieve $S_j = S[j \cdot b, (j+1) \cdot b - 1]$ in constant time. To achieve this, we take the following steps:

1. We calculate $h = j$ div c, $h' = (j+1)$ div c and $u = U[R_g[h] + R_l[j] \ldots R_g[h'] + R_l[j+1] - 1]$, then
   - if $W[j] = 0$ then we have $S_j = u$.
   - if $W[j] = 1$ then we have $S_j = T[C[rank(W, j)], u']$, where u' is u padded with $b' - |u|$ dummy bits.

   We note that $|u| \leq b'$ and thus it can be manipulated in constant time.

Lemma 5.1. *For a given sequence $S[1, n]$ over an alphabet Σ of size σ, we can access any aligned substring of S of b symbols in $O(1)$ time using the EBDS.*

5.1.3 Space Requirement

Let us now consider the storage size of our structures.

- We use the constant-time solution to answer the *rank* queries over W (Section 2.4), totalizing $\frac{2n}{\log_\sigma n}(1 + o(1))$ bits.
- Table C requires at most $\frac{2n}{\log_\sigma n} k \log \sigma$ bits.
- The size of U is $|U| = \sum_{i=0}^{\lfloor n/b \rfloor} |\tilde{E}_i| \leq \sum_{i=0}^{\lfloor n/b \rfloor} |E_i| = nH_k(S) + O(k \log n) + \sum_{i=0}^{\lfloor n/b \rfloor} f_k(E, S_i)$, which depends on the statistical encoder E used, recall Section 2.2. For example, in the case of Huffman coding, we have $f_k(\text{Huffman}, S_i) < b$, and thus we achieve $nH_k(S) + O(k \log n) + n$ bits. For the case of Arithmetic coding, we have $f_k(\text{Arithmetic}, S_i) \leq 2$, and thus we have $nH_k(S) + O(k \log n) + \frac{4n}{\log_\sigma n}$ bits.
- The size of T is $\sigma^k 2^{b'} b \log \sigma = \sigma^k \ \frac{\log n}{2} \ n^{1/2}$ bits.

- Finally, let us consider tables R_g and R_l. Table R_g has $\lceil n/(bc) \rceil$ entries of size $\lceil \log n \rceil$, totalizing $\frac{2n}{\log_\sigma n}$ bits. Table R_l has $\lceil n/b \rceil$ entries of size $\lceil \log(b'c) \rceil$, totalizing $\frac{4n \log\log n}{\log_\sigma n}$ bits.

Since any substring of $\Theta(\log_\sigma n)$ symbols can be extracted in constant time by applying $O(1)$ times the procedure of Section 5.1.2, we have the final theorem.

Theorem 5.1. *Let $S[1,n]$ be a sequence over an alphabet Σ of size σ. The EBDS uses $nH_k(S)+O(\frac{n}{\log_\sigma n}(k\log\sigma+\log\log n))$ bits of space for any $k < (1-\epsilon)\log_\sigma n$ and any constant $0 < \epsilon < 1$, and it supports access to any substring of S of size $\Theta(\log_\sigma n)$ symbols in $O(1)$ time.*

Note that, in our scheme, the size of T can be neglected only if $k \leq (\frac{1}{2}-\epsilon)\log_\sigma n$, but this can be pushed to $(1-\epsilon)\log_\sigma n$, for any constant $0<\epsilon<1$, by choosing $\beta = \frac{\epsilon}{2}\log_\sigma n$. Thus, the size of T will be $\sigma^k n^{\frac{\epsilon}{2}} \frac{\log n}{s}$, which is negligible for, say, $k \leq (1-\frac{\epsilon}{2}-\frac{\epsilon}{2})\log_\sigma n$. The price is that now we must decode $O(1/\epsilon)$ blocks to extract $O(\log_\sigma n)$ bits from S, but this is still constant.

Corollary 5.1. *The EBDS takes space $nH_k(S)+o(n\log\sigma)$ if $k = o(\log_\sigma n)$.*

We note that, if we use Arithmetic coding, we will never have the problem of symbols with very low probability (which can require many bits, recall Section 2.2), because we do not encode any sequence that requires more than $\frac{\log n}{2}$ bits. As soon as a representation exceeds $\frac{\log n}{2}$ bits we switch to plain symbol-wise encoding. We also notice that we run into no efficiency problems at all at decoding time, as we will use the $\frac{\log n}{2}$-bit compressed stream as an index to a precomputed table that will directly yield the uncompressed symbols.

Our results match exactly those of [SG06b], once one corrects their k to $k\log\sigma$ as explained. Our method is simpler than theirs. However, their result holds simultaneously for all k, while in our structure k must be chosen beforehand.

Note that we are storing some redundant information that can be eliminated. The last characters of block S_i are stored both within $\tilde{E}_i$ and as C_{i+1}. Instead, we can choose to explicitly store the first k characters of *all* blocks S_i, and encode only the remaining $b-k$ symbols, $S_i[k+1,b]$, either in explicit or compressed form. This improves the space in practice, but in theory we cannot prove it to be better than the scheme we have given.

5.2 Supporting Appends

We can extend our scheme to support appending of symbols, while maintaining the same space and query complexity, with each appended symbol having constant amortized cost.

Assume our current static structure holds n symbols. We use a buffer of $n' = n/\log n$ symbols that are stored explicitly. When the buffer is full, we use our EBDS (Section 5.1) to represent those n' symbols and then we empty the buffer. We repeat this until we have $\log n$ EBDS's. At this moment we reencode all the structures plus our original n symbols, generating a new single EBDS, and restart the process with $2n$ symbols.

5.2.1 Data Structures

We describe the additional structures needed to append symbols to the EBDS.

- $BF[1, n']$ is the sequence of at most $n' = n/\log n$ uncompressed symbols.
- AP_i is the i-th EBDS, with $0 \le i \le N$. $N \le \log n$ is the number of EBDS we currently have. We call AS_i the sequence AP_i represents. AP_0 is the original EBDS. So $|AS_0| = n$ and $|AS_i| = n/\log n$, $i > 0$.

5.2.2 Substring Decoding Algorithm

We want to retrieve $S_j = S[j \cdot b, (j+1) \cdot b - 1]$. To achieve this, we algebraically calculate the index $0 \le t \le N+1$ where the position $j \cdot b$ belongs; $N+1$ represents BF.

If $t = N+1$ the symbols of S_j are explicitly represented in BF. Else, we apply the technique of Section 5.1.2 to AP_t.

5.2.3 Construction Time

Just after we reencode everything, we have that $n/2$ symbols have been reencoded once, $n/4$ symbols twice, $n/8$ symbols 3 times and so on. The total number of reencodings is $\sum_{i \ge 1} n \frac{i}{2^i} = 2n$. On the other hand, we are using a semi-static statistical encoder, which takes $O(1)$ time to encode each symbol. Thus each symbol has a worst-case amortized appending cost of $O(1)$.

5.2.4 Space Requirement

Let us now consider the storage of the appended structures.

- Buffer BF requires $n/\log_\sigma n$ bits.

- Each AP_i is an EBDS, using $|AS_i|H_k(AS_i) + O(\frac{|AS_i|}{\log_\sigma |AS_i|}(k \log \sigma + \log\log |AS_i|))$ bits of space.

Lemma 5.2. *The space requirement of all AP_i, for $0 \leq i \leq N$, is $\sum_{i=0}^{\log n} |AP_i| \leq |S\ AS_1 \ldots AS_N| H_k(S\ AS_1 \ldots AS_N) + O(\frac{n}{\log_\sigma n}(k \log \sigma + \log\log n))$ $+O(\sigma^{k+1} \log^2 n) + O(k \log^2 n)$ bits, where $n = |S| \leq |S\ AS_1 \ldots AS_N|/2$.*

Proof. Consider summing any two entropies (recall Eqs. (2.1) and (2.2)).

$$\begin{aligned}
&|AS_1|H_k(AS_1) + |AS_2|H_k(AS_2) = \\
&\quad = \textstyle\sum_{w\in\Sigma^k} |w_{AS_1}|H_0(w_{AS_1}) + \sum_{w\in\Sigma^k} |w_{AS_2}|H_0(w_{AS_2}) \\
&\quad \leq \textstyle\sum_{w\in\Sigma^k} \left(\log \binom{|w_{AS_1}|}{|n^{a_1}_{AS_1}|,|n^{a_2}_{AS_1}|,\ldots,|n^{a_\sigma}_{AS_1}|} + \log \binom{|w_{AS_2}|}{|n^{a_1}_{AS_2}|,|n^{a_2}_{AS_2}|,\ldots,|n^{a_\sigma}_{AS_2}|} \right) + O(\sigma^{k+1} \log n) \\
&\quad \leq \textstyle\sum_{w\in\Sigma^k} \log \binom{|w_{AS_1}|+|w_{AS_2}|}{|n^{a_1}_{AS_1}|+|n^{a_1}_{AS_2}|,|n^{a_2}_{AS_1}|+|n^{a_2}_{AS_2}|,\ldots,|n^{a_\sigma}_{AS_1}|+|n^{a_\sigma}_{AS_2}|} + O(\sigma^{k+1} \log n) \\
&\quad \leq |AS_1AS_2|H_k(AS_1AS_2) + O(\sigma^{k+1} \log n) + O(k \log n)
\end{aligned}$$

where $O(\sigma^{k+1} \log n)$ comes from the relationship between the zero-order entropy and the combinatorials, and $O(k \log n)$ comes from considering the symbols in the border between AS_1 and AS_2. Note that $\sigma^{k+1} \log n = o(n)$ if $k < (1-\epsilon)\log_\sigma n$. Then the lemma follows by adding up the $N \leq \log n$ EBDS's. □

Theorem 5.2. *The structure of Theorem 5.1 supports appending symbols in constant amortized time and retains the same space and query time complexities, being n the current length of the sequence.*

5.3 Application to Full-Text Indexing

In this section, we give some positive and negative results about the application of the technique to full-text indexing, as explained in the beginning of the chapter. We have a text $T[1, n]$ over alphabet Σ and wish to compress a transformed version X of T with our technique. Then, the question is how does $H_k(X)$ relate to $H_k(T)$.

5.3.1 The Burrows-Wheeler Transform

The Burrows-Wheeler Transform, $S = T^{bwt}$, is used by many compressed full-text self-indexes [FM05, FMMN07, MN05]. We have introduced it in Section 2.8.

We show that there is a relationship between the k-th order entropy of a text T and the first order entropy of $S = T^{bwt}$. For this sake, we will compress S with a first-order compressor, whose output size is an upper bound to $nH_1(S)$.

A *run* in S is a maximal substring formed by a single letter. Let $rl(S)$ be the number of runs in S. In [MN05] they prove that $rl(S) \leq nH_k(T) + \sigma^k$ for any k. Our first-order encoder exploits this property, as follows:

- If $i > 1$ and $s_i = s_{i-1}$ we output bit 0.
- Otherwise, we output bit 1 followed by s_i in plain form ($\log \sigma$ bits).

Thus, we encode each symbol of S by considering only its preceding symbol. The total number of bits is $n + rl(S)\log\sigma \leq n(1 + H_k(S)\log\sigma + \frac{\sigma^k \log\sigma}{n})$. The latter term is negligible for $k < (1-\epsilon)\log_\sigma n$, for any $0 < \epsilon < 1$. On the other hand, the total space obtained by our first-order encoder cannot be less than $nH_1(S)$. Thus, we get our result:

Lemma 5.3. *Let $S = T^{bwt}$, where $T[1,n]$ is a text over an alphabet of size σ. Then $H_1(S) \leq 1 + H_k(T)\log\sigma + o(1)$ for any $k < (1-\epsilon)\log_\sigma n$ and any constant $0 < \epsilon < 1$.*

We can improve this upper bound if we use Arithmetic encoding to encode the 0 and 1 bits that distinguish run heads. Their zero-order probability is $p = H_k(T) + \frac{\sigma^k}{n}$, thus we spend $-p\log p - (1-p)\log(1-p) \leq 1$ bits per symbol. Likewise, we can encode the run heads s_i up to their zero-order entropy. These improvements, however, do not translate into clean formulas.

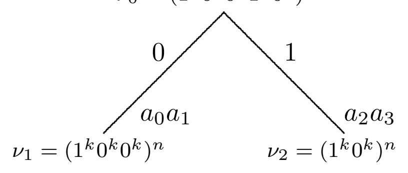
This shows, for example, that we can get (at least) about the same results of the Run-Length FM-Index (Section 3.3.1) by compressing T^{bwt} using our structure.

5.3.2 The Wavelet Tree

Several FM-Index variants [MN05, FMMN07] use wavelet trees to represent $S = T^{bwt}$, while others [GGV03] use them for other purposes. As explained in Section 2.4, $wt(S)$ is composed of several binary sequences. By compressing each such sequence B to $|B|H_0(B)$ bits, one achieves $nH_0(S)$ bits overall. The natural question is, thus, whether we can prove any bound on the overall space if we encode sequences B to $|B|H_k(B)$ bits using our technique. Let $nH_k(wt(S))$ be the space usage in bits that comes from encoding each binary sequence B to $|B|H_k(B)$ bits in the wavelet tree of S. We present next two negative examples.

- First, we show a case where $H_k(S) < H_k(wt(S))$. We choose $S = (a_3^k a_1^k a_0^k a_2^k a_0^k)^n$, then

$wt(S) =$

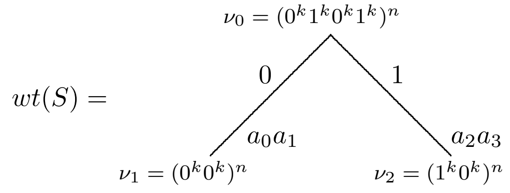


Let us compute $H_k(S)$ according to Section 2.1. Note that $H_0(w_S) = 0$ for all contexts except $w = a_0^k$, where $w_S = a_2(a_3a_2)^{n-1}\$$, being "\$" the sequence terminator. Thus $|w_S| = 2n$ and $H_0(w_S) = -\frac{n}{2n}\log\frac{n}{2n} - \frac{n-1}{2n}\log\frac{n-1}{2n} - \frac{1}{2n}\log\frac{1}{2n} = 1 + O(\frac{\log n}{n})$. Therefore $H_k(S) \simeq \frac{2}{5k}$.

On the other hand, $H_k(wt(S)) = \sum_{i=0}^{2} H_k(\nu_i) \simeq \underbrace{\frac{2}{5k}\log k}_{\nu_0} + \underbrace{\frac{1}{3k} + \frac{\log k}{3k}}_{\nu_1}$, as $H_k(\nu_2) \simeq 0$.

Therefore, in this case, $H_k(S) < H_k(wt(S))$, by a $\Theta(\log k)$ factor.

- Second, we show a case where $H_k(S) > H_k(wt(S))$. Now we choose $S = (a_0^k a_3^k a_0^k a_2^k)^n$, then



In this case, $H_k(S) \simeq \frac{2}{4k}$ and $H_k(wt(S)) = \sum_{i=0}^{2} H_k(\nu_i) = O(\frac{\log n}{n})$. Thus $H_k(S) > H_k(wt(S))$ by a factor of $\Theta(n/(k\log n))$.

Lemma 5.4. *The ratio between the k-th order entropy of the wavelet tree representation of a sequence S, $H_k(wt(S))$, and that of S itself, $H_k(S)$, can be at least $\Omega(\log k)$. More precisely, $H_k(wt(S))/H_k(S)$ can be $\Omega(\log k)$ and $H_k(S)/H_k(wt(S))$ can be $\Omega(n/(k\log n))$.*

What is most interesting is that $H_k(wt(S))$ can be $\Theta(\log k)$ times larger than $H_k(S)$. We have not been able to produce a larger gap. Whether $H_k(wt(S)) = O(H_k(S)\log k)$ remains open.

There are other results in this line of analysis. In [FGM06], they show that if we compress the wavelet tree bitmaps using run-length encoding and γ-encode the run lengths (Section 2.3), we obtain at most $4nH_0(S) + 2\sigma + 2\log n - 1$ bits of space. In [MN07], they show that if we compress T^{bwt} using a balanced wavelet tree where each bitmap is compressed using the structure presented in [RRR02] (Section 2.4), we obtain $nH_k(T) + o(n\log\sigma)$ bits of space for any $k \leq \alpha\log_\sigma(n) - 1$ and any constant $0 < \alpha < 1$.

Chapter 6

A Compressed Text Index on Secondary Memory

Compressed full-text self-indexing, as explained in Section 2.9, can run in main memory in cases where a traditional index would have to resort to secondary memory. In those situations, a compressed index is extremely attractive.

There are, however, cases where even the compressed index is too large to fit in main memory. One would still expect some benefit from compression in this case (apart from the obvious space savings). For example, sequentially searching a compressed text is much faster than a plain text, because fewer disk blocks must be scanned [ZMNBY00]. However, this has not been usually the case on indexed searching. The existing compressed text indexes for secondary memory are usually slower than their uncompressed counterparts, due to their poor locality of access.

The most relevant text indexes for secondary memory follow (refer to Section 2.6 and 2.9 for the description of the base structures). We have different ways to express the size of a disk block: $\bar{b}$ will be the number of bits, $b = \bar{b}/\log\sigma$ the number of symbols, and $\tilde{b} = \bar{b}/\log n$ the number of integers in a block.

- The String B-tree [FG99] is based on a combination between B-trees and Patricia tries. In this index $locate(P_{1,m})$ takes $O(\frac{m+occ}{\tilde{b}} + \log_{\tilde{b}} n)$ worst-case I/O operations. This time complexity is optimal, yet the string B-tree is not a compressed index. Its static version takes about 5–6 times the text size, plus text.
- The Compact Pat Tree (CPT) [CM96] represents a suffix tree in secondary memory in compact form. It does not provide theoretical space or time guarantees, but the index works well in practice, requiring 2–3 I/Os per query. Still, its size is 4–5 times the text size, plus text.

- The disk-based Suffix Array [BYBZ96] is a suffix array on disk plus some memory-resident structures that improve the cost of the search. The suffix array is divided into blocks of h elements, and for each block the first m symbols of its first suffix are stored. At best [1], it takes $4 + m/h$ times the text size, plus text, and needs $2(1 + \log h)$ I/Os for counting and $1 + \lceil (occ - 1)/\tilde{b} \rceil$ extra I/Os for locating. This is not yet a compressed index.

- The disk-based Compressed Suffix Array (CSA) [MNS04] adapts the CSA [Sad03] to secondary memory. It requires $n(H_0 + O(\log\log\sigma))$ bits of space. It takes $O(m \log_{\tilde{b}} n)$ I/O time for $count(P_{1,m})$. Locating requires $O(\log n)$ accesses per occurrence, which is too expensive.

- The disk-based LZ-Index [AN07] builds on the LZ-index [Nav04]. It uses $8nH_k(T) + o(n\log\sigma)$ bits, for any $k = o(\log_\sigma n)$. It does not provide theoretical bounds on time complexity, but it is quite competitive in practice.

- The disk-based Geometric Burrows-Wheeler Transform (GBWT) [CHSV08] uses $O(n\log\sigma)$ bits, with constant > 2. Locating takes $O(m/\tilde{b} + \log_\sigma n \log_{\tilde{b}} n + occ \log_{\tilde{b}} n)$ I/Os. There is no implementation nor simulations as far as we know.

- The LOF-SA index [SPMT08] is a suffix array enriched with longest common prefix (LCP) information, the characters that distinguish each pair of consecutive suffixes, and a truncated suffix tree in RAM that distinguishes suffixes in consecutive disk blocks. The whole structure requires more than $13n$ bytes, text included. It permits counting with at most $2 + \lceil (m-1)/\tilde{b} \rceil$ accesses to disk. Locating takes $O(occ/b)$ I/Os, with constant at least 2.

In this chapter, we present a practical self-index for secondary memory, which is built from three components: for *count*, we develop a novel secondary-memory version of backward searching; for *locate*, we adapt our LCSA (Chapter 4), and for *extract*, we adapt our EBDS that compresses sequences to k-th order entropy while retaining random access (Chapter 5). Depending on the available main memory, our data structure requires $2(m-1)$ to $4(m-1)$ accesses to disk for *count* in the worst case. It locates the occurrences in $1 + \lceil (occ-1)/\tilde{b} \rceil$ I/Os in the worst case, and on average in $1 + (cr \cdot occ - 1)/\tilde{b}$ I/Os, $0 < cr \leq 1$ being the *suffix array compression ratio* achieved: the compressed size divided by the original suffix array size. Similarly, the time to extract $T_{l,r}$ is at most $1 + \lceil (r-l)/b \rceil$ I/Os in the worst case. On average, this is $1 + (cs \cdot (r-l+1) - 1)/b$, $0 < cs \leq 1$ being the *text compression ratio* achieved: the compressed size divided by the original text size. With sufficient main memory, our index takes $O(H_k \log(1/H_k) n \log n)$ bits of space (see restrictions in the next section), which in practice can be up to 4 times smaller than classical suffix arrays. Thus, our index is the first

[1]Here we are assuming m is known at indexing time, which is rather optimistic. Times would be worse otherwise, and this is the meaning of "at best". Still, we refer to worst cases.

in being compressed and at the same time taking advantage of compression in secondary memory, as its *locate* and *extract* times are faster when the text is compressible. Counting time does not improve with compression but it is usually better than, for example, disk-based suffix arrays and CSAs. We show experimentally that our index is very competitive against the alternatives, offering a relevant space/time tradeoff when the text is compressible.

6.1 An Entropy-Compressed Rank Dictionary on Secondary Memory

As we will require several bitmaps in our structure with few bits set, we describe an entropy-compressed rank dictionary, suitable for secondary memory, to represent a binary sequence $B_{1,n}$. In case it fits in main memory, we use *BSGAP* (Section 2.4), which encodes the gaps between ones. Otherwise we will use *DEB*, a disk-based form of *BSGAP*: we δ-encode the gaps between consecutive 1's in B (Section 2.3). If s is the number of one-bits in $B_{1,n}$ then *DEB* uses at most $s \log \frac{n}{s} + 2s \log\log \frac{n}{s} + O(\log n)$ bits of space [MN07, Section 3.4.1]. We split *DEB* into blocks of at most $\bar{b}$ bits: if a δ-encoding spans two blocks, we move it to the next block. Each block is stored in secondary memory and, at the beginning of block i, we also store the number of 1's accumulated up to block $i-1$; we call this value OB_i. To access *DEB*, we use in main memory an array B^a, where $B^a[i]$ is the number of bits of B represented in blocks 1 to $i-1$. B^a uses $(s \log \frac{n}{s} + 2s \log\log \frac{n}{s} + O(\log n))\frac{\log n}{\bar{b}}$ bits of space.

To answer $rank_1(B,i)$ with this structure, we carry out the following steps: (1) We binary search B^a to find j such that $B^a[j] \leq i < B^a[j+1]$. (2) Our initial position is $p \leftarrow B^a[j]$ and our initial rank is $r \leftarrow OB_j$. (3) We read block j from disk. (4) We decompress the δ-encodings x in block j, adding x to p, and adding 1 to r if $p \leq i$. (5) We stop when $p \geq i$; $rank_1(B,i)$ will be r.

Overall this costs $O(\log \frac{s}{\bar{b}} + \log\log \frac{n}{s} + \bar{b})$ CPU time and just one disk access. When we use these structures in this chapter, s will be $\Theta(n/b)$ and the CPU time will be $O(\log \frac{s}{\bar{b}} + \bar{b})$. Table 6.1 shows some real sizes and times obtained for the structures, when $s = n/b$. As it can be seen, we require very little main memory for the second scheme. For moderate-size bitmaps even the *BSGAP* option is good, and gives $O(\log \frac{n}{b})$ CPU time.

6.2 An Entropy-Bounded Data Structure for Secondary Memory

We now modify our data structure presented in Chapter 5 to operate on secondary memory.

Structure	Space (bits)	CPU time for *rank*
BSGAP	$s\log\frac{n}{s} + s\frac{\log n}{\log s} + 2s\log\log\frac{n}{s} + O(\log n)$	$O(\log s)$
DEB+	$s\log\frac{n}{s} + 2s\log\log\frac{n}{s} + O(\log n)\ +$	$O(\log s + \bar{b}$
B^a	$(s\log\frac{n}{s} + 2s\log\log\frac{n}{s} + O(\log n))\frac{\log n}{b}$	$+\log\log\frac{n}{s})$

Structure	Real space if $s = n/b$			
	$n =$ 1 Tb	1 Gb	1 Gb	1 Mb
	$b =$ 32 KB	8 KB	4 KB	4 KB
BSGAP	100 MB	354 KB	667 KB	$<$ 1KB
DEB+	93 MB	326 KB	613 KB	$<$ 1KB
B^a	14 KB	$<$ 1KB	$<$ 1KB	$<$ 1KB

Table 6.1: Different sizes and times obtained to answer *rank*, for some relevant choices of n and b. *DEB* is stored in secondary memory and is accessed using B^a. B^a and *BSGAP* reside in main memory. Tb, Gb, etc. mean terabits, gigabits, etc. TB, GB, etc. mean terabytes, gigabytes, etc.

Structures maintained in main memory. We store in main memory the data generated by the modeler, that is, table T, which requires $\sigma^k\ \frac{\log n}{2}\ n^{1/2}$ bits. This restricts the maximum possible k to be used.

Structures in secondary memory. To store the structure in secondary memory, we split the sequence $U = \tilde{E}_0\tilde{E}_1\ldots\tilde{E}_{\lfloor n/\beta\rfloor}$ and W into disk blocks of $\bar{b}$ bits (thus the overhead over the entropy is $\frac{n}{\bar{b}}f_k(EN,\bar{b})$, where EN is the statistical encoder used and function $f_k(,)$ is as defined in Section 2.2). Also each block will contain the context C_j of order k of the first entry of U, $\tilde{E}_j$, stored in the disk block (using $k\log\sigma$ bits).

To know where a symbol of the sequence S is stored, we need a compressed rank dictionary ER (Section 6.1), which is built over a bitmap of length n, that has marked (with a one) the position in S of the first symbol of each disk block. This replaces tables R_g and R_l (Section 5.1.1). ER can be chosen to reside in main or in secondary memory, the latter choice requiring one more I/O access.

The algorithm to extract $S_{l,r}$ is: (1) Find the block $j = rank_1(ER,l)$ where S_l is stored. (2) Read block j and decompress it using T and the context of the first entry. (3) Continue reading and decompressing them until reaching S_r.

Using this scheme, we have at most $1+\lceil(r-l+1)/b\rceil$ I/O operations, which on average is $1+((r-l+1)H_k(S)-1)/\bar{b}$. We add one I/O operation if we use the secondary memory version of the rank dictionary. The total CPU time is $O(\frac{r-l}{\log_\sigma n}+\bar{b}+\log n)$. Term $\bar{b}$ can be removed by directly accessing inside the block. This requires maintaining in each disk block the R_l of each $\tilde{E}_i$ stored inside the block, which adds other $o(n \log \sigma)$ bits of space. We have assumed $\bar{b} = \omega(k \log \sigma)$ in this analysis for simplicity and practicality.

6.3 A Compressed Secondary Memory Structure

We introduce a structure on secondary memory which is able to answer *count*, *locate* and *extract* queries. It is composed of three substructures, each responsible for one type of query, and allows diverse trade-offs depending on how much main memory space they occupy.

6.3.1 Counting

We run the algorithm of Figure 2.1 (Section 2.9.1) to answer a counting query. Table C uses $\sigma \log n$ bits and easily fits in main memory, thus the problem is how to calculate $rank_c$ over T^{bwt}.

To calculate $rank_c(T^{bwt}, i)$, we need to know the number of occurrences of symbol c before each block on disk. To do so, we store a two-level structure: the first level stores for every t-th block the number of occurrences of every c from the beginning, and the second level stores the number of occurrences of every c from the last t-th block. The first level is maintained in main memory and the second level on disk, together with the representation of T^{bwt} (i.e., the entry of each block is stored within the block). Let K be the total number of blocks. We define:

- $E_c(j)$: number of occurrences of symbol c in blocks 0 to $(j-1)\cdot t$, with $E_c(0) = 0$, $1 \leq j < \lfloor K/t \rfloor$.
- $E'_c(j)$: j goes from 0 to $K-1$. For $j \bmod t = 0$ we have $E'_c(j) = 0$, and for the rest we have that $E'_c(j)$ is the number of occurrences of symbol c in blocks from $\lfloor j/t \rfloor \cdot t$ to $j-1$.

Now we can compute $rank_c(T^{bwt}, i) = E_c(j \text{ div } t) + E'_c(j) + rank_c(B_j, \textit{offset})$, where j is the block where i belongs, *offset* is the position of i within block j, and $rank_c(B_j, \textit{offset})$ is the number of occurrences of symbol c within block B_j up to *offset*. Now we present four ways to represent T^{bwt}, each with its pros and cons. This will give us four different ways to calculate j, *offset*, and $rank_c(B_j, \textit{offset})$.

Version 1. The simplest choice is to store T^{bwt} directly without any compression. As a disk block can store b symbols, we will have $K = \lceil n/b \rceil$ blocks. $rank_c(B_j, \textit{offset})$ is calculated by traversing the block and counting the occurrences of c up to *offset*. As the layout of blocks is regular, we know that that $T^{bwt}[i]$ belongs to block $j = \lfloor i/b \rfloor$, and $\textit{offset} = i - j \cdot b$.

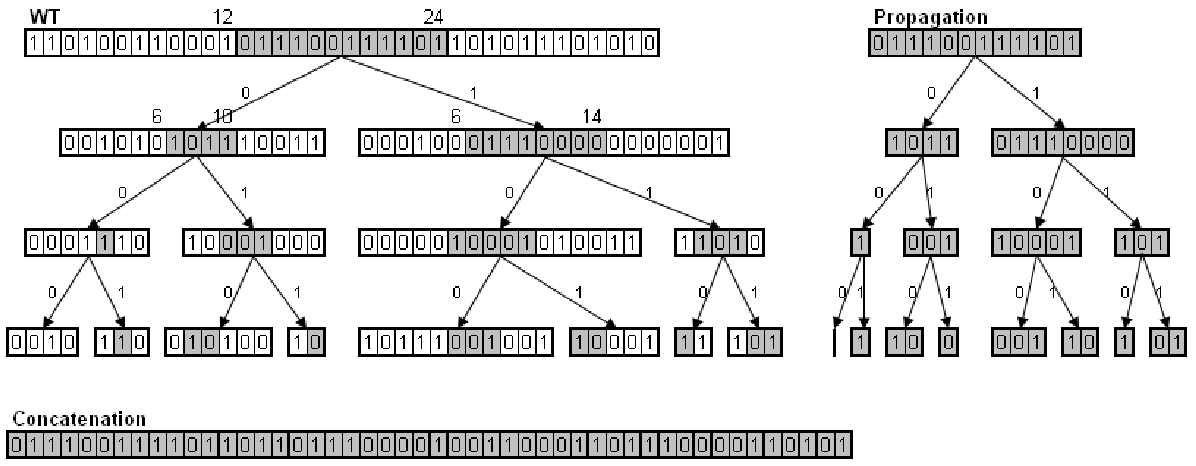


Figure 6.1: Block propagation over the wavelet tree WT. Making *ranks* over the first level of WT ($rank_0(12) = 6$, $rank_0(24) = 10$ and $rank_1(i) = i - rank_0(i)$), we determine the propagation over the second level of WT, and so on.

Version 2. We represent the T^{bwt} chunks with a wavelet tree (Section 2.4) to speed up the scanning of the block. We divide the first level of $WT = wt(T^{bwt})$ into blocks of b bits. Then, for each block, we gather its propagation over WT by concatenating the subsequences in breadth-first order, thus forming a sequence of $b \log \sigma$ bits (just like the plain storage of the chunk of T^{bwt}). In this case, the division of T^{bwt} is uniform and uncompressed, thus we can still easily determine j and *offset*. Figure 6.1 illustrates. Note that this propagation generates $2^{\ell-1}$ intervals at level ℓ of WT. Some definitions follow:

- B_i^ℓ: the i-th interval of level ℓ, with $1 \leq \ell \leq \lceil \log \sigma \rceil$ and $1 \leq i \leq 2^{\ell-1}$.
- L_i^ℓ: the length of interval B_i^ℓ.
- O_i^ℓ / Z_i^ℓ: the number of 1's/0's in interval B_i^ℓ.
- $D_\ell = B_1^\ell \ldots B_{2^{\ell-1}}^\ell$ with $1 \leq \ell \leq \lceil \log \sigma \rceil$: all concatenated intervals from level ℓ.
- $B = D_1 D_2 \ldots D_{\lceil \log \sigma \rceil}$: concatenation of all the D_ℓ, with $1 \leq \ell \leq \lceil \log \sigma \rceil$.

Some relationships hold: (1) $L_i^\ell = O_i^\ell + Z_i^\ell$. (2) $Z_i^\ell = rank_0(B_i^\ell, L_i^\ell)$. (3) $L_i^\ell = Z_{(i+1)/2}^{\ell-1}$ if i is odd (B_i^ℓ is a left child); $L_i^\ell = O_{i/2}^{\ell-1}$ otherwise. (4) $|D_\ell| = L_1^1 = b$ for $\ell < \lfloor \log \sigma \rfloor$, the last level can be different if σ is not a power of 2. With those properties, L_i^ℓ, O_i^ℓ and Z_i^ℓ are determined recursively from B and b. We only store B plus the structures to answer *rank* on it in constant time. Note that any $rank(B_i^\ell)$ is answered via two *ranks* over B.

Figure 6.2 shows how we calculate $rank_c$ in $O(\log \sigma)$ constant-time steps. Some precisions are in order:

```
**Algorithm** $rank_c(B, j)$
$node \leftarrow 1; ans \leftarrow j; des \leftarrow 0; B_1^1 = B[1, b];$
**for** $\ell \leftarrow 1$ **to** $\lceil \log \sigma \rceil$ **do**
    **if** $c$ belongs to the left subtree of $node$ **then**
        $ans \leftarrow rank_0(B_{node}^{\ell}, ans);$
        $len \leftarrow Z_{node}^{\ell};$
        $node \leftarrow 2 \cdot node - 1;$
    **else** $ans \leftarrow rank_1(B_{node}^{\ell}, ans);$
        $len \leftarrow O_{node}^{\ell};$   $des \leftarrow Z_{node}^{\ell};$
        $node \leftarrow 2 \cdot node;$
    $B_{node}^{\ell} = B[\ell \cdot b + des + 1, \ell \cdot b + des + len];$
**return** $ans;$
```

Figure 6.2: Algorithm to obtain the number of occurrences of c inside a disk block, for Version 2.

1. Block D_ℓ begins at bit $(\ell - 1) \cdot b + 1$ of B, and $|B| = b \log \sigma$.
2. To know where B_i^ℓ begins, we only need to add to the beginning of D_ℓ the length of $B_1^\ell, \ldots, B_{i-1}^\ell$. Each B_k^ℓ, with $1 \leq k \leq i - 1$, belongs to a left branch that we do not follow to reach B_i^ℓ from the root. So, when we descend through the wavelet tree to B_i^ℓ, every time we take a right branch we accumulate the number of bits of the left branch (zeroes of the parent).
3. *node* is the number of the current interval at the current ℓ.
4. We do not calculate B_{node}^ℓ, we just maintain its position within B.

The extra space on top of the $n \log \sigma$ bits is still $O(\frac{n \log \sigma \log \log n}{\log n}) = o(n \log \sigma)$, even if compression is local to the block. This is achieved by maintaining the block sizes of $\frac{1}{2} \log n$ bits in the *rank* structures (Section 2.4). The consequence is a small table of $O(\sqrt{n}$ polylog$(n))$ bits in main memory.

Version 3. We aim at compressing T^{bwt} so as to achieve k-th order compression of T. We compress the blocks B from Version 2 using the (c, o)-pair compression [RRR02] (Section 2.4). In this case, the division of T^{bwt} is not uniform; rather we add symbols from T^{bwt} to the disk block as long as its compressed WT fits in the block. By doing this, we compress T^{bwt} to at most $nH_k + \sigma^{k+1} \log n + o(n \log \sigma)$ bits for any k [MN07]. To calculate $rank_c(B, \mathit{offset})$, we apply the same algorithm of Version 2, but now the bitmap is not stored explicity. Constant time *ranks* on the bitmaps are supported by the compressed representation [RRR02].

As the block size is variable, determining j is not as simple as before. Compression ensures that there are at most $(n + o(n))/b$ blocks. We use a binary sequence $EB_{1,n}$ to mark where each block starts. Thus, the block of $T^{bwt}[i]$ is $j = rank_1(EB, i)$. We use an entropy-compressed rank dictionary (*BSGAP*, Section 2.4) for *EB*. If we need to use the *DEB* variant, we add up one more I/O per access to T^{bwt} (Section 6.1).

Version 4. We aim at compressing T^{bwt} directly without wavelet trees. We represent T^{bwt} with our entropy-bounded data structure on secondary memory (Section 6.2). Again, the division of T^{bwt} is not uniform, rather we add symbols from T^{bwt} to the disk block as long as its compressed T^{bwt} fits in the block. By doing this, we compress T^{bwt} to $nH_k(T^{bwt}) + o(n \log \sigma)$ bits for $k = o(\log_\sigma n)$. To calculate $rank_c(B, \mathit{offset})$, we decompress block B and apply the decoding algorithm presented in Section 6.2.

Space usage of E ***and*** E'***.*** In Versions 1 and 2, if we sum up all the entries, E uses $\lceil K/t \rceil \cdot \sigma \log n$ bits and E' uses $K\sigma \log \frac{t \cdot n}{K}$ bits. In Version 3, the numbering scheme [RRR02] (Section 2.4) has a compression limit $n/K \leq b \cdot \log n/(2 \log \log n)$. Thus, for Version 3, E' uses at most $K \cdot \sigma \log(t \cdot \frac{b \log n}{2 \log \log n})$ bits. In Version 4, there is no upper bound to how many original symbols can fit in a particular compressed block. To avoid an excessively large E', we can impose an artificial limit: if more than $b \log n$ symbols are compressed into a single disk block, we stop adding symbols there. This guarantees that $\log(t \cdot b \log n)$ bits are sufficient for each entry of E'. The growth in the compressed file we cause cannot be more than $\bar{b}$ bits per $b \log n$ symbols, that is, $O(\frac{n\bar{b}}{b \log n}) = o(n \log \sigma)$ bits overall.

Costs per call to $rank_c$***.*** In Versions 1 and 2, we pay one I/O per call to $rank_c$. In Versions 3 and 4, we pay one or two I/Os per call to $rank_c$. In Versions 1 and 4, we spend $O(b)$ CPU operations per call to $rank_c$. In Versions 2 and 3, this is reduced to $O(\log \sigma)$ per call to $rank_c$.

Table 6.2 shows the different sizes and times needed for our four versions. We added the times to do *rank* on the entropy-compressed bit arrays. Versions 3a and 4a use an in-memory rank dictionary *BSGAP* costing $O(\log \frac{n}{b})$ CPU time per access, while 3b and 4b use the *DEB* variant costing $O(\log \frac{n}{b} + \bar{b})$ CPU time (Section 6.1). The space complexity of Version 3 depends on $H_k(T)$ but Version 4 depends on $H_k(T^{bwt})$. Note that, as shown in Section 5.3, there is no obvious connection between $H_k(T)$ and $H_k(T^{bwt})$, except $H_1(T^{bwt}) \leq 1 + H_k(T) \log \sigma + o(1)$ for any $k < (1 - \epsilon) \log_\sigma n$ and any constant $0 < \epsilon < 1$. We have assumed $\bar{b} = \omega(\sigma \log(tb))$ (Versions 1 and 2) and $\bar{b} = \omega(\sigma \log(tb \log n))$ (Versions 3 and 4) for simplicity, otherwise E' must be stored separately and the disk accesses doubled.

Version	Main Memory	Secondary Memory	I/O	CPU
1	$\frac{n}{bt}\cdot \sigma \log n$	$n \log \sigma + \frac{n}{b}\cdot \sigma \log(t\cdot b)$	$2(m-1)$	$O(m\cdot b)$
2	$\frac{n}{bt}\cdot \sigma \log n + o(\sqrt{n}\log^2 n)$	$n \log \sigma(1+o(1)) + \frac{n}{b}\cdot \sigma \log(t\cdot b)$	$2(m-1)$	$O(m \log \sigma)$
3a	$\frac{n}{bt}\cdot \sigma \log n$ $+o(\sqrt{n}\log^2 n) + bsgap$	$nH_k(T) + o(n\log\sigma) + \sigma^{k+1}\log n$ $+\frac{n}{b}\cdot \sigma \log(t \cdot b \log n)$	$2(m-1)$	$O(m(\log\sigma + \log\frac{n}{b}))$
3b	$\frac{n}{bt}\cdot \sigma \log n$ $+o(\sqrt{n}\log^2 n) + gap\frac{\log n}{b}$	$nH_k(T) + o(n\log\sigma) + \sigma^{k+1}\log n$ $+gap + \frac{n}{b}\cdot \sigma \log(t \cdot b \log n)$	$4(m-1)$	$O(m(b + \log\frac{n}{b}))$
4a	$\frac{n}{bt}\cdot \sigma \log n$ $\sigma^k\sqrt{n}\frac{\log n}{2} + bsgap$	$nH_k(T^{bwt}) + o(n\log\sigma) + \sigma^{k+1}\log n$ $+\frac{n}{b}\cdot \sigma \log(t \cdot b \log n)$	$2(m-1)$	$O(m(b + \log\frac{n}{b}))$
4b	$\frac{n}{bt}\cdot \sigma \log n$ $\sigma^k\sqrt{n}\frac{\log n}{2} + gap\frac{\log n}{b}$	$nH_k(T^{bwt}) + o(n\log\sigma) + \sigma^{k+1}\log n$ $+gap + \frac{n}{b}\cdot \sigma \log(t \cdot b \log n)$	$4(m-1)$	$O(m(b + \log\frac{n}{b}))$

$gap = \frac{n}{b}(\log b + 2\log\log b) + O(\log n) = O(\frac{n}{b}\log n)$
$bsgap = \frac{n}{b}(\log b + \frac{\log n}{\log b} + 2\log\log b) + O(\log n) = O(\frac{n}{b}\log n)$.

Table 6.2: Different sizes and times obtained to answer $count(P_{1,m})$.

6.3.2 Locating

Our locating structure will be a variant of the LCSA (Chapter 4). The array C from LCSA (Section 2.10) will be split into disk blocks of $\tilde{b}$ integers. Also, we will store in each block the absolute value of the suffix array at the beginning of the block. To minimize the I/Os, the dictionary will be maintained in main memory. So we compress the differential suffix array until we reach the desired dictionary size. Finally, we need a compressed bitmap LB (Section 6.1) to mark the beginning of each disk block. LB is entropy-compressed and can reside in main or secondary memory.

For locating every match of a pattern $P_{1,m}$, we first use our counting substructure to obtain the interval $[sp, ep]$ of the suffix array of T (see Section 2.9.1). Then, we find the block where sp belongs to, $j = rank_1(LB, sp)$. Finally, we read the necessary blocks until we reach ep, decompressing them using the dictionary of the LCSA.

We define $occ = ep - sp + 1$ and $occ' = cr\cdot occ$, where $0 < cr \leq 1$ is the compression ratio of the LCSA (more precisely, of the C sequence). This process takes, without counting, $1 + \lceil (occ-1)/\tilde{b} \rceil$ I/O accesses, plus one if we store LB in secondary memory. This I/O cost is optimal and on average improves, thanks to compression, to $1 + (occ'-1)/\tilde{b}$. We perform $O(occ + \tilde{b})$ CPU operations to decompress the LCSA interval.

6.3.3 Extracting

To extract arbitrary portions of the text, we store T in compressed form using the variant of our entropy-bounded data structure for secondary memory, see Section 6.2.

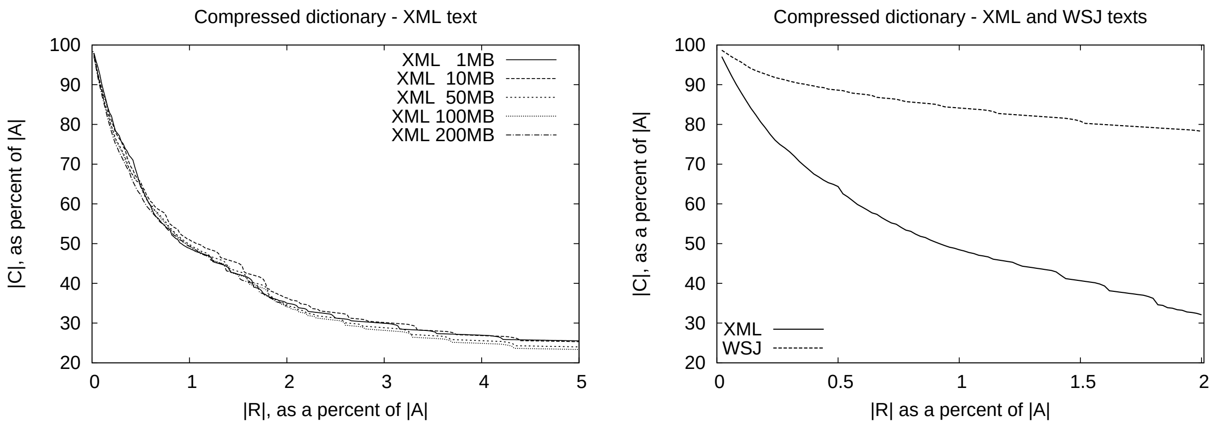


Figure 6.3: On the left, compression ratio achieved on XML (for different lengths) as a function of the percentage allowed to the dictionary (R). Both are percentages over the size of A. On the right, the different texts.

6.4 Experiments

We consider two text files for the experiments: the text WSJ (Wall Street Journal) from the TREC collection from year 1987, of size 126 MB, and the 200 MB XML file provided in *Pizza&Chili*. We searched for 5,000 random patterns, of length from 5 to 50, generated from these files. As in previous work [FG96, AN07], we assume a disk page size of 32 KB. We first study the compressibility we achieve as a function of the size of the compressed dictionary, as it must reside in RAM. Let $|R|$ be the size of the structures obtained from compressing R (Section 4.1.4). Recall that A is the suffix array. Figure 6.3 (left) shows that the compressibility depends on the percentage $|R|/|A|$ and not on the absolute size $|R|$. Figure 6.3 (right) shows the relation between $|R|/|A|$ and $|C|/|A|$ for the texts used in the experiments. In the following, we let our dictionary use 2% of the suffix array size $|R|/|A| = 2\%$. For counting, we use Version 1 (Section 6.3.1) with $t = \log n$, and *BSGAP* for the *LB* locating structure (Section 6.3.2). With this setting our index uses 16.13 MB of RAM for XML ($\sigma = 97$), and 10.58 MB for WSJ ($\sigma = 91$), for LB, R, and E_c. It compresses the suffix array of XML to 34.30% and that of WSJ to 80.28% of its original size.

We compared our results against String B-tree [FG99], Compact Pat Tree (CPT) [CM96], disk-based Suffix Array (SA) [BYBZ96] and disk-based LZ-Index [AN07]. We add our results to those of [AN07, Section 4]. We omit the disk-based CSA [MNS04] and the disk-based GBWT [CHSV08] as they are not implemented (even for simulations), but also because they can be predicted to be strictly worse than ours in these experiments. We also omit the LOF-SA index [SPMT08] because it largely exceeds our range of space consumption of interest, and it would not be competitive for locating for the same reason (fewer entries fit in a disk block). For counting it would need usually less than 4 accesses to disk.

We add our results to those of [AN07, Section 4]. Albeit our RAM usage is moderate, other data structures [FG99, CM96, AN07] can operate with a (small) constant number of disk blocks in main memory. We now consider the impact of giving these other structures the same amount of RAM we use, using it in the most reasonable (obvious) way we can devise. For the String B-tree, it was shown [FG96] that the arity of the tree is $b/12.25$ for the static version. For our page size $b = 8192$ integers, one would need less than 21 MB of RAM to hold the first tree level, thus we will subtract one disk access from the results given in [AN07]. For the CPT, we have that the pages are formed mostly by pointers to children and are filled to about 50% [CM96]. Thus one would need near 100 MB to fit the first level in RAM. The effect of fitting as much as possible in, say, 20 MB of RAM, is negligible and thus we have not changed the results used in [AN07]. For the disk-based LZ-index, in both texts one could store one level of $LZTrie$ and $RevTrie$ in about 12 MB of RAM. Yet, this time the top-down tree traversal is just a part of the total number of accesses, as there are also many direct accesses to the tries. As the potential benefit is hard to predict and implementations taking advantage of main memory do not exist yet[2], we do not change the results of [AN07]. Finally, the disk-based SA needs to hold the extra nm/h bytes in RAM, and thus we have extended the range studied in [AN07] to include the point where nm/h is as small as our RAM usage.

Figure 6.4 shows counting experiments (GN-index being ours). Our structure needs at most $2(m-1)$ disk accesses, but usually less as both ends of the suffix array interval tend to fall within the same disk block as the counting progresses. We present our index with and without the substructures for locating. It can be seen that our structure is extremely competitive for counting, being much smaller and/or faster than all the alternatives.

Figure 6.5 shows locating experiments. This time our structure grows due to the inclusion of the LCSA. Note that, for $m = 5$, we are able to report *more* occurrences than those the block could store in raw format. This time the competitiveness of our structure depends a lot on the compressibility of the text. In the highly-compressible XML, our index occupies a very relevant niche in the tradeoff curves, whereas in WSJ it is subsumed by String B-trees.

We have used texts up to 200 MB, but our results show that the compression ratio stays similar if we maintain a fixed percentage for the dictionary size (Figure 6.3 (left)), that the counting cost is at most $2(m-1)$, and that the locating cost depends on the number of occurrences of P and on the LCSA compression ratio. Thus it is very easy to predict other scenarios.

6.5 LCSA Construction in Secondary Memory

Particularly for the application described in Section 6.3.2, where the LCSA does not fit in main memory, a natural question is whether is it possible to efficiently build it in secondary

[2]D. Arroyuelo. Personal communication.

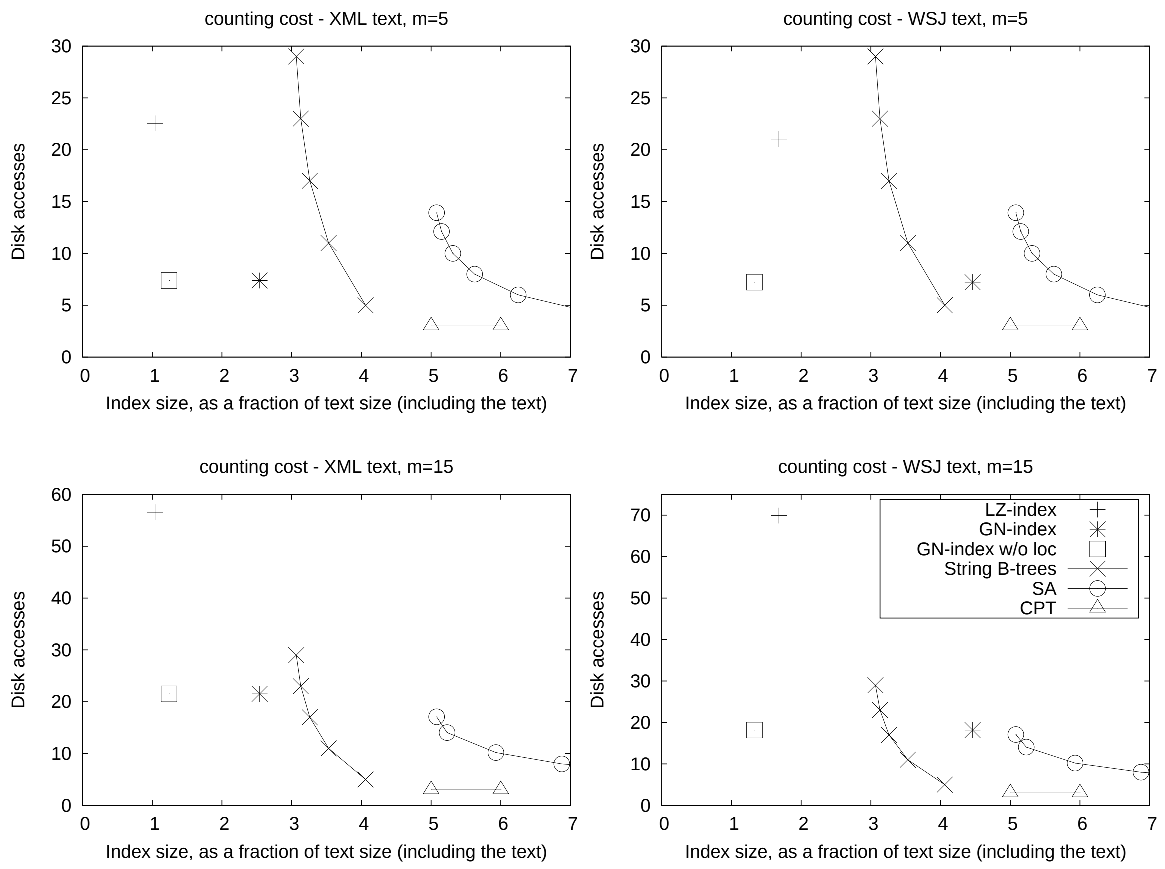


Figure 6.4: Counting cost vs. space requirement for the different texts and indexes tested, lower is faster. Recall that m is the pattern length.

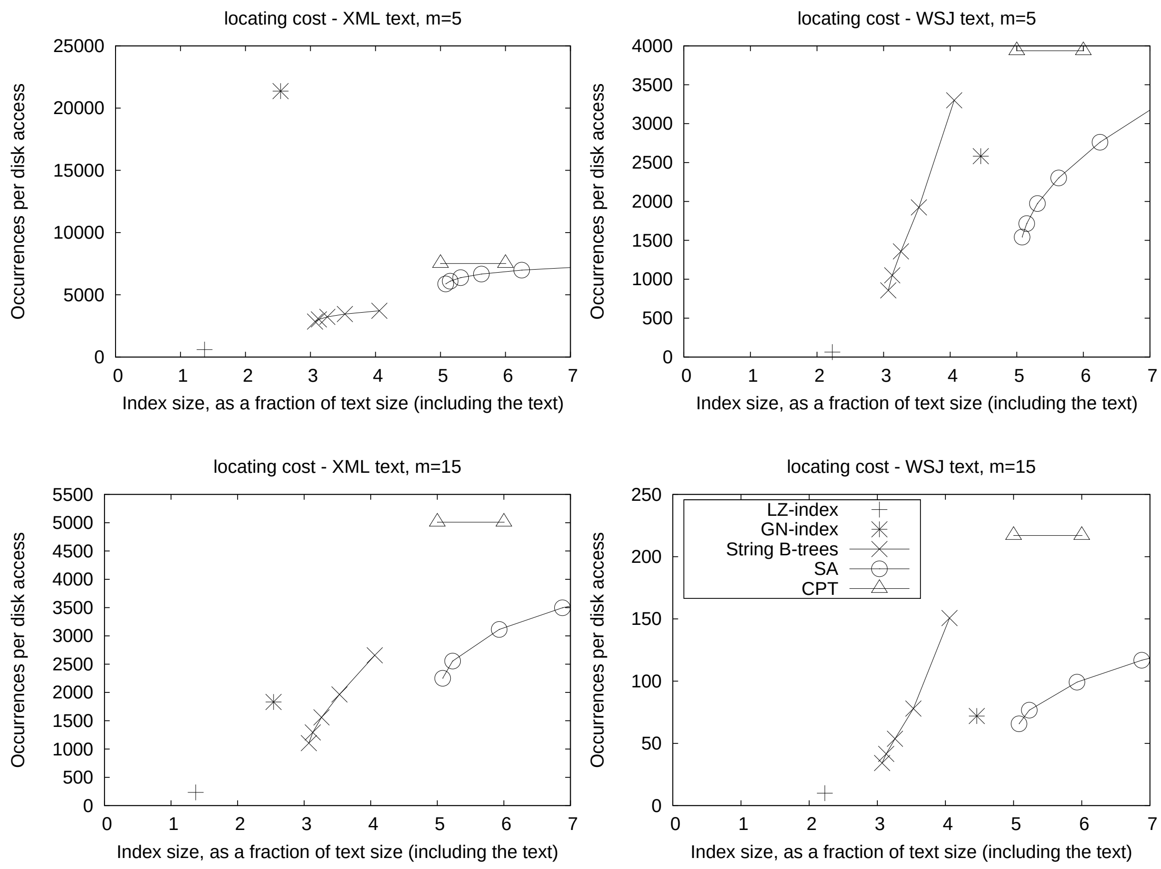


Figure 6.5: Locating cost vs. space requirement for the different texts and indexes tested, higher is faster. Recall that m is the pattern length.

memory. Secondary memory algorithms to build the suffix array A are well-known [KR03, CF02, DKMS05], yet the algorithms we have presented in Section 4.1 for compressing A' are highly non-local. We now show that the algorithms to compress A' by using Ψ can be adapted efficiently to secondary memory, and also show how to compress the dictionary in secondary memory.

6.5.1 Compressing the Differential Suffix Array

When we compress in main memory by using Ψ (Sections 4.1.2 and 4.1.3), we notice that Ψ traverses the suffix array in increasing values of $A[\cdot]$. That is, if j is the position where $A[j] = 1$, then $A[\Psi[j]] = 2$, $A[\Psi[\Psi[j]]] = 3$ and so on. The idea is to store for each position i of A' the information that permits us to compress $A'[i]$ after we sort it by increasing values of A, that is, by text order. For each position in A' we define:

- $A'[i] = A[i] - A[i-1], 1 < i \leq n; A'[1] = A[1]$. This is the differential form of A.
- $A''[i] = A'[i+1], 1 < i < n$; $A''[n] = \perp$ denotes an invalid value.
- $V'[i] = A[i], 1 \leq i \leq n$. We will use this array to sort the rest.
- $NV'[i] = A[i+1], 1 \leq i < n$; $NV'[n] = \perp$. This is the position of the next valid symbol of $A'[i]$ after sorting.
- $N^2V'[i] = A[i+2], 1 \leq i < n-1$; $NV'[n-1] = NV'[n] = \perp$. This is the position of the next-next valid symbol of $A'[i]$ after sorting.
- $PV'[i] = A[i-1], 1 < i \leq n$; $PV'[1] = \perp$, the position of the previous valid symbol of $A'[i]$ after sorting.

Now we sort $\{A'[i], A''[i], V'[i], NV'[i], N^2V'[i], PV'[i]\}_{1 \leq i \leq n}$ by the values of $V' = A$. Given that A is a permutation of $\{1, \ldots, n\}$, the effect of the sorting is that of composing the arrays with A^{-1}. We call the reordered arrays as follows:

- $\tilde{A}'[j] = A'[A^{-1}[j]] = A[A^{-1}[j]] - A[A^{-1}[j]-1], 1 \leq j \leq n, j \neq A[1]; \tilde{A}'[A[1]] = A[1]$.
- $\tilde{A}''[j] = A''[A^{-1}[j]] = A[A^{-1}[j]+1] - A[A^{-1}[j]], 1 \leq j \leq n, j \neq A[n]; \tilde{A}''[A[n]] = \perp$.
- $V[j] = V'[A^{-1}[j]] = A[A^{-1}[j]] = j, 1 \leq j \leq n$.
- $NV[j] = NV'[A^{-1}[j]] = A[A^{-1}[j]+1], 1 \leq j \leq n, j \neq A[n]; NV[A[n]] = \perp$.
- $N^2V[j] = N^2V'[A^{-1}[j]] = A[A^{-1}[j]+2], 1 \leq j \leq n, j \neq A[n], j \neq A[n-1]; NV[A[n-1]] = NV[A[n]] = \perp$.

- $PV[j] = PV'[A^{-1}[j]] = A[A^{-1}[j]-1], 1 \leq j \leq n, j \neq A[1]; PV[A[1]] = \perp$.

Now that we have the arrays $\{V[j], \tilde{A}'[j], \tilde{A}''[j], NV[j], N^2V[j], PV[j]\}_{1\leq j\leq n}$, the sequential traversal of these arrays is equivalent to navigating the original ones using Ψ. More precisely, let $j = A[i]$ (and thus $i = A^{-1}[j]$), then $\tilde{A}'[j] = A'[i]$ and $\tilde{A}''[j] = A''[i] = A'[i+1]$. Moreover, $\tilde{A}'[j+1] = A'[\Psi(i)]$ and $\tilde{A}''[j+1] = A'[\Psi(i)+1]$. Hence, the check for pair equality between $A'[i]A'[i+1]$ and $A'[\Psi(i)]A'[\Psi(i)+1]$ reduces to checking whether $\tilde{A}'[j]\tilde{A}''[j] = \tilde{A}'[j+1]\tilde{A}''[j+1]$, which can be carried out sequentially on $\tilde{A}'$ and $\tilde{A}''$.

The other arrays are used to maintain consistency upon changes in A': When we change $\tilde{A}'[j]$ and $\tilde{A}''[j]$, corresponding to $A'[i]$ and $A'[i+1]$, we have the problem that the pair $A'[i-1]A'[i]$, which is explicitly stored at $\tilde{A}'[PV[j]]$ and $\tilde{A}''[PV[j]]$, must be updated as well. Similarly, NV serves to locate the place where the pair corresponding to $A'[i+1]A'[i+2]$ is stored after the sorting. Thus, as the arrays are now sorted by $j = A[i]$, arrays PV and NV serve as a doubly-linked list to let us move to $i-1$ and $i+1$ in the sorted array. Those lists must be updated upon removals across the compression process, and they are also useful to maintain the current values of $\tilde{A}''[j]$ up to date when elements in the chain are removed.

Let τ be the total size in integers of these arrays. We divide them into l chunks of size τ/l and, for each chunk, we keep in main memory a buffer of $\tilde{b}$ integers. Let M be the size in integers of the main memory, then $\tau/l + l\cdot\tilde{b} \leq M$ must hold (we consider later the case where M is smaller). The algorithm to carry out a single pass on A' in secondary memory is as follows. Note that this is the main subroutine of both approximate methods based on Ψ (Sections 4.1.2 and 4.1.3).

1. We read the first chunk from disk and initialize empty buffers.
2. We find sequentially in the chunk the first j satisfying $\tilde{A}'[j]\tilde{A}''[j] = \tilde{A}'[j+1]\tilde{A}''[j+1]$. If no such j is found we go on with the next chunk. In the stronger approximate method, we require equality between several $\tilde{A}'[j+r]\tilde{A}''[j+r]$ pairs before proceeding to the next step.
3. From that j, we start a chain of replacements: We add a new pair $s \leftarrow \tilde{A}'[j]\tilde{A}''[j]$ to R, make the replacements at j and $j+1$ and move on with $j \leftarrow j+1$, replacing until the pair changes. When the pair changes, that is $\tilde{A}'[j]\tilde{A}''[j] \neq \tilde{A}'[j+1]\tilde{A}''[j+1]$, we restart the search for pairs at Step (2).
4. If we reach the end of the block, the replacement chain may continue at the next one.

To consistently perform a replacement $\tilde{A}'[j] : \tilde{A}''[j] \leftarrow s : \perp$ in Step (3), maintaining also the linked lists, we must carry out the following actions (in parallel; we overline variables to indicate that we use their original values prior to any assignment): (a) $\tilde{A}'[j] \leftarrow s$, (b) $\tilde{A}'[\overline{NV}[j]] \leftarrow \perp$, (c) $\tilde{A}''[j] \leftarrow \tilde{A}''[\overline{NV}[j]]$, (d) $NV[j] \leftarrow \overline{N^2V}[j]$, (e) $PV[\overline{N^2V}[j]] \leftarrow j$, (f)

$N^2V[j] \leftarrow N^2V[\overline{NV}[j]]$, (g) $\tilde{A}''[PV[j]] \leftarrow s$, and (h) $N^2V[PV[j]] \leftarrow \overline{N^2V}[j]$. From those, only (a) and (d) can be executed locally, whereas the others may require reading/writing data from/to other chunks not yet in main memory. If this is the case, we "send messages" to read/update other chunks. Those will be stored in their corresponding buffers, and carried out right before those chunks are processed. (If a buffer gets full, it is written out to disk into a log of actions the chunk must execute.) Some of those messages will then send messages back to the current chunk to update its values, and this update will be executed when the current chunk is read again. This is not a problem because we will not access cell j again until the next pass. (The updates that happen to belong to the current chunk, instead, must be executed immediately.)

Each message is of the form $action(dest, parameters)$, where $dest$ is the destination position that determines the chunk $\lceil dest/l \rceil$ that will execute it; see Table 6.3 for the meaning of each action. The global instructions we described are then translated into the following instructions and messages sent: (a) $\tilde{A}'[j] \leftarrow s$, (b) send $DL(NV[j], j)$ (will solve (c) and (f) in the next pass), (d) $NV[j] \leftarrow N^2V[j]$, (e) send $UP(N^2V[j], j)$, (g) send $UA''(PV[j], s)$, and (h) send $UN^2(PV[j], N^2V[j])$. Figure 6.6 shows the operations that are performed after a replacement; this occurs in three steps. Thus, at the end of the pass, we must carry out two extra passes to process the messages sent and their responses, before finishing the pass properly.

action	parameter	what it does at position $dest$
UA'	sym	Updates $\tilde{A}'$ to sym, $\tilde{A}'[dest] \leftarrow sym$.
UA''	sym	Updates $\tilde{A}''$ to sym, $\tilde{A}''[dest] \leftarrow sym$.
UN^2	$next$	Updates N^2V to $next$, $N^2V[dest] \leftarrow next$.
DL	$from$	Marks $dest$ as deleted and responds with
		$UA''(from, \tilde{A}''[dest])$ and $UN^2(from, N^2V[dest])$
UP	$prev$	Updates PV to $prev$, $PV[dest] \leftarrow prev$.

Table 6.3: Message types and meanings used by the secondary memory construction algorithm.

The invalid entries we produce must be compacted for the next passes. Apart from removing the invalid entries, we must update the pointers NV, N^2V, and PV. We first sort all the arrays by the NV values. The result will be an increasing sequence of $NV[i]$ values, with some missing integers due to the removed entries. Thus, we assign $NV[i] \leftarrow i$ to effectively remove the invalid entries. We repeat the process of sorting and reassigning for N^2V and PV. Finally, we sort again by V, and are ready for the next pass.

Thus, each pass of the original algorithm over an array of size n' costs us, in I/O terms, $O(n'/\tilde{b})$ for the traversal plus $O((n'/\tilde{b}) \log_{M/\tilde{b}}(n'/M)) = O(Sort(n'))$ for the sortings. It follows that, because the n's are of the form $(1-\alpha)^i n$, the linear-time algorithm of

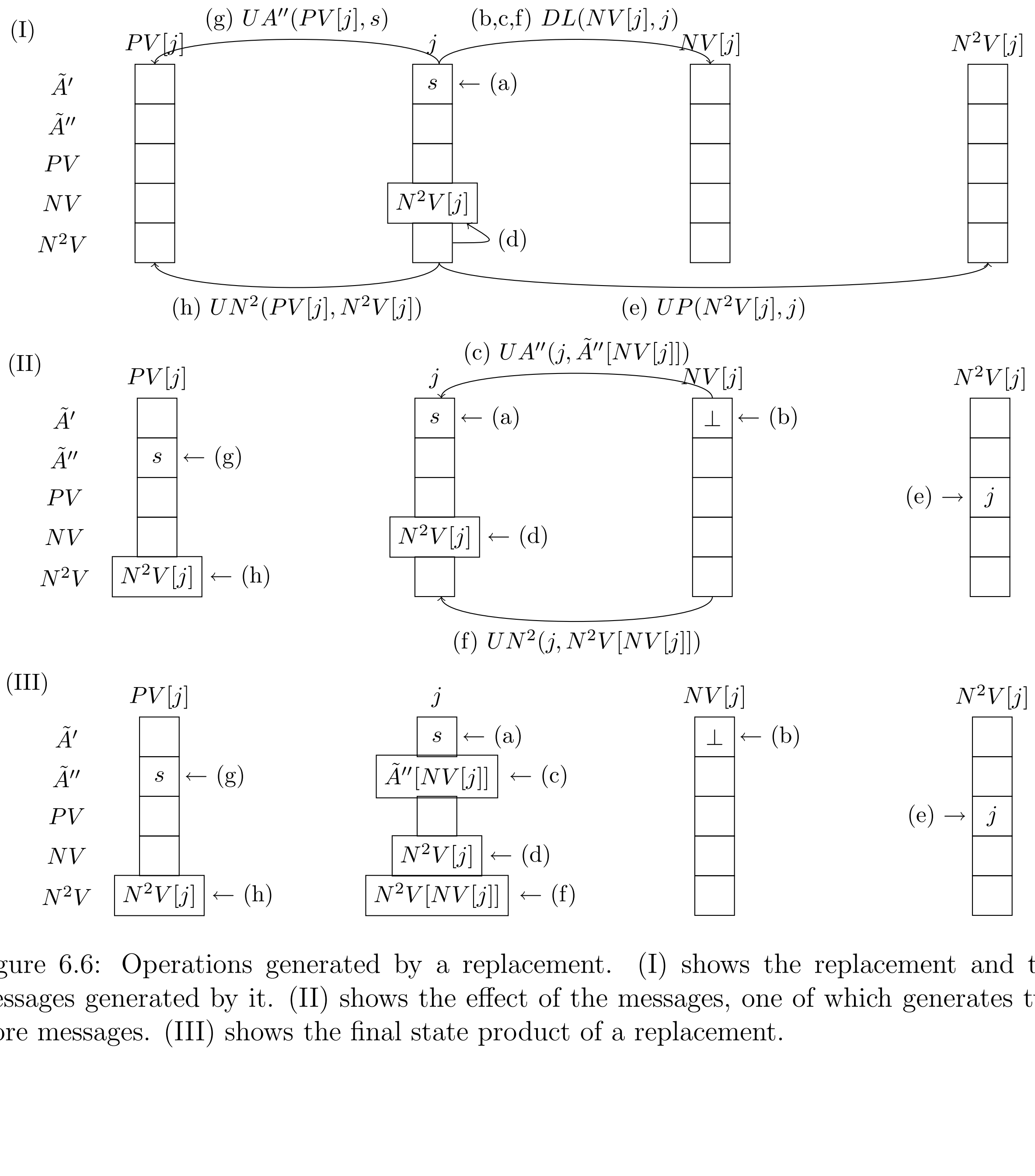


Figure 6.6: Operations generated by a replacement. (I) shows the replacement and the messages generated by it. (II) shows the effect of the messages, one of which generates two more messages. (III) shows the final state product of a replacement.

Section 4.1.2 costs $O(Sort(n))$ in secondary memory. Similarly, the $O(n \log n)$ time algorithm of Section 4.1.3 costs $O(Sort(n) \log n)$ time. This is almost I/O optimal with respect to the original algorithms.

As a practical note, Figure 4.3 shows that indeed 8 passes are sufficient, even on `xml`, to achieve most of the compression on the variants we tried.

With respect to the extra secondary memory needed, it is $O(n)$ words for the arrays, plus the message logs. Since each position can receive $O(1)$ messages from elsewhere, all the message logs add up to $O(n)$ words as well.

Finally, we consider the case $\tau/l + l \cdot \tilde{b} > M$, that is, there is no space to store one disk page per chunk in main memory. In this case, we replace the in-memory buffers by a secondary-memory priority queue, where the messages are inserted with priority given by *dest*. Those sent from a chunk i to a position $dest < i$ will be inserted with priority $n + dest$, so that they are processed in a second traversal. Each new chunk reads from the priority queue (by means of extracting minima) all the messages that correspond to it, and inserts new messages. As there are overall $O(n)$ messages, optimal-I/O priority queues (e.g., [BCFM00]) yield $O((n'/\tilde{b}) \log_{M/\tilde{b}}(n'/\tilde{b}))$ overall time, which raises only slightly the $O(Sort(n'))$ time per pass we had.

6.5.2 Compressing the Dictionary

When we compress the dictionary in main memory, we start expanding the first (i.e., earliest) rule s_j that has not been used by another one, $U[j] = 0$ (see Section 4.1.4). Then, we expand the next rule $s_{j'}$ that has $U[j'] = 0$, and so on. The idea here is to group together all the rules that are used in the same expansion of a rule with value $U[\cdot] = 0$, so that when we expand it we will have almost all the information needed to do the expansion in main memory.

We can regard the dictionary as $R = \{s_1 \rightarrow a_1 b_1, s_2 \rightarrow a_2 b_2, \ldots, s_\nu \rightarrow a_\nu b_\nu\}$, where $s_i = i + n$. For each rule we define:

- $R_0[i] = s_i$, the value of the i-th rule.
- $R_1[i] = a_i$, the left symbol of the i-th rule. It can be a rule or an original symbol.
- $R_2[i] = b_i$, the right symbol of the i-th rule. It can be a rule or an original symbol.
- $Q[i] = \begin{cases} \min\{s_j, s_i \text{ is used by } s_j \wedge U[j] = 0\} \text{ if } U[i] = 1, \\ s_i \text{ otherwise.} \end{cases}$
  The min is the lowest (i.e., earliest) rule that contains s_i and has value $U[\cdot] = 0$. Rules with the same value of Q will be used in the same expansion, so Q will be a kind of group identifier.

Given the non-locality of reference between rules, to calculate the values of Q we need to send messages of the form $UQ(dest, parameter)$, to achieve $Q[dest] \leftarrow parameter$, with $1 \leq dest, parameter \leq \nu$. Note that a position $dest$ could receive multiple messages and that always $dest < parameter$.

We maintain a secondary-memory priority queue PQ, where the messages are inserted with priority given by $dest$ (larger first). We process the arrays from higher values of R_0 first, i.e., in reverse order. To process cell i, we extract from PQ all the messages for $dest = i$ and set $Q[i]$ to the minimum $parameter$ for that i. If there are no messages for i, we assign a new $Q[i] \leftarrow s_i$ (i.e., we do not need to store U). Then, we insert message $UQ(R_1[i], Q[j])$ ($UQ(R_2[i], Q[j])$) into PQ, if $R_1[i]$ ($R_2[i]$) is not an original symbol. Since any rule $R_0[j]$ that uses rule $R_0[i]$ has been visited before we reach position i, necessarily the message $UQ(R_0[i], Q[j])$ is in PQ by that time. Thus, we compute array Q in one pass.

Now, we sort these arrays by increasing values of the pair $(Q[i], R_0[i])$, i.e., by Q and using R_0 to break ties, obtaining $\tilde{Q}, \tilde{R}_0, \tilde{R}_1, \tilde{R}_2$. These arrays can be partitioned into η groups, each one with the same value of Q. Let η_k be the position where the k-th group finishes. Note that if the length of the longest phrase is μ then any group has at most μ elements. Now, for each group, we compress the dictionary almost the same way as in algorithm Expand_Rule (see Figure 4.2, page 63). There are four differences with the original algorithm:

1. We write down R_B and R_S to disk instead of maintaining them in main memory. The array NV is not used.
2. We expand $\tilde{R}_0[\eta_k]$, the last element of the k-th group, because by construction these rules use all the other rules in the same group.
3. To process rule $\tilde{R}_0[i]$, which is expanded to $\tilde{R}_1[i]\tilde{R}_2[i]$, we do as follows. If $\tilde{R}_1[i]$ is not an original symbol, and it does not belong to the same group of $\tilde{R}_0[i]$, it must have been defined in a previous group. Hence, we must not expand it further, but instead write its final value in R_S. Yet this value is only know within the other group. We send message $NR_S(pos, \tilde{R}_1[i])$, where pos is the current position where we are writing in R_S. The same goes for $\tilde{R}_2[i]$.
4. We also write down to disk the pair $(s_j, LR_B[s_j])$ (second line of algorithm Expand_Rule), where s_j is the rule and $LR_B[s_j]$ is its final value (i.e., the current value of variable LR_B).

After carrying out the previous steps, we still need to execute the messages NR_S to update R_S. We sort the messages by $dest$ (their first component) obtaining $\overline{NR_S} = (\overline{dest}, \overline{parameter})$, and sort the pairs $(s_j, LR_B[s_j])$ by its first coordinate, obtaining $(\overline{s}_j, \overline{LR_B})$. Because the compressed dictionary will hold at least ν integers in RAM, we can use that RAM space across the process. From now on, $\overline{LR_B}$ will reside completely in main memory, so we can access it at random. To apply the messages, we traverse $\overline{NR_S}$ and R_S making the necessary replacements in R_S, that is, $R_S[\overline{dest}[i]] = \overline{LR_B}[\overline{parameter}[i] - n]$. These replacements are done by increasing values of $\overline{dest}$, so we traverse only once the arrays $\overline{NR_S}$ and R_S. Using again $\overline{LR_B}$, we traverse C, changing the values of the rules to their final ones.

Let M be the size in integers of the main memory. The breakdown of the cost is as follows:

- When we calculate Q there are overall $O(\nu)$ messages. Using optimal-I/O priority queues (e.g., [BCFM00]) yields $O((\nu/\tilde{b}) \log_{M/\tilde{b}}(\nu/\tilde{b}))$ time.
- When we expand a rule we need to find both children. Each of these searches within their group takes at most $\log \mu$ CPU time. Overall, there are at most 2ν of these searches, totalizing $O(\nu \log \mu)$ CPU time.
- We take $O(\nu/\tilde{b})$ I/Os to read/write all the needed arrays from disk.
- There are three sortings, which in total take $O(Sort(\nu))$ time.
- Updating C to the new rule values takes $O(n'/\tilde{b})$ I/Os .

Overall, time is $O((\nu/\tilde{b}) \log_{M/\tilde{b}}(\nu/\tilde{b}) + n'/\tilde{b})$ I/Os. The extra space on disk is $O(\nu)$ integers. Note that $4\mu \leq M$ must hold to be able to compress a group in main memory.

If $\overline{LR_B}$ does not fit in main memory, we pre-process the messages first, that is, we first sort NR_S by *parameter*, then we traverse it in synchronization with LR_B updating $parameter \leftarrow LR_B(parameter)$, and then we sort again NR_S by *dest*. Now we only need to traverse R_S and NR_S together, to update R_S. This add an extra $O(Sort(\nu))$ time to the cost, which does not affect our complexity. Something similar can be done to update C, which incurs an extra cost of $O(Sort(n'))$. This would affect the complexity, but is still within the cost paid in Section 6.5.1 to build C.

Chapter 7

Rank/Select on Dynamic Compressed Sequences

In this chapter, we are interested in the case where the collections can be updated via insertions and deletions of symbols, while maintaining *rank* and *select* capabilities. Two current solutions stand out as the best in the tradeoff of space versus time (when considering all the operations). One solution, by Mäkinen and Navarro [MN08], achieves compressed space (i.e., $nH_0 + o(n \log \sigma)$ bits) and $O(\log n \log \sigma)$ worst-case time for all the operations. The other solution, by Lee and Park [LP07], achieves $O(\log n(1 + \frac{\log \sigma}{\log \log n}))$ amortized time and uncompressed space, i.e. $n \log \sigma + O(n) + o(n \log \sigma)$ bits.

In this chapter, we show that the best of both worlds can be achieved. We combine the solutions to obtain $nH_0 + o(n \log \sigma)$ bits of space and $O(\log n(1 + \frac{\log \sigma}{\log \log n}))$ worst-case time for all the operations. Apart from the best current solution to the problem, we obtain several byproducts of independent interest applicable to partial sums, text indexes, suffix arrays, the Burrows-Wheeler transform, and others.

We remind that there is a static sequence representation [GMR06] that requires $n \log \sigma + n\, o(\log \sigma)$ bits and answers the queries in $O(\log \log \sigma)$ time. There has been work on dynamizing this structure [GHSV07b], where they achieve the same space plus $o(n)$ bits, the query times are increased by $O(\frac{1}{\epsilon} \log \log n)$, and the update times are $O(\frac{1}{\epsilon} n^{\epsilon})$ amortized, for any constant $0 < \epsilon < 1$. In fact, the method can be used to dynamize any other scheme (such as the wavelet-tree-based ones [FMMN07]), at the same extra cost. This is extremely relevant when there are more queries than updates. In this chapter, we focus on achieving the best time *for all the operations*. In particular, this is crucial when using the scheme to achieve good construction times within compressed space.

In Section 7.1, we describe a solution to handle a collection of several synchronized partial sums. This is used in Section 7.2 to design a dynamic *rank/select* solution for small alphabets ($O(\log n)$) with no compression. In Section 7.3, we introduce compression, first for even smaller alphabets ($o(\log n/\log\log n)$), and then generalizing for arbitrary alphabets via multi-ary wavelet trees. In Section 7.4, we explore general applications of our result, in particular to compressed text indexes.

As for the model of computation, our results (and all the mentioned ones) assume a RAM model with word size $w = \Omega(\log n)$, so that operations on $O(\log n)$ contiguous bits can be carried out in constant time. For the dynamic structures, we always allocate $\omega(\log n)$-bit chunks of the same size (or a finite set of sizes), which can be handled in constant time and asymptotically no extra space [RR03].

7.1 Collection of Searchable Partial Sums with Indels

In this section, we generalize the well-known partial sums problem (Section 2.5) to handle a collection of somehow “synchronized” sequences. Apart from having independent interest, this will be an essential tool for the main development in the chapter. We now define our extension of this problem.

The *Collection of Searchable Partial Sums with Indels (CSPSI)* problem consists in maintaining a collection of σ sequences $C = \{S^1, \ldots, S^\sigma\}$ of nonnegative integers $\{s_i^j\}_{1\le j\le\sigma, 1\le i\le n}$, each one of $k = O(\log n)$ bits. The following operations must be supported:

- $sum(C, j, i)$ is $\sum_{l=1}^{i} s_l^j$;
- $search(C, j, y)$ is the smallest i' such that $sum(C, j, i') \ge y$;
- $update(C, j, i, x)$ updates s_i^j to $s_i^j + x$;
- $insert(C, i)$ inserts 0 between s_{i-1}^j and s_i^j for all $1 \le j \le \sigma$.;
- $delete(C, i)$ deletes s_i^j from the sequence S^j for all $1 \le j \le \sigma$; To perform $delete(C, i)$ it must hold $s_i^j = 0$ for all $1 \le j \le \sigma$.

Note the limitations about inserting/deleting only zeros, and at the same place in all sequences. In the sequel, we show how to solve the CSPSI problem in $O(\sigma + \log n)$ time, using $O(\sigma k n)$ bits of space.

Data structure. We construct a red-black tree over C [CLRS01, Chapter 13], where each leaf contains a non-empty *superblock*, whose size goes from $\frac{1}{2}\log^2 n$ to $2\log^2 n$ bits. The leftmost leaf contains $s_1^1 \cdots s_{b_1}^1 s_1^2 \cdots s_{b_1}^2 \cdots\ s_1^\sigma \cdots s_{b_1}^\sigma$, the second leftmost leaf contains $s_{b_1+1}^1 \cdots s_{b_2}^1$ $s_{b_1+1}^2 \cdots s_{b_2}^2 \cdots s_{b_1+1}^\sigma \cdots s_{b_2}^\sigma$, and so on. The size of the leftmost leaf is $\sigma k b_1$ bits, the size of the second leftmost leaf is $\sigma k(b_2 - b_1)$ bits, and so on. The size of the leaves is variable and bounded, so $b_1, b_2, \ldots$ are such that $\frac{1}{2}\log^2 n \leq \sigma k b_1, \sigma k(b_2 - b_1), \ldots \leq 2\log^2 n$.[1] Each internal node v stores counters $\{r^j(v)\}_{1\leq j\leq\sigma}$ and $p(v)$, where $r^j(v)$ is the sum of the integers in the left subtree for sequence S^j and $p(v)$ is the number of positions stored in the left subtree (for any sequence).

Each superblock is further divided into *blocks* of $\sqrt{\log n}\log n$ bits, so each superblock has between $\frac{1}{2}\sqrt{\log n}$ and $2\sqrt{\log n}$ blocks. We maintain these blocks using a linked list. Only the last block in the list could have some free space, all the other use all of their bits. To scan a leaf we proceed block by block. To directly access an arbitrary element in a leaf, we must also follow the links of the blocks until we arrive at the correct block. This takes $O(\sqrt{\log n})$ steps.

Computing *sum*(C, j, i)**.** We traverse the tree to find the leaf containing the i-th position. We start with $sum \leftarrow 0$ and $v \leftarrow root$. If $p(v) \geq i$ we enter the left subtree, otherwise we enter the right subtree with $i \leftarrow i - p(v)$ and $sum \leftarrow sum + r^j(v)$. We reach the leaf that contains the i-th position in $O(\log n)$ time. Then, we scan the leaf, summing up from where the sequence S^j begins, in chunks of size $\frac{1}{2}\log n$ bits using a universal precomputed table Y, until we reach position i. Table Y receives any possible sequence of dk bits, for $d = \lfloor\frac{\frac{1}{2}\log n}{k}\rfloor$, and gives the sum of the d k-bit numbers encoded. The last (at most $d-1$) integers must be added individually.[2] The *sum* query takes in total $O(\log n)$ time, and table Y adds only $O(\sqrt{n}\ \mathrm{polylog}(n))$ bits of space.

Block boundaries do not affect the procedure. If the sequence of dk bits we must input to Y is split between the current and next block, we read the corresponding bits from both blocks to compose the sequence before applying Y. Thus, the complexities are not affected.

Computing *search*(C, j, y)**.** We traverse the tree to find the smallest i' such that $sum(C, j, i') \geq y$. We start with $pos \leftarrow 0$ and $v \leftarrow root$. If $r^j(v) \geq y$ we enter the left subtree, otherwise we enter the right subtree with $y \leftarrow y - r^j(v)$ and $pos \leftarrow pos + p(v)$. We reach the leaf that contains the i'-th position in $O(\log n)$ time. Then, we scan the leaf, summing up from where the sequence S^j begins, in chunks of size $\frac{1}{2}\log n$ bits using table Y, until this sum is greater than y after adding up i' integers; the answer is then $pos + i'$. (More precisely, once an application of the table exceeds y, we must reprocess the last chunk number-wise.) The *search* query takes in total $O(\log n)$ time.

Operation *update*(C, j, i, x)**.** We proceed similarly to *sum*, updating $r^j(v)$ as we traverse the tree. That is, we update $r^j(v)$ to $r^j(v)+x$ each time we go left from v. When we reach the

[1]If $\sigma k > 2\log^2 n$, we just store σk bits per leaf. All the algorithms in the sequel get simplified and the complexities are maintained.

[2]Note that if $k > \frac{1}{2}\log n$ we can just add each number individually within the time bounds.

leaf, we directly update s_i^j to $s_i^j + x$ in $O(\sqrt{\log n})$ time (direct access). The *update* operation takes in total $O(\log n)$ time.

For the next operations, we note that a leaf has at most $m = \lfloor \frac{2\log^2 n}{\sigma k} \rfloor$ integers from any sequence. Then, a subsequence of a given sequence has at most mk bits. So if we copy a subsequence in chunks of $\frac{1}{2}\log n$ bits, the process will take $1 + \lceil \frac{2mk}{\log n} \rceil = O(1 + \frac{\log n}{\sigma})$ time in the RAM model[3]. As we have σ sequences, we can copy a given subsequence of them all in $O(\sigma + \log n)$ time. The next operations are solved by a constant number of applications of these copying operations. Again, block boundaries do not affect the complexities.

Operation *insert*(C, i)**.** We traverse the tree similarly to *sum*, updating $p(v)$ as we traverse the tree. That is, we increase $p(v)$ by 1 each time we go left from v. Then, we create a new copy of the leaf arrived at (by allocating new blocks as needed), adding a 0 between s_{i-1}^j and s_i^j for all j. This is done by first copying the subsequences $\ldots s_{i-1}^j$ for all j, then adding 0 to each sequence, and finally copying the subsequences $s_i^j \ldots$ for all j. As we have just explained, this can be done in $O(\sigma + \log n)$ time.

If the new leaf uses more than $2\log^2 n$ bits, it is split into two. An overflowed leaf has $m = \lfloor \frac{2\log^2 n}{\sigma k} \rfloor + 1$ integers in each sequence. So we store in the left leaf the first $\lfloor m/2 \rfloor$ integers of each sequence and in the right leaf we store the rest. These two copies can be done again in $O(\sigma + \log n)$ time. The new leaves are made children of a new node μ. We compute each $r^j(\mu)$ by scanning and summing on the left leaf. This summing can be done in $O(\sigma + \log n)$ time using table Y. We also set $p(\mu) = \lfloor m/2 \rfloor$. Finally, we check if we need to rebalance the tree. If needed, the red-black tree is rebalanced with $O(1)$ rotations and $O(\log n)$ red-black tag updates [CLRS01, Chapter 13.3]. After a rotation, we need to update $r^j(\cdot)$ and $p(\cdot)$ only for one tree node, which is easily done in $O(\sigma)$ time. The *insert* operation takes in total $O(\sigma + \log n)$ time.

Operation *delete*(C, i)**.** We traverse the tree similarly to *sum*, updating $p(v)$ while we traverse the tree. That is, we decrease $p(v)$ by 1 each time we go left from v. Then, similarly to *insert*, we make a new copy of the leaf (allocating blocks as needed), deleting s_i^j for all j. This takes $O(\sigma + \log n)$ time.

There are three possibilities after this deletion: (i) The new leaf uses more than $\frac{1}{2}\log^2 n$ bits, in which case we are done. (ii) The new leaf uses less than $\frac{1}{2}\log^2 n$ and its sibling is also a leaf, in which case we merge it with its sibling, again in $O(\sigma + \log n)$ time. Note that this merging removes the leaf's parent but does not require any recomputation of $r^j(\cdot)$ or $p(\cdot)$. (iii) The new leaf uses less than $\frac{1}{2}\log^2 n$ and its sibling is an internal node μ, in which

[3]This requires shifting bits, which in case it is not supported by the model, can be handled using small universal tables of the kind of Y.

case by the red-black tree properties we have that μ must have two leaf children[4]. In this case, we merge our new leaf with the closest child of μ, updating the counters of μ in $O(\sigma)$ time, and letting μ replace the parent of our original leaf.

In cases (*ii*) and (*iii*), the merged leaf might use more than $2\log^2 n$ bits. In this case, we split it again into two halves, just as we do in *insert* (and including the recomputation of $r^j(\cdot)$ and $p(\cdot)$). The tree might have to be rebalanced as well. The *delete* operation takes in total $O(\sigma + \log n)$ time.

The breakdown of the space requirement for the structure is as follows.

- All the sequence representations add up to σkn bits of space.
- Each pointer of the linked list of blocks uses $O(\log n)$ bits and we have $O(\frac{\sigma kn}{\sqrt{\log n}\log n})$ full blocks, totalizing $O(\frac{\sigma kn}{\sqrt{\log n}})$ bits.
- The last block in each superblock is not necessarily fully used. We have at most $\lceil \frac{2\sigma kn}{\log^2 n} \rceil$ superblocks, each of which can waste an underused block of size $\sqrt{\log n}\log n$ bits, totalizing $O(\frac{\sigma kn}{\sqrt{\log n}})$ bits.
- For each internal node we have two pointers, red-black data, a counter $p(\cdot)$, and σ counters $r^j(\cdot) \leq 2^k \cdot n$, totalizing $O(\log n) + \sigma(k + \log n) = O(\sigma \log n)$ bits per node. So, the internal nodes use $O(\frac{\sigma kn}{\log^2 n}\,\sigma\log n) = O(\frac{\sigma^2 kn}{\log n})$ bits overall.

We have proved our main result in this section.

Theorem 7.1. *The Collection of Searchable Partial Sums with Indels problem with σ sequences of n numbers of k bits can be solved, in a RAM machine of $w = \Omega(\log n)$ bits, using $\sigma kn(1 + O(\frac{1}{\sqrt{\log n}} + \frac{\sigma}{\log n}))$ bits of space, supporting all the operations in $O(\sigma + \log n)$ worst-case time. Note that, if $\sigma = O(\log n)$ the space is $O(\sigma kn)$ and the time is $O(\log n)$.*

If we had tried to solve the CSPSI problem by just managing σ SPSI individual problems, the time complexities would have raised to $O(\sigma \log n)$.

We note that we have actually assumed that $w = \Theta(\log n)$ in our space computation (as we have used w-bit system pointers). The general case $w = \Omega(\log n)$ can be addressed using the same technique developed in previous work [MN08, Sections 4.5, 4.6, and 6.4], which uses a more refined memory management with pointers of $(\log n) \pm 1$ bits, and splits the sequence into three in a way that retains the worst-case complexities.

[4]For each node, all paths from the node to descendant leaves contain the same number of black nodes and all the leaves are black. In particular, if the sibling of the deleted leaf is an internal node and it is red then its children must be black leaves.

The three subsequences are called *previous*, *current* and *next* [MN08, Section 4.5]. Let $l = \lceil \log n \rceil$ be the current pointer width in use, where n is the current length of the sequences. A prefix of all sequences is in *previous* using $l - 1$ bits, and a suffix in *next* using $l + 1$ bits. The middle part is in *current* and uses l bits. Upon insertions and deletions, some elements are moved across the three structures so as to ensure that, when n becomes a new power of 2 (i.e., $\lceil \log n \rceil$ changes), all the elements reside in *previous* (if n becomes $n/2$) or in *next* (if n becomes $2n$) and we can smoothly change l.

To carry out the queries over this split structure, we must maintain, for each of the three trees, summary $p(\cdot)$ and $r^j(\cdot)$ data on the whole trees. This allows us to know on which of the trees to operate and also gives us information to translate the local result of one tree into the final answer of the structure.

7.2 Uncompressed Dynamic Rank-Select Structures for a Small Alphabet

We now turn our attention into the dynamic *rank/select* problem on a sequence $T[1, n]$ over an alphabet Σ of size σ. We start with a simpler setting, where the alphabet is small, $\sigma = O(\log n)$, and we do not yet attempt to achieve compressed space. In the next section, we build on this one to achieve our stronger result.

Data structure. We construct a red-black tree over $T[1, n]$ where each leaf contains a non-empty *superblock* of size up to $2 \log^2 n$ bits. Each internal node v stores counters $r(v)$ and $p(v)$, where $r(v)$ is the number of superblocks in the left subtree and $p(v)$ is the number of symbols stored in the left subtree.

A superblock storing less than $\log^2 n$ bits will be called *sparse*. Operations *insert* and *delete* will maintain the invariant that no two consecutive sparse superblocks may exist. This ensures that every consecutive pair of superblocks holds at least $\log^2 n$ bits from T, and thus there are at most $1 + \frac{2n \log \sigma}{\log^2 n}$ superblocks.

For each superblock i, we maintain s_i^j, the number of occurrences of symbol j in superblock i, for $1 \leq j \leq \sigma$. We store all these sequences of numbers using a *Collection of Searchable Partial Sums with Indels*, C (Section 7.1). The length of each sequence will be at most $1 + \frac{2n \log \sigma}{\log^2 n}$ integers, we assume $\sigma = O(\log n)$, and $k = O(\log \log n)$ holds because $s_i^j \leq \frac{2 \log^2 n}{\log \sigma}$. So the partial sums operate in $O(\log n)$ worst-case time (Theorem 7.1).

Just as in Section 7.1, each superblock is further divided into *blocks* of $\sqrt{\log n} \log n$ bits, so each superblock has up to $2\sqrt{\log n}$ blocks. We maintain these blocks using a linked list. Only the last block could be not fully used, the rest use all of their bits.

The overall space usage of our structure is $n\log\sigma + O(\frac{n\log\sigma}{\sqrt{\log n}})$, as $\sigma = O(\log n)$:

- The text itself uses $n\log\sigma$ bits of space.
- The *CSPSI* C uses $O(\sigma\log\log n\frac{n\log\sigma}{\log^2 n}) = O(\frac{n\log\log n\log\sigma}{\log n})$ bits of space.
- Each pointer of the linked list of blocks uses $O(\log n)$ bits and we have full $O(\frac{n\log\sigma}{\sqrt{\log n}\log n})$ blocks, totalizing $O(\frac{n\log\sigma}{\sqrt{\log n}})$ bits.
- The last block in each superblock is not necessarily fully used. We have at most $1+\frac{2n\log\sigma}{\log^2 n}$ superblocks, each of which can waste an underused block of size $\sqrt{\log n}\log n$ bits, totalizing $O(\frac{n\log\sigma}{\sqrt{\log n}})$ bits.
- The tree pointers and counters use $O(\frac{n\log\sigma}{\log^2 n}\cdot\log n) = O(\frac{n\log\sigma}{\log n})$ bits.

Now, we show how to carry out all the queries/operations in $O(\log n)$ time. First, it is important to notice, as in Section 7.1, that each block can be scanned or shifted in $O(\sqrt{\log n})$ time, using tables that process chunks of $\frac{1}{2}\log n$ bits[5]. Given that there are $O(\sqrt{\log n})$ blocks in a superblock, we can scan or shift elements within a superblock in $O(\log n)$ time, even considering block boundaries.

Computing $access(T, i)$**.** We traverse the tree to find the leaf containing the i-th position. We start with $sb \leftarrow 1$ and $pos \leftarrow i$. If $p(v) \geq pos$, we enter the left subtree, otherwise we enter the right subtree with $sb \leftarrow sb + r(v)$ and $pos \leftarrow pos - p(v)$. We reach the leaf that contains the i-th position in $O(\log n)$ time. Then, we directly access the pos-th symbol of superblock sb.[6] Note that, within the same $O(\log n)$ time, we can extract any $O(\log^2 n)$-bit long sequence of symbols from T (by moving to next leaves if necessary).

Computing $rank_c(T, i)$**.** We find the leaf containing the i-th position, just as for *access*. Then, we scan superblock sb from the first block summing up the occurrences of c up to the position pos, using a table Z to sum the c's. Z receives a symbol c and $\lfloor\frac{1}{2}\log_\sigma n\rfloor$ symbols ($\leq \frac{1}{2}\log n$ bits), and tells how many times does c appear in the sequence (again, we can just proceed symbolwise if $\log\sigma > \frac{1}{2}\log n$). We add to this quantity $sum(C, c, sb-1)$, the number of times that c appears before superblock sb. The *rank* query takes in total $O(\log n)$ time. Table Z requires $O(\sigma\sqrt{n}\ \mathrm{polylog}(n)) = O(\sqrt{n}\ \mathrm{polylog}(n))$ bits.

[5]Again, if $\log\sigma > \frac{1}{2}\log n$, we can process each symbol individually within the time bounds. This can happen even if $\sigma = O(\log n)$.

[6]Actually we do not need to know the superblock number sb for the *access* query, but we need it for the next ones.

Computing $select_c(T,i)$**.** We calculate $j = search(C,c,i)$; this way we know that the i-th c belongs to superblock j and it is the i'-th appearance of c within superblock j, for $i' = i - sum(C,c,j-1)$. Then, we traverse the tree to find the leaf representing superblock j. We start with $sb \leftarrow j$ and $pos \leftarrow 0$. If $r(v) \geq sb$ we enter the left subtree, otherwise we enter the right subtree with $sb \leftarrow sb - r(v)$ and $pos \leftarrow pos + p(v)$. We reach the correct leaf in $O(\log n)$ time. Then, we scan superblock j from the first block, searching for the position of the i'-th appearance of symbol c within superblock j, using table Z. To this position we add *pos* to obtain the final result. The *select* query takes in total $O(\log n)$ time.

Operation $insert_c(T,i)$**.** We obtain *sb* and *pos* just like in the *access* query, except that we start with $pos \leftarrow i-1$, so as to insert right after position $i-1$. Then, if superblock *sb* contains room for one more symbol, we insert c right after the *pos*-th position of *sb*, by shifting the symbols through the blocks as explained. If the insertion causes an overflow in the last block of *sb*, we simply add a new block at the end of the linked list to hold the trailing bits.

We also carry out $update(C,c,sb,1)$ and retraverse the path from the root to *sb* adding 1 to $p(v)$ each time we go left from v. In this case, we finish in $O(\log n)$ time.

If, instead, the superblock is full, we cannot carry out the insertion yet. We first move one symbol to the previous superblock (creating a new one if this is not possible): We first $delete(T,d)$ the first symbol c' from block *sb* (the global position of c' is $d = i - pos$), and this cannot cause an underflow of *sb*. Now, we check how many symbols does superblock $sb-1$ have (this is easy by subtracting the *pos* numbers corresponding to accessing blocks $sb-1$ and *sb*). If superblock $sb-1$ can hold one more symbol, we insert the removed symbol c' at the end of superblock $sb-1$. This is done by calling $insert_{c'}(T,d)$, a recursive invocation that now will arrive at block $sb-1$ and will not overflow it (thus no further recursion will occur).[7]

If superblock $sb-1$ is also full or does not exist, then we are entitled to create a sparse superblock between $sb-1$ and *sb*, without breaking the invariant on sparse superblocks. We create such an empty superblock and insert symbol c' into it, using the following procedure: We retraverse the path from the root to *sb*, updating $r(v)$ to $r(v)+1$ each time we go left from v. When we arrive again at leaf *sb*, we create a new node μ with $r(\mu) = 1$ and $p(\mu) = 1$. Its left child is the new empty superblock, where the single symbol c' is inserted, and its right child is *sb*. We also execute $insert(C,sb)$ and $update(C,sb,c',1)$.

After creating μ, we must check if we need to rebalance the tree. If it is needed, it can be done with $O(1)$ rotations and $O(\log n)$ red-black tag updates. After a rotation, we need to update $r(\cdot)$ and $p(\cdot)$ only for one tree node. These updates can be done in constant time.

[7]We note that, if one deletes the first symbol of a block and reinserts it at the same position, it will get inserted into the previous block.

Now that we have finally made room to carry out the original insertion, we rerun $insert_c(T, i)$ and it will not overflow again. The whole *insert* operation takes $O(\log n)$ time.

Operation $delete(T, i)$**.** We obtain *sb* and *pos* just as in the *access* query, updating $p(v)$ to $p(v) - 1$ each time we go left from v. Then, we delete the *pos*-th position (let c be the symbol deleted) of the *sb*-th superblock, by shifting the symbols back through the blocks. If this deletion empties the last block, we free it. In any case, we call $update(C, c, sb, -1)$ on the partial sums.

There are three possibilities after this deletion: (*i*) superblock *sb* is not sparse after the deletion, in which case we are done; (*ii*) *sb* was already sparse before the deletion, in which case we have only to check that it has not become empty; (*iii*) *sb* turned to sparse due to the deletion, in which case we have to care about the invariant on sparse superblocks.

If superblock *sb* becomes empty, we retraverse the path from the root to it, updating $r(v)$ to $r(v) - 1$ each time we go left from v, in $O(\log n)$ time. When we arrive at leaf *sb* again, we remove it and invoke $delete(C, sb)$. Finally, we check if we need to rebalance the tree, in which case $O(1)$ rotations and $O(\log n)$ red-black tag updates suffice, just as for insertion. After a rotation we also need to update $r(\cdot)$ and $p(\cdot)$ only for one tree node. These updates take constant time.

If, instead, superblock *sb* turned to sparse, we make sure that neither superblocks $sb - 1$ or $sb + 1$ are also sparse. If they are not, then superblock *sb* can become sparse and hence we finish without further intervention.

If superblock $sb + 1$ is sparse, we $delete(T, d)$ its first symbol c' (at position d), and $insert_{c'}(T, d)$ at the end of superblock *sb* (as done for the insertion). This recursive call brings no problems because $sb + 1$ is already sparse, and we restore the non-sparse status of *sb*. If superblock $sb + 1$ becomes empty, we remove it just as explained for the case of superblock *sb*. The action is symmetric if $sb + 1$ is not sparse but $sb - 1$ is.[8]

The *delete* operation takes in total $O(\log n)$ time.

Theorem 7.2. *Given a text T of length n over a small alphabet of size $\sigma = O(\log n)$, the Dynamic Sequence with Indels problem under RAM model with word size $w = \Omega(\log n)$ can be solved using $n \log \sigma + O(\frac{n \log \sigma}{\sqrt{\log n}})$ bits of space, supporting all the queries access, rank, select, insert and delete, in $O(\log n)$ worst-case time.*

We note again that we have actually assumed that $w = \Theta(\log n)$ in our space computation. The general case $w = \Omega(\log n)$ can be obtained using exactly the same techniques developed previously [MN08, Sections 4.5, 4.6, and 6.4], with no changes.

[8]For the symmetric case one needs a slightly different version of procedure *insert*, which inserts after, not before, position i.

7.3 Compressed Dynamic Rank-Select Structures

We now extend our results to use a compressed sequence representation, by just changing the way we store/manage the *blocks*. The key idea is to detach the *representational* and the *physical* (i.e., compressed) sizes of the storage units at different levels.

We use the same red-black tree over $T[1,n]$, where each leaf contains a non-empty superblock *representing* up to $2\log^2 n$ bits *of the original text* T (they will actually store more or less bits depending on how compressible is the portion of T they represent). The same superblock splitting/merging policy related to sparse superblocks is used. Each internal node has the same counters and they are managed in the same way. So all the queries/operations are exactly the same up to the superblock level. Compression is encapsulated inside the superblocks.

In *physical* terms, a superblock is divided into blocks just as before, and they are still of the same *physical* size, $\sqrt{\log n}\log n$ bits. Depending on compressibility, blocks will represent more or less symbols of the original text, as their physical size is fixed.

In *logical* terms, a superblock is divided into *segments* representing $\lfloor \frac{1}{2}\log_\sigma n \rfloor$ original symbols[9] from T. We represent each segment using the (c,o)-pair encoding of Ferragina et al. [FMMN07]: The c part is of fixed width and tells how many occurrences of each alphabet symbol are there in the segment; whereas the o part is of variable width and gives the identifier of the segment among those sharing the same c component. Each c component uses at most $\sigma\log\log n$ bits; while the o components use at most $\frac{1}{2}\log n$ bits each, and overall add up to $nH_0(T) + O(n\log\sigma/\log n)$ bits [FMMN07, Section 3.1].

In a block of $\sqrt{\log n}\log n$ bits, we store as many bits as they fit. The universal tables (like Y) used to sequentially process the blocks in chunks of $\frac{1}{2}\log n$ bits must now be modified to process the compressed sequence of (c,o) pairs. This is complex because an insertion in a segment introduces a displacement that propagates over all the segments of the superblock, which must be completely recomputed and rewritten (and it can even cause the physical size of the whole superblock to double!). Fortunately, all those tedious details have been already sorted out in previous work [MN08, Sections 5.2, 6.1, and 6.2], where their “superblocks” play the role of our “blocks”, and their tree rearrangements are not necessary for us because we are within a leaf now. Their “partial blocks” mechanism is also not useful for us, because we can tolerate those propagations to extend over all the blocks of our superblocks. Hence, only the last block of our superblocks is not completely full.

The time achieved in there [MN08] is $O(1)$ per $\Theta(\log n)$ physical bits. Even in the worst case (where compression does not work at all in the superblock), the number of physical bits will be $\frac{2\log^2 n}{\frac{1}{2}\log n}(\sigma\log\log n + \frac{1}{2}\log n) = O(\log^2 n + \sigma\log n\log\log n)$, and thus the time to solve any query or carry out any update on a superblock will be $O(\log n + \sigma\log\log n)$.

[9]Or just one symbol if $\frac{1}{2}\log_\sigma n < 1$.

Let us now consider the space usage of these new structures, focusing only on what differs from the uncompressed version:

- The text itself (as a sequence of pairs (c, o)) uses $nH_0(T) + O(\frac{\sigma n \log\log n}{\log_\sigma n})$ bits.
- The number of full blocks is $O(\frac{nH_0(T)+\frac{\sigma n \log\log n}{\log_\sigma n}}{\sqrt{\log n}\log n})$, and thus the space wasted by their pointers is $O(\frac{n\log\sigma(\sigma\log\log n+\log n)}{\sqrt{\log n}\log n})$ bits.
- The extra space for the tables to operate the (c, o) encoding is $O(\sqrt{n}\ \sigma\ \text{polylog}(n))$ bits.

It can be seen that the time and space complexities depend sharply on σ. Thus, the solution is indeed of interest only for rather small $\sigma = o(\log n/\log\log n)$. For such a small alphabet, we have the following theorem. Again, all the issues of varying $\lceil\log n\rceil$ and the case $w = \omega(\log n)$ are handled just as in previous work [MN08, Sections 4.5, 4.6, and 6.4]

Theorem 7.3. *Given a text T of length n over a small alphabet of size $\sigma = O(\frac{\sqrt{\log n}}{\log\log n})$ and zero-order entropy $H_0(T)$, the Dynamic Sequence with Indels problem under RAM model with word size $w = \Omega(\log n)$ can be solved using $nH_0(T) + O(\frac{n\log\sigma}{\sqrt{\log n}})$ bits of space, supporting the queries access, rank, select, insert and delete in $O(\log n)$ worst-case time.*

To extend our results to a larger alphabet of size $\sigma = \Omega(\sqrt{\log n}/\log\log n)$, we use a generalized ρ-ary wavelet tree [FMMN07] over T, where $\rho = \Theta(\sqrt{\log n}/\log\log n)$. Essentially, this generalized wavelet tree makes a sequence with the first $\log\rho$ bits of the symbols at the first level, the next $\log\rho$ bits at the second level (where the symbols with the same first $\log\rho$ bits are grouped in the same child of the root), and so on. The tree has $O(\log_\rho\sigma) = O(\frac{\log\sigma}{\log\log n})$ levels. We store on each level a sequence over an alphabet of size ρ, which is handled using the solution of Theorem 7.3, for which ρ is small enough. Hence, each query and operation takes $O(\log n)$ time per level, adding up $O(\log n\frac{\log\sigma}{\log\log n})$ worst-case time overall.

As shown by Ferragina et al. [FMMN07], the sum of the zero-order entropy representations of the sequences at each level adds up to the zero-order entropy of T. In addition, the generalized ρ-ary wavelet tree handles changes in $\lceil\log n\rceil$ automatically, as this is encapsulated within each level. We thus obtain our main theorem, where we have included the case of small σ as well. We recall that, within the same time, *access* can retrieve $O(\log_\sigma n\log n)$ consecutive symbols from T.

Theorem 7.4. *Given a text T of length n over an alphabet of size σ and zero-order entropy $H_0(T)$, the Dynamic Sequence with Indels problem under RAM model with word size $w = \Omega(\log n)$ can be solved using $nH_0(T)+O(\frac{n\log\sigma}{\sqrt{\log n}})$ bits of space, supporting queries access, rank, select, insert and delete in $O(\log n(1 + \frac{\log\sigma}{\log\log n}))$ worst-case time.*

7.4 Applications

Very recently [MN08, MN07] it has been shown that a wavelet tree built over the Burrows-Wheeler Transform T^{bwt} of a text T [BW94], and compressed using the (c, o) pair technique (Section 7.3), achieves high-order entropy space, namely $nH_k(T) + o(n \log \sigma)$ for any $k+1 \leq \alpha \log_\sigma n$ and constant $0 < \alpha < 1$. This is used by Mäkinen and Navarro [MN08] to obtain a dynamic text index that handles a collection $\mathcal{C}$ of texts and permits searching for patterns, extracting text snippets, and inserting/deleting texts in/from the collection. Using their definitions [MN08, Section 7] and using their same sampling step, we can state a stronger version of those theorems:

Theorem 7.5. *The Dynamic Text Collection problem can be solved with a data structure of size* $nH_k(\mathcal{C}) + o(n \log \sigma) + O(\sigma^{k+1} \log n + m \log n + w)$ *bits, simultaneously for all* k*. Here* n *is the length of the concatenation of* m *texts,* $\mathcal{C} = 0\ T_1 0\ T_2 \cdots\ 0\ T_m$*, and we assume that* $\sigma = o(n)$ *is the alphabet size and* $w = \Omega(\log n)$ *is the machine word size under the RAM model. The structure supports counting of the occurrences of a pattern* P *in* $O(|P| \log n(1 + \frac{\log \sigma}{\log \log n}))$ *time, and inserting and deleting a text* T *in* $O(|T| \log n(1 + \frac{\log \sigma}{\log \log n}))$ *time. After counting, any occurrence can be located in time* $O(\log^2 n(1 + \frac{\log \log n}{\log \sigma}))$*. Any substring of length* ℓ *from any* T *in the collection can be displayed in time* $O(\log^2 n(1 + \frac{\log \log n}{\log \sigma}) + \ell \log n(1 + \frac{\log \sigma}{\log \log n}))$*. For* $k \leq (\alpha \log_\sigma n) - 1$*, for any constant* $0 < \alpha < 1$*, the space complexity simplifies to* $nH_k(\mathcal{C}) + o(n \log \sigma) + O(m \log n + w)$ *bits.*

When the alphabet is of moderate size, that is, $\sigma = O(\text{polylog}(n))$, the times obtained above become $O(|P| \log n)$ for counting, $O(|T| \log n)$ for text insertion/deletion, $O(\log_\sigma n \log \log n)$ for locating, and $O(\log_\sigma n \log \log n + \ell)$ for displaying.

Another important application that derives from Theorem 7.5 is the compressed construction of text indexes. For example, a variant of the FM-index [FMMN07] (Section 2.9.1) requires k-th entropy space once built, but in order to build it we need $O(n \log n)$ bits of space. The previous theorem can be used to build the FM-index of a text by starting with an empty collection and inserting the text T of interest. Our new results make this process faster.

Theorem 7.6. *The Alphabet-Friendly FM-index of a text* $T[1, n]$ *over an alphabet of size* σ *can be built using* $nH_k(T) + o(n \log \sigma)$ *bits, simultaneously for all* $k \leq (\alpha \log_\sigma n) - 1$ *and any constant* $0 < \alpha < 1$*, in time* $O(n \log n(1 + \frac{\log \sigma}{\log \log n}))$*.*

We note that this is the same asymptotic space required for the final, static, FM-index [FMMN07]. This FM-index is not only relevant by itself, but also as an intermediate step to compute other important structures such as the *suffix array* [MM93] and the *Burrows-Wheeler Transform (BWT)* [BW94] of T. Both are easily derived from our dynamic FM-index. Although the final product takes in this case more space than our intermediate representation, we can output the result in order, so that we do not need to maintain the large representation in memory. Our next discussion assumes this model: we must output the suffix array or the BWT sequentially (as otherwise there is no point in building them in little space).

The BWT is the simpler problem for us. We can easily derive it sequentially from the FM-index, by obtaining one by one the symbols in $O(1+\frac{\log \sigma}{\log \log n})$ time each, and sending them to the output. The best previous result we know of, in terms of space complexity [Kär04], achieves $O(n \log^2 n)$ time ($O(n \log n)$ on average) using $O(n)$ bits in addition to the $n \log \sigma$ bits of the text. This is asymptotically worse than our space and time for any σ. We note that, using previous work [MN08], one achieves $O(n \log n \log \sigma)$ time, which may be as bad as $O(n \log^2 n)$ for large σ.

Using our result to build the suffix array is a bit more complicated. Let us focus on the case $\sigma = O(\text{polylog}(n))$, where our FM-index worst-case construction time becomes $O(n \log n)$. To obtain the suffix array sequentially, we must carry out one *locate* operation for each cell, which can be made as fast as $O(\log n)$ time per cell if we spend $O(n)$ additional bits of space. Thus, we can build the suffix array sequentially within $n \log \sigma + o(n \log \sigma) + O(n)$ bits (even on uncompressible texts) and in $O(n \log n)$ time. This was indeed the best known time complexity to build the suffix array until a few years ago [PST07]. Nowadays, linear-time algorithms exist, yet all of them require $O(n \log n)$ bits of space. On the other hand, the best current result on compressed suffix array construction [HLS+07] takes $O(n \log \sigma)$ bits of space and $O(n \log n)$ time for arbitrary alphabets (note that the space is not compressed and its constant term is not 1).

Chapter 8

Conclusions

Since year 2000, a rapid sequence of achievements showed how to relate information theory with string matching concepts, in a way that index regularities that show up when the text is compressible were discovered and exploited to reduce index occupancy. The overall result has been the design of full-text indexes whose size is proportional to that of the compressed text. Moreover, those indexes are able to reproduce any text portion without accessing the original text, and thus they replace the text. This way, compressed text indexes allow one to add search and random access functionalities to compressed data. For example, it is feasible today to index the 3 GB Human genome on a 1 GB RAM desktop PC.

These compressed text indexes, however, presented several weaknesses (see Section 1.1). In this thesis, we have addressed these weaknesses and proposed some solutions to them. In the following, we summarize our main contributions. At the end, we give some directions for further work.

8.1 Contribution of this Thesis

8.1.1 Compressed Text Indexes: From Theory to Practice

We have first considered the basic problem of performing rank/select queries over a binary sequence, focusing on succinct data structures that pose a sublinear space overhead. This is an intensive area of theoretical research with lots of applications. Yet, there had been little effort in assessing how practical are the best theoretical solutions, compared to more naive alternatives that not always guarantee the same complexities. Our conclusions are that, in many cases, a clever implementation of a theoretically more naive solution can outperform the best theoretical solution. In other cases, the theoretical solution is a good choice provided it is cleverly implemented.

Although a comprehensive survey on compressed text indexing has recently appeared [NM07], the implementation of these indexes requires a significant programming skill, a deep engineering effort, and a strong algorithmic background. Only isolated implementations and focused comparisons of compressed indexes had been reported, and they missed a common API, which prevented their re-use or deploy within other applications. The research presented in Chapter 3 had therefore three main purposes:

1. We review the most successful compressed indexes that have been implemented so far, and present them in a way that may be useful for software developers, by focusing on implementation choices as well on their limitations. This point of view complements the existing survey [NM07] and fixes the state-of-the-art for this technology, possibly stimulating improvements in the design of such sophisticated algorithmic tools.

   In addition, we introduce novel implementations of compressed indexes, which combine the best existing theoretical guarantees with a competitive space/time tradeoff in practice.

2. We experimentally compare a selected subset of implementations. This not only serves to help programmers in choosing the best index for their needs, but also gives a grasp of the practical relevance of this fascinating technology.

3. We introduce the *Pizza&Chili* site, which was developed with the aim of providing publicly available implementations of compressed indexes. Each implementation is well-tuned and adheres to a suitable API of functions, which should allow any programmer to easily plug the provided compressed indexes within his/her own software. The site also offers a collection of texts and tools for experimenting and validating the proposed compressed indexes. We hope that this simple API and the good performance of those indexes will spread their use in several applications.

The use of compressed indexes is obviously not limited to plain text searching, but to storing, searching and retrieving collections of strings in little space.

8.1.2 Locally Compressed Suffix Arrays

We have presented a suffix array compression method that retains fast locating of the occurrences of a pattern. We have proved analytically that the resulting size is related to the k-th order entropy of the text. The method has been used to obtain a compressed self-index with fast *locate* (where the norm is to be extremely slow), and a small index that is a viable alternative to classical suffix arrays. Our experiments show that our structure is very practical and relevant. In particular, we showed that the regularities that Re-Pair exploits on the differential suffix array are closely related to the runs in Ψ. Thus, we have taken advantage of the locality properties of Re-Pair, and also used the close relation with Ψ to analyze the compression achieved and design faster Re-Pair algorithms. These new algorithms are approximations, yet we show that their compression loss is negligible.

A byproduct, which might be of general interest, is a compact data structure to represent the Re-Pair dictionary. This structure can reduce the dictionary space by up to 50%, and operates in compressed form, that is, it permits decompressing parts of the text without decompressing the dictionary.

8.1.3 Statistical Encoding of Sequences

We have presented a scheme based on k-th order modeling plus statistical encoding to convert any data structure on sequences into a compressed data structure. This structure permits retrieving any string of S of $\Theta(\log_\sigma n)$ symbols in constant time. This is an alternative to the first work achieving the same result [SG06b], which is based on Ziv-Lempel compression. We also show how to append symbols to the original sequence within the same space complexity and with constant amortized cost per appended symbol. This method also works on the structure presented in [SG06b].

We also analyze the behavior of this technique when applied to full-text self-indexes, as advocated in [SG06b]. We prove some results on the application of this technique over the Burrows-Wheeler Transform [BW94] and over the wavelet tree [GGV03] of a sequence.

As a followup of our work, Ferragina and Venturini [FV07] proposed an even simpler storage scheme, which does not use any compressor (either statistical or Ziv-Lempel) and achieves the same space usage of our scheme. To prove their space usage, however, they build on our results.

8.1.4 A Compressed Text Index on Secondary Memory

We have presented a practical self-index for secondary memory that, when the text is compressible, takes much less space than the suffix array. It also provides good I/O times for locating, which in particular improve when the text is compressible. In this aspect, our index is unique, as most compressed indexes are slower than their classical counterparts on secondary memory. We show experimentally that our index is very competitive against the alternatives, offering very relevant space/time tradeoffs.

We also presented a secondary memory construction for those approximations, which run almost I/O-optimally and extends the applicability of the methods to compress suffix arrays that do not fit in main memory.

8.1.5 Rank/Select on Dynamic Compressed Sequences

We combine previous works [MN08, LP07] to obtain, for a sequence S, a structure that (1) takes $nH_0(S) + o(n \log \sigma)$ bits of space, and (2) performs all the operations (*access*, *rank*, *select*, *insert* and *delete*) in $O(\log n(1 + \frac{\log \sigma}{\log \log n}))$ worst-case time. (This is achieved even for the case where $\lceil \log n \rceil$ changes and so does the length of the structure pointers in order to maintain the promised space bounds.) The result becomes the most efficient dynamic representation of sequences, both in time and space, and its benefits have immediate applications to other compressed data structures such as compressed text indexes.

Lee and Park [LP07] spend $O(n)$ extra bits in bitmaps that maintain leaf-granularity information on *rank/select*. We show that this can be replaced by dynamic partial sums, which use sublinear space. However, we need σ partial sums and cannot afford to update them individually upon a leaf insertion/deletion. Hence, we create a new structure where a collection of σ sequences are maintained in synchronization, and this can be of independent interest. The second problem was that leaf splitting/merging in Lee and Park's work [LP07] triggered too many updates to summarization data, which could not be handled in $O(\log n)$ worst-case time, only in $O(\log n)$ amortized time. To get rid of this problem, we redefined the leaf fill ratio invariants, preferring a weaker condition that still ensures that leaves are sufficiently full and can be maintained within the $O(\log n)$-worst-case-time bound. This can also be of independent interest.

Our result is not only interesting by itself, but also yields the best current algorithm to maintain a dynamic collection of texts that can be searched for patterns, and to build indexes for static text collections within compressed space. In addition, our results permit building suffix arrays [MM93] in competitive time, improving in particular the best algorithm to build it within $O(n \log \sigma)$ bits of space when the alphabet is not too large. Finally, we derive the best current algorithm to compute the Burrows-Wheeler Transform [BW94] within $n \log \sigma + O(n)$ bits of space.

8.2 Further Work

Our work on practical compressed text indexes opens several future development lines. In the short term, we plan to do some upgrades on the *Pizza&Chili* site: To add new indexes; to add new functionalities to the indexes; to improve the API to support several simultaneously active indexes and to add iterators for *locate* and *display*. As an ambitious longer-term project, we want to convert *Pizza&Chili* into a more general site on compressed data structures. We also seek to implement our secondary memory index, which is right now a theoretical proposal (our experimental results are obtained with simulations). In this latter line, we are working on merging the CPT structure [CM96] with our LCSA. We believe the result would be extremely competitive in practice. We also seek to implement the secondary memory construction of our LCSA.

Our work on compressed sequences also leaves several future challenges. High-order entropy compression requires better understanding of how entropies behave upon transformations such as Burrows-Wheeler or the wavelet tree. Also, there is no good *rank* and *select* support for this space occupancy. Finally, the dynamic support for high-entropy compressed sequences, even considering just *access*, is currently null or very rudimentary [SG06b, GN06, FV07].

On the other hand, we showed that one can give good support for dynamic sequences while compressing them to zero-order entropy. Yet, more can be done. Chan et al. [CHLS07] recently showed that *rank* and *select* on bits ($\sigma = 2$) can be solved in $O(\frac{\log n}{\log\log n})$ time for all operations, using $O(n)$ bits of space (this is striking because the *rank*/*select* problem was conjectured to have the same $\Omega(\log n)$ lower bound of partial sums [PD06]). Combining with multiary wavelet trees, one immediately achieves $O(n \log \sigma)$ bits of space and $O(\frac{\log n}{\log\log n}(1 + \frac{\log \sigma}{\log\log n}))$ time for general alphabets. This time matches the lower bound of Fredman and Saks for *rank*/*select* [FS89] as long as $\sigma = O(\text{polylog}(n))$, whereas it is not known whether the result would be time-optimal for larger σ. In any case, this raises the challenge of achieving that complexity within $nH_0 + o(n \log \sigma)$ bits of space, or even $nH_k + o(n \log \sigma)$.

Finally, one can wish to handle a stronger set of operations. In particular, our wavelet trees are markedly static in shape, and thus supporting changes in the alphabet Σ looks challenging. This would have applications in a dynamic scenario where the set of symbols is not known in advance.

In a more conceptual view, we believe this thesis is a first step towards compressed text indexes with competitive locating times, in particular via locality of access. Our resulting index is still far from achieving the space used by the smallest self-indexes, which are however extremely slow to locate. This raises some questions: Is there a fundamental lower bound to the tradeoff one can achieve between space and time for locating? Is there a limit to what can be achieved via local compression?

Appendix A

API for Text Indexes

The *Pizza&Chili* indexes are used through an API written in C/C++ language. This API is sketched below. We use `uchar` for denoting unsigned char and `ulong` for unsigned long. The interface assumes that each text symbol is represented with one byte. The integer `e` returned by any procedure indicates an error code, if it is different of zero. The error message can be retrieved by calling the procedure `char *error_index(e)`. Text and pattern indexes start at zero.

- **Error management**
  - `char *error_index (int e)` returns a string describing the error associated with *e*. The string must not be freed, and it will be overwritten with subsequent calls.
- **Building the index**
  - `int build_index (uchar *text, ulong length, char *build_options, void **index)` creates an index from `text[0..length-1]`. Note that the index is an opaque data type (i.e., it uses `void`-type). Build options must be passed in string `build_options`, whose syntax depends on the index and is described in its accompanying documentation. The index must always work with some default parameters, if `build_options` is `NULL`. The returned index is ready to be queried.
  - `int save_index (void *index, char *filename)` saves `index` on disk by using single or multiple files, using proper extensions.
  - `int load_index (char *filename, void **index)` loads `index` from one or more files named `filename` with proper extensions.
  - `int free_index (void *index)` frees the memory occupied by `index`.
  - `int index_size(void *index, ulong *size)` tells the memory occupied by `index` in bytes. This must be the internal memory the index needs to operate.

- **Querying the index**
  - `int count (void *index, uchar *pattern, ulong length, ulong *numocc)` writes in `numocc` the number of occurrences of the substring `pattern[0..length-1]` found in the text indexed by `index`.
  - `int locate (void *index, uchar *pattern, ulong length, ulong **occ, ulong *numocc)` writes in `numocc` the number of occurrences of `pattern[0..length-1]` in the text indexed by `index`. It also allocates `*occ` (which must be freed by the caller) that contains the locations of the `*numocc` occurrences, in arbitrary order.
- **Accessing the indexed text**
  - `int extract (void *index, ulong from, ulong to, uchar **snippet, ulong *snippet_length)` allocates `snippet` (which must be freed by the caller) and writes the substring `text[from..to]` into it. Returns in `snippet_length` the length of the text snippet actually extracted, since this could be less than `to-from+1` if `to` is larger than the text size.
  - `int display (void *index, uchar *pattern, ulong length, ulong numc, ulong *numocc, uchar **snippet_text, ulong **snippet_lengths)` displays the text snippet surrounding every occurrence of the substring `pattern[0..length-1]` within the indexed text. The snippet must include `numc` symbols before and after the pattern occurrence, totalizing `length+2*numc` symbols, or less if the text boundaries are reached. The number of pattern occurrences is stored in `numocc`, and their snippets are stored in the arrays `snippet_text` and `snippet_lengths` (which must be freed by the caller). The first array is a symbol array of `numocc*(length+2*numc)` symbols, with a new snippet starting at every multiple of `length+2*numc`. The second array contains integers, each indicating the real length of each of the `numocc` extracted snippets.
  - `int length (void *index, ulong *length)` obtains the length of the text indexed by index, in bytes.

Bibliography

[AN05] D. Arroyuelo and G. Navarro. Space-efficient construction of LZ-index. In *Proc. 16th Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS 3827, pages 1143–1152, 2005.

[AN07] D. Arroyuelo and G. Navarro. A Lempel-Ziv text index on secondary storage. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 83–94, 2007.

[AN08] D. Arroyuelo and G. Navarro. Practical approaches to reduce the space requirement of Lempel-Ziv-based compressed text indices. Technical Report DCC-2008-9, Dept. of Computer Science, University of Chile, 2008.

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

[ANS08] D. Arroyuelo, G. Navarro, and Kunihiko Sadakane. Stronger Lempel-Ziv based compressed text indexing. Technical Report DCC-2008-2, Dept. of Computer Science, University of Chile, 2008.

[Ash65] Robert B. Ash. *Information Theory.* Dover Publications, 1965.

[BB04] D. Blandford and G. Blelloch. Compact representations of ordered sets. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 11–19, 2004.

[BCFM00] K. Brengel, A. Crauser, P. Ferragina, and U. Meyer. An experimental study of priority queues in external memory. *ACM Journal of Experimental Algorithmics*, 5(17), 2000.

[BCW90] T. Bell, J. Cleary, and I. Witten. *Text compression.* Prentice Hall, 1990.

[BDMR99] D. Benoit, E. Demaine, I. Munro, and V. Raman. Representing trees of higher degree. In *Proc. 6th International Workshop on Algorithms and Data Structures (WADS)*, LNCS 1663, pages 169–180, 1999.

[BGMR06] J. Barbay, A. Golynski, I. Munro, and S. Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 24–35, 2006.

[BHMR07] J. Barbay, M. He, I. Munro, and S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 680–689, 2007.

[Bur93] Michael Buro. On the maximum length of Huffman codes. *Information Processing Letters*, 45(5):219–223, 1993.

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

[BYBZ96] R. Baeza-Yates, E. F. Barbosa, and N. Ziviani. Hierarchies of indices for text searching. *Information Systems*, 21(6):497–514, 1996.

[CF02] A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[CHL04] H.-L. Chan, W.-K. Hon, and T.-W. Lam. Compressed index for a dynamic collection of texts. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 3109, pages 445–456, 2004.

[CHLS07] H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms (TALG)*, 3(2):21, 2007.

[CHSV08] Y.-F. Chien, W.-K. Hon, R. Shah, and J. Vitter. Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In *Proc. 18th Data Compression Conference (DCC)*, pages 252–261, 2008.

[CKL06] R. Cole, T. Kopelowitz, and M. Lewenstein. Suffix trays and suffix trists: Structures for faster text indexing. In *Proc. 33th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 358–369, 2006.

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

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill, second edition, 2001.

[CM96] D. Clark and I. Munro. Efficient suffix trees on secondary storage. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 383–391, 1996.

[CN07] F. Claude and G. Navarro. A fast and compact Web graph representation. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 105–116, 2007.

[DKMS05] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. In *Proc. 7th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 86–97, 2005.

[FG96] P. Ferragina and R. Grossi. Fast string searching in secondary storage: theoretical developments and experimental results. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 373–382, 1996.

[FG99] P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[FGM06] P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. In *Proc. 33rd International Colloquium on Automata, Languages and Programming (ICALP)*, pages 560–571, 2006.

[FGMS05] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52:688–713, 2005.

[FLMM05] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 184–196, 2005.

[FM00] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.

[FM01] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 269–278, 2001.

[FM04] P. Ferragina and G. Manzini. Compression boosting in optimal linear time using the Burrows-Wheeler transform. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 655–663, 2004.

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

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

[FS89] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proc. 43th ACM Symposium on Theory of Computing (STOC)*, pages 345–354, 1989.

[FV07] P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. *Theoretical Computer Science (TCS)*, 372(1):115–121, 2007.

[GGG$^+$07] A. Golynski, R. Grossi, A. Gupta, R. Raman, and S. Rao. On the size of succinct indices. In *Proc. 15th Annual European Symposium (ESA)*, LNCS 4698, pages 371–382, 2007.

[GGMN05] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005.

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

[GHSV07a] A. Gupta, W.-K. Hon, R. Shah, and J. Vitter. Compressed data structures: Dictionaries and data-aware measures. *Theoretical Computer Science (TCS)*, 387(3):313–331, 2007.

[GHSV07b] A. Gupta, W.-K. Hon, R. Shah, and J. Vitter. A framework for dynamizing succinct data structures. In *Proc. 34th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 521–532, 2007.

[GMR06] A. Golynski, I. Munro, and S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

[GN06] R. González and G. Navarro. Statistical encoding of succinct data structures. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009, pages 295–306, 2006.

[GN07a] R. González and G. Navarro. A compressed text index on secondary memory. In *Proc. 18th International Workshop on Combinatorial Algorithms (IWOCA)*, pages 80–91. College Publications, UK, 2007.

[GN07b] R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 216–227, 2007.

[GN08] R. González and G. Navarro. Improved dynamic rank-select entropy-bound structures. In *Proc. 8th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 4957, pages 374–386, 2008.

[GNP$^+$06] Sz. Grabowski, G. Navarro, R. Przywarski, A. Salinger, and V. Mäkinen. A simple alphabet-independent FM-index. *International Journal of Foundations of Computer Science (IJFCS)*, 17(6):1365–1384, 2006.

[Gol07] A. Golynski. Optimal lower bounds for rank and select indexes. *Theoretical Computer Science (TCS)*, 387(3):348–359, 2007.

[GRRR04] R. Geary, N. Rahman, V. Raman, and R. Raman. A simple optimal representation for balanced parentheses. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 159–172, 2004.

[GS03] R. Giancarlo and M. Sciortino. Optimal partitions of strings: A new class of Burrows-Wheeler compression algorithms. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 129–143, 2003.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[GV00] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd ACM Symposium on Theory of Computing (STOC)*, pages 397–406, 2000.

[GV06] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2006.

[HLS$^+$07] W.-K. Hon, T.-W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 48(1):23–36, 2007.

[HSS03a] W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 251–260, 2003.

[HSS03b] W.-K. Hon, K. Sadakane, and W.-K. Sung. Succinct data structures for searchable partials sums. In *Proc. 14th Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS 2906, pages 505–516, 2003.

[Huf52] D. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the Institute of Radio Engineers*, 40(9):1090–1101, 1952.

[Jac89] G. Jacobson. Succinct static data structures. Technical Report CMU-CS-89-112, Carnegie Mellon Univ., 1989.

[Kär04] J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. In *Proc. DIMACS Workshop on the Burrows-Wheeler Transform: Ten Years Later*, 2004.

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

[KR03] J. Kärkkäinen and S. Rao. *Algorithms for Memory Hierarchies*, chapter 7: Full-text indexes in external memory, pages 149–170. LNCS 2625. 2003.

[Kur99] S. Kurtz. Reducing the space requirement of suffix trees. In *Software - Practice and Experience*, pages 1149–1171, 1999.

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

[LP00] K. Lemström and S. Perttu. SEMEX – an efficient music retrieval prototype. In *Proc. 1st International Symposium on Music Information Retrieval (ISMIR)*, 2000.

[LP07] S. Lee and K. Park. Dynamic rank-select structures with applications to run-length encoded texts. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 95–106, 2007.

[LS99] J. Larsson and K. Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, Department of Computer Science, Lund University, Sweden, 1999.

[LSSY02] T.-W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. In *Proc. 8th Annual International Conference on Computing and Combinatorics (COCOON)*, pages 401–410, 2002.

[Mäk03] V. Mäkinen. Compact suffix array — a space-efficient full-text index. *Fundamenta Informaticae*, 56(1–2):191–210, 2003.

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

[MF04] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

[Mil05] P. Miltersen. Lower bounds on the size of selection and rank indexes. In *Proc. 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 11–12, 2005.

[ML00] R. Milidiú and E. Laber. The WARM-UP algorithm: A Lagrangian construction of length restricted Huffman codes. *SIAM Journal on Computing*, 30(5):1405–1426, 2000.

[MM93] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 22:935–948, 1993.

[MN04] V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 3109, pages 420–433, 2004.

[MN05] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

[MN06a] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009, pages 307–318, 2006.

[MN06b] V. Mäkinen and G. Navarro. Position-restricted substring searching. In *Proc. 7th Latin American Symposium on Theoretical Inform atics (LATIN)*, LNCS 3887, pages 703–714, 2006.

[MN07] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. 14th String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 214–226, 2007.

[MN08] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 4(3):article 32, 2008. 38 pages.

[MNS04] V. Mäkinen, G. Navarro, and K. Sadakane. Advantages of backward searching — efficient secondary memory and distributed implementation of compressed suffix arrays. In *Proc. 15th Annual International Symposium on Algorithms and Computation (ISAAC)*, 2004.

[MR97] I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. 38th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 118–126, 1997.

[MR04] I. Munro and S. Rao. Succinct representations of functions. In *Proc. 31th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 1006–1015, 2004.

[MRRR03] I. Munro, R. Raman, V. Raman, and S. Rao. Succinct representations of permutations. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 345–356, 2003.

[Mun96] I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.

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

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

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

[Pag99] R. Pagh. Low redundancy in dictionaries with $O(1)$ worst case lookup time. In *Proc. 26th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 595–604, 1999.

[PD06] M. Patrascu and E. Demaine. Logarithmic lower bounds in the cell probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.

[PST07] S. Puglisi, W. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):article 4, 2007.

[RR03] R. Raman and S. Rao. Succinct dynamic dictionaries and trees. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 2719, pages 357–368, 2003.

[RRR02] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[Sad00] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. 11th Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS 1969, pages 410–421, 2000.

[Sad02] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 225–232, 2002.

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

[SG06a] K. Sadakane and R. Grossi. Personal communication, 2006.

[SG06b] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1230–1239, 2006.

[Sha48] C.E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423,623–656, 1948.

[SPMT08] R. Sinha, S. Puglisi, A. Moffat, and A. Turpin. Improving suffix array locality for fast pattern matching on disk. In *Proc. 28th ACM International Conference on Management of Data (SIGMOD)*, pages 661–672, 2008.

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

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

[ZMNBY00] N. Ziviani, E. Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, 2000.