

UNIVERSITÀ DEGLI STUDI DI PISA

DIPARTIMENTO DI INFORMATICA  
DOTTORATO DI RICERCA IN INFORMATICA  
SETTORE SCIENTIFICO DISCIPLINARE INF/01

PH.D. THESIS

# **On searching and extracting strings from compressed textual data**

Rossano Venturini

SUPERVISOR

Prof. Paolo Ferragina

REFEREE

Prof. Ricardo Baeza-Yates

REFEREE

Prof. Jeffrey S. Vitter



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Basic concepts</b>	<b>7</b>
2.1	Models of computation . . . . .	7
2.1.1	RAM Model . . . . .	8
2.1.2	External-Memory Model . . . . .	8
2.2	Notation . . . . .	9
2.3	Classical full-text indexes . . . . .	9
2.3.1	Suffix tree . . . . .	10
2.3.2	Suffix array . . . . .	12
2.4	Compression . . . . .	13
2.4.1	Empirical entropy . . . . .	13
2.4.2	Burrows-Wheeler Transform . . . . .	15
<b>3</b>	<b>Optimally partitioning a text to improve its compression</b>	<b>19</b>
3.1	The PPC-paradigm and motivations for optimal partitioning . .	20
3.2	The problem and our solution . . . . .	24
3.2.1	A pruning strategy . . . . .	25
3.2.2	Space and time efficient algorithms for generating $\mathcal{G}_\epsilon(T)$	26
3.3	On zero-th order compressors . . . . .	28
3.3.1	On optimal partition and booster . . . . .	32

3.4	On $k$ -th order compressors . . . . .	33
3.5	On BWT-based compressors . . . . .	34
<b>4</b>	<b>Bit-complexity of Lempel-Ziv compression</b>	<b>39</b>
4.1	Notation and terminology . . . . .	42
4.2	An efficient and bit-optimal greedy parsing . . . . .	43
4.3	On the bit-efficiency of the greedy LZ77-parsing . . . . .	47
4.4	On Bit-Optimal Parsings and Shortest-Path problems . . . . .	48
4.4.1	A useful, small, subgraph of $\mathcal{G}(T)$ . . . . .	49
4.4.2	An efficient bit-optimal parser . . . . .	51
4.4.3	On the optimal construction of $\mathcal{T}_B$ . . . . .	57
4.5	Some experiments . . . . .	58
<b>5</b>	<b>Fast random access on compressed data</b>	<b>61</b>
5.1	Our storage scheme for strings . . . . .	62
5.2	Huffman on blocks of symbols . . . . .	65
5.3	BWT compression and access . . . . .	66
<b>6</b>	<b>Experiments on compressed full-text indexing</b>	<b>69</b>
6.1	Basics and background . . . . .	72
6.1.1	Backward Search . . . . .	73
6.1.2	Rank Query . . . . .	73
6.2	Compressed Indexes . . . . .	75
6.2.1	The FM-index Family . . . . .	75
6.2.2	Implementing the FM-index . . . . .	77
6.2.3	The Compressed Suffix Array (CSA) . . . . .	78
6.2.4	The Lempel-Ziv Index . . . . .	80
6.2.5	Novel Implementations . . . . .	81
6.3	The Pizza&Chili Site . . . . .	84
6.3.1	Indexes . . . . .	85
6.3.2	Texts . . . . .	86
6.4	Experimental Results . . . . .	89

6.4.1	Construction	90
6.4.2	Counting	91
6.4.3	Locate	92
6.4.4	Extract	96
6.4.5	Final comparison	96
<b>7</b>	<b>Dictionary indexes</b>	<b>101</b>
7.1	Background	104
7.2	Compressed Permuterm Index	105
7.2.1	A simple, but inefficient solution	105
7.2.2	A simple and efficient solution	106
7.3	Dynamic Compressed Permuterm Index	112
7.3.1	Deleting one dictionary string	113
7.3.2	Inserting one dictionary string	115
7.4	Experimental Results	117
7.5	Further considerations	122
<b>8</b>	<b>Future directions of research</b>	<b>123</b>
	<b>References</b>	<b>127</b>

---

CHAPTER

ONE

---

## Introduction

A large fraction of the data we process every day consists of a sequence of symbols over an alphabet, and hence is a *text*. Unformatted natural language documents, XML and HTML file collections, program codes, music sequences, DNA and protein sequences, are the typical examples that come to our mind when thinking of text incarnations. Although the scientific literature offers many solutions to the storage, access and search of textual data, the current growth and availability of massive amounts of texts gathered and processed by applications – Web search engines, textual and biological databases, just to cite a few – has changed the algorithmic requirements of these basic processing and mining tools and provide ample motivation for a great deal of new theoretical research on algorithms and data structures. In fact the memory hierarchies on current PCs and workstations are very complex because they consist of multiple levels: L1 and L2 caches, internal memory, one or more disks, other external storage devices (like CD-ROMs and DVDs), and memories of multiple hosts over a network. Although the virtualization of the memory permits the address space to be larger than the internal memory, it is well-known that not all memory references are equal. In fact each of these memory levels has its own cost, capacity, latency and bandwidth, and thus the memory accesses at the highest levels of the hierarchy are orders of magnitude faster than the accesses at the lowest levels. Therefore, applications working on large data sets should carefully organize their data in order to *help* the underlying Virtual Memory to guarantee efficient performance to the underlying applications. Knuth, in its famous *The Art of Computer Programming*, observed indeed that the *“space optimization is closely related to time optimization in a disk*

*memory*". Obviously this consideration is valid for all levels of the memory hierarchy.

In this scenario data compression seems mandatory because it may induce a twofold advantage: fitting more data into high (fast) memory levels reduces the transfer time from the slow levels, and may speed up the execution of algorithms. It goes without saying that storing data in compressed format is beneficial whenever the cost of accessing them out-weights their decompression time. Until recently data compression had its most important applications in areas of data transmission and storage: texts were compressed only for reducing their size and, except for some heuristic approaches, any access or search to their contents required their whole decompression [164]. Thus, the focus was primarily posed on designing compression schemes that achieve the maximum level of compression and that are usually very slow both in compression and decompression. Recently, a bunch of algorithmic tools and theoretical machineries have been made available to access compressed information without incurring in their whole decompression (e.g., see [134] and references therein). The exact meaning of access to compressed information deeply varies by application to application. Different applications usually ask to provide different operations on compressed text that range from allowing fast decompression of the whole text, to permitting random accesses to its substrings, up to answering sophisticated pattern matching queries on it. In literature is known a plethora of solutions: some of them support most of these operations while other are more specialized and provide just few of them. At a first glance, tools with fewer supported operations seems to be unnecessary since, for example, by providing efficient random access to substrings we also derive fast decompression of whole text. However, by designing specialized schemes we can obtain faster and/or highly compressed solutions. It is the case of compressed full-text indexes which are compressed tools that answer efficiently pattern matching queries on the indexed text [134]. Most of them also allow random accesses to substrings of the compressed text. However, these tools are slower than ad hoc solutions (e.g., [57, 79, 148]) in solving the latter operation primarily due to inefficiencies induced by their compression strategies. Observation above forced the research community to design different and specialized solutions that better fit in different applicative scenarios. These scenarios can be identified by observing the behavior of applications on the compressed data they manage. Sometimes data are accessed by applications only few times in their lifetime. We can, thus, admit some time inefficiencies on compression/decompression speed and focus our efforts on achieving best possible compression. Consider, as an example, applications that manage huge files that have to be stored on a disk for backup purposes or transmitted from an

host to an other through a network with small bandwidth. Other applications, instead, access data very often once it has been compressed. There are two possible pattern of accesses: the application performs its computation by scanning the whole text, even more times, from left to right, or by random accessing its substrings. For example, the first type of accesses may suffice in applications that compute some statistics on the text for mining purposes while the task of generating snippets in a search engine requires random accesses on compressed text. Finally, applications that manage texts require to perform some relative simple pattern matching queries on them. Such queries require to efficiently report or count the number of the occurrences of a given pattern on the underlying text. These queries usually suffice in many contexts but may be also used as building blocks whenever we require more sophisticated queries such as approximate pattern matching, searching with regular expressions, and so on.

We can summarize the contributions of this thesis with the following points that introduce our solutions for different applicative scenarios described above.

**Maximizing achieved compression.** As we said before, in some scenarios compression/decompression speed is not the main concern since we are more interested in maximizing the achieved compression. In Chapter 3 we describe a way to optimize the performance of a given compressor over an input text. More precisely, we investigate the problem of partitioning an input string  $T$  in such a way that compressing individually its parts via the given compressor gets a compressed output that is shorter than applying the compressor over the entire  $T$  at once. Our solution is the first known algorithm which is guaranteed to compute in  $O(n \log_{1+\epsilon} n)$  time a partition of  $T$  whose compressed output is guaranteed to be no more than  $(1 + \epsilon)$  times worse in size than the optimal one, where  $\epsilon$  is an arbitrary positive value.

The content of this chapter is based on paper [54].

**Fast whole decompression.** LZ77 compressor is a compression scheme that has been introduced by Lempel and Ziv about 30 years ago [166]. It has gained a lot of popularity due to its good compression performance and its very fast decompression algorithm which makes it the default choice in any scenario in which we often decompress the original text. In Chapter 4 we provide algorithms that permit to optimize the compression performance of LZ-based compressors. These theoretical results are sustained by some experiments that compare our novel LZ-based compressors against the most popular compression tools (like `gzip`, `bzip2`) and state-of-the-art compressors (like the *booster* of [41, 43]). These results show that our solutions significantly improve compression performance of other LZ-based compressors still retaining very fast



decompression speed.

The content of this chapter is based on paper [55].

**Random access to the compressed data.** The task of providing efficient random accesses to the compressed data is a key problem in many applications. For example, search engines and textual databases often need to extract pieces of data (e.g. web pages or records) from a compressed collection of files, without incurring in their whole decompression. Many solutions for lossless data compression (e.g. the ones discussed in the previous two points) fail in providing efficient random access to the compressed data and in achieving provable good compression ratio. In Chapter 5 we describe a compressed storage scheme for strings which improves known solutions in compression ratio and supports the retrieval of any of its substrings in optimal time. The importance of the problem is confirmed by the fact that our scheme has been used as building block in many subsequent papers to reduce space requirements of data structures and algorithms (e.g., see [12, 15, 27, 39, 61, 87, 91, 136]).

The content of this chapter is based on papers [57] and [58].

**Compressed full-text indexes.** Suffix trees and suffix arrays [83] are well-known classical full-text indexes that solve the so-called text searching problem efficiently in time, but they are greedy of space requiring  $\Theta(n \log n)$  bits to index a text of  $n$  symbols. In practice this means that such indexes are from 4 to 20 times larger than the input text, and thus their use is unfeasible for large data sets. Recent results [134] have shown astonishing and strong connections between indexing data structures and compressor design. This connection, at a first glance, might appear paradoxical because these tools have antithetical goals. In fact, index design aims at augmenting data with routing information (i.e. data structures) that allow the efficient retrieval of patterns or the extraction of some information. Conversely, compressors aim at removing the repetitiveness present in the data to squeeze them in a reduced space occupancy. The resulting *compressed full-text indexes* carefully combine ideas born in both fields to obtain the search power of suffix arrays/trees and a space occupancy close to the one achievable by the best known compressors, like `gzip` and `bzip2`. These studies were mainly at a theoretical stage. The content of Chapter 6 complemented those theoretical achievements with a significant experimental and algorithmic engineering effort. This is the first extensive experimental comparison among the most important compressed indexes known in the literature. This result has led to the design and develop of the *Pizza&Chili* site (<http://pizzachili.di.unipi.it>), that offers publicly available implementations of the best known compressed indexes, and a collection of texts and tools for experimenting and validating these indexes.

The content of this chapter is based on paper [44].

**Compressed indexes for dictionaries of strings.** Many applications require to index *dictionary of strings* instead of texts, like terms and URLs in web search engines. The main difference between these indexes and full-text ones relies in the basic operations to be supported: membership of a string, ranking of a string in the sorted dictionary, or selection of the  $i$ -th string from it. Tries are classical data structures providing those primitives but they require a large amount of extra space [104]. Since dictionaries of strings are getting larger and larger, it becomes crucial to devise implementations for the above primitives which are fast and work in compressed space. In Chapter 7 we describe a compressed index having these characteristics. In addition, we also devise a *dynamic* compressed index that is able to maintain the dictionary under insertions and deletions of an individual string. This theoretical study has been complemented with a rich set of experiments which have shown that our compressed index is also of practical relevance. In fact, it improves known approaches based on front-coding [121, 164] by more than 50% in absolute space occupancy, still guaranteeing comparable query time.

The content of this chapter is based on papers [56] and [59].

We conclude the thesis (Chapter 8) by presenting some of the most important and challenging problems of this fascinating area of research.



---

CHAPTER  
TWO

---

## Basic concepts

In this thesis we will present and solve problems concerning combinatorial pattern matching and data compression. In the last decade more than before researchers have shown that these two tools are strongly related. On one hand, pattern matching data structures and techniques are used to develop time efficient implementations of almost any data compression algorithm. On the other hand, many classical pattern matching data structures suffer of space inefficiencies that can be mitigated by reducing the redundancy present in the indexed text via data compression.

The aim of the present chapter is that of introducing the most important models of computation and defining useful notation. After that, we describe some well-known results on combinatorial pattern matching and data compression that will be often referred to in subsequent chapters.

The content of this chapter can be safely skipped by readers who are familiar with the fields of textual data compression and combinatorial pattern matching.

### 2.1 Models of computation

In order to reason about algorithms and data structures, we need *models of computation* that grasp the essence of real situations so that algorithms that are good in a model are also good in practice. The next subsections present two of the most important models: *RAM Model* and *External-Memory Model*.

### 2.1.1 RAM Model

The *RAM model* tries to model a realistic computer. RAM stands for Random Access Machine, which differentiates the model from classic but unrealistic computation models such as a tape-based Turing Machine. The machine is formed of a CPU, which executes primitive operations, and a memory, which stores the program and the data. The memory is divided in cells, called *words*, each having size  $w$  bits. It is usually assumed that  $\log n \leq w = \Theta(\log n)$ , where  $n$  is the size of the problem.<sup>1</sup> This assumption (usually referred as *trans-dichotomous assumption* [64]) is actually very realistic: a word must be large enough to store pointers and indices into the data, since otherwise we cannot even address the input. We can operate on words using at least a basic instruction set consisting of: Direct and indirect addressing, and a number of computational instructions, including addition, subtraction, multiplication, division, bitwise boolean operations and left and right shifts. Each of these operations requires a constant amount of time and can only manipulate  $O(1)$  words at a time.

### 2.1.2 External-Memory Model

The RAM model does not reflect the *memory hierarchy* of real computers that have more than one layer of memory, with different access characteristics. The *external-memory model* (or I/O model) was introduced by Aggarwal and Vitter in 1988 [1] to capture memory hierarchy with two layers. The model abstracts a computer which consists of two memory levels: a fast and small (internal) memory of size  $M$ , and a slow but potentially unbounded disk. Actually, this model can be used to abstract any two different layers in the memory hierarchy (e.g., disk and network). Both the memory and disk are divided into *blocks* of size  $B$ . The CPU can only operate directly on the data stored in memory, which consists of  $\frac{M}{B}$  blocks. Algorithms can make memory transfer operations, that is they can either read one block from disk to memory, or write one block from memory to disk. The cost of an algorithm is the number of memory transfers required to complete the task. In this model, thus, the cost of performing any other operation on data in memory is considered of secondary importance. Clearly any algorithm that has running time  $T(N)$  in the RAM model can be trivially converted into an external-memory algorithm that requires no more than  $T(N)$  memory transfers. However, faster algorithms can be often designed by carefully organizing and orchestrating accesses to data on the disk.

---

<sup>1</sup>Throughout this document we assume that all logarithms are taken to the base 2, whenever not explicitly indicated, and we assume  $0 \log 0 = 0$ .

## 2.2 Notation

It is convenient to fix a common notation regarding strings that we will use in the whole thesis. Let  $T[1, n]$  be a string drawn from an totally ordered alphabet  $\Sigma$  of size  $\sigma$ <sup>2</sup>. We will refer to the  $i$ th symbol of  $T$  as  $T[i]$ . Given any two positions  $i, j \in [n]$ <sup>3</sup> such that  $i \leq j$ , we will use  $T[i : j]$  to denote the substring  $T[i]T[i + 1] \dots T[j]$ . We will use  $T_i = T[i : n]$  to denote the  $i$ -th suffix of  $T$ .

In the thesis we will often compare strings resorting to their *lexicographic order*. The lexicographic ordering, denoted by  $\leq$ , is a total ordering induced by an ordering of symbols in the alphabet  $\Sigma$  which is defined as follows. Given two strings  $S$  and  $S'$  drawn from alphabet  $\Sigma$ , we say that  $S$  is lexicographically smaller than  $S'$  ( $S < S'$ ) if and only if  $S[1] < S'[1]$  or both  $S[1] = S'[1]$  and  $S_1 < S'_1$ . The empty string is considered lexicographically smaller than any non-empty string.

## 2.3 Classical full-text indexes

The aim of this section is that of introducing the *Text Searching Problem* and classical full-text indexes that will be intensively referred in the next chapters. The Text Searching Problem is stated as follows.

**Problem 1.** *Given a text  $T[1, n]$  and a pattern  $P[1, p]$ , we wish to answer the following queries:*

1.  $\text{COUNT}(P)$  that returns the number of occurrences (*occ*) of  $P$  in  $T$ ;
2.  $\text{LOCATE}(P)$  that reports the *occ* positions in  $T$  where  $P$  occurs.

In literature are known several algorithms to solve this problem in linear time via a sequential scan of  $T$  [83]. Despite the increase in processing speeds, sequential text searching long ago ceased to be a viable alternative for many applications, and indexed text searching has become mandatory. A *(full-)text index* is a data structure built over a text  $T$  which significantly speeds up sub-sequential searches for *arbitrary* pattern strings. This speed up came at the cost of additional space consumption (namely, the space required to store the index). Many different indexing data structures have been proposed in the literature, most notably suffix trees and suffix arrays (e.g., see [2, 83] and references therein).

---

<sup>2</sup>Unless otherwise stated, in most of this thesis we assume that  $\sigma = O(\text{poly}(n))$ .

<sup>3</sup> $[n]$  denotes the set  $\{1, 2, \dots, n\}$ .

### 2.3.1 Suffix tree

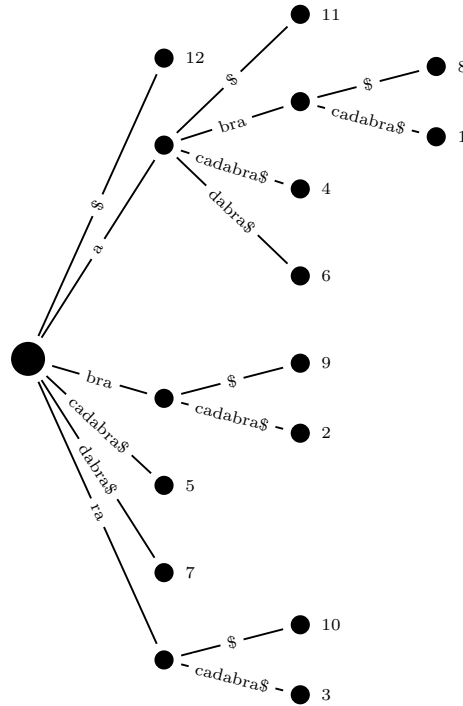
Let  $\mathcal{S}$  be a sorted set of strings drawn from an alphabet  $\Sigma$  such that no string is a proper prefix of any other string. The (*compact*) *trie* of  $\mathcal{S}$  is the rooted ordered tree defined as follows:

- each edge is labeled with a non-empty string;
- the labels of any two edges leaving the same node begin with different symbols;
- all internal nodes, except possibly the root, are branching;
- the tree has  $|\mathcal{S}|$  leaves;
- the  $i$ -th leaf is associated with the  $i$ -th lexicographic smaller string  $S$  of  $\mathcal{S}$  and the concatenation of the labels on the path from the root to this leaf is exactly equal to  $S$ .

The *suffix tree* of a text  $T[1, n]$  is the compacted trie, denoted as  $\mathcal{ST}(T)$  or simply  $\mathcal{ST}$ , built on all the  $n$  suffixes of  $T$ . We can ensure that no suffix is a proper prefix of another suffix by simply assuming that a special symbol, say \$, terminates the text  $T$ . The symbol \$ does not appear anywhere else in  $T$  and is lexicographically smaller than any other symbol in  $\Sigma$ . This constraint immediately implies that each suffix of  $T$  has its own unique leaf in the suffix tree, since any two suffixes of  $T$  will eventually follow separate branches in the tree.

For a given edge, the *edge label* is simply the substring in  $T$  corresponding to the edge. For edge between nodes  $u$  and  $v$  in  $\mathcal{ST}$ , the edge label (denoted  $label(u, v)$ ) is always a non-empty substring of  $T$ . For a given node  $u$  in the suffix tree, its *path label* (denoted  $pathlabel(u)$ ) is defined as the concatenation of edge labels on the path from the root to  $u$ . The *string depth* of node  $u$  is simply  $|pathlabel(u)|$ . For any two suffixes  $T_i$  and  $T_j$ , if  $w$  is their longest common prefix, then there exists a node  $u$  in  $\mathcal{ST}$  whose path label is equal to  $w$ . This node  $u$  is the lowest common ancestor of the two leaves labeled  $i$  and  $j$ . In the following, we will use  $Lcp(T_i, T_j)$  to denote the length of  $w$ . Figure 2.1 shows the suffix tree of the example text  $T = abracadabra\$$ .

In order to allow a linear space representation of the tree, each edge label is represented by a pair of integers denoting, respectively, the starting and ending positions in the  $T$  of the substring describing the edge label. If the edge label corresponds to a repeat substring, the indices corresponding to any these occurrences could be used. In this way, the suffix tree can be stored in  $\Theta(n \log n)$  bits of space which may be, however, much more than the  $n \lceil \log \sigma \rceil$  bits needed



**Figure 2.1:** The suffix tree of the example text  $T = abracadabra\$$ .

to represent text itself. We notice that this space bound is not optimal for any  $\sigma$ , since there are just  $\sigma^n$  different possible strings while  $n \log n$  bits are suffice to represent  $n^n$  different elements.

An optimal algorithm for building the suffix tree of a text  $T$  is the elegant solution by Farach-Colton [37] which requires  $O(n)$  time<sup>4</sup>.

In order to solve the Text Searching Problem with the suffix tree we observe that if a pattern  $P$  occurs in  $T$  starting from position  $i$ , then  $P$  is a prefix of  $T_i$ . This implies that the searching algorithm should identify the highest node  $u$  in  $\mathcal{ST}$  such that  $P$  is prefix of  $pathlabel(u)$ . From this observation we can derive the following algorithm for  $COUNT(P)$ : Start from the root of  $\mathcal{ST}$  and follow the path matching symbols of  $P$ , until a mismatch occurs or  $P$  is completely matched. In the former case  $P$  does not occur in  $T$ . In the latter case, each leaf in the subtree below the matching position gives an occurrence of  $P$ . This algorithm counts the  $occ$  occurrences of any pattern  $P[1, p]$  in time  $O(p \log \sigma)$ . These positions can be located by traversing the subtree in time proportional to its size. It is easy to see that this size is  $O(occ)$ . The complexity of counting can be reduced to  $O(p)$  by placing a (minimal) perfect hashing function [84] in each node to speed up percolation. This will increase the space just by a

<sup>4</sup>Recall the assumption  $\sigma = O(\text{poly}(n))$ .



constant factor.

For further details, the reader can consult the books by Gusfield [83] and, Crochemore and Rytter [34] that provide a comprehensive treatment of suffix trees, their construction, and their applications.

### 2.3.2 Suffix array

The *suffix array* [119] is a compact version of the suffix tree, obtained by storing in a array  $SA[1, n]$  the starting positions of the suffixes of  $T$  listed in lexicographic order. As the suffix tree, this array requires  $\Theta(n \log n)$  bits in the worst case. The main practical advantage is given by the fact that the constant hidden in the big-Oh notation is smaller (namely, it is less than 4).  $SA$  can be obtained by traversing the leaves of the suffix tree, or it can be built directly in optimal linear time via ad-hoc sorting methods [83, 139]. Since any substring of  $T$  is the prefix of a text suffix, the solution to the Text Searching Problem consists in finding the interval of positions in  $SA$  corresponding all text suffixes that start with  $P$ . Once this interval  $SA[sp, ep]$  has been identified,  $COUNT(P)$  is solved by returning the value  $occ = ep - sp + 1$ , and  $LOCATE(P)$  is solved by retrieving the entries  $SA[sp], SA[sp + 1], \dots, SA[ep]$ . The interval  $SA[sp, ep]$  can be binary searched in  $O(p \log n)$  time, since each binary search step requires to compare up to  $p$  symbols of a text suffix and the pattern. This time can be reduced to  $O(p + \log n)$  by using an auxiliary array called  $LCP$  that doubles the space requirement of the suffix array. The array  $LCP[1, n]$  essentially captures information on the heights of the internal nodes in the suffix tree of  $T$ . It is defined such that its entry  $LCP[i]$  is equal to the length of the longest common prefix of the  $(i - 1)$ -st and  $i$ -th lexicographically smallest suffixes in  $T$  (namely,  $LCP[i] = \text{Lcp}(T_{SA[i-1]}, T_{SA[i]})$ ). The  $LCP$  array can be computed in linear time starting from the suffix array and, in conjunction with an auxiliary data structure to solve *Range Minimum Queries* (RMQ), it suffices to compute the length of longest common prefix between any pair of suffixes of  $T$ .

$SA$	12	11	8	1	4	6	9	2	5	7	10	3
$SA^{-1}$	4	8	12	5	9	6	10	3	7	11	2	1
$LCP$	–	0	1	4	1	1	0	3	0	0	0	2

Figure 2.2: The table shows  $SA$ ,  $SA^{-1}$  and  $LCP$  arrays for the example text  $T = \text{abracadabra\$}$ .

We conclude this section by introducing the *inverse suffix array*. Even if it is not directly related to the Text Searching Problem, it will be often referred in

the next chapters. The *inverse suffix array*, denoted as  $SA^{-1}$ , is an array of  $n$  elements defined as

$$SA^{-1}[i] = j \iff SA[j] = i.$$

In other words, since  $SA$  is just a permutation of  $[n]$ ,  $SA^{-1}$  is defined to be its inverse. Therefore,  $SA^{-1}[i] = j$  tells us that suffix  $T_i$  is the  $j$ -th suffix in lexicographic order. We finally notice that  $SA$  and  $SA^{-1}$  can be easily computed one from the other in linear time. Table ?? shows  $SA$ ,  $SA^{-1}$  and  $LCP$  arrays for the example text  $T = \text{abracadabra\$}$ .

## 2.4 Compression

In this section we present some concepts related to compression that will be very useful in the next pages. In particular, in Section 2.4.1 we introduce the notion of *empirical entropy* which is a well-known measure of the compressibility of a text. This measure will be used in almost all the chapters either to optimize the performance of different compressors (Chapter 3) or to establish upper bounds to performance of compression schemes or compressed indexes (Chapters 5–7). In Section 2.4.2 we introduce the *Burrows-Wheeler Transform* which is an amazing algorithm for data compression. Even if we will introduce some other compression algorithms in the next chapters, we decided to introduce this tool earlier since it plays a central rule in all the thesis.

### 2.4.1 Empirical entropy

The *empirical entropy* resembles the entropy defined in the probabilistic setting (for example, when the input comes from a Markov source), but now it is defined for any finite individual string and can be used to measure the performance of compression algorithms without any assumption on the input distribution [123].

The 0-th order empirical entropy is currently a well-established measure of compressibility for a single string [123], and it is defined as follows. For each  $c \in \Sigma$ , we let  $n_c$  be the number of occurrences of  $c$  in  $T$ . The zero-th order *empirical entropy* of  $T$  is defined as

$$H_0(T) = \frac{1}{|T|} \sum_{c \in \Sigma} n_c \log \frac{n}{n_c}. \quad (2.4.1)$$

Note that  $|T|H_0(T)$  provides an information-theoretic lower bound to the out-

put size of any compressor that encodes each symbol of  $T$  with a fixed code [164]. The so-called zero-th order statistical compressors (such as Huffman or Arithmetic [164]) achieve an output size which is very close to this bound. However, they require to know information about frequencies of input symbols (called the *model* of the source). Those frequencies can be either known in advance (*static* model) or computed by scanning the input text (*semistatic* model). In both cases the model must be stored in the compressed file to be used by the decompressor. The compressed size achieved by zero-th order compressors over  $T$  is bounded by  $|\mathcal{C}_0(T)| \leq \lambda n H_0(T) + f_0(n, \sigma)$  bits, where  $\lambda$  is a positive constant and  $f_0(n, \sigma)$  is a function including the extra costs of encoding the source model and/or other inefficiencies of  $\mathcal{C}$ . As an example, for Huffman  $f_0(n, \sigma) = \sigma \log \sigma + O(\sigma) + n$  bits and  $\lambda = 1$ , and for Arithmetic  $f_0(n, \sigma) = O(\sigma \log n)$  bits and  $\lambda = 1$ .

In order to evict the cost of the model, we can resort to zero-th order *adaptive* compressors that do not require to know the symbols' frequencies in advance, since they are computed incrementally during the compression. The zero-th order *adaptive empirical* entropy of  $T$  [90] is then defined as

$$H_0^a(T) = \frac{1}{|T|} \sum_{c \in \Sigma} \log \frac{n!}{n_c!}. \quad (2.4.2)$$

The compress size achieved by zero-th order adaptive compressors over  $T$  is bounded by  $|\mathcal{C}_0^a(T)| \leq n H_0^a(T) + f_0(n, \sigma)$  bits where  $f_0(n, \sigma)$  is a function including inefficiencies of  $\mathcal{C}$ .

Let us now come to more powerful compressors. For any string  $u$  of length  $k$ , we denote by  $u_T$  the string of single symbols following the occurrences of  $u$  in  $T$ , taken from left to right. For example, if  $T = \text{abracadabra\$}$  and  $u = \text{ab}$ , we have  $u_T = \text{rr}$  since the two occurrences of  $\text{ab}$  in  $T$  are both followed by symbol  $\text{r}$ . The  $k$ -th order *empirical* entropy of  $T$  is defined as

$$H_k(T) = \frac{1}{|T|} \sum_{u \in \Sigma^k} |u_T| H_0(u_T). \quad (2.4.3)$$

Analogously, the  $k$ -th order *adaptive empirical* entropy of  $T$  is defined as

$$H_k^a(T) = \frac{1}{|T|} \sum_{u \in \Sigma^k} |u_T| H_0^a(u_T). \quad (2.4.4)$$

We have  $H_k(T) \geq H_{k+1}(T)$  for any  $k \geq 0$ . As usual in data compression [123], the value  $n H_k(T)$  is an information-theoretic lower bound to the output size of any compressor that encodes each symbol of  $T$  with a fixed code

that depends on the symbol itself and on the  $k$  immediately preceding symbols. Recently (see e.g. [42, 43, 48, 107, 116, 123] and references therein) authors have provided *upper bounds* in terms of  $H_k(|T|)$  for sophisticated data compression algorithms, such as dictionary based [107], Bwt-based [43, 94, 123], and PPM-like. These bounds have the form  $|\mathcal{C}(T)| \leq \lambda|T| H_k(T) + f_k(|T|, \sigma)$ , where  $\lambda$  is a positive constant and  $f_k(|T|, \sigma)$  is a function including the extra-cost of encoding the source model and/or other inefficiencies of  $\mathcal{C}$ . The smaller are  $\lambda$  and  $f_k()$ , the better is the compressor  $\mathcal{C}$ . As an example, the bound of the compressor in [116] has  $\lambda = 1$  and  $f(|T|, \sigma) = O(\sigma^{k+1} \log |T| + |T| \log \sigma \log \log |T| / \log |T|)$ . Similar bounds that involve the adaptive  $k$ -th order entropy are known [42, 43, 123] for many compressors. In these cases the bound takes the form  $|\mathcal{C}_k^a(T)| \leq \lambda|T|H_k^*(T) + f_k(|T|, \sigma)$  bits where  $f_k(|T|, \sigma)$  is a function including the inefficiencies of  $\mathcal{C}$ .

### 2.4.2 Burrows-Wheeler Transform

In [24] Burrows and Wheeler introduced a new compression algorithm based on a reversible transformation, now called the *Burrows-Wheeler Transform* (BWT from now on). The BWT transforms the input string  $T$  into a new string that is easier to compress. The BWT of  $T$ , hereafter denoted by  $\text{Bwt}(T)$  or simply  $\text{Bwt}$ , is built with three basic steps (see Figure 2.3):

1. append at the end of  $T$  a special symbol  $\$$  smaller than any other symbol of  $\Sigma$ ;
2. form a *conceptual* matrix  $\mathcal{M}_T$  whose rows are the cyclic rotations of string  $T\$$  in lexicographic order;
3. construct string  $L$  by taking the last column of the sorted matrix  $\mathcal{M}_T$ . We set  $\text{Bwt}(T) = L$ .

Every column of  $\mathcal{M}_T$ , hence also the transformed string  $L$ , is a permutation of  $T\$$ . In particular the first column of  $\mathcal{M}_T$ , call it  $F$ , is obtained by lexicographically sorting the symbols of  $T\$$  (or, equally, the symbols of  $L$ ). Note that the sorting of the rows of  $\mathcal{M}_T$  is essentially equal to the sorting of the suffixes of  $T$ , because of the presence of the special symbol  $\$$ . This shows that: (1) symbols following the same substring (*context*) in  $T$  are grouped together in  $L$ , and thus give raise to clusters of nearly identical symbols; (2) there is an obvious relation between  $\mathcal{M}_T$  and  $SA$ . Property 1 is the key for devising modern data compressors (see e.g. [123]), Property 2 is crucial for designing compressed indexes (see e.g. [134]) and, additionally, suggests a way to compute the  $\text{Bwt}$

		F	L
abracadabra\$		\$ abracadabr a	
bracadabra\$a		a \$abracadabr	
racadabra\$ab		a bra\$abraca d	
acadabra\$abr		a bracadabra \$	
cadabra\$abra		a cadabra\$abr	
adabra\$abrac	$\implies$	a dabra\$abra c	
dabra\$abraca		b ra\$abracad a	
abra\$abracad		b racadabra\$ a	
bra\$abracada		c adabra\$abr a	
ra\$abracadab		d abra\$abrac a	
a\$abracadabr		r a\$abracada b	
\$abracadabra		r acadabra\$a b	

**Figure 2.3:** Example of Burrows-Wheeler transform for the string  $T = \text{abracadabra}\$$ . The matrix on the right has the rows sorted in lexicographic order. The output of the BWT is the column  $L = \text{ard}\$rcaaaabb$ .

through the construction of the suffix array of  $T$ :  $L[0] = T[n]$  and, for any  $1 \leq i \leq n$ , set  $L[i] = T[SA[i] - 1]$ .

Burrows and Wheeler [24] devised two properties for the invertibility of the Bwt:

- (a) Since the rows in  $\mathcal{M}_T$  are cyclically rotated,  $L[i]$  precedes  $F[i]$  in the original string  $T$ .
- (b) For any  $c \in \Sigma$ , the  $\ell$ -th occurrence of  $c$  in  $F$  and the  $\ell$ -th occurrence of  $c$  in  $L$  correspond to the *same* character of the string  $T$ .

As a result, the original text  $T$  can be obtained backwards from  $L$  by resorting to function  $LF$  (also called Last-to-First column mapping or LF-mapping) that maps row indexes to row indexes, and is defined as:

$$LF(i) = C[L[i]] + \text{RANK}_{L[i]}(L, i),$$

where  $C[L[i]]$  counts the number of occurrences in  $T$  of symbols smaller than  $L[i]$  and  $\text{RANK}_{L[i]}(L, i)$  is a function that returns the number of times symbol  $L[i]$  occurs in the prefix  $L[1 : i]$ . We talk about LF-mapping because the symbol  $c = L[i]$  is located in the first column of  $\mathcal{M}_T$  at position  $LF(i)$ . The LF-mapping allows one to navigate  $T$  backwards: if  $T[k] = L[i]$ , then  $T[k - 1] = L[LF(i)]$

---

because row  $LF(i)$  of  $\mathcal{M}_T$  starts with  $T[k]$  and thus ends with  $T[k - 1]$ . In this way, we can reconstruct  $T$  backwards by starting at the first row, equal to  $\$T$ , and repeatedly applying  $LF$  for  $n$  steps. As an example, see Figure 2.3 in which the 3rd  $a$  in  $L$  lies onto the row which starts with `bracadabra$` and, correctly, the 3rd  $a$  in  $F$  lies onto the row which starts with `abracadabra$`. That symbol  $a$  is  $T[1]$ .



---

CHAPTER

THREE

---

## Optimally partitioning a text to improve its compression

Reorganizing data in order to improve the performance of a given compressor  $\mathcal{C}$  is a recent and important paradigm in data compression (see e.g. [22, 43]). The basic idea consists of *permuting* the input data  $T$  to form a new string  $T'$  which is then *partitioned* into substrings  $T' = T'_1 T'_2 \cdots T'_k$  that are finally compressed *individually* by the base compressor  $\mathcal{C}$ . The goal is to find the best instantiation of the two steps Permuting+Partitioning so that the compression of the individual substrings  $T'_i$  minimizes the total length of the compressed output. This approach (hereafter abbreviated as PPC) is clearly *at least* as powerful as the classic data compression approach that applies  $\mathcal{C}$  to the entire  $T$ : just take the identity permutation and set  $k = 1$ . The question is whether it can be *more powerful* than that!

Intuition leads to think favorably about it: by grouping together objects that are “related”, one can hope to obtain better compression even using a very weak compressor  $\mathcal{C}$ . Surprisingly enough, this intuition has been sustained by convincing theoretical and experimental results only recently. These results have investigated the PPC-paradigm under various angles by considering: different data formats (strings [43], trees [48], tables [22], etc.), different granularities for the items of  $T$  to be permuted (chars, node labels, columns, blocks [18, 108], files [26, 154, 157], etc.), different permutations (see e.g. [26, 72, 157, 160]), different base compressors to be boosted (0-th order compressors, `gzip`, `bzip2`, etc.). Among these plethora of proposals, we survey below the most notable examples which are useful to introduce the prob-



lem we attack in this chapter, and refer the reader to the cited bibliography for other interesting results.

### 3.1 The PPC-paradigm and motivations for optimal partitioning

The PPC-paradigm was introduced in [21], and further elaborated upon in [22]. In these papers  $T$  is a *table* formed by fixed size columns, and the goal is to permute the columns in such a way that individually compressing contiguous groups of them gives the shortest compressed output. The authors of [22] showed that the PPC-problem in its full generality is MAX-SNP hard, devised a link between PPC and the classical asymmetric TSP problem, and then resorted known *heuristics* to find approximate solutions based on several measures of correlations between the table's columns. For the grouping they proposed either an optimal but very slow approach, based on Dynamic Programming (see below), or some very simple and fast algorithms which however did not have any guaranteed bounds in terms of efficacy of their grouping process. Experiments showed that these heuristics achieve significant improvements over the classic `gzip`, when it is applied on the serialized original  $T$  (row- or column-wise). Furthermore, they showed that the combination of the TSP-heuristic with the DP-optimal partitioning is even better, but it is too slow to be used in practice even on small file sizes because of the DP-cubic time complexity.<sup>1</sup>

When  $T$  is a text string, the most famous instantiation of the PPC-paradigm has been obtained by combining the Burrows and Wheeler Transform [24] with a context-based grouping of the input symbols, which are finally compressed via proper 0-th order-entropy compressors (like Mtf, Rle, Huffman, Arithmetic, or their combinations, see e.g. [164]). Here the PPC-paradigm takes the name of *compression booster* [43] because the net result it produces is to boost the performance of the base compressor  $\mathcal{C}$  from 0-th order-entropy bounds to  $k$ -th order entropy bounds, simultaneously over all  $k \geq 0$ . In this scenario the permutation acts on single symbols, and the partitioning/permuting steps deploy the context (substring) following each symbol in the original string in order to identify “related” symbols which must be therefore compressed together. Recently [72] investigated whether there exist other permutations of the symbols

---

<sup>1</sup>Page 836 of [22] says: “computing a good approximation to the TSP reordering before partitioning contributes significant compression improvement at minimal time cost. [...] This time is negligible compared to the time to compute the optimal, contiguous partition via DP.”

of  $T$  which admit effective compression and can be computed/inverted fast. Unfortunately they found a connection between table compression and the Bwt, so that many natural similarity-functions between contexts turned out to induce MAX-SNP hard permuting problems! Interesting enough, the Bwt seems to be the unique highly compressible permutation which is fast to be computed and achieves effective compression bounds. Several other papers have given an analytic account of this phenomenon [42, 94, 116, 123] and have shown, also experimentally [41], that the partitioning of the BW-transformed data is a key step for achieving effective compression ratios. Optimal partitioning is actually even more mandatory in the context of labeled-tree compression where a BWT-inspired transform, called XBW-transform in [48, 49], allows to produce permuted strings with a strong clustering effect. Starting from these premises [73] attacked the computation of the optimal partitioning of  $T$  via a DP-approach, which turned to be very costly; then [43] (and subsequently many other authors, see e.g. [42, 48, 116]) proposed solutions which are not optimal but, nonetheless, achieve interesting  $k$ -th order-entropy bounds. This is indeed a subtle point which is frequently neglected when dealing with compression boosters, especially in practice, and for this reason we detail it more clearly in Subsection 3.3.1 in which we show an infinite class of strings for which the compression achieved by the booster is far from the optimal-partitioning by a multiplicative factor  $\Omega(\sqrt{\log n})$ .

Finally, there is another scenario in which the computation of the optimal partition of an input string for compression boosting can be successful and occurs when  $T$  is a single (possibly long) file on which we wish to apply classic data compressors, such as `gzip`, `bzip2`, PPM, etc. [164]. Note that how much redundancy can be detected and exploited by these compressors depends on their ability to “look back” at the previously seen data. However, such ability has a cost in terms of memory usage and running time, and thus most compression systems provide a facility that controls the amount of data that may be processed at once — usually called the *block size*. For example the classic tools `gzip` and `bzip2` have been designed to have a small memory footprint, up to few hundreds KBs. More recent and sophisticated compressors, like PPM [164] and the family of BWT-based compressors [41], have been designed to use block sizes of up to a few hundreds MBs. But using larger blocks to be compressed at once does not necessarily induce a better compression ratio! As an example, let us take  $\mathcal{C}$  as the simple Huffman or Arithmetic coders and use them to compress the text  $T = 0^{n/2}1^{n/2}$ : There is a clear difference whether we compress individually the two halves of  $T$  (achieving an output size of about  $O(\log n)$  bits) or we compress  $T$  as a whole (achieving  $n + O(\log n)$  bits). The impact of the block size is even more significant as

we use more powerful compressors, such as the  $k$ -th order entropy encoder PPM which compresses each symbol according to its preceding  $k$ -long context. In this case take  $T = (2^k 0)^{n/(2(k+1))} (2^k 1)^{n/(2(k+1))}$  and observe that if we divide  $T$  in two halves and compress them individually, the output size is about  $O(\log n)$  bits, but if we compress the entire  $T$  at once then the output size turns to be much longer, i.e.  $\frac{n}{k+1} + O(\log n)$  bits. Therefore the choice of the block size cannot be underestimated and, additionally, it is made even more problematic by the fact that it is not necessarily the same along the whole file we are compressing because it depends on the distribution of the repetitions within it. This problem is even more challenging when  $T$  is obtained by concatenating a collection of files via any permutation of them: think to the serialization induced by the Unix `tar` command, or other more sophisticated heuristics like the ones discussed in [26, 137, 154, 157]. In these cases, the partitioning step looks for *homogeneous* groups of contiguous files which can be effectively compressed together by the base-compressor  $\mathcal{C}$ . More than before, taking the largest memory-footprint offered by  $\mathcal{C}$  to group the files and compress them at once is not necessarily the best choice because real collections are typically formed by homogeneous groups of dramatically different sizes (e.g. think to a Web collection and its different kinds of pages). Again, in all those cases we could apply the optimal DP-based partitioning approach of [22, 73], but this would take more than cubic time (in the overall input size  $|T|$ ) thus resulting unusable even on small input data of few MBs.

In summary the efficient computation of an optimal partitioning of the input text for compression boosting is an important and still open problem of data compression (see [23]). The goal of our solution is to make a step forward by providing the first efficient approximation algorithm for this problem, formally stated as follows.

Let  $\mathcal{C}$  be the base compressor we wish to boost, and let  $T[1, n]$  be the input string we wish to partition and then compress by  $\mathcal{C}$ . So, we are assuming that  $T$  has been (possibly) permuted in advance, and we are concentrating on the last two steps of the PPC-paradigm. Now, given a partition  $\mathcal{P}$  of the input string into contiguous substrings, say  $T = T_1 T_2 \cdots T_k$ , we denote by  $\text{Cost}(\mathcal{P})$  the cost of this partition and measure it as  $\sum_{i=1}^k |\mathcal{C}(T_i)|$ , where  $|\mathcal{C}(\alpha)|$  is the length in bit of the string  $\alpha$  compressed by  $\mathcal{C}$ . The problem of *optimally partitioning*  $T$  according to the base-compressor  $\mathcal{C}$  consists then of computing the partition  $\mathcal{P}_{\text{opt}}$  achieving the minimum cost, namely  $\mathcal{P}_{\text{opt}} = \min_{\mathcal{P}} \text{Cost}(\mathcal{P})$ , and thus the shortest compressed output.<sup>2</sup>

---

<sup>2</sup>We are assuming that  $\mathcal{C}(\alpha)$  is a prefix-free encoding of  $\alpha$ , so that we can concatenate the compressed output of many substrings and still be able to recover them via a sequential scan.

As we mentioned above  $\mathcal{P}_{\text{opt}}$  might be computed via a Dynamic-Programming approach [22, 73]. Define  $E[i]$  as the cost of the optimum partitioning of  $T[1, i]$ , and set  $E[0] = 0$ . Then, for each  $i \geq 1$ , we can compute  $E[i]$  using the recurrence  $\min_{0 \leq j \leq i-1} E[j] + |\mathcal{C}(T[j+1 : i])|$ . At the end  $E[n]$  gives the cost of  $\mathcal{P}_{\text{opt}}$ , which can be explicitly determined by standard back-tracking over the DP-array. Unfortunately, this solution requires to run  $\mathcal{C}$  over  $\Theta(n^2)$  substrings of average length  $\Theta(n)$ , for an overall  $\Theta(n^3)$  time cost in the worst case which is clearly unfeasible even on small input sizes  $n$ .

In order to overcome this computational bottleneck we make two crucial observations: (1) instead of applying  $\mathcal{C}$  over each substring of  $T$ , we use an entropy-based estimation of  $\mathcal{C}$ 's compressed output that can be computed efficiently and incrementally by suitable dynamic data structures; (2) we relax the requirement for an exact solution to the optimal partitioning problem, and aim at finding a partition whose cost is no more than  $(1 + \epsilon)$  worse than  $\mathcal{P}_{\text{opt}}$ , where  $\epsilon$  may be any positive constant. Item (1) takes inspiration from the heuristics proposed in [21, 22], but it is executed in a more principled way because our entropy-based cost functions reflect the real behavior of modern compressors, and our dynamic data structures allow the efficient estimation of those costs without their re-computation from scratch at each substring (as instead occurred in [21, 22]). For item (2) it is convenient to resort to a well-known reduction from solutions of dynamic programming recurrences to Single Source Shortest path (SSSP) computation over weighted DAGs [35]. In our case, the solution for the optimal partitioning problem can be rephrased as a SSSP-computation over a weighted DAG consisting of  $n$  nodes and  $O(n^2)$  edges whose costs are derived from item (1). By exploiting some interesting structural properties of this graph, we are able to restrict the computation of that SSSP to a subgraph consisting of  $O(n \log_{1+\epsilon} n)$  edges only. The technical part of our solution (see Section 3.2) will show that we can build this graph on-the-fly as the SSSP-computation proceeds over the DAG via the proper use of time-space efficient dynamic data structures. The final result will be to show that we can  $(1 + \epsilon)$ -approximate  $\mathcal{P}_{\text{opt}}$  in  $O(n \log_{1+\epsilon} n)$  time and  $O(n)$  space, for both 0-th order compressors (like Huffman and Arithmetic [164]) and  $k$ -th order compressors (like PPM [164]). We will also extend these results to the class of BWT-based compressors, when  $T$  is a collection of texts.

We point out that the result on 0-th order compressors is interesting in its own from both the experimental side, since *Huffword* compressor is the standard choice for the storage of Web pages [164], and from the theoretical side since it can be applied to the compression booster of [43] to fast obtain an approximation of the optimal partition of  $\text{Bwt}(T)$  in  $O(n \log_{1+\epsilon} n)$  time. This may be better than the algorithm of [43] both in time complexity, since that takes

$O(n|\Sigma|)$  time where  $\Sigma$  is the alphabet of  $T$ , and in compression ratio (as we have shown above, see Subsection 3.3.1). The case of a large alphabet (namely,  $|\Sigma| = \Omega(\text{polylog}(n))$ ) is particularly interesting whenever we consider either a word-based Bwt [127] or the Xbw-transform over labeled trees [43]. Finally, we mention that our results apply also to the practical case in which the base compressor  $\mathcal{C}$  has a maximum (block) size  $B$  of data it can process at once (see above the case of `gzip`, `bzip2`, etc.). In this situation the time performance of our solution reduces to  $O(n \log_{1+\epsilon}(B \log \sigma))$ .

The map of the chapter is as follows. Section 3.2 describes reduction from the optimal partitioning problem of  $T$  to a SSSP problem over a weighted DAG in which edges represent substrings of  $T$  and edge costs are entropy-based estimations of the compression of these substrings via  $\mathcal{C}$ . After that, we show some properties of this DAG that permit our fast solution to the SSSP problem. The subsequent Sections will address the problem of incrementally and efficiently computing those edge costs as they are needed by the SSSP-computation, distinguishing the two cases of 0-th order estimators (Section 3.3) and  $k$ -th order estimators (Section 3.4), and the situation in which  $\mathcal{C}$  is a BWT-based compressor and  $T$  is a collection of files (Section 3.5).

## 3.2 The problem and our solution

In our solution we will use entropy-based upper bounds for the estimation of  $|\mathcal{C}(T[i : j])|$  described in Section 2.4.1. In what follows we will assume that the function  $f_0(n, \sigma)$  can be computed in constant time given  $n$  and  $\sigma$ . Even if we use these entropy-based bounds for the estimation of  $|\mathcal{C}(T[i : j])|$  instead of the real compress size, this will not be enough to achieve a fast DP-based algorithm for our optimal-partitioning problem. We cannot re-compute from scratch those estimates for every substring  $T[i : j]$  of  $T$ , being them  $\Theta(n^2)$  in number. So we will show below some structural properties of our problem and introduce few novel technicalities (Sections 3.3–3.4) that will allow us to compute  $H_k(T[i : j])$  only on a *reduced* subset of  $T$ 's substrings, having size  $O(n \log_{1+\epsilon} n)$ , by taking  $O(\text{polylog}(n))$  time per substring and  $O(n)$  space overall.

The optimal partitioning problem, stated in Section 3.1 can be reduced to a single source shortest path computation (SSSP) over a directed acyclic graph  $\mathcal{G}(T)$  defined as follows. The graph  $\mathcal{G}(T)$  has a vertex  $v_i$  for each text position  $i$  of  $T$ , plus an additional vertex  $v_{n+1}$  marking the end of the text, and an edge connecting vertex  $v_i$  to vertex  $v_j$  for any pair of indices  $i$  and  $j$  such that  $i < j$ . Each edge  $(v_i, v_j)$  has associated the cost  $c(v_i, v_j) = |\mathcal{C}(T[i : j - 1])|$  that

corresponds to the size in bits of the substring  $T[i : j - 1]$  compressed by  $\mathcal{C}$ . We remark the following crucial, but easy to prove, property of the cost function defined on  $\mathcal{G}(T)$ :

**Fact 1.** For any vertex  $v_i$ , it is  $0 < c(v_i, v_{i+1}) \leq c(v_i, v_{i+2}) \leq \dots \leq c(v_i, v_{n+1})$

There is a one-to-one correspondence between paths from  $v_1$  to  $v_{n+1}$  in  $\mathcal{G}(T)$  and partitions of  $T$ : every edge  $(v_i, v_j)$  in the path identifies a contiguous substring  $T[i : j - 1]$  of the corresponding partition. Therefore the cost of a path is equal to the (compression-)cost of the corresponding partition. Thus, we can find the optimal partition of  $T$  by computing the shortest path in  $\mathcal{G}(T)$  from  $v_1$  to  $v_{n+1}$ . Unfortunately this simple approach has two main drawbacks:

1. the number of edges in  $\mathcal{G}(T)$  is  $\Theta(n^2)$ , thus making the **SSSP** computation inefficient (i.e.  $\Omega(n^2)$  time) if executed directly over  $\mathcal{G}(T)$ ;
2. the computation of the each edge cost might take  $\Theta(n)$  time over most  $T$ 's substrings, if  $\mathcal{C}$  is run on each of them from scratch.

In the following sections we will successfully address both these two drawbacks. First, we sensibly reduce the number of edges in the graph  $\mathcal{G}(T)$  to be examined during the **SSSP** computation and show that we can obtain a  $(1 + \epsilon)$  approximation using only  $O(n \log_{1+\epsilon} n)$  edges, where  $\epsilon > 0$  is a user-defined parameter (Section 3.2.1). Second, we show some sufficient properties that  $\mathcal{C}$  needs to satisfy in order to compute efficiently every edge's cost. These properties hold for some well-known compressors— e.g. 0-order compressors, PPM-like and bzip-like compressors— and for them we show how to compute each edge cost in constant or polylogarithmic time (Sections 3.3—3.5).

### 3.2.1 A pruning strategy

The aim of this section is to design a *pruning* strategy that produces a subgraph  $\mathcal{G}_\epsilon(T)$  of the original DAG  $\mathcal{G}(T)$  in which the shortest path distance between its leftmost and rightmost nodes,  $v_1$  and  $v_{n+1}$ , increases by no more than a factor  $(1 + \epsilon)$ . We define  $\mathcal{G}_\epsilon(T)$  to contain all edges  $(v_i, v_j)$  of  $\mathcal{G}(T)$ , recall  $i < j$ , such that at least one of the following two conditions holds:

1. there exists a positive integer  $k$  such that  $c(v_i, v_j) \leq (1 + \epsilon)^k < c(v_i, v_{j+1})$ ;
2.  $j = n + 1$ .

In other words, by Fact 1, we are keeping for each integer  $k$  the edge of  $\mathcal{G}(T)$  that approximates at the best the value  $(1 + \epsilon)^k$  from below. Given this, we

will call  $\epsilon$ -maximal the edges of  $\mathcal{G}_\epsilon(T)$ . Clearly, each vertex of  $\mathcal{G}_\epsilon(T)$  has at most  $\log_{1+\epsilon} n = O(\frac{1}{\epsilon} \log n)$  outgoing edges, which are  $\epsilon$ -maximal by definition. Therefore the total size of  $\mathcal{G}_\epsilon(T)$  is at most  $O(\frac{n}{\epsilon} \log n)$ . Hereafter, we will denote with  $d_G(-, -)$  the shortest path distance between any two nodes in a graph  $G$ .

The following lemma states a basic property of shortest path distances over our special DAG  $\mathcal{G}(T)$ :

**Lemma 1.** *For any triple of indices  $1 \leq i \leq j \leq q \leq n + 1$  we have:*

1.  $d_{\mathcal{G}(T)}(v_j, v_q) \leq d_{\mathcal{G}(T)}(v_i, v_q)$
2.  $d_{\mathcal{G}(T)}(v_i, v_j) \leq d_{\mathcal{G}(T)}(v_i, v_q)$

**Proof.** We prove just 1, since 2 is symmetric. It suffices by induction to prove the case  $j = i + 1$ . Let  $(v_i, w_1)(w_1, w_2)\dots(w_{h-1}, w_h)$ , with  $w_h = v_q$ , be a shortest path in  $\mathcal{G}(T)$  from  $v_i$  to  $v_q$ . By fact 1,  $c(v_j, w_1) \leq c(v_i, w_1)$  since  $i \leq j$ . Therefore the cost of the path  $(v_j, w_1)(w_1, w_2)\dots(w_{h-1}, w_h)$  is at most  $d_{\mathcal{G}(T)}(v_i, v_q)$ , which proves the claim. ■

The correctness of our pruning strategy relies on the following theorem:

**Theorem 1.** *For any text  $T$ , the shortest path in  $\mathcal{G}_\epsilon(T)$  from  $v_1$  to  $v_{n+1}$  has a total cost of at most  $(1 + \epsilon) d_{\mathcal{G}(T)}(v_1, v_{n+1})$ .*

**Proof.** We prove a stronger assertion:  $d_{\mathcal{G}_\epsilon(T)}(v_i, v_{n+1}) \leq (1 + \epsilon) d_{\mathcal{G}(T)}(v_i, v_{n+1})$  for any index  $1 \leq i \leq n + 1$ . This is clearly true for  $i = n + 1$ , because in that case the distance is 0. Now let us inductively consider the shortest path  $\pi$  in  $\mathcal{G}(T)$  from  $v_i$  to  $v_{n+1}$  and let  $(v_k, v_{t_1})(v_{t_1}, v_{t_2})\dots(v_{t_h}, v_{n+1})$  be its edges. By the definition of  $\epsilon$ -maximal edge, it is possible to find an  $\epsilon$ -maximal edge  $(v_k, v_r)$  with  $t_1 \leq r$ , such that  $c(v_k, v_r) \leq (1 + \epsilon) c(v_k, v_{t_1})$ . By Lemma 1,  $d_{\mathcal{G}(T)}(v_r, v_{n+1}) \leq d_{\mathcal{G}(T)}(v_{t_1}, v_{n+1})$  while, by induction,  $d_{\mathcal{G}_\epsilon(T)}(v_r, v_{n+1}) \leq (1 + \epsilon) d_{\mathcal{G}(T)}(v_r, v_{n+1})$ . Combining this with the triangle inequality we get the thesis. ■

### 3.2.2 Space and time efficient algorithms for generating $\mathcal{G}_\epsilon(T)$

Theorem 1 ensures that, in order to compute a  $(1 + \epsilon)$  approximation of the optimal partition of  $T$ , it suffices to compute the SSSP in  $\mathcal{G}_\epsilon(T)$  from  $v_1$  to  $v_{n+1}$ . This can be easily computed in  $O(|\mathcal{G}_\epsilon(T)|) = O(n \log_\epsilon n)$  time since

$\mathcal{G}_\epsilon(T)$  is a DAG [31], by making a single pass over its vertices and relaxing all edges going out from the current one.

However, generating  $\mathcal{G}_\epsilon(T)$  in efficient time is a non-trivial task for three main reasons. First, the original graph  $\mathcal{G}(T)$  contains  $\Omega(n^2)$  edges, so that we cannot check each of them to determine whether it is  $\epsilon$ -maximal or not, because this would take  $\Omega(n^2)$  time. Second, we cannot compute the cost of an edge  $(v_i, v_j)$  by executing  $\mathcal{C}(T[i : j - 1])$  *from scratch*, since this would require time linear in the substring length, and thus  $\Omega(n^3)$  time over all  $T$ 's substrings. Third, we cannot materialize  $\mathcal{G}_\epsilon(T)$  (e.g. its adjacency lists) because it consists of  $\Theta(n \text{ polylog}(n))$  edges, and thus its space occupancy would be super-linear in the input size.

The rest of this section is devoted to design an algorithm which overcomes the three limitations above. The specialty of our algorithm consists of materializing  $\mathcal{G}_\epsilon(T)$  on-the-fly, as its vertices are examined during the SSSP-computation, by spending only polylogarithmic time per edge. The actual time complexity per edge will depend on the entropy-based cost function we will use to estimate  $|\mathcal{C}(T[i : j - 1])|$  and on the dynamic data structure we will deploy to compute that estimation efficiently.

The key tool we use to make a fast estimation of the edge costs is a dynamic data structure built over the input text  $T$  and requiring  $O(|T|)$  space. We state the main properties of this data structure in an abstract form, in order to design a general framework for solving our problem; in the next sections we will then provide implementations of this data structure and thus obtain real time/space bounds for our problem. So, let us assume to have a dynamic data structure that maintains a set of *sliding windows* over  $T$  denoted by  $w_1, w_2, \dots, w_{\log_{1+\epsilon} n}$ . The sliding windows are substrings of  $T$  which start at the same text position  $l$  but have different lengths: namely,  $w_i = T[l : r_i]$  and  $r_1 \leq r_2 \leq \dots \leq r_{\log_{1+\epsilon} n}$ . The data structure must support the following three operations:

1. REMOVE() moves the starting position  $l$  of all windows one position to the right (i.e.  $l + 1$ );
2. APPEND( $w_i$ ) moves the ending position of the window  $w_i$  one position to the right (i.e.  $r_i + 1$ );
3. SIZE( $w_i$ ) computes and returns the value  $|\mathcal{C}(T[l : r_i])|$ .

This data structure is enough to generate  $\epsilon$ -maximal edges via a single pass over  $T$ , using  $O(|T|)$  space. More precisely, let  $v_l$  be the vertex of  $\mathcal{G}(T)$  currently examined by our SSSP computation, and thus  $l$  is the current posi-



tion reached by our scan of  $T$ . We maintain the following invariant: the sliding windows correspond to all  $\epsilon$ -maximal edges going out from  $v_l$ , that is, the edge  $(v_l, v_{1+r_t})$  is the  $\epsilon$ -maximal edge satisfying  $c(v_l, v_{1+r_t}) \leq (1 + \epsilon)^t < c(v_l, v_{1+(r_t+1)})$ . Initially all indices are set to 0. To maintain the invariant, when the text scan advances to the next position  $l + 1$ , we call operation REMOVE() once to increment index  $l$  and, for each  $t = 1, \dots, \log_{1+\epsilon}(n)$ , we call operation APPEND( $w_t$ ) until we find the largest  $r_t$  such that  $\text{SIZE}(w_t) = c(v_l, v_{1+r_t}) \leq (1 + \epsilon)^t$ . The key issue here is that APPEND and SIZE are paired so that our data structure should take advantage of the rightward sliding of  $r_t$  for computing  $c(v_l, v_{1+r_t})$  efficiently. Just one symbol is entering  $w_t$  to its right, so we need to deploy this fact for making the computation of  $\text{SIZE}(w_t)$  fast (given its previous value). Here comes into play the second contribution of our solution that consists of adopting the entropy-bounded estimates for the compressibility of a string to estimate indeed the edge costs  $\text{SIZE}(w_t) = |\mathcal{C}(w_t)|$ . This idea is crucial because we will be able to show that these functions do satisfy some structural properties that admit a *fast incremental computation*, as the one required by APPEND + SIZE. These issues will be discussed in the following sections, here we just state that, overall, the SSSP computation over  $\mathcal{G}_\epsilon(T)$  takes  $O(n)$  calls to operation REMOVE, and  $O(n \log_{1+\epsilon} n)$  calls to operations APPEND and SIZE.

**Theorem 2.** *If we have a dynamic data structure occupying  $O(n)$  space and supporting operation REMOVE in time  $L(n)$ , and operations APPEND and SIZE in time  $R(n)$ , we can compute the shortest path in  $\mathcal{G}_\epsilon(T)$  from  $v_1$  to  $v_{n+1}$  taking  $O(n L(n) + (n \log_{1+\epsilon} n) R(n))$  time and  $O(n)$  space.*

### 3.3 On zero-th order compressors

In this section we explain how to implement the data structure above whenever  $\mathcal{C}$  is a 0-th order compressor, and thus  $H_0$  is used to provide a bound to the compression cost of  $\mathcal{G}(T)$ 's edges. The key point is actually to show how to efficiently compute  $\text{SIZE}(w_i)$  as the sum of  $|T[l : r_i]| H_0(T[l : r_i]) = \sum_{c \in \Sigma} n_c \log((r_i - l + 1)/n_c)$  (see its definition in Section 2.4.1) plus  $f_0(r_i - l + 1, |\Sigma_{T[l:r_i]}|)$ , where  $n_c$  is the number of occurrences of symbol  $c$  in  $T[l : r_i]$  and  $|\Sigma_{T[l:r_i]}|$  denotes the number of different symbols in  $T[l : r_i]$ .

The first solution we are going to present is very simple and uses  $O(\sigma)$  space per window. The idea is the following: for each window  $w_i$  we keep in memory an array of counters  $A_i[c]$  indexed by symbol  $c$  in  $\Sigma$ . At any step of our algorithm, the counter  $A_i[c]$  stores the number of occurrences of symbol  $c$  in

$T[l : r_i]$ . For any window  $w_i$ , we also use a variable  $E_i$  that stores the value of  $\sum_{c \in \Sigma} A_i[c] \log A_i[c]$ . It is easy to notice that:

$$|T[l : r_i]| H_0(T[l : r_i]) = (r_i - l + 1) \log(r_i - l + 1) - E_i. \quad (3.3.1)$$

Therefore, if we know the value of  $E_i$ , we can answer to a query  $\text{SIZE}(w_i)$  in constant time. So, we are left with showing how to implement efficiently the two operations that modify  $l$  or any  $r$ s value and, thus, modify appropriately the  $E$ 's value. This can be done as follows:

1. **REMOVE()**: For each window  $w_i$ , we subtract from the appropriate counter and from variable  $E_i$  the contribution of the symbol  $T[l]$  which has been evicted from the window. That is, we decrease  $A_i[T[l]]$  by one, and update  $E_i$  by subtracting  $(A_i[T[l]] + 1) \log(A_i[T[l]] + 1)$  and then summing  $A_i[T[l]] \log A_i[T[l]]$ . Finally we set  $l = l + 1$ .
2. **APPEND( $w_i$ )**: We add to the appropriate counter and variable  $E_i$  the contribution of the symbol  $T[r_i + 1]$  which has been appended to window  $w_i$ . That is, we increase  $A_i[T[r_i + 1]]$  by one, then we update  $E_i$  by subtracting  $(A_i[T[r_i + 1]] - 1) \log(A_i[T[r_i + 1]] - 1)$  and summing  $A_i[T[r_i + 1]] \log A_i[T[r_i + 1]]$ . Finally we set  $r_i = r_i + 1$ .

In this way, operation **REMOVE** requires constant time per window, hence  $O(\log_{1+\epsilon} n)$  time overall. **APPEND( $w_i$ )** takes constant time. The space required by the counters  $A_i$  is  $O(\sigma \log_{1+\epsilon} n)$  words. Unfortunately, the space complexity of this solution can be too much when it is used as the basic-block for computing the  $k$ -th order entropy of  $T$  (see Section 2.4.1) as we will do in Section 3.4. In fact, we would achieve  $\min(\sigma^{k+1} \log_{1+\epsilon} n, n \log_{1+\epsilon} n)$  space, which may be superlinear in  $n$  depending on  $\sigma$  and  $k$ .

The rest of this section is therefore devoted to provide an implementation of our dynamic data structure that takes the same query time above for these three operations, but within  $O(n)$  space, which is independent of  $\sigma$  and  $k$ . The new solution still uses  $E$ 's value but the counters  $A_i$  are computed on-the-fly by exploiting the fact that all windows share the same value of  $l$ . We keep an array  $B$  indexed by symbols whose entry  $B[c]$  stores the number of occurrences of  $c$  in  $T[1 : l]$ . We can keep these counters updated after a **REMOVE** by simply decreasing  $B[T[l]]$  by one. We also maintain an array  $R$  with an entry for each text position. The entry  $R[j]$  stores the number of occurrences of symbol  $T[j]$  in  $T[1 : j]$ . The number of elements in both  $B$  and  $R$  is no more than  $n$ , hence they take  $O(n)$  space.

These two arrays are enough to correctly update the value  $E_i$  after **APPEND( $w_i$ )**, which is in turn enough to estimate  $H_0$  (see Eqn 3.3.1). In fact, we can compute

the value  $A_i[T[r_i + 1]]$  by computing  $R[r_i + 1] - B[T[r_i + 1]]$  which correctly reports the number of occurrences of  $T[r_i + 1]$  in  $T[l : r_i + 1]$ . Once we have the value of  $A_i[T[r_i + 1]]$ , we can update  $E_i$  as explained in the above item 2.

We are left with showing how to support  $\text{REMOVE}()$  whose computation requires to evaluate the value of  $A_i[T[l]]$  for each window  $w_i$ . Each of these values can be computed as  $R[t] - B[T[l]]$  where  $t$  is the last occurrence of symbol  $T[l]$  in  $T[l : r_i]$ . The problem here is given by the fact that we do not know the position  $t$ . We solve this issue by resorting to a doubly linked list  $L_c$  for each symbol  $c$ . The list  $L_c$  links together the last occurrences of  $c$  in all those windows, ordered by increasing position. Notice that a position  $j$  may be the last occurrence of symbol  $T[j]$  for different (but consecutive) windows. In this case we force that position to occur in  $L_{T[j]}$  just once. These lists are sufficient to compute values  $A_i[T[l]]$  for all the windows together. In fact, since any position in  $L_{T[l]}$  is the last occurrence of at least one sliding window, each of them can be used to compute  $A_i[T[l]]$  for the appropriate indices  $i$ . Once we have all values  $A_i[T[l]]$ , we can update all  $E_i$ 's as explained in the above item 1. Since list  $L_{T[l]}$  contains no more than  $\log_{1+\epsilon} n$  elements, all  $E$ s can be updated in  $O(\log_{1+\epsilon} n)$  time. Notice that the number of elements in all the lists  $L$  is bounded by the text length. Thus, they are stored using  $O(n)$  space.

It remains to explain how to keep lists  $L$  correctly updated. Notice that only one list may change after a  $\text{REMOVE}()$  or an  $\text{APPEND}(w_i)$ . In the former case we have possibly to remove position  $l$  from list  $L_{T[l]}$ . This operation is simple because, if that position is in the list, then  $T[l]$  is the last occurrence of that symbol in  $w_1$  (recall that all the windows start at position  $l$ , and are kept ordered by increasing ending position) and, thus, it must be the head of  $L_{T[l]}$ . The case of  $\text{APPEND}(w_i)$  is more involved. Since the ending position of  $w_i$  is moved to the right, position  $r_i + 1$  becomes the last occurrence of symbol  $T[r_i + 1]$  in  $w_i$ . Recall that  $\text{APPEND}(w_i)$  inserts symbol  $T[r_i + 1]$  in  $w_i$ . Thus, it must be inserted in  $L_{T[r_i+1]}$  in its correct (sorted) position, if it is not present yet. Obviously, we can do that in  $O(\log_{1+\epsilon} n)$  time by scanning the whole list. This is too much, so we show how to spend only constant time. Let  $p$  the rightmost occurrence of the symbol  $T[r_i + 1]$  in  $T[0 : r_i]$ .<sup>3</sup> If  $p < l$ , then  $r_i + 1$  must be inserted in the front of  $L_{T[r_i+1]}$  and we have done. In fact,  $p < l$  implies that there is no occurrence of  $T[r_i + 1]$  in  $T[l : r_i]$  and, thus, no position can precede  $r_i + 1$  in  $L_{T[r_i+1]}$ . Otherwise (i.e.  $p \geq l$ ), we have that  $p$  is in  $L_{T[r_i+1]}$ , because it is the last occurrence of symbol  $T[r_i + 1]$  for some window  $w_j$  with  $j \leq i$ . We observe that if  $w_j = w_i$ , then  $p$  must be replaced by  $r_i + 1$  which is now the last occurrence of  $T[r_i + 1]$  in  $w_i$ ; otherwise  $r_i + 1$  must be inserted after  $p$  in

<sup>3</sup>Notice that we can precompute and store the last occurrence of symbol  $T[j + 1]$  in  $T[1 : j]$  for all  $j$ s in linear time and space.

$L_{T[r_i+1]}$  because  $p$  is still the last occurrence of this symbol in the window  $w_j$ . We can decide which one is the correct case by comparing  $p$  and  $r_{i-1}$  (i.e., the ending position of the preceding window  $w_{r_{i-1}}$ ). In any case, the list is kept updated in constant time.

The following Lemma derives by the discussion above:

**Lemma 2.** *Let  $T[1, n]$  be a text drawn from an alphabet of size  $\sigma = \text{poly}(n)$ . If we estimate  $\text{SIZE}()$  via 0-th order entropy, then we can design a dynamic data structure that takes  $O(n)$  space and supports the operations REMOVE in  $R(n) = O(\log_{1+\epsilon} n)$  time, and APPEND and SIZE in  $L(n) = O(1)$  time.*

In order to evict the cost of the model from the compressed output (see Section 2.4.1), authors typically resort to zero-th order *adaptive* compressors which do not store the symbols' frequencies, since they are computed *incrementally* during the compression [90]. A similar approach can be used in this case to achieve the same time and space bounds of Lemma 2. Here, we require that  $\text{SIZE}(w_i) = |\mathcal{C}_0^a(T[l : r_i])| = |T[l : r_i]|H_0^a(T[l : r_i])$ . Recall that with these type of compressors the model must not be stored. We use the same tools above but we change the values stored in variables  $E_i$  and the way in which they are updated after a REMOVE or an APPEND.

Observe that in this case we have that

$$|\mathcal{C}_0^a(T[l : r_i])| = |T[l : r_i]|H_0^a(T[l : r_i]) = \log((r_i - l + 1)!) - \sum_{c \in \Sigma} \log(n_c!)$$

where  $n_c$  is the number of occurrences of symbol  $c$  in  $T[l : r_i]$ . Therefore, if the variable  $E_i$  stores the value  $\sum_{c \in \Sigma} \log(A_i[c]!)$ , then we have that  $|T[l : r_i]|H_0^a(T[l : r_i]) = \log((r_i - l + 1)!) - E_i$ .<sup>4</sup>

After the two operations, we change  $E$ 's value in the following way:

1. REMOVE(): For any window  $w_i$  we update  $E_i$  by subtracting  $\log(A_i[T[l]])$ . We also increase  $l$  by one.
2. APPEND( $w_i$ ): We update  $E_i$  by summing  $\log A[T[r_i + 1]]$  and we increase  $r_i$  by one.

By the discussion above and Theorem 2 we obtain:

**Theorem 3.** *Given a text  $T[1, n]$  drawn from an alphabet of size  $\sigma = \text{poly}(n)$ , we can find an  $(1 + \epsilon)$ -optimal partition of  $T$  with respect to a 0-th order (adaptive) compressor in  $O(n \log_{1+\epsilon} n)$  time and  $O(n)$  space, where  $\epsilon$  is any positive constant.*

<sup>4</sup>Notice that the value  $\log((r_i - l + 1)!)$  can be stored in a variable and updated in constant time since the size of the value  $r_i - l + 1$  changes just by one after a REMOVE or an APPEND.

We point out that these results can be applied to the compression booster of [43] to fast obtain an approximation of the optimal partition of  $\text{Bwt}(T)$ . This may be better than the algorithm of [43] both in time complexity, since that algorithm took  $O(n\sigma)$  time, and in compression ratio by a factor up to  $\Omega(\sqrt{\log n})$  (see discussion and class of strings in Subsection 3.3.1). The case of a large alphabet (namely,  $\sigma = \Omega(\text{polylog}(n))$ ) is particularly interesting whenever we consider either a word-based  $\text{Bwt}$  [127] or the  $\text{Xbw}$ -transform over labeled trees [43]. We notice that our result is interesting also for the *Huffman* compressor which is the standard choice for the storage of Web pages [164]; here  $\Sigma$  consists of the distinct words constituting the Web-page collection.

### 3.3.1 On optimal partition and booster

An  $O(n\sigma)$ -exetime algorithm to partition  $\text{Bwt}(T)$  has been presented in [43]. Even if it achieves interesting  $k$ -th order-entropy bounds, there are cases in which their greedy algorithm does not find the optimal solution. This is indeed a subtle point which is frequently neglected when dealing with compression boosters, especially in practice. In this section we prove that there exists an infinite class of strings for which the partition selected by booster [43] is far from the optimal one by a factor  $\Omega(\sqrt{\log n})$ . Consider an alphabet  $\Sigma = \{c_1, c_2, \dots, c_\sigma\}$  and assume that  $c_1 < c_2 < \dots < c_\sigma$ . We divide it into  $l = \sigma/\alpha$  groups of  $\alpha$  consecutive symbols each, where  $\alpha > 0$  will be defined later. Let  $\Sigma_1, \Sigma_2, \dots, \Sigma_l$  denote these sub-alphabets. For each  $\Sigma_i$ , we build a De Bruijn sequence  $T_i$  in which each pair of symbols of  $\Sigma_i$  occurs exactly once. By construction each sequence  $T_i$  has length  $\alpha^2$ . Then, we define  $T = T_1 T_2 \dots T_l$ , so that  $|T| = \sigma\alpha$  and each symbol of  $\Sigma$  occurs exactly  $\alpha$  times in  $T$ . Therefore, the first column of  $\text{Bwt}$  matrix is equal to  $(c_1)^\alpha (c_2)^\alpha \dots (c_\sigma)^\alpha$ . We denote with  $L_c$  the portion of  $\text{Bwt}(T)$  that has symbol  $c$  as prefix in the  $\text{Bwt}$  matrix. By construction, if  $c \in \Sigma_i$ , we have that any  $L_c$  has either one occurrence of each symbol of  $\Sigma_i$  or one occurrence of these symbols of  $\Sigma_i$  minus one plus one occurrence of some symbol of  $\Sigma_{i-1}$  (or  $\Sigma_l$  if  $i = 1$ ). In both cases, each  $L_c$  has  $\alpha$  symbols, which are all distinct. Notice that by construction, the longest common prefix among any two suffixes of  $T$  is at most 1. Therefore, since the booster can partition only using prefix-close contexts (see [43]), there are just three possible partitions: (1) one substring containing all symbols of  $L$ , (2) one substring per  $L_c$ , or (3) as many substrings as symbols of  $L$ . Assuming that the cost of each model is at least  $\log \sigma$  bits<sup>5</sup>, then the costs of all possible booster's

<sup>5</sup>Here we assume that it contains at least one symbol. Nevertheless, as we will see, the compression gap between booster's partition and the optimal one grows as the cost of the

partitions are:

1. Compressing the whole  $L$  at once has cost at least  $\sigma\alpha \log \sigma$  bits. In fact, all the symbols in  $\Sigma$  have the same frequency in  $L$ .
2. Compressing each string  $L_c$  costs at least  $\alpha \log \alpha + \log \sigma$  bits, since each  $L_c$  contains  $\alpha$  distinct symbols. Thus, the overall cost for this partition is at least  $\sigma\alpha \log \alpha + \sigma \log \sigma$  bits.
3. Compressing each symbol separately has overall cost at least  $\sigma\alpha \log \sigma$  bits.

We consider the alternative partition which is not achievable by the booster that subdivides  $L$  into  $\sigma/\alpha^2$  substrings denoted  $S_1, S_2, \dots, S_{\sigma/\alpha^2}$  of size  $\alpha^3$  symbols each (recall that  $|T| = \sigma\alpha$ ). Notice that each  $S_i$  is drawn from an alphabet of size smaller than  $\alpha^3$ .

The strings  $S_i$  are compressed separately. The cost of compressing each string  $S_i$  is  $O(\alpha^3 \log \alpha^3 + \log \sigma) = O(\alpha^3 \log \alpha + \log \sigma)$ . Since there are  $\sigma/\alpha^2$  strings  $S_i$ s, the cost of this partition is  $P = O(\sigma\alpha \log \alpha + (\sigma/\alpha^2) \log \sigma)$ . Therefore, by setting  $\alpha = O(\sqrt{\log \sigma / \log \log \sigma})$ , we have that  $P = O(\sigma \sqrt{\log \sigma})$  bits. As far as the booster is concerned, the best compression is achieved by its second partition whose cost is  $O(\sigma \log \sigma)$  bits. Therefore, the latter is  $\Omega(\sqrt{\log \sigma})$  times larger than our proposed partition. Since  $\sigma \geq \sqrt{n}$ , the ratio among the two partitions is  $\Omega(\sqrt{\log n})$ .

### 3.4 On $k$ -th order compressors

In this section we make one step further and consider the more powerful  $k$ -th order compressors, for which there exist  $H_k$  bounds for estimating the size of their compressed output (see Section 2.4.1). Here  $\text{SIZE}(w_i)$  must compute  $|\mathcal{C}(T[l : r_i])|$  which is estimated by

$$(r_i - l + 1)H_k(T[l : r_i]) + f_k(r_i - l + 1, |\Sigma_{T[l:r_i]}|),$$

where  $\Sigma_{T[l:r_i]}$  denotes the number of different symbols in  $T[l : r_i]$ .

Let us denote with  $T_q[1 : n - q]$  the text whose  $i$ -th symbol  $T[i]$  is equal to the  $q$ -gram  $T[i : i + q - 1]$ . Actually, we can remap the symbols of  $T_q$  to integers in  $[n]$  without modifying its zero-th order entropy. In fact the number of distinct  $q$ -grams occurring in  $T_q$  is less than  $n$ , the length of  $T$ . Thus  $T_q$ 's symbols take

---

model becomes bigger.

$O(\log n)$  bits and  $T_q$  can be stored in  $O(n)$  space. This remapping takes linear time and space, whenever  $\sigma$  is polynomial in  $n$ .

A simple calculation shows that the  $k$ -th order (adaptive) entropy of a string (see definition Section 2.4.1) can be expressed as the difference between the zero-th order (adaptive) entropy of its  $k + 1$ -grams and its  $k$ -grams. This suggests that we can use the solution of the previous section in order to compute the zero-th order entropy of the appropriate substrings of  $T_{k+1}$  and  $T_k$ . More precisely, we use two instances of the data structure of Theorem 3 (one for  $T_{k+1}$  and one for  $T_k$ ), which are kept *synchronized* in the sense that, when operations are performed on one data structure, then they are also executed on the other.

**Lemma 3.** *Let  $T[1, n]$  be a text drawn from an alphabet of size  $\sigma = \text{poly}(n)$ . If we estimate  $\text{SIZE}()$  via  $k$ -th order entropy, then we can design a dynamic data structure that takes  $O(n)$  space and supports the operations REMOVE in  $R(n) = O(\log_{1+\epsilon} n)$  time, and APPEND and SIZE in  $L(n) = O(1)$  time.*

Essentially the same technique is applicable to the case of  $k$ -th order *adaptive* compressor  $\mathcal{C}$ , in this case we keep up-to-date the 0-th order *adaptive* entropies of the strings  $T_{k+1}$  and  $T_k$ .

**Theorem 4.** *Given a text  $T[1, n]$  drawn from an alphabet of size  $\sigma = \text{poly}(n)$ , we can find an  $(1 + \epsilon)$ -optimal partition of  $T$  with respect to a  $k$ -th order (adaptive) compressor in  $O(n \log_{1+\epsilon} n)$  time and  $O(n)$  space, where  $\epsilon$  is any positive constant.*

We point out that this result applies also to the practical case in which the base compressor  $\mathcal{C}$  has a maximum (block) size  $B$  of data it can process at once (this is the typical scenario for `gzip`, `bzip2`, etc.). In this situation the time performance of our solution reduces to  $O(n \log_{1+\epsilon}(B \log \sigma))$ .

### 3.5 On BWT-based compressors

In literature we know entropy-bounded estimates for the output size of BWT-based compressors [123]. So we could apply Theorem 4 to compute the optimal partitioning of  $T$  for such a type of compressors. Nevertheless, it is also known [41] that such compression-estimates are rough in practice because of the features of the compressors that are applied to the  $\text{Bwt}(T)$ -string. Typically,  $\text{Bwt}$  is encoded via a sequence of simple compressors such as Mtf encoding, Rle encoding (which is optional), and finally a 0-order encoder like Huffman or Arithmetic [164]. For each of these compression steps, a 0-th entropy bound is known [123], but the combination of these bounds may result much far from

the final compressed size produced by the overall sequence of compressors in practice [41].

In this section, we propose a solution to the optimal partitioning problem for BWT-based compressors that introduces a  $\Theta(\sigma \log n)$  slowdown in the time complexity of Theorem 4, but with the advantage of computing the  $(1 + \epsilon)$ -optimal solution wrt the real compressed size, thus without any estimation by any entropy-cost functions. Since in practice it is  $\sigma = \text{polylog}(n)$ , this slowdown should be negligible. In order to achieve this result, we need to address a slightly different (but yet interesting in practice) problem which is defined as follows. The input string  $T$  has the form  $S[1]\#_1S[2]\#_2 \dots S[m]\#_m$  where each  $S[i]$  is a text (called *page*) drawn from an alphabet  $\Sigma$ , and  $\#_1, \#_2, \dots, \#_m$  are special symbols greater than any symbol of  $\Sigma$ . A partition of  $T$  must be page-aligned, that is it must form *groups of contiguous pages*  $S[i]\#_i \dots S[j]\#_j$ , denoted also  $S[i : j]$ . Our aim is to find a page-aligned partition whose cost is at most  $(1 + \epsilon)$  the minimum possible cost, for any fixed  $\epsilon > 0$ . We notice that this problem generalizes the table partitioning problem [22], since we can assume that  $S[i]$  is a column of the table.

To simplify things we will drop the RLE encoding step of a Bwt-based algorithm. We start by noticing that a close analog of Theorem 2 holds for this variant of the optimal partitioning problem, which implies that a  $(1 + \epsilon)$ -approximation of the optimum cost (and the corresponding partition) can be computed using a data structure supporting operations APPEND, REMOVE, and SIZE; with the only difference that the windows  $w_1, w_2, \dots, w_m$  subject to the operations are groups of contiguous pages of the form  $w_i = S[l, r_i]$ .

It goes without saying that there exist data structures designed to dynamically maintain a dynamic text compressed with a Bwt-based compressor under insertions and deletions of symbols (see Chapter 7). But they do not fit our context for two reasons: (1) their underlying compressor is significantly different from the scheme above; (2) in the worst case, they would spend linear space per window yielding a super-linear overall space complexity.

Instead of keeping a given window  $w$  in compressed form, our approach only store the frequency distribution of the integers in the string  $w' = \text{Mtf}(\text{Bwt}(w))$  since this is enough to compute the compressed output size produced by the final step of the Bwt-based algorithm, which is usually implemented via Huffman or Arithmetic [164]. Indeed, since Mtf produces a sequence of integers from 0 to  $\sigma$ , we can store their number of occurrences for each window  $w_i$  into an array  $F_{w_i}$  of size  $\sigma$ . The update of  $F_{w_i}$  due to the insertion or the removal of a page in  $w_i$  incurs two main difficulties: (1) how to update  $w'_i$  as pages are added/removed from the extremes of the window  $w_i$ , (2) perform this up-



date implicitly over  $F_{w_i}$ , because of the space reasons mentioned above. Our solution relies on two key facts about **Bwt** and **Mtf**:

1. Since the pages are separated in  $T$  by distinct separators, inserting or removing one page into a window  $w$  does not alter the relative lexicographic order of the original suffixes of  $w$  (see [59]).
2. If a string  $s'$  is obtained from string  $s$  by inserting or removing a char  $c$  into an arbitrary position, then  $\text{Mtf}(s')$  differs from  $\text{Mtf}(s)$  in at most  $\sigma$  symbols. More precisely, if  $c'$  is the next occurrence in  $s$  of the newly inserted (or removed) symbol  $c$ , then the **Mtf** has to be updated only in the first occurrence of each symbol of  $\Sigma$  among  $c$  and  $c'$ .

We can now describe a data structure that supports operations  $\text{APPEND}(w)$  and  $\text{REMOVE}()$ . We assume that the separator symbols in the  $\text{Bwt}(T)$  are ignored by the **Mtf** step, which means that when the **Mtf** encoder finds a separator in  $\text{Bwt}(T)$ , this is replaced with the corresponding integer without altering the **Mtf**-list. This variant does not introduce any compression penalty (because every separator occurs just once) but simplifies the discussion that follows. Given a range  $I = [a, b]$  of positions of  $T$ , an occurrence of a symbol of  $\text{Bwt}(T)$  is called *active*<sub>[a,b]</sub> if it corresponds to a symbol in  $T[a : b]$ . For any range  $[a, b] \subset [n]$  of positions in  $T$ , we define  $\text{rBwt}(T[a : b])$  as the string obtained by concatenating the *active*<sub>[a,b]</sub> symbols of  $\text{Bwt}(T)$  by preserving their relative order. In the following, we will not indicate the interval when it will be clear from the context. Notice that, due to the presence of separators,  $\text{rBwt}(T[a : b])$  coincides with  $\text{Bwt}(T[a : b])$  when  $T[a : b]$  spans a group of contiguous pages (see Chapter 7). Moreover,  $\text{Mtf}(\text{rBwt}(T[a : b]))$  is the string obtained by performing the **Mtf** algorithm on  $\text{rBwt}(T[a : b])$ . We will call the symbol  $\text{Mtf}(\text{rBwt}(T[a : b]))[i]$  as the **Mtf**-encoding of the symbol  $\text{rBwt}(T[a : b])[i]$ .

For each window  $w$ , our solution will not explicitly store neither  $\text{rBwt}(w)$  or  $\text{Mtf}(\text{rBwt}(T[a : b]))$  since this might require a superlinear amount of space. Instead, we maintain only an array  $F_w$  of size  $\sigma$  whose entry  $F_w[e]$  keeps the number of occurrences of the encoding  $e$  in  $\text{Mtf}(\text{rBwt}(w))$ . The array  $F_w$  is enough to compute the 0-order entropy of  $\text{Mtf}(\text{rBwt}(w))$  in  $\sigma$  time (or eventually the exact cost of compressing it with Huffman in  $\sigma \log \sigma$  time).

We are left with showing how to correctly keep updated  $F_w$  after a  $\text{REMOVE}()$  or an  $\text{APPEND}(w)$ . In the following we will concentrate only on  $\text{APPEND}(w)$  since  $\text{REMOVE}()$  is symmetrical. The idea underlying the implementation of  $\text{APPEND}(w)$ , where  $w = S[l, r]$ , is to *conceptually insert* the symbols of the next page  $S[r + 1]$  into  $\text{rBwt}(w)$  one at a time from left to right. Since the relative order among the symbols of  $\text{rBwt}(w)$  is preserved in  $\text{Bwt}(T)$ , it is more convenient to work with active symbols of  $\text{Bwt}(T)$  by resorting to a data structure,

whose details are given later, which is able to efficiently answer the following two queries with parameters  $c$ ,  $I$  and  $h$ , where  $c \in \Sigma$ ,  $I = [a, b]$  is a range of positions in  $T$  and  $h$  is a position in  $\text{Bwt}(T)$ :

- $\text{PREV}_c(I, h)$ : locate the last active <sub>$[a, b]$</sub>  occurrence in  $\text{Bwt}(T)[0, h - 1]$  of symbol  $c$ ;
- $\text{NEXT}_c(I, h)$ : locate the first active <sub>$[a, b]$</sub>  occurrence in  $\text{Bwt}(T)[h + 1, n]$  of symbol  $c$ .

This data structure is built over the whole text  $T$  and requires  $O(|T|)$  space.

Let  $c$  be the symbol of  $S[r_i + 1]$  we have to conceptually insert in  $\text{rBwt}(T[a : b])$ . We can compute the position (say,  $h$ ) of this symbol in  $\text{Bwt}(T)$  by resorting to the inverse suffix array of  $T$ . Once we know position  $h$ , we have to determine what changes in  $\text{Mtf}(\text{rBwt}(w))$  the insertion of  $c$  has produced and update  $F_w$  accordingly. It is not hard to convince ourselves that the insertion of symbol  $c$  changes no more than  $\sigma$  encodings in  $\text{Mtf}(\text{rBwt}(w))$ . In fact, only the first active occurrence of each symbol in  $\Sigma$  after position  $h$  may change its  $\text{Mtf}$  encoding. More precisely, let  $h_p$  and  $h_n$  be respectively the last active occurrence of  $c$  before  $h$  and the first active occurrence of  $c$  after  $h$  in  $\text{Bwt}(w)$ , then the first active occurrence of a symbol after  $h$  changes its  $\text{Mtf}$  encoding if and only if it occurs active both in  $\text{Bwt}(w)[h_p, h]$  and in  $\text{Bwt}(w)[h, h_n]$ . Otherwise, the new occurrence of  $c$  has no effect on its  $\text{Mtf}$  encoding. Notice that  $h_p$  and  $h_n$  can be computed via proper queries  $\text{PREV}_c$  and  $\text{NEXT}_c$ . In order to correctly update  $F_w$ , we need to recover for each of the above symbols their old and new encodings. The first step consists of finding the last active occurrence before  $h$  of each symbols in  $\Sigma$  using  $\text{PREV}$  queries. Once we have these positions, we can recover the status of the  $\text{Mtf}$  list, denoted  $\lambda$ , before encoding  $c$  at position  $h$ . This is simply obtained by sorting the symbols ordered by decreasing position. In the second step, for each distinct symbol that occurs active in  $\text{Bwt}(w)[h_p, h]$ , we find its first active occurrence in  $\text{Bwt}(w)[h, h_n]$ . Knowing  $\lambda$  and these occurrences sorted by increasing position, we can simulate the  $\text{Mtf}$  algorithm to find the old and new encodings of each of those symbols.

This provides an algorithm to perform  $\text{APPEND}(w)$  by making  $O(\sigma)$  queries of types  $\text{PREV}$  and  $\text{NEXT}$  for each symbol of the page to append in  $w$ . To complete the proof of the time bounds in Theorem 5 we have to show how to support queries of type  $\text{PREV}$  and  $\text{NEXT}$  in  $O(\log n)$  time and  $O(n)$  space. This is achieved by a straightforward reduction to a classic geometric range-searching problem. Given a set of points  $P = \{(x_1, y_1), (x_2, y_2), \dots, (x_p, y_p)\}$  from the set  $[n] \times [n]$  (notice that  $n$  can be larger than  $p$ ), such that no pair of

points shares the same  $x$ - or  $y$ -coordinate, there exists a data structure [115] requiring  $O(p)$  space and supporting the following two queries in  $O(\log p)$  time:

- $\text{RANGEMAX}([l, r], h)$ : return among the points of  $P$  contained in  $[l, r] \times [-\infty, h]$  the one with maximum  $y$ -value
- $\text{RANGEMIN}([l, r], h)$ : return among the points of  $P$  contained in  $[l, r] \times [h, +\infty]$  the one with minimum  $y$ -value

Initially we compute  $SA$  and  $SA^{-1}$  of  $T$  in  $O(n \log \sigma)$  time then, for each symbol  $c \in \Sigma$ , we define  $P_c$  as the set of points  $\{(i, SA^{-1}[i + 1]) \mid T[i] = c\}$  and build the above geometric range-searching structure on  $P_c$ . It is easy to see that  $\text{PREV}_c(l, h)$  can be computed in  $O(\log n)$  time by answering query  $\text{RANGEMAX}(l, SA^{-1}[h + 1])$  on the set  $P_c$ , and the same holds for  $\text{NEXT}_c$  by using  $\text{RANGEMIN}$  instead of  $\text{RANGEMAX}$ , this completes the reduction and the proof of the following theorem.

**Theorem 5.** *Given a sequence of texts of total length  $n$  and alphabet size  $\sigma = \text{poly}(n)$ , we can compute an  $(1 + \epsilon)$ -approximate solution to the optimal partitioning problem for a BWT-based compressor, in  $O(n(\log_{1+\epsilon} n) \sigma \log n)$  time and  $O(n + \sigma \log_{1+\epsilon} n)$  space.*

## Bit-complexity of Lempel-Ziv compression

One of the most famous lossless data-compression schemes is the one introduced by Lempel and Ziv in the late 70s, and indeed many (non-)commercial programs are currently based on it— like `gzip`, `zip`, `pkzip`, `arj`, `rar`, just to cite a few. This compression scheme is known as *dictionary-based compressor*, and consists of squeezing an input string  $T[1, n]$  by replacing some of its substrings with (shorter) *codewords* which are actually pointers to a dictionary of phrases. The dictionary can be either *static* (in that it has been constructed before the compression starts) or *dynamic* (in that it is built as the input string is compressed). The well-known LZ77 and LZ78 compressors, proposed by Lempel and Ziv in [166, 167], and all their variants [149], are interesting examples of *dynamic* dictionary-based compressors.

Many theoretical and experimental results have been dedicated to LZ-compressors in these thirty years and, although today there are alternative solutions to the lossless data-compression problem (e.g., Burrows-Wheeler compression and Prediction by Partial Matching [164]), dictionary-based compression is still widely used for its unique combination of compression power and compression/decompression speed. Over the years dictionary-based compression has also gained importance as a general algorithmic tool, being employed in the design of compressed text indexes [134], in *universal* clustering [28] or classification tools [165], in designing *optimal* pre-fetching mechanisms [159], and in streaming or on-the-fly compression applications [32, 66].

In this chapter we address some key issues which arise when dealing with

the output-size in bits of the so called *LZ77-parsing scheme*, namely, the one in which the dictionary consists of all substrings starting in the last  $M$  scanned positions of the text, where  $M$  is called the *window size* (and possibly depends on the text length), and phrase-codewords consist of triples  $\langle d, \ell, c \rangle$  where  $d$  is the relative *offset* of the copied phrase ( $d \leq M$ ),  $\ell$  is the length of the phrase and  $c$  is the single (new) character following it. Classically, the LZ77-parser adopts a *greedy* rule, namely, one that at each step takes the *longest* dictionary phrase which is a prefix of the currently unparsed suffix of the input string. This greedy parsing can be computed in  $O(n \log \sigma)$  time and  $O(M)$  space [60].<sup>1</sup> The greedy parsing is optimal with respect to the *number of phrases* in which  $T$  can be parsed by any suffix-complete dictionary (like the LZ77-dictionary). Of course, the number of parsed phrases influences the compression ratio and, indeed, various authors [107, 166] proved that greedy parsing achieves asymptotically the (*empirical*) *entropy* of the source generating the input string  $T$ . But these fundamental results have *not yet closed* the problem of optimally compressing  $T$  because the optimality in the number of parsed phrases *is not necessarily equal* to the optimality in the number of bits output by the final compressor on *each individual* input string  $T$ . In fact, if the phrases are compressed via an *equal-length* encoder, like in [107, 149, 166], then the produced output is *bit optimal*. But if one aims for higher compression, *variable-length encoders* should be taken into account (see e.g. [150, 164], and the software `gzip` [86]), and in this situation the greedy-parsing scheme is *no longer optimal* in terms of the number of bits output by the final compressor (see Section 4.3).

Starting from these premises we address in this chapter four main problems, both on the theoretical and the experimental side, which pertain with the bit-optimal compression of the input string  $T$  via parsers that deploy the LZ77-dictionary built on an unbounded window (namely it is  $M = n$ ). Our results extend easily to windows of arbitrary size  $M$ .

**Problem 1.** Let us consider the greedy LZ77-parser, and assume that we encode every parsed phrase  $w_i$  with a variable-length encoder. The value of  $\ell_i = |w_i|$  is in some sense fixed by the greedy choice, being the length of the longest phrase occurring in the current LZ77-dictionary. Conversely, the value of  $d_i$  depends on the position of the copy of  $w_i$  in  $T$ . In order to minimize the number of bits output by the final compressor, the greedy parser should obviously select the *closest* copy of each phrase  $w_i$  in  $T$ , and thus the smallest possible  $d_i$ . Surprisingly enough, known implementations of greedy parsers are time optimal but not bit-optimal, because they select an arbitrary

<sup>1</sup>Recently, [33] showed how to achieve the optimal  $O(n)$  time and space when the alphabet has size  $O(n)$  and the window is unbounded, i.e.  $M = n$ .

or the leftmost occurrence of the longest copied phrase (see [33] and references therein), or they select the closest copy but take  $O(n \log n)$  suboptimal time [3]. In Section 4.2 we provide an elegant, yet simple, algorithm which computes at each parsing step the closest copy of the longest dictionary phrase in  $O(n(1 + \frac{\log \sigma}{\log \log n}))$  overall time and  $O(n)$  space (Lemma 4). This is optimal in terms of time/space performance when the alphabet has size  $\text{polylog}(n)$  (hence almost all texts of practical interest).

**Problem 2.** How good is the greedy LZ77-parsing of  $T$  whenever the compression cost is measured in terms of number of bits produced in output? We show that the greedy selection of the longest dictionary phrase at each parsing step is not optimal, and this may be larger than the bit-optimal parsing by a multiplicative factor  $\Omega(\log n / \log \log n)$ , which is unbounded asymptotically (Section 4.3). Additionally, we show that this lower-bound is tight up to a factor  $\Theta(\log \log n)$ , and we support these theoretical figures with some experimental results which stress the practical importance of finding the bit-optimal parsing of  $T$ .

**Problem 3.** How much efficiently (in time and space) can we compute the bit-optimal (LZ77-)parsing of  $T$ ? Several solutions are indeed known for this problem but they are either inefficient [151], in that they take  $\Theta(n^2)$  worst-case time and space, or they are approximate [101], or they rely on heuristics [14, 29, 103, 153] which do not provide any guarantee on the time/space performance of the compression process. This is the reason why Rajpoot and Sahinalp stated in [141, pag. 159] that “*We are not aware of any on-line or off-line parsing scheme that achieves optimality when the LZ77-dictionary is in use under any constraint on the codewords other than being of equal length*”. In Section 4.4.2 we investigate this question by considering a general class of variable-length codeword encodings which are typically used in data compression (e.g. gzip) and in the design of search engines and compressed indexes [134, 149, 164]. Our final result is a time efficient (possibly, optimal) and space optimal solution for the problem above (Theorem 7).

Technically speaking, the problem of finding bit-optimal parsing of an input string  $T$  can be solved via Dynamic Programming [151]. As we have done in Chapter 3, we use the reduction from solutions of Dynamic Programming recurrences to *single-source shortest path* problem (SSSP) on a *weighted DAG*  $\mathcal{G}(T)$ . This DAG consists of  $n$  nodes, one per character of  $T$ , and  $e$  edges, one per possible parsing step. Every edge is weighted according to the length in bits of the codeword adopted to compress the corresponding phrase. Since these codewords are tuples of integers (see above), we consider a natural class of codeword encoders which satisfy the so called *increasing cost property*: the

greater is the integer to be encoded, the longer is the codeword. This class encompasses most of the encoders used in the literature to design data compressors (see [150] and `gzip` [86]), compressed full-text indexes [134] and search engines [164]. We prove new combinatorial properties for this SSSP-problem and show that the computation of the SSSP in  $\mathcal{G}(T)$  can be restricted onto a subgraph  $\tilde{\mathcal{G}}(T)$  whose structure depends on the integer-encoding functions adopted to compress the LZ77-phrases, and whose size is *provably smaller* than the complete graph generated by [151] (see Theorem 6). Additionally, we design an algorithm that solves the SSSP on the subgraph  $\tilde{\mathcal{G}}(T)$  without materializing it *all at once*, but creating and exploring its edges *on-the-fly* in optimal  $O(1)$  amortized time per edge and  $O(n)$  optimal space overall. As a result, our novel LZ77-compressor achieves bit-optimality in  $O(n)$  optimal working space and in time proportional to  $|\tilde{\mathcal{G}}(T)|$  (hence, it is optimal in its size). The latter is  $O(n \log n)$  for a large class of integer encoders, like Elias and Fibonacci codes [150, 164], and it is optimal  $O(n)$  for (most of) the encodings used by `gzip` [86]. This is the first result providing a *positive answer* to Rajpoot-Sahinalp’s question above.

**Problem 4.** How much efficient are *in practice* our bit-optimal LZ77-compressor? To establish this, we have taken several freely available text collections, and compared our proposal against the classic `gzip` and `bzip2`, as well as against the state-of-the-art *boosting* compressor of [41, 43]. Section 4.5 reports on some experimental figures, and comments on our theoretical findings as well as on possible algorithm-engineering research directions which deserve further attention.

## 4.1 Notation and terminology

As usual, let  $T[1, n]$  be a string drawn from an alphabet  $\Sigma$  of size  $\sigma$ .<sup>2</sup>

In the rest of the chapter we concentrate on LZ77-compression with an unbounded window size, so we will drop the specification “LZ77” unless this will be required to make things unambiguous. The compressor, as any dictionary-based compressor, will work in two intermingled phases: *parsing* and *encoding*. Let  $w_1, w_2, \dots, w_{i-1}$  be the phrases in which a prefix of  $T$  has been already parsed. At this step, the dictionary consists of all substrings of  $T$  starting in the last  $M$  positions of  $w_1 w_2 \dots w_{i-1}$ , where  $M$  is called the *window*

---

<sup>2</sup>Recall that we are assuming that  $\sigma \leq n$ . In case of a larger alphabet, our algorithms are still correct but we need to add an the additive term  $\text{Time}_{\text{sort}}(n, \sigma)$  to their time complexities where  $\text{Time}_{\text{sort}}(n, \sigma)$  is the time required to sort/remap distinct symbols of  $T$  into  $[n]$ .

size (hereafter assumed unbounded, for simplicity). The classic parsing-rule adopted by most LZ77-compressors selects the next phrase according to the so called *longest match heuristic*: that is, this phrase is taken as the *longest* phrase in the current dictionary which prefixes the remaining suffix of  $T$ . This is usually called *greedy parsing*. After such a phrase is selected, the parser adds one further symbol to it and thus forms the next phrase  $w_i$  of  $T$ 's parsing. For simplicity of exposition we will restrict to the LZ77-variant which avoids the additional symbol per phrase. This means that  $w_i$  is represented by the integer pair  $\langle d_i, \ell_i \rangle$ , where  $d_i$  is the relative *offset* of the copied phrase  $w_i$  within the prefix  $w_1 \cdots w_{i-1}$  and  $\ell_i$  is its length  $|w_i|$ . Every first occurrence of a new symbol  $c$  is encoded as  $\langle 0, c \rangle$ .

Once phrases are identified and represented via pairs of integers, their components are compressed via *variable-length integer encoders* which eventually produce the compressed output of  $T$  as a sequence of bits. In order to study and design bit-optimal parsing schemes, we therefore need to deal with such integer encoders. Let  $f$  be an integer-encoding function that maps any integer  $x \in [n]$  into a (bit-)codeword  $f(x)$  whose length is denoted by  $|f(x)|$  bits. We consider variable-length encodings which use longer codewords for greater integers:

**Property 1** (Increasing Cost Property). *For any  $x, y \in [n]$ , if  $x \leq y$ , then  $|f(x)| \leq |f(y)|$ .*

This property is satisfied by most of known integer encoders— like equal-length codewords, Elias codes [164], Fibonacci's codes [150]— which are used to design data compressors [149], compressed full-text indexes [134] and search engines [164].

## 4.2 An efficient and bit-optimal greedy parsing

In this section we describe how to compute efficiently the *greedy* parsing that minimizes the final compress size. We remark that the minimization here is done with respect to all the LZ77-parsings that follow the greedy strategy for selecting their phrases.

Let  $f$  and  $g$  be two integer encoders which satisfy the Increasing Cost Property (possibly  $f = g$ ). We denote by  $\text{LZ}_{f,g}(T)$  the compressed output produced by the greedy-parsing strategy in which we have used  $f$  to compress the distance  $d_i$ , and  $g$  to compress the length  $\ell_i$  of any parsed phrase  $w_i$ . Thus, in  $\text{LZ}_{f,g}(T)$  any phrase  $w_i$  is encoded in  $|f(d_i)| + |g(\ell_i)|$  bits. Given that the parsing is the



greedy one,  $\ell_i$  is in some sense fixed (being the length of the longest copy), so we minimize  $|\text{LZ}_{f,g}(T)|$  by minimizing the *distance*  $d_i$  of  $w_i$ 's copy in  $T$ . If  $p_i$  is the starting position of  $w_i$  in  $T$  (namely,  $T[p_i, p_i + \ell_i - 1] = w_i$ ), many copies of the phrase  $w_i$  could be present in  $T[1, p_i - 1]$ . To minimize  $|\text{LZ}_{f,g}(T)|$  we should choose the copy which is the closest one to  $p_i$ , and thus requires the minimum number of bits to encode its distance  $d_i$  (recall the assumption  $M = n$ ).

In this section we propose an elegant, yet simple, algorithm that selects the rightmost copy of each phrase  $w_i$  in  $O(n(1 + \log \sigma / \log \log n))$  time. This algorithm is the fastest known in the literature [33], and results to be optimal for alphabets with a *polylog*( $n$ ) size (i.e., almost all texts in practice). It requires the suffix tree  $\mathcal{ST}$  of  $T$ , preprocessed to support constant-time `lca`-queries, and the parsing of  $T$  which consists of, say,  $k \leq n$  phrases. All these machineries can be computed in linear time and stored in linear space. We say that a node  $u$  of  $\mathcal{ST}$  is *marked* iff the string spelled out by the root-to- $u$  path in  $\mathcal{ST}$  is equal to some phrase  $w_i$ . In this case we use the notation  $u_{p_i}$  to denote the node marked by phrase  $w_i$  which starts at position  $p_i$  of  $T$ . Since the same node may be marked by different phrases, but any phrase marks just one node, the total number of marked nodes is bounded by the number of phrases, hence  $k$ . Furthermore, if a node is assigned with many phrases, since the greedy LZ77-parsing takes the longest one, it must be the case that every such occurrences of  $w_i$  is followed by a distinct character. So the number of phrases assigned to the same marked node is bounded by  $\sigma$ .

All marked nodes can be computed in  $O(k)$  time by executing  $k$  `lca`-queries on  $\mathcal{ST}$ . Let us now define  $\mathcal{ST}_{\mathcal{C}}$  as the contracted version of  $\mathcal{ST}$ , namely a tree whose internal nodes are the marked nodes of  $\mathcal{ST}$  and whose leaves are the leaves of  $\mathcal{ST}$ . The parent of any node in  $\mathcal{ST}_{\mathcal{C}}$  is its lowest marked ancestor in  $\mathcal{ST}$ . It is easy to see that  $\mathcal{ST}_{\mathcal{C}}$  consists of  $O(k)$  internal nodes and  $n$  leaves, and that it can be built in  $O(n)$  time via a top-down visit of  $\mathcal{ST}$ .

Given the properties of suffix trees, we can now rephrase our problem as follows: for each position  $p_i$ , we need to compute the largest position  $x_i$  which is smaller than  $p_i$  and whose leaf in  $\mathcal{ST}_{\mathcal{C}}$  lies within the subtree rooted at  $u_{p_i}$ . Our algorithm processes the input string  $T$  from left to right and, at each position  $j$ , it maintains the following invariant: the parent  $v$  of any leaf in  $\mathcal{ST}_{\mathcal{C}}$  stores the maximum position  $h < j$  such that the leaf labeled  $h$  is attached to  $v$ . Maintaining this invariant is trivial: after that position  $j$  is processed,  $j$  is assigned to the leaf parent of the leaf labeled  $j$  in  $\mathcal{ST}_{\mathcal{C}}$ . The key point now is how to compute the position  $x_i$  of the rightmost-copy of  $w_i$  whenever we discover that  $j$  is the starting position of a phrase (i.e.  $j = p_i$  for some  $i$ ). In

this case, the algorithm visits the subtree of  $\mathcal{ST}_C$  rooted at  $u_j$  and computes the maximum position stored in its internal nodes. By the invariant, this position is the rightmost copy of the phrase  $w_i$ . This process takes  $O(n + \sigma \sum_{i=1}^k \#(u_{p_i}))$  time, where  $\#(u_{p_i})$  is the number of internal nodes in the subtree rooted at  $u_{p_i}$  in  $\mathcal{ST}_C$ . In fact, by construction, there can be at most  $\sigma$  repetitions of the same phrase in the parsing of  $T$ , and for each of them the algorithm performs a visit of the corresponding subtree.

As a final step we prove that  $\sum_{i=1}^k \#(u_{p_i}) = O(n)$ . By properties of suffix trees, the depth of  $u_{p_i}$  is smaller than  $\ell_i = |w_i|$ , and each (marked) node of  $\mathcal{ST}_C$  is visited as many times as the number of its (marked) ancestors in  $\mathcal{ST}_C$  (with their multiplicities). For each (marked) node  $u_{p_i}$ , this number can be bounded by  $\ell_i = O(|w_i|)$ . Summing up on all nodes, we get  $\sum_{i=1}^k O(|w_i|) = O(n)$ . Thus, the above algorithm requires  $O(\sigma \times n)$  time, which is trivially optimal whenever  $\sigma = O(1)$ .

Now we will show how to further reduce the time complexity to  $O(n(1 + \log \sigma / \log \log n))$  by properly combining a slightly modified variant of the tree covering procedure of [69] with a dynamic Range Maximum Query data structure [126, 162] applied on properly composed arrays of integers. Notice that this improvement leads to an algorithm requiring optimal  $O(n)$  time for alphabets of size poly-logarithmic in  $n$ .

Given  $\mathcal{ST}_C$  and an integer parameter  $P \geq 2$  (in our case  $P = \sigma$ ) this procedure covers the  $k$  internal nodes of  $\mathcal{ST}_C$  in a number of connected subtrees, all of which have size  $\Theta(P)$ , except the one which contains the root of  $\mathcal{ST}_C$  that has size  $O(P)$ . Any two of these subtrees are either disjoint or intersect at their common root. (We refer to Section 2 of [69] for more details.) In our modification we impose that there is no node in common to two subtrees, because we move their common root to the subtree that contains its parent. None of the above properties change, except for the fact that each cover could now be a subforest instead of subtree of  $\mathcal{ST}_C$ . Let  $F_1, F_2, \dots, F_t$  be the subforests obtained by the above covering, where we clearly have that  $t = O(k/P)$ .

We define the tree  $\mathcal{ST}_{SC}$  whose leaves are the leaves of  $\mathcal{ST}_C$  and whose internal nodes are the above subforests. With a little abuse of notation, let us refer with  $F_i$  to the node in  $\mathcal{ST}_{SC}$  corresponding to the subforest  $F_i$ . The leaf  $l$  having  $u$  as parent in  $\mathcal{ST}_C$ , is thus connected to the node  $F_i$  in  $\mathcal{ST}_{SC}$ , where  $F_i$  is the forest that contains the node  $u$ . Notice that roots of subtrees in any subforest  $F_i$  have common parent in  $\mathcal{ST}_C$ .

The computation of the rightmost copy for a phrase  $p_i$  is now divided in two phases. Let  $F_i$  be the subforest that contains  $u_{p_i}$ , the node spelled out by the phrase starting at  $T[p_i]$ . In the first phase, we compute the rightmost copy for

the phrase starting at  $p_i$  among the descendants of  $u_{p_i}$  in  $\mathcal{ST}_{SC}$  that belong to subforests different from  $F_i$ . In the second phase, we compute its rightmost copy among the descendants of  $u_{p_i}$  in  $F_i$ . The maximum between these two values will give the rightmost copy for  $p_i$ , of course. To solve the former problem, we execute our previous algorithm on  $\mathcal{ST}_{SC}$ . It simply visits all subforests descendant from  $F_i$  in  $\mathcal{ST}_{SC}$ , each of them maintaining the rightmost position among its already scanned leaves, and returns the maximum of these value. Since groups of  $P = \sigma$  nodes of  $\mathcal{ST}_{\mathcal{C}}$  have single nodes in  $\mathcal{ST}_{SC}$ , in this case our previous algorithm requires  $O(n)$  time.

The latter problem is solved with a new algorithm exploiting the fact that the number of nodes in  $F_i$  is  $O(\sigma)$  and resorting to dynamic Range Maximum Queries (RMQ) on properly defined arrays [126]. To be precise, we assign to each node of  $F_i$  a unique identifier in  $[m]$  that corresponds to the time of its visit in a depth-first traversal of  $F_i$ . Notice that the nodes in the subtree rooted at some node  $u$  receive integers spanning the whole range from the starting time to the ending time of the DFS-visit of  $u$ . We use an array  $A_{F_i}$  that has an entry for each node of  $F_i$ . Initially, all entries are set to  $-\infty$ . The entry corresponding to any node has index equal to the time of its visit. We build on each array  $A_{F_i}$  a dynamic data structure that answers range maximum queries. For this purpose we use a simple balanced tree augmented with the maximum of the descending leaves in each nodes of  $F_i$ . This way Range-Max queries and updates on  $A_{F_i}$  take  $O(\log \sigma)$  time in the worst case. Now, we proceed, as in our first algorithm, by processing string  $T$  from left to right. When a position  $j$  of  $T$  is processed, we identify the subforest  $F_i$  containing the father of the leaf labeled  $j$  in  $\mathcal{ST}_{\mathcal{C}}$  and we set to  $j$  the corresponding entry in  $A_{F_i}$  (this induces a change in the underlying RMQ data structure). If  $j$  is the starting position of a phrase, we identify the subforest  $F_x$  containing the node  $u_j$  and compute its rightmost copy in  $F_x$ , by resorting to a RMQ on  $A_{F_x}$ . The left and right indexes for the range query are, respectively, the starting and ending time of the visit of  $u_j$  in  $F_x$ .

It is easy to notice that the overall complexity of the algorithm is dominated by the  $O(n)$  updates to the RMQ data structures and the  $O(k)$  queries onto them (recall that  $k$  is the number of phrases of the LZ77-greedy parsing of  $T$ ). Our algorithm then takes  $O(n \log \sigma)$  time and  $O(n)$  space. A further improvement can be obtained by adopting an idea similar to the one in [162][Section 5] to reduce the height of that balanced tree and, consequently, our time complexity by a factor  $O(\log \log n)$ . This proves the following Lemma.

**Lemma 4.** *Given a string  $T[1, n]$  drawn from an alphabet of size  $\sigma$ , there exists an algorithm that computes the greedy parsing of  $T$  and reports the rightmost copy of*

each phrase in the LZ77-dictionary taking  $O(n(1 + \frac{\log \sigma}{\log \log n}))$  time and  $O(n)$  space.

### 4.3 On the bit-efficiency of the greedy LZ77-parsing

We have already noticed above that the greedy strategy used by  $\text{LZ}_{f,g}(T)$  is not necessarily bit-optimal, so we will hereafter use  $\text{OPT}_{f,g}(T)$  to denote the *Bit-Optimal LZ77-parsing* of  $T$  relative to  $f$  and  $g$ , namely a parsing of  $T$  which uses phrases extracted from the LZ77-dictionary and *minimizes* the total number of bits produced by using the encoding functions  $f$  and  $g$ . Of course  $|\text{LZ}_{f,g}(T)| \geq |\text{OPT}_{f,g}(T)|$ , but this does not provide us with any estimate of how much worse the greedy parsing can be with respect to the bit-optimal one. In what follows we identify an infinite family of strings  $T$  for which  $\frac{|\text{LZ}_{f,g}(T)|}{|\text{OPT}_{f,g}(T)|} = \Omega(\frac{\log n}{\log \log n})$ , so the gap may be asymptotically unbounded thus stressing the need for an  $(f, g)$ -optimal parser, as requested by [141].

Our argument holds for any choice of  $f$  and  $g$  from the family of encoding functions that represent an integer  $x$  with a bit string of size  $\Theta(\log x)$  bits (thus the well-known Elias' and Fibonacci's coders belong to this family). Taking inspiration from the proof of Lemma 4.2 in [107], we consider the infinite family of strings  $T_l = ba^l c^{2^l} ba^l ba^{2^l} ba^{2^l} \dots ba^l$ , parameterized in the positive value  $l$ . The greedy LZ77-parser partitions  $T_l$  as<sup>3</sup>:

$$(b) (a) (a^{l-1}) (c) (c^{2^l-1}) (ba) (ba^2) (ba^3) \dots (ba^l),$$

where the symbols forming a parsed phrase have been delimited within a pair of brackets. Thus it copies the latest  $l$  phrases from the beginning of  $T_l$  and takes at least  $l \times |f(2^l)| = \Theta(l^2)$  bits.

A more parsimonious parser selects the copy of  $ba^{i-1}$  (with  $i > 1$ ) from its immediately previous occurrence thus parsing  $T_l$  as:

$$(b) (a) (a^{l-1}) (c) (c^{2^l-1}) (b) (a) (ba) (a) (ba^2) (a) \dots (ba^{l-1}) (a).$$

Hence the encoding of this parsing, called  $\text{rOPT}(T_l)$ , takes  $|g(2^l - 1)| + |g(l - 1)| + \sum_{i=2}^l [|f(i)| + |g(i)| + |f(0)|] + O(l) = O(l \log l)$  bits.

**Lemma 5.** *There exists an infinite family of strings such that, for any of its elements  $T$ , it is  $|\text{LZ}_{f,g}(T)| \geq \Theta(\log |T| / \log \log |T|) |\text{OPT}_{f,g}(T)|$ .*

<sup>3</sup>Recall the variant of LZ77 we are considering here, which uses just a pair of integers per phrase, and thus drops the char following that phrase in  $T$ . See section 4.1.

**Proof.** Since  $|\text{OPT}(T_l)| \leq |\text{rOPT}(T_l)|$ , we can conclude that:

$$\frac{|\text{LZ}_{f,g}(T_l)|}{|\text{OPT}_{f,g}(T_l)|} \geq \frac{|\text{LZ}_{f,g}(T_l)|}{|\text{rOPT}(T_l)|} \geq \Theta\left(\frac{l}{\log l}\right).$$

Since  $|T_l| = 2^l + l^2 - O(l)$ , we have that  $l = \Theta(\log |T_l|)$  for sufficiently long strings. ■

The experimental results reported in Table ?? will show that this gap is not negligible in practice too.

Additionally we can prove that this lower bound is tight up to a  $\log \log |T|$  multiplicative factor, by easily extending to the LZ77-dictionary (which is dynamic), a result proved in [102] for static dictionaries. Precisely, it holds that  $\frac{|\text{LZ}_{f,g}(T)|}{|\text{OPT}_{f,g}(T)|} \leq \frac{|f(|T|)| + |g(|T|)|}{|f(0)| + |g(0)|}$ , which is upper bounded by  $O(\log |T|)$  because  $|f(|T|)| = |g(|T|)| = \Theta(\log |T|)$  and  $|f(0)| = |g(0)| = O(1)$ . To see this, let us assume that  $\text{LZ}_{f,g}(T)$  and  $\text{OPT}_{f,g}(T)$  are formed by  $\ell_{\text{LZ}}$  and  $\ell_{\text{opt}}$  phrases respectively. Of course,  $\ell_{\text{LZ}} \leq \ell_{\text{opt}}$  because the greedy parsing is optimal with respect to the number of parsed phrases for  $T$ . We then assume the worst-case scenario in which every phrase is encoded by  $\text{LZ}_{f,g}(T)$  with the longest encoding (namely,  $f(|T|)$  and  $g(|T|)$  bits each) while  $\text{OPT}_{f,g}(T)$  uses the shortest one (namely,  $f(0)$  and  $g(0)$  bits each). Therefore, we have  $\frac{|\text{LZ}_{f,g}(T)|}{|\text{OPT}_{f,g}(T)|} \leq \frac{\ell_{\text{LZ}}(|f(|T|)| + |g(|T|)|)}{\ell_{\text{opt}}(|f(0)| + |g(0)|)} \leq \frac{|f(|T|)| + |g(|T|)|}{|f(0)| + |g(0)|} = \Theta(\log |T|)$ .

## 4.4 On Bit-Optimal Parsings and Shortest-Path problems

In this section we will describe how to compute efficiently the parsing that minimizes the final compress size of  $T$  with respect to all possible LZ77-parsings. Following [151], we model the design of a bit-optimal LZ77-parsing strategy for a string  $T$  as a Single-Source Shortest Path problem (shortly, SSSP-problem) on a weighted DAG  $\mathcal{G}(T)$  defined as follows. Graph  $\mathcal{G}(T) = (V, E)$  has one vertex per symbol of  $T$  plus a dummy vertex  $v_{n+1}$ , and its edge set  $E$  is defined so that  $(v_i, v_j) \in E$  iff (1)  $j = i + 1$  or (2) the substring  $T[i : j - 1]$  occurs in  $T$  starting from a (previous) position  $p < i$ . Clearly  $i < j$  and thus  $\mathcal{G}(T)$  is a DAG. Every edge  $(v_i, v_j)$  is labeled with the pair  $\langle d_{i,j}, \ell_{i,j} \rangle$  which is set to  $\langle 0, T[i] \rangle$  in case (1), or it is set to  $\langle i - p, j - i \rangle$  in case (2). The second case corresponds to copying a phrase longer than one single character.

It is easy to see that the edges outgoing from  $v_i$  denote all possible parsing steps that can be taken by any parsing strategy which uses a LZ77-dictionary. Hence, there exists a correspondence between paths from  $v_1$  to  $v_{n+1}$  in  $\mathcal{G}(T)$  and LZ77-parsings of the whole string  $T$ . If we weight every edge  $(v_i, v_j) \in E$  with an integer  $c(v_i, v_j) = |f(d_{i,j})| + |g(\ell_{i,j})|$ , which accounts for the cost of encoding its label (phrase) via the encoding functions  $f$  and  $g$ , then the length in bits of the encoded parsing is equal to the cost of the corresponding weighted path in  $\mathcal{G}(T)$ . The problem of determining  $\text{OPT}_{f,g}(T)$  is thus reduced to computing the shortest path from  $v_1$  to  $v_{n+1}$  in  $\mathcal{G}(T)$ .

Given that  $\mathcal{G}(T)$  is a DAG, its shortest path from  $v_1$  to  $v_{n+1}$  can be computed in  $O(|E|)$  time and space. However, this is  $\Theta(n^2)$  in the worst case (consider e.g.  $T = a^n$  [101, 151]) thus resulting inefficient and actually unfeasible in practice even for strings of few Megabytes. In what follows we show that the computation of the SSSP can be restricted to a subgraph of  $\mathcal{G}(T)$  whose size depends on the choice of  $f$  and  $g$  satisfying Property 1, and is  $O(n \log n)$  for most known integer-encoding functions. Then we will design efficient algorithms and data structures that will allow us to generate this subgraph *on-the-fly* by taking  $O(1)$  amortized time per edge and  $O(n)$  space overall. These algorithms will be therefore time-and-space optimal for the subgraph in hand.

#### 4.4.1 A useful, small, subgraph of $\mathcal{G}(T)$ .

We use  $FS(v)$  to denote the *forward star* of a vertex  $v$ , namely the set of vertices pointed to by  $v$  in  $\mathcal{G}(T)$ ; and we use  $BS(v)$  to denote the *backward star* of  $v$ , namely the set of vertices pointing to  $v$  in  $\mathcal{G}(T)$ . It is easy to notice that all edges of in this graph are oriented rightward, so  $\mathcal{G}(T)$  is a DAG. More importantly, we can prove that the indices of the vertices in  $FS(v)$  and  $BS(v)$  form a *contiguous range*:

**Fact 2.** *Given a vertex  $v_i$  and let  $v_{i+x}$  and  $v_{i-y}$  be respectively the vertex with greatest index in  $FS(v_i)$  and the smallest index in  $BS(v_i)$ , it holds*

- $FS(v_i) = \{v_{i+1}, \dots, v_{i+x-1}, v_{i+x}\}$  and
- $BS(v_i) = \{v_{i-y}, \dots, v_{i-2}, v_{i-1}\}$ .

Furthermore,  $x, y$  are smaller than the length of the longest repeated substring in  $T$ .

**Proof.** By definition of  $(v_i, v_{i+x})$ , string  $T[i : i + x - 1]$  occurs at some position  $p < i$  in  $T$ . Any prefix  $T[i : k - 1]$  of  $T[i : i + x - 1]$  also occurs at that position  $p$ , thus  $v_k \in FS(v_i)$ . The bound on  $x$  derives from the definition of  $(v_i, v_{i+x})$ . A similar argument holds for  $BS(v_i)$ . ■

This means that if an edge does exist in  $\mathcal{G}(T)$ , then there exist also all the edges which are *nested* within it and are incident into one of its extremes. The following property relates the indices of the vertices  $v_j \in FS(v_i)$  with the cost of their connecting edge  $(v_i, v_j)$ , and not surprisingly shows that the smaller is  $j$  (i.e. the shorter is the edge), the smaller is the cost of encoding the phrase  $T[i : j - 1]$ <sup>4</sup>

**Fact 3.** *Given a vertex  $v_i$ , for any pair of vertices  $v_{j'}, v_{j''} \in FS(v_i)$  such that  $j' < j''$ , we have that  $c(v_i, v_{j'}) \leq c(v_i, v_{j''})$ . The same property holds for  $v_{j'}, v_{j''} \in BS(v_i)$ .*

**Proof.** We have that  $d_{i,j'} \leq d_{i,j''}$  and  $\ell_{i,j'} < \ell_{i,j''}$  because  $T[i : j' - 1]$  is a prefix of  $T[i : j'' - 1]$  and thus the first substring occurs wherever the latter occurs. The property holds because  $f$  and  $g$  satisfy the Increasing Cost Property. ■

Given these monotonicity properties, we are ready to characterize a special subset of the vertices in  $FS(v_i)$ , and their connecting edges.

**Definition 1.** *An edge  $(v_i, v_j) \in E$  is called*

- *$d$ -maximal iff the next edge from  $v_i$  takes more bits to encode its distance (i.e.,  $|f(d_{i,j})| < |f(d_{i,j+1})|$ ).*
- *$\ell$ -maximal iff the next edge from  $v_i$  takes more bits to encode its length (i.e.,  $|g(\ell_{i,j})| < |g(\ell_{i,j+1})|$ ).*

*Edge  $(v_i, v_j)$  is called maximal if it is either  $d$ -maximal or  $\ell$ -maximal (i.e.,  $c(v_i, v_j) < c(v_i, v_{j+1})$ ).*

The number of maximal edges depends on the functions  $f$  and  $g$  (which satisfy Property 1). Let  $Q(f, n)$  (resp.  $Q(g, n)$ ) be the number of different codeword lengths generated by  $f$  (resp.  $g$ ) when applied to integers in the range  $[n]$ . We can partition  $[n]$  into contiguous sub-ranges  $I_1, I_2, \dots, I_{Q(f, n)}$  such that the integers in  $I_i$  are mapped by  $f$  to codewords (strictly) shorter than the codewords for the integers in  $I_{i+1}$ . Similarly,  $g$  partitions the range  $[n]$  in  $Q(g, n)$  contiguous sub-ranges.

**Lemma 6.** *There are at most  $Q(f, n) + Q(g, n)$  maximal edges outgoing from any vertex  $v_i$ .*

**Proof.** By Fact 2, vertices in  $FS(v_i)$  have indices in a range  $R$ , and by Fact 3,  $c(v_i, v_j)$  is monotonically non-decreasing as  $j$  increases in  $R$ . Moreover we know that  $f$  (resp.  $g$ ) cannot change more than  $Q(f, n)$  (resp.  $Q(g, n)$ ) times. ■

<sup>4</sup>Recall that  $c(v_i, v_j) = |f(d_{i,j})| + |g(\ell_{i,j})|$ , if the edge does exist, otherwise we set  $c(v_i, v_j) = +\infty$ .

To speed up the computation of a SSSP from  $v_1$  to  $v_{n+1}$ , we construct a subgraph  $\tilde{\mathcal{G}}(T)$  of  $\mathcal{G}(T)$  which is formed by maximal edges only, it is smaller than  $\mathcal{G}(T)$  and contains one of those SSSP.

**Theorem 6.** *There exists a shortest path in  $\mathcal{G}(T)$  from  $v_1$  to  $v_{n+1}$  that traverses maximal edges only.*

**Proof.** By contradiction assume that every such shortest path contains at least one non-maximal edge. Let  $\pi = v_{i_1}v_{i_2}\dots v_{i_k}$ , with  $i_1 = 1$  and  $i_k = n + 1$ , be one of these shortest paths, and let  $\gamma = v_{i_1}\dots v_{i_r}$  be the longest initial subpath of  $\pi$  which traverses maximal edges only. Assume w.l.o.g. that  $\pi$  is the shortest path maximizing the value of  $|\gamma|$ . We know that  $(v_{i_r}, v_{i_{r+1}})$  is a non-maximal edge, and thus we can take the maximal edge  $(v_{i_r}, v_j)$  that has the same cost. By definition of maximal edge, it is  $j > i_{r+1}$ ; furthermore, we must have  $j < n + 1$  because we assumed that no path is formed by maximal edges only. Now, since  $\mathcal{G}(T)$  is a DAG and indices in  $\pi$  are increasing, it must exist an index  $i_h \geq i_r$  such that the index of that maximal edge  $j$  lies in  $[i_h, i_{h+1}]$ . Since  $(v_{i_h}, v_{i_{h+1}})$  is an edge of  $\pi$ , it does exist the edge  $(v_j, v_{i_{h+1}})$  (by Fact 2), and by Fact 3 on  $BS(v_{i_{h+1}})$  we can conclude that  $c(v_j, v_{i_{h+1}}) \leq c(v_{i_h}, v_{i_{h+1}})$ . Consequently, the path  $v_{i_1}\dots v_{i_r}v_jv_{i_{h+1}}\dots v_{i_k}$  is also a shortest path but its longest initial subpath of maximal edges consists of  $|\gamma| + 1$  vertices, which is a contradiction! ■

Theorem 6 implies that the distance between  $v_1$  and  $v_{n+1}$  is the same in  $\mathcal{G}(T)$  and  $\tilde{\mathcal{G}}(T)$ , with the advantage that computing SSSP in  $\tilde{\mathcal{G}}(T)$  can be done faster and in reduced space, because  $|FS(v)| \leq Q(f, n) + Q(g, n)$  (Lemma 6). Thus, subgraph  $\tilde{\mathcal{G}}(T)$  consists of  $n + 1$  vertices and at most  $n(Q(f, n) + Q(g, n))$  edges. For Elias' codes [36], Fibonacci's codes [150], and most practical integer encoders used for search engines and data compressors [149, 164], it is  $Q(f, n) = Q(g, n) = O(\log n)$ . Therefore  $|\tilde{\mathcal{G}}(T)| = O(n \log n)$ , so it is smaller than the complete graph built and used by previous papers [101, 151]. For the encoders used in gzip, it is  $Q(f, n) = Q(g, n) = O(1)$  and thus  $|\tilde{\mathcal{G}}(T)| = O(n)$ .

#### 4.4.2 An efficient bit-optimal parser

From a high level, our solution is a variant of a classic linear-time algorithm for SSSP over a DAG (see [31, Section 24.2]), here applied to work on the subgraph  $\tilde{\mathcal{G}}(T)$ . Therefore its correctness follows directly from Theorem 24.5 of [31] and our Theorem 6. However, the key difficulty in implementing this approach consists of how to generate *on-the-fly and efficiently (in time and space) the maximal edges outgoing from vertex  $v_i$* . We will refer to this problem as the



forward-star generation problem, and use FSG for brevity. In what follows we show that FSG takes  $O(1)$  amortized time per edge and  $O(n)$  space in total. Since we have  $n$  vertices, with no more than  $Q(f, n) + Q(g, n)$  maximal edges each (Lemma 6), we will obtain the following:

**Theorem 7.** *Given a string  $T[1, n]$  drawn from an alphabet of size  $\sigma$ , and two integer-encoding functions  $f$  and  $g$  that satisfy Property 1, there exists an algorithm that computes the  $(f, g)$ -optimal parsing of  $T$  based on a LZ77-dictionary by taking  $O(n)$  space and  $O(n(Q(f, n) + Q(g, n)))$  time in the worst case.*

Most of the integer-encoding functions used in practice are such that  $Q(e, n) = O(\log n)$  [150]. Thus, by Theorem 7 we derive the following corollary that instantiates our result for many of such integers-encoding functions.

**Corollary 1.** *Given a string  $T[1, n]$  drawn from an alphabet of size  $\sigma$ , and let  $f$  and  $g$  be chosen among Elias Gamma code, Elias Delta code, Elias Omega code, Even-Rodeh codes, Fibonacci code, Variable Bytes, Nibbles code or Boldi-Vigna Zeta codes, the  $(f, g)$ -optimal parsing of  $T$  based on a LZ77-dictionary by taking  $O(n \log n)$  time and  $O(n)$  space in the worst case.*

To the best of our knowledge, result stated above is the first one that answers positively to the question posed by Rajpoot and Sahinalp in [141, pag. 159].

The rest of this section will be devoted to prove Theorem 7, which is indeed our main contribution.

From Lemma 6 we know that the edges outgoing from  $v_i$  can be partitioned into no more than  $Q(f, n)$  groups, according to the distance from  $T[i]$  of the copied string they represent. Let  $I_1, I_2, \dots, I_{Q(f, n)}$  be the intervals of distances such that all distances in  $I_k$  are encoded with the same number of bits by  $f$ . Take now the  $d$ -maximal edge  $(v_i, v_{h_k})$  for the interval  $I_k$ . We can infer that substring  $T[i : h_k - 1]$  is the longest substring having a copy at distance within  $I_k$  because, by Definition 1 and Fact 3, any edge following  $(v_i, v_{h_k})$  denotes a longer substring which must lie in a subsequent interval (by  $d$ -maximality of  $(v_i, v_{h_k})$ ), and thus must have longer distance from  $T[i]$ . Once  $d$ -maximal edges are known, the computation of the  $\ell$ -maximal edges is then easy because it suffices to further decompose the edges between successive  $d$ -maximal edges, say between  $(v_i, v_{h_{k-1}+1})$  and  $(v_i, v_{h_k})$ , according to the distinct values assumed by the encoding function  $g$  on the lengths in the range  $[h_{k-1}, \dots, h_k - 1]$ . This takes  $O(1)$  time per  $\ell$ -maximal edge, because it needs some algebraic calculations, and the corresponding copied substring can then be inferred as a prefix of  $T[i : h_k - 1]$ .

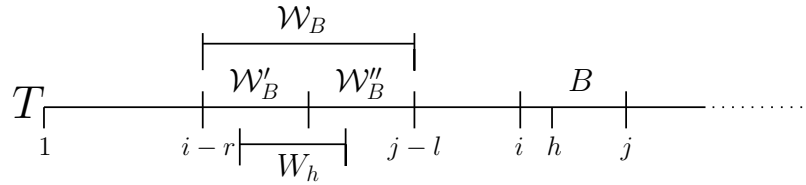
So, let us concentrate on the computation of  $d$ -maximal edges outgoing from vertex  $v_i$ . We remark that we could use the solution proposed in [60] on each

of the  $Q(f, n)$  ranges of distances in which a phrase copy can be found. Unfortunately, this approach would pay another multiplicative factor  $\log \sigma$  per symbol and its space complexity would be super-linear in  $n$ . Conversely, our solution overcomes these drawbacks by deploying two key ideas:

1. The first idea aims at achieving the optimal  $O(n)$  working-space bound. It consists of proceeding in  $Q(f, n)$  passes, one per interval  $I_k$  of possible  $d$ -costs for the edges in  $\tilde{\mathcal{G}}(T)$ . During the  $k$ th pass, we logically partition the vertices of  $\tilde{\mathcal{G}}(T)$  in blocks of  $|I_k|$  contiguous vertices, say  $v_{i_k}, v_{i_k+1}, \dots, v_{i_k+|I_k|-1}$ , and compute all  $d$ -maximal edges which spread out from that block and have copy-distance within  $I_k$  (thus they all have the same  $d$ -cost, say  $c(I_k)$ ). These edges are kept in memory until they are used by our bit-optimal parser, and discarded as soon as the first vertex of the next block, i.e.  $v_{i_k+|I_k|}$ , needs to be processed. The next block of  $|I_k|$  vertices is then fetched and the process repeats. All passes are executed in *parallel* to guarantee that all  $d$ -maximal edges of  $v_i$  are available when processing this vertex. There are  $n/|I_k|$  distinct blocks at each pass, and each  $d$ -maximal edge of a vertex is considered in some pass (because it has  $d$ -cost in some  $I_k$ ). The space is  $\sum_{k=1}^{Q(f,n)} |I_k| = O(n)$  because we keep one  $d$ -maximal edge per vertex at any pass.
2. The second key idea aims at computing the  $d$ -maximal edges for that block of  $|I_k|$  contiguous vertices in  $O(|I_k|)$  time and space. This is what we address below, being the most sophisticated technicality of our solution. As a result, we show that the time complexity of FSG is no more than  $\sum_{k=1}^{Q(f,n)} (n/|I_k|)O(|I_k|) = O(n Q(f, n))$ , i.e.,  $O(1)$  amortized time per  $d$ -maximal edge. Combining this fact with the previous observation on the computation of the  $\ell$ -maximal edges, we get Theorem 7 above.

Consider the  $k$ th pass of FSG in which we assume that  $I_k = [l, r]$ . Recall that all distances in  $I_k$  can be  $f$ -encoded in the same number of, say,  $c(I_k)$  bits. Let  $B = [i, i + |I_k| - 1]$  be the block of (indices of) vertices for which we wish to compute *on-the-fly* the  $d$ -maximal edges of cost  $c(I_k)$ . This means that the  $d$ -maximal edge from vertex  $v_h$ ,  $h \in B$ , represents a phrase that starts at  $T[h]$  and has a copy starting in the *window* (of indices)  $W_h = [h - r, h - l]$ . Thus the distance of that copy can be  $f$ -encoded in  $c(I_k)$  bits, and so we will say that the edge has  $d$ -cost  $c(I_k)$ . Since this computation must be done for all vertices in  $B$ , it is useful to consider the window  $\mathcal{W}_B = W_i \cup W_{i+|I_k|-1}$  which merges the first and last window of positions that can be the (copy-)reference of any  $d$ -maximal edge outgoing from  $B$ . Note that  $|\mathcal{W}_B| = 2|I_k|$  (see Figure 4.1) and

it spans all positions where the copy of a  $d$ -maximal edge outgoing from  $B$  can occur.



**Figure 4.1:** Interval  $B = [i, j]$  with  $j = i + |I_k| - 1$ , window  $\mathcal{W}_B$  and its two halves  $\mathcal{W}'_B, \mathcal{W}''_B$ .

The next fact is crucial to fast compute all these  $d$ -maximal edges via indexing data structures built over  $T$ :

**Fact 4.** *If there exists a  $d$ -maximal edge outgoing from  $v_h$  and having  $d$ -cost  $c(I_k)$ , then this edge can be found by determining a position  $s \in W_h$  whose suffix  $T_s$  shares the maximum longest common prefix (shortly, **Lcp**) with  $T_h$ .*

**Proof.** Among all positions  $s$  in  $W_h$  take one whose suffix  $T_s$  shares the maximum **Lcp** with  $T_h$ , and let  $q$  be the length of this **Lcp**. Of course, there may exist many such positions, we take just one of them. The edge  $(v_h, v_{h+q+1})$  has  $d$ -cost  $c(I_k)$ , because  $s \in W_h$ , and is  $d$ -maximal because any other position  $s' \in W_h$  induces an edge  $(v_h, v_{h+q'+1})$  whose length  $q' \leq q$ , by maximality of  $q$ . So any edge  $(v_h, v_{h+q''})$ , with  $q'' > q$ , must reference a copy before  $W_h$ , if any. ■

Hereafter we call the position  $s$  of Fact 4 *maximal position* for vertex  $v_h$ , and note that it does exist only if  $v_h$  has a  $d$ -maximal edge of cost  $c(I_k)$ . Our algorithm will compute the maximal positions of every vertex  $v_h$  in  $B$  and every cost  $c(I_k)$ . If such maximal position does not exist,  $v_h$  will be assigned an *arbitrary* position. The net result is that we will generate a *supergraph* of  $\tilde{\mathcal{G}}(T)$  which is still guaranteed to have the size stated in Lemma 6 and can be created efficiently in  $O(|I_k|)$  time and space, for each block of distances  $I_k$ , as we required above.

Fact 4 relates the computation of maximal positions for the vertices in  $B$  to **Lcp**-computations between suffixes in  $B$  and suffixes in  $\mathcal{W}_B$ . Therefore it is natural to resort some indexing data structure, like the compact trie  $\mathcal{T}_B$ , built over the suffixes of  $T$  which start in the range of positions  $B \cup \mathcal{W}_B$ . Trie  $\mathcal{T}_B$  takes  $O(|B| + |\mathcal{W}_B|) = O(|I_k|)$  space, and this bound is within our required space complexity. It is not easy to build  $\mathcal{T}_B$  in  $O(|I_k|)$  time and space, because this time complexity is *independent* of the length of the indexed suffixes and

the alphabet size. The proof of this result may be of independent interest, and it is deferred to Subsection 4.4.3. The key idea there is to deploy the fact that the algorithm we detail below does not make any assumption on the edge-ordering of  $\mathcal{T}_B$ , because it just computes (sort of) Lca-queries on its structure.

So, let us assume that we are given the trie  $\mathcal{T}_B$ . We notice that the maximal position  $s$  for a vertex  $v_h$  in  $B$  having  $d$ -cost  $c(I_k)$  can be computed by *finding the leaf of  $\mathcal{T}_B$  which is labeled with an index  $s$  that belongs to the range  $W_h$  and has the maximum Lcp with the leaf labeled  $h$* . This actually corresponds to find the leaf whose label  $s \in W_h$  and has the deepest Lca with the leaf labeled  $h$ . We need to answer this query in  $O(1)$  amortized time per vertex  $v_h$ , since we aim at achieving an  $O(|I_k|)$  time complexity over all vertices in  $B$ . This is not easy because this is *not* the classic Lca-query since we do not know  $s$ , which is actually the position we are searching for! Furthermore, since the leaf  $s$  is the closest one to  $h$  in  $\mathcal{T}_B$  among the leaves with index in  $W_h$ , one could think to use proper predecessor/successor queries on a suitable dynamic set of suffixes in  $W_h$ . Unfortunately, this would take  $\omega(1)$  time because of well-known lower bounds [13]. Therefore, in order to answer this query in constant (amortized) time per vertex of  $B$ , we deploy proper structural properties of the trie  $\mathcal{T}_B$  and the problem at hand.

Let  $u$  be the Lca of the leaves labeled  $h$  and  $s$  in  $\mathcal{T}_B$ . For simplicity, we assume that the window  $W_h$  strictly precedes  $B$  and that  $s$  is the unique maximal position for  $v_h$  (our algorithm deals with these cases too, see the proof of Lemma 7). We observe that  $h$  must be the smallest index that lies in  $B$  and labels a leaf descending from  $u$  in  $\mathcal{T}_B$ . In fact assume, by contradiction, that a smaller index  $h' < h$  does exist. By definition  $h' \in B$  and thus  $v_h$  would not have a  $d$ -maximal edge of  $d$ -cost  $c(I_k)$  because it could copy from the closer  $h'$  a possibly longer phrase, instead of copying from the farther set of positions in  $W_h$ . This observation implies that we have to search only for one maximal position per node  $u$  of  $\mathcal{T}_B$ , and this position refers to the vertex  $v_{a(u)}$  whose index  $a(u)$  is the smallest one that lies in  $B$  and labels a leaf descending from  $u$ . Computing the value  $a(\cdot)$  for all nodes  $u$  in  $\mathcal{T}_B$  takes  $O(|\mathcal{T}_B|) = O(|I_k|)$  time and space via a traversal of the trie  $\mathcal{T}_B$ .

Now we need to compute the maximal position for  $v_{a(u)}$ , for each node  $u \in \mathcal{T}_B$ . We cannot traverse the subtree of  $u$  searching for the maximal position for  $v_{a(u)}$ , because this would take quadratic time complexity overall. Conversely, we define  $\mathcal{W}'_B$  and  $\mathcal{W}''_B$  to be the first and the second half of  $\mathcal{W}_B$ , respectively, and observe that any window  $W_h$  has its left extreme in  $\mathcal{W}'_B$  and its right extreme in  $\mathcal{W}''_B$  (see Figure 4.1). Therefore the window  $W_{a(u)}$  containing the maximal position  $s$  for  $v_{a(u)}$  overlaps both  $\mathcal{W}'_B$  and  $\mathcal{W}''_B$ . If  $s$  does exist for  $v_{a(u)}$ ,

then  $s$  belongs to either  $\mathcal{W}'_B$  or to  $\mathcal{W}''_B$ , and the leaf labeled  $s$  descends from  $u$ . Hence the maximum (resp. minimum) among the elements in  $\mathcal{W}'_B$  (resp.  $\mathcal{W}''_B$ ) that label leaves descending from  $u$  must belong to  $W_{a(u)}$ .

This suggests to compute for each node  $u$  the rightmost position in  $\mathcal{W}'_B$  and the leftmost position in  $\mathcal{W}''_B$  that label a leaf descending from  $u$ , denoted respectively by  $\max(u)$  and  $\min(u)$ . This takes  $O(|I_k|)$  time with a post-order visit of  $\mathcal{T}_B$ . We can now efficiently compute  $\text{mp}[h]$  as the maximal position for  $v_h$ , if it exists, or otherwise set  $\text{mp}[h]$  arbitrarily. We initially set all  $\text{mp}$ 's entries to `nil`; then we visit  $\mathcal{T}_B$  in post-order and perform, at each node  $u$ , the following two checks whenever  $\text{mp}[a(u)] = \text{nil}$ : If  $\min(u) \in W_{a(u)}$ , we set  $\text{mp}[a(u)] = \min(u)$ ; if  $\max(u) \in W_{a(u)}$ , we set  $\text{mp}[a(u)] = \max(u)$ . At the end of the visit, if  $\text{mp}[a(u)]$  is still `nil` we set  $\text{mp}[a(u)] = a(\text{parent}(u))$  whenever  $a(u) \neq a(\text{parent}(u))$ . This last check is needed (see proof of Lemma 7) to manage the case in which  $T[a(u)]$  can copy the phrase starting at its position from position  $a(\text{parent}(u))$  and, additionally, we have that  $B$  overlaps  $\mathcal{W}_B$  (which may occur depending on  $f$ ). Since  $\mathcal{T}_B$  has size  $O(|I_k|)$ , the overall algorithm requires  $O(|I_k|)$  time and space in the worst case, and hence Theorem 7 follows. Correctness follows from lemma below.

**Lemma 7.** *For each position  $h \in B$ , if there exists a  $d$ -maximal edge outgoing from  $v_h$  and having  $d$ -cost  $c(I_k)$ , then  $\text{mp}[h]$  is equal to its maximal position.*

**Proof.** Recall that  $B = [i, i + |I_k| - 1]$  and consider the longest path  $\pi = u_1 u_2 \dots u_z$  in  $\mathcal{T}_B$  that starts from the leaf  $u_1$  labeled with  $h \in B$  and goes upward until the traversed nodes satisfy the condition  $a(u_j) = h$ , here  $j = 1, \dots, z$ . By definition of  $a$ -value (see above), we know that all leaves descending from  $u_z$  and occurring in  $B$  are labeled with an index which is larger than  $h$ . Clearly, if  $\text{parent}(u_z)$  does exist, then it is  $a(\text{parent}(u_z)) < h$ . There are two cases for the final value stored in  $\text{mp}[h]$ .

**Case 1.** Suppose that  $\text{mp}[h] \in W_h$ . We want to prove that  $\text{mp}[h]$  is the index of the leaf which has the deepest **Lca** with  $h$  among all the other leaves labeled with an index in  $W_h$ . Let  $u_x \in \pi$  be the node in which the value of  $\text{mp}[h]$  is assigned. Then, by our algorithm it is  $a(u_x) = h$ . Assume now that there exists at least another index in  $W_h$  whose leaf has a deeper **Lca** with leaf  $h$ . This **Lca** must lie on  $u_1 \dots u_{x-1}$ , say  $u_l$ . Since  $W_h$  is a window having its left extreme in  $\mathcal{W}'_B$  and its right extreme in  $\mathcal{W}''_B$ , the value  $\max(u_l)$  or  $\min(u_l)$  must lie in  $W_h$  and thus the algorithm has set  $\text{mp}[h]$  to one of these positions, because of the post-order visit of  $\mathcal{T}_B$  and the check on  $\text{mp}[a(u_x)] = \text{nil}$ . Therefore  $\text{mp}[h]$  must be the index of the leaf having the deepest **Lca** with  $h$ , and thus by Fact 4 it is its maximal position.

**Case 2.** Suppose that  $\text{mp}[h] \notin W_h$  and, thus, it cannot be a maximal position for  $v_h$ . We have to prove that it does not exist a  $d$ -maximal edge outgoing from the vertex  $v_h$  with cost  $c(I_k)$ . Let  $T_s$  be the suffix in  $W_h$  having the maximum Lcp with  $T_h$ , and let  $l$  be the Lcp-length. Values  $\min(u_i)$  and  $\max(u_i)$  do not belong to  $W_h$ , for any node  $u_i \in \pi$  with  $a(u_i) = h$ , otherwise  $\text{mp}[h]$  would have been assigned with an index in  $W_h$  (contradicting the hypothesis). Thus the value of  $\text{mp}[h]$  remains nil up to node  $u_z$ . This implies that no suffix descending from  $u_z$  starts in  $W_h$  and, in particular,  $T_s$  does not descend from  $u_z$ . Therefore, the Lca between leaves  $h$  and  $s$  is a node in the path from  $\text{parent}(u_z)$  to the root of  $\mathcal{T}_B$ , and the  $\text{Lcp}(T_{a(\text{parent}(u_z))}, T_h) \geq \text{Lcp}(T_s, T_h) = l$ . Since  $a(\text{parent}(u_z)) < a(u_z)$  and it belongs to  $B$  (by definition of  $a$ -value), this position is nearer to  $h$  than any other position in  $W_h$ , and shares a longer prefix with  $T_h$ . So we found a longer edge from  $v_h$  with smaller  $d$ -cost. This way  $v_h$  has no  $d$ -maximal edge of cost  $c(I_k)$  in  $\tilde{\mathcal{G}}(T)$ . ■

#### 4.4.3 On the optimal construction of $\mathcal{T}_B$

In the discussion above we left out the explanation on how to build  $\mathcal{T}_B$  in  $O(|I_k|)$  time and space, thus within a time complexity which is *independent* of the length of the  $|I_k|$  indexed suffixes and the alphabet size  $\sigma$ . To achieve this result we deploy the crucial fact that the algorithm of the previous section does not make any assumption on the ordering of the edges in  $\mathcal{T}_B$ , because it just computes (sort of) Lca-queries on its structure.

First of all, we build the suffix array of the whole string  $T$  and a data structure that answers constant-time Lcp-queries between pair of suffixes (see e.g. [139]). This takes  $O(n)$  time and space.

Let us first assume that  $B$  and  $\mathcal{W}_B$  are contiguous and form the range  $[i, i + 3|I_k| - 1]$ . If we had the sorted sequence of suffixes starting in  $T[i : i + 3|I_k| - 1]$ , we could easily build  $\mathcal{T}_B$  in  $O(|I_k|)$  time and space by deploying the above Lcp-data structure. Unfortunately, it is unclear how to obtain from the suffix array of the whole  $T$ , the sorted sub-sequence of suffixes *starting* in the range  $[i, i + 3|I_k| - 1]$  by taking  $O(|B| + |\mathcal{W}_B|) = O(|I_k|)$  time (notice that these suffixes have length  $\Theta(n - i)$ ). We cannot perform a sequence of predecessor/successor queries because they would take  $\omega(1)$  time each [13]. Conversely, we resort the key observation above that  $\mathcal{T}_B$  does not need to be ordered, and thus devise a solution which builds an *unordered*  $\mathcal{T}_B$  in  $O(|I_k|)$  time and space, passing through the construction of the suffix array of a *transformed* string. The transformation is simple. We first map the distinct symbols of  $T[i, i + 3|I_k| - 1]$  to the first  $O(|I_k|)$  integers. This mapping *does not need to re-*

flect their lexicographic order, and thus can be computed in  $O(|I_k|)$  time by a simple scan of those symbols and the use of a table  $T$  of size  $\sigma = O(n)$ . Then, we define  $\hat{T}$  as the string  $T$  which has been transformed by re-mapping some of the symbols according to table  $T$  (namely, those occurring in  $T[i : i + 3|I_k| - 1]$ ). We can prove the following Lemma.

**Lemma 8.** *Let  $T_i, \dots, T_j$  be a contiguous sequence of suffixes in  $T$ . The re-mapped suffixes  $\hat{T}_i \dots \hat{T}_j$  can be lexicographically sorted in  $O(j - i + 1)$  time.*

**Proof.** Consider the string of pairs  $w = \langle \hat{T}[i], b_i \rangle \dots \langle \hat{T}[j], b_j \rangle \$$ , where  $b_h$  is 1 if  $\hat{T}_{h+1} > \hat{T}_{j+1}$ , -1 if  $\hat{T}_{h+1} < \hat{T}_{j+1}$ , or 0 if  $h = j$ . The ordering of the pairs is defined component-wise, and we assume that  $\$$  is a special “pair” larger than any other pair in  $w$ . For any pair of indices  $p, q \in [1 \dots j - i]$ , it is  $\hat{T}_{p+i} > \hat{T}_{q+i}$  iff  $w_p > w_q$ . In fact, suppose that  $w_p > w_q$  and set  $r = \text{Lcp}(w_p, w_q)$ . We have that  $w[p + r] = \langle \hat{T}[p + i + r], b_{p+i+r} \rangle > \langle \hat{T}[q + i + r], b_{q+i+r} \rangle = w[q + i + r]$ . Hence we have that either  $\hat{T}[p + i + r] > \hat{T}[q + i + r]$  or  $b_{p+i+r} > b_{q+i+r}$ . The latter actually means that  $\hat{T}_{p+i+r+1} > \hat{T}_{j+1} \geq \hat{T}_{q+i+r+1}$ . In any case, it follows that  $\hat{T}_{p+i+r} > \hat{T}_{q+i+r}$  and thus  $\hat{T}_{p+i} > \hat{T}_{q+i}$ , since their first  $r$  symbols are equal.

This implies that sorting the suffixes  $\hat{T}_i, \dots, \hat{T}_j$  reduces to computing the suffix array of  $w$ , and this takes  $O(|w|)$  time given that the alphabet size is  $O(|w|)$  [139]. Clearly,  $w$  can be constructed in that time bound because comparing  $\hat{T}_z$  with  $\hat{T}_{j+1}$  takes  $O(1)$  time via an Lcp-query on  $T$  (using the proper data structure above) and a check at their first mismatch. ■

Lemma 8 allows us to generate the compact trie of  $\hat{T}_i, \dots, \hat{T}_{i+3|I_k|-1}$ , which is equal to the (unordered) compacted trie of  $T_i, \dots, T_{i+3|I_k|-1}$  after replacing every ID assigned by table  $T$  with its original symbol in  $T$ . We finally notice that if  $B$  and  $\mathcal{W}_B$  are not contiguous (as instead we assumed above), we can use a similar strategy to sort separately the suffixes in  $B$  and the suffixes in  $\mathcal{W}_B$ , and then merge these two sequences together by deploying the Lcp-data structure mentioned at the beginning of this section.

## 4.5 Some experiments

In this section we provide an experimental support to our findings of Section 4.3, and compare our proposals of Sections 4.2 and 4.4 with some state-of-the-art compression tools over few freely available text collections. Table ?? reports our findings.

Compressor	english	sources
gzip -9	37.52%	23.29%
bzip2 -9	28.40%	19.78%
boosterOpt	20.62%	17.36%
Fixed-LZ77	26.19%	24.63%
Rightmost-LZ77	23.81%	20.14%
BitOptimal-LZ77	21.62%	17.62%
Compressor	html [19]	Avg Dec. time (sec)
gzip -9	20.09%	0.7
bzip2 -9	10.63%	6.3
boosterOpt	3.89%	20.2
Fixed-LZ77	4.98%	0.8
Rightmost-LZ77	4.27%	0.9
BitOptimal-LZ77	3.87%	0.9

**Table 4.1:** Each text collection consists of 50 Mbytes of data. All the experiments were executed on a 2.6 GHz Pentium 4, with 1.5 GB of main memory, and running Fedora Linux.

Let us first consider algorithm `Fixed-LZ77`, which uses an unbounded window and equal-length encoders for the distance of the copied phrases. Its compression performance shows that an unbounded window may introduce a significant compression gain wrt to a bounded one, as used by `gzip` and `bzip2` (see e.g. `html`), thus witnessing the presence in current (Web/text) collections of surprisingly many long repetitions at large distances. This motivates our main study for  $M = n$ , even if our results extend to the bounded-window case too.

Then we consider `Rightmost-LZ77` (Section 4.2), which uses an unbounded window and selects the rightmost copy of the currently longest phrase. As expected, this parsing combined with the use of variable-length encoders improves `Fixed-LZ77`, thus sustaining in practice the starting point of our theoretical investigation.

Finally, we tested our bit-optimal compressor `BitOptimal-LZ77` finding that it improves `Rightmost-LZ77`, as theoretically predicted in Lemma 4. Surprisingly, `BitOptimal-LZ77` significantly improves `bzip2` (which uses a bounded window) and comes close to the booster (which uses an unbounded window [43]). Additionally, since `BitOptimal-LZ77` adopts the same decompression algorithm of `gzip`, it retains its *fast decompression speed* which is at least one order of magnitude faster than decompressing `bzip2`'s or `booster`'s compressed files. This is a nice combination which makes `BitOptimal-LZ77`



practically relevant for a wide range of applications in which the paradigm is “compress once & decompress many times” (like in Web search engines and IR systems), or where the decompression system is less powerful than the compressor one (like a server that distributes data to clients, possibly mobile phones).

## Fast random access on compressed data

Starting from [52] and [81, 145], the design of compressed (self)indexes for strings became an active field of research (some of these results will be presented in Chapters 6 and 7). The key problem addressed in these papers consists of representing a string  $T[1, n]$  drawn from an alphabet  $\Sigma$  of size  $\sigma$  within compressed space, and still be able to solve the Text Searching Problem in efficient time, without incurring in the whole decompression of the compressed data. In these results, compressed space usually means space close to the  $k$ -th order empirical entropy of  $T$ , and efficient time means something depending on the length of the searched string and as much independent as possible of  $T$ 's length.

Recently, Sadakane and Grossi [148] addressed the foundational problem of designing a compressed storage scheme for a string  $T$  in which the query operation to be supported is the retrieval of any  $\ell$ -long substring of  $T$  in optimal  $O(1 + \frac{\ell}{\log_{\sigma} n})$  time. The previous known solutions [134] were based on compressed indexes. The main drawback of these solutions is given by the fact that they incur in an additional sub-logarithmic time overhead. Instead, the Sadakane-Grossi's storage scheme achieves the optimal time bound and occupies a number of bits upper bounded by the following function:<sup>1</sup>  $nH_k(T) + O(\frac{n}{\log_{\sigma} n} ((k + 1) \log \sigma + \log \log n))$ , where, as usual,  $H_k(T)$  is the  $k$ -th order entropy of string  $T$  (recall Definition 2.4.3). This storage scheme is based on a

---

<sup>1</sup>As stated in [79], the term  $(k + 1) \log \sigma$  appears erroneously as  $k$  in [148]. We therefore use the correct bound in this chapter.

sophisticated combination of various techniques: Ziv-Lempel's string encoding [167], succinct dictionaries [143], and some novel succinct data structures for supporting navigation and path-decoding in LZ-tries. Since storing  $T$  by means of a *plain* array of symbols takes  $\Theta(n \log \sigma)$  bits, the scheme in [148] is effective when  $k = o(\log_\sigma n)$ .

González and Navarro [79] proposed a simpler storage scheme achieving the same query time and a slightly improved space bound:

$$nH_k(T) + O\left(\frac{n}{\log_\sigma n} (k \log \sigma + \log \log n)\right) \quad (5.0.1)$$

This storage scheme exploits a statistical encoder (namely, Arithmetic) on most of  $T$ 's substrings but, unlike [148], requires to fix the order  $k$  of the entropy bound in advance.

In what follows we propose a very simple storage scheme that: (1) drops the use of any compressor (either statistical or LZ-like), and deploys only binary encodings and tables; (2) matches the space bound of Eqn. (5.0.1) simultaneously over all  $k = o(\log_\sigma n)$ . We then exploit this storage scheme to achieve two corollary results. The first one provides a novel bound in terms of  $H_k(T)$  on the compression ratio achievable by any 0-th order compressor applied on blocks of  $l$  contiguous symbols of  $T$ , with  $k \leq l$  (see Theorem 10). The second result shows that our storage scheme can be used upon the string  $\text{Bwt}(T)$  in order to achieve an interesting compressed-space bound which depends on the  $k$ -th order entropy of both the strings  $T$  and  $\text{Bwt}(T)$  (see Theorem 11).

Recently, Theorem 10 has been used in [77] to design a scheme which achieves the same query time and has a slightly improved space bound. Essentially, they use our construction combined with recent techniques in a way that allows them to remove the term  $O\left(\frac{n}{\log_\sigma n} \log \log n\right)$  in Eqn. (5.0.1), so that its space bound becomes  $nH_k(T) + O\left(\frac{n}{\log_\sigma n} k \log \sigma\right)$  bits. This implies that their solution is better than ours only for very small value of  $k$  (namely,  $k = o\left(\frac{\log \log n}{\log \sigma}\right)$ ).

## 5.1 Our storage scheme for strings

Let  $T[1, n]$  be a string drawn from an alphabet  $\Sigma$ , and assume that  $n$  is a multiple of  $b = \lfloor \frac{1}{2} \log_\sigma n \rfloor$ . If this is not the case, we append to  $T$  the missing symbols taking them as the special *null symbol*.<sup>2</sup> We partition  $T$  into blocks

<sup>2</sup>This will add to the entropy estimation a negligible additive term equal to  $O(\log \sigma \log_\sigma n) = O(\log n)$  bits.

$T^i$  of size  $b$  each. Let  $\mathcal{S}$  be the set of distinct blocks of  $T$ . The number of all blocks is  $\frac{n}{b}$ ; the number of distinct blocks is  $|\mathcal{S}| = O(\sigma^b) = O(n^{1/2})$ . We define  $\mathcal{B} = [\epsilon, 0, 1, 00, 01, 10, 11, 000, \dots]$  to denote the infinite sequence of binary strings ordered first by length and then lexicographically by their content, with  $\epsilon$  denotes the empty string.

**The encoding scheme.** We sort the elements of  $\mathcal{S}$  per decreasing frequency of occurrence in  $T$ 's partition. Let  $r(T^i)$  be the rank of the block  $T^i$  in this ordering, and let  $r^{-1}(j)$  be its inverse function (namely, the one that returns the block having the given rank  $j$ ). The storage scheme for  $T$  consists of the following information.

- Each block  $T^i$  is assigned a codeword  $\text{enc}(i)$  consisting of the binary string that has rank  $r(T^i)$  in  $\mathcal{B}$ . It is simple to see that  $|\text{enc}(i)| \leq \log i \leq \frac{1}{2} \log n$ . Of course,  $\text{enc}(i)$  is not a *uniquely decodable code*, but the additional tables built below will allow us to decode it in constant time and within a space bounded by Eqn. (5.0.1).
- We build a bit sequence  $V$  obtained by juxtaposing the binary encodings of all  $T$ 's blocks in the order of their appearance in  $T$ . Namely  $V = \text{enc}(1) \cdots \text{enc}(\frac{n}{b})$ .
- We store  $r^{-1}$  as a table of  $O(\sigma^b)$  entries, taking  $O(\sigma^b \log n) = o(n)$  bits.
- To guarantee constant-time access to the encodings of  $T$ 's blocks and to ensure their decodings, we use a *two-level* storage scheme for the starting positions of  $\text{enc}$ s (see [128]). Specifically, we *logically* group every  $c = \Theta(\log n)$  contiguous blocks into one *superblock*, having thus size  $bc \log \sigma = \Theta(\log^2 n)$  bits. Table  $D_{Sblk}[1, \frac{n}{bc}]$  stores the starting position of the encoding of every super-block in  $V$ , and table  $D_{blk}[1, \frac{n}{b}]$  stores the starting position in  $V$  of the encoding of every block *relative to* the beginning of its enclosing super-block. Note that the starting position of each super-block is no more than  $|V| = O(\frac{n}{b} \log n) = O(n \log \sigma)$ , whereas the relative position of each block within its super-block is  $O(\log^2 n)$ . Consequently, tables  $D_{Sblk}$  and  $D_{blk}$  occupy  $O(\frac{n}{bc} \log |V| + \frac{n}{b} \log \log n) = O(\frac{n \log \log n}{\log_\sigma n})$  bits overall, and guarantee a constant-time access to every codeword  $\text{enc}(i)$  and its length.<sup>3</sup>

**Theorem 8.** *Our storage scheme encodes  $T[1, n]$  in  $|V| + O(\frac{n \log \log n}{\log_\sigma n})$  bits, which is upper bounded by Eqn. (5.0.1), simultaneously over all  $k = o(\log_\sigma n)$ .*

<sup>3</sup>It suffices to compute the starting position of  $\text{enc}(i)$  and  $\text{enc}(i+1)$ , if any.

**Proof.** For every position  $i$ ,  $k \leq i \leq n$ , let us denote by  $f_i$  denotes the empirical probability of seeing  $T[i]$  after the  $k$ -order context  $T[i - k, i - 1]$ . According to Definition 2.4.3, this can be rephrased by saying that  $f_i$  is the frequency of occurrence of symbol  $T[i]$  within  $u_T$ , where  $u = T[i - k, i - 1]$ . It is easy to see that a (semi-static)  $k$ -order modeler can compute all the frequencies  $f_i$  via two passes over  $T$ , hence in  $O(n)$  time.

Arithmetic encoding is the most effective statistical encoder [164]. Given the  $f_i$ s, it represents the string  $T$  with a range of size  $F = f_1 \times f_2 \times \dots \times f_n$ . It is well known [164] that  $2 + \log(1/F) = 2 + \sum_{i=1}^n \log(1/f_i)$  bits are enough to distinguish a number within that range. The binary representation of this number is the Arithmetic compression of  $T$ . If we compute  $\sum_{i=k+1}^n \log(1/f_i)$ , and then group all the terms referring to the same  $k$ -th order context, we obtain a summation upper bounded by  $nH_k(T)$  (see Eqn. (2.4.3)). Additionally, since  $f_i \geq 1/n$ , we have that  $\sum_{i=1}^k \log(1/f_i) = O(k \log n)$ . As a result, a (semi-static)  $k$ -th order Arithmetic encoder compresses the whole  $T$  within  $nH_k(T) + 2 + O(k \log n)$  bits.

Let us introduce a compressor  $E$  that encodes each block  $T^i$  of  $T$  individually: the first  $k$  symbols of  $T^i$  are represented explicitly, the remaining  $b - k$  symbols of  $T^i$  are compressed via the above  $k$ -order Arithmetic encoder (hence using their  $k$ -th order frequencies  $f$ s). It is easy to observe that the codeword so assigned to  $T^i$  uniquely identifies it. This blocking approach increases the above Arithmetic encoding of the whole  $T$  by  $O((n/b)k \log \sigma)$  bits, which accounts for the cost of explicitly storing the first  $k$  symbols of each  $T^i$ .

To show that the string  $V$  produced by our storage scheme `enc` is shorter than the compressed string produced by  $E$ , it suffices to note that the codewords assigned by  $E$  are a subset of  $\mathcal{B}$ , whereas the codewords assigned by `enc` are the first  $|\mathcal{S}|$  binary strings of  $\mathcal{B}$ . Given that  $\mathcal{B}$  is the set of the shortest codewords assignable to  $\mathcal{S}$ 's strings, our encoding `enc` is better than  $E$  because it follows the *golden rule* of data compression: it assigns shorter codewords to more frequent symbols. Summing up the cost of the block's encodings and the space occupancy of the decoding table, we get the space bound of Eqn. (5.0.1), whenever  $k = o(\log_{\sigma} n)$  and independently of it. ■

We now show how to decode in constant time a generic block  $T^k$ . This will be enough to prove the result for any  $\ell$ -long substring of  $T$ . We first derive the starting position  $p(k)$  of the string `enc(k)` that encodes  $T^k$  in  $V$ . Namely, we compute the super-block number  $h = \lceil k/c \rceil$  containing `enc(k)`, and its starting bit-position  $y = D_{Sblk}[h]$  within  $V$ . Then, we compute  $x = D_{blk}[k]$  as the relative bit-position of `enc(k)` within its enclosing super-block. Thus  $p(k) = x + y$ . Similarly, we derive the starting position  $p(k + 1)$  of `enc(k + 1)` in  $V$  (if any, oth-

erwise we set  $p(k+1) = |V| + 1$ . We can thus fetch  $\text{enc}(k) = V[p(k), p(k+1) - 1]$  in constant time since  $|\text{enc}(k)| = p(k+1) - p(k) = O(\log n)$  bits.

We finally decode  $\text{enc}(k)$  as follows. Let  $v$  be the integer value represented by the binary string  $\text{enc}(k)$ , where  $v = 0$  if  $\text{enc}(k) = \epsilon$ . Because of the canonical ordering of  $\mathcal{S}$ ,  $T^k$  is computed as the block having rank  $z = 2^{|\text{enc}(k)|} + v$ . That is,  $T^k = r^{-1}(z)$ .

**Theorem 9.** *Our storage scheme retrieves any substring of  $T$  of length  $\ell$  in optimal  $O(1 + \frac{\ell}{\log_\sigma n})$  time.*

**Proof.** The algorithm described above allows to retrieve any block  $T^k$  in constant time. The theorem follows by observing that any  $\ell$ -long substring  $T[j, j + \ell - 1]$  spans  $O(1 + \frac{\ell}{\log_\sigma n})$  blocks of  $T$ . ■

## 5.2 Huffman on blocks of symbols

It is well-known that the Huffman compressor cannot represent a symbol with less than one bit. To circumvent this, the string  $T$  is usually partitioned into  $\frac{n}{l}$  blocks of length  $l$  each, and then Huffman is applied onto the alphabet  $\Sigma_l$  of these new symbols, i.e.  $l$ -long blocks. This blocking strategy spreads the per-symbol inefficiency over the entire block, thus reducing it to  $\frac{1}{l}$  bits. It is natural to ask what is the compression ratio of this block-Huffman algorithm. The following theorem bounds the 0-th order entropy of  $T_l$  by the  $k$ -th order empirical entropy of original string  $T$ .

**Theorem 10.**  $H_0(T_l) \leq lH_k(T) + O(k \log \sigma)$ , simultaneously over all  $k \leq l$ .

**Proof.** Consider the compressor  $E$  in the proof of Theorem 8.  $E$  does not depend on the size of the blocks in which  $T$  has been decomposed. Hence, we can set  $b = l$ , apply  $E$  onto the blocked  $T$  and thus assign a distinct prefix-free codeword to each distinct block in  $\mathcal{S}$  (i.e. symbol of  $\Sigma_l$ ). As seen in that proof, the space necessary to represent the whole  $T_l$  is bounded by  $nH_k(T) + O((n/l)k \log \sigma)$  bits. The stated theorem follows by the classical Information-Theory lower bound, since every prefix-free encoder needs on  $T_l$  at least  $|T_l|H_0(T_l) = \frac{n}{l}H_0(T_l)$  bits. ■

We note that the case  $k = 0$  of Theorem 10 has been proved in [145]. Given Theorem 10, the output produced by Huffman over  $T_l$  is bounded by  $\frac{n}{l}H_0(T_l) + \frac{n}{l} \leq nH_k(T) + O(\frac{n}{l}(k \log \sigma + 1))$ .

### 5.3 BWT compression and access

In this section we show that our storage scheme can be used upon the string  $\text{Bwt}(T)$  in order to achieve an interesting compressed-space bound which depends on both  $H_k(T)$  and  $H_k(\text{Bwt}(T))$ . The relation between these two entropies will be commented below.

**Theorem 11.** *Our storage scheme applied on the string  $L = \text{Bwt}(T)$  takes no more than  $\min\{nH_k(L), nH_k(T)\} + o(n \log \sigma)$  bits, simultaneously over all  $k = o(\log_\sigma n)$ . Any  $\ell$ -long substring of  $L$  can be retrieved in optimal  $O(1 + \frac{\ell}{\log_\sigma n})$  time.*

**Proof.** Let  $C_k(j)$  be the  $k$ -long prefix of the  $j$ -th row of  $\mathcal{M}_T$ . By the properties of  $\text{Bwt}(T)$  (see Section 2.4.2),  $C_k(j)$  follows  $L[j]$  in  $T$  and thus  $C_k(j)$  is the *following*  $k$ -long context of  $L[j]$ . The Definition 2.4.3 of  $H_k(T)$  can be changed by replacing the notion of *preceding*  $k$ -long contexts with the one of *following*  $k$ -long contexts. The difference between these two quantities results negligible [52]. Therefore, to ease our discussion we consider the *following*  $k$ -long contexts and work with  $H_k(T)$  as it were defined over them.

We partition  $L$  into substrings of length  $b = \lfloor \frac{1}{2} \log_\sigma n \rfloor$ , say  $L_1 L_2 \dots L_{n/b}$  (called hereafter *blocks*). Note that each block  $L_i$  corresponds to a range of  $b$  rows in the BWT-matrix  $\mathcal{M}_T$ . We say that the block  $L_i$  is *k-prefix-equal* if all rows in  $\mathcal{M}_T[(i-1)b+1, ib]$  have the same prefix of length  $k$ . Otherwise,  $L_i$  is said to be *k-prefix-different*. The total number of  $k$ -prefix-different blocks is  $O(\sigma^k)$ , because that is the number of distinct strings of length  $k$  over  $\Sigma$ . Moreover, we note that the first  $k$  symbols of  $T$  belong to  $k$ -prefix-different blocks because of the special symbol  $\$$ .

To prove the Theorem we consider a preliminary encoding scheme for  $L$ 's blocks. A  $k$ -prefix-different block  $L_i$  is written without any compression as  $k$  occurrences of a special *null* symbol plus  $L_i$  itself. This takes  $O((b+k) \log \sigma) = O(\log n + k \log \sigma)$  bits. Since there are  $O(\sigma^k)$   $k$ -prefix-different blocks and we assumed  $k = o(\log_\sigma n)$ , the (plain) encoding of the  $k$ -prefix-different blocks takes  $O(\sigma^k \log n) = o(n)$  bits.

As far as  $k$ -prefix-equal blocks are concerned, we encode a block  $L_i$  as follows: we write explicitly the  $k$ -long following context shared by all  $L_i$ 's symbols, using  $k \log \sigma$  bits; and then use a  $k$ -th order Arithmetic encoder on the individual symbols of  $L_i$ . This encoder computes for any symbol  $L[j]$  the empirical probability  $f_j$  of seeing this symbol *followed* by the context  $C_k(j)$  in  $T$ . In the proof of Theorem 8, we considered  $f$  as the preceding contexts and showed that a (semi-static)  $k$ -th order Arithmetic encoder that uses the  $f$ s, compresses  $T$  within  $H_k(T) + (n/b)k \log \sigma + O(k \log n) + 2$  bits. Here we are compress-

ing  $L$ , which is a permutation of string  $T$ , and we are considering the following contexts of  $T$ 's symbols. Given our comment above on the definition of  $H_k(T)$ , we can conclude that this bound still holds for the Arithmetic encoder applied on  $L$ . Summing up, the space required by this encoding scheme over all blocks of  $L$  is  $nH_k(T) + O((n/b)k \log \sigma)$  bits.

Let us now take our storage scheme `enc` of Section 5.1, apply it onto string  $L$ , and compare the length of the resulting compressed string against the previous encoding. By Theorem 8, we know that `enc` encodes  $L$  within  $nH_k(L) + O((n/b)k \log \sigma)$  bits, and this proves one part of the theorem. As far as the other term  $H_k(T)$  is concerned, we observe that any block  $L_i$  may occur many times in the partition of  $L$  and each occurrence may have associated a different  $k$ -long following context. As a result, the above scheme encodes all occurrences of  $L_i$  with at most  $O(\sigma^k)$  different codewords, because it has at most  $\sigma^k$  distinct  $k$ -contexts (as a  $k$ -prefix-equal block) and at most  $\sigma^k$  plain encodings (as a  $k$ -prefix-different block). If we re-assign to all these codewords the shortest one, we have that each distinct block of  $L$  gets one codeword in  $\mathcal{B}$ . Following an argument similar to the one used in the proof of Theorem 8, this encoding of  $L$ 's blocks is worse than `enc` because this latter assigns the shortest codewords of  $\mathcal{B}$  to the distinct blocks of  $L$ . Therefore `enc` takes no more than  $nH_k(T) + O((n/b)k \log \sigma)$  bits. ■

The relation between  $H_k(T)$  and  $H_k(L)$  is not fully known. In [79] they proved that  $H_1(L) \leq 1 + H_k(T) \log \sigma + o(1)$  for any  $k < (1 - \epsilon) \log_\sigma n$  and any constant  $0 < \epsilon < 1$ . Actually the gap may be quite large. For example, let us consider the string  $T = (bba)^m$  and set  $k = 1$ . By Eqn. 2.4.3 we have

$$nH_1(T) = (m-1)H_0(b^{m-1}) + 2mH_0((ba)^m) = 2m = \frac{2}{3}n$$

On the other hand, since  $L = \mathbf{Bwt}(T) = b^{2m}a^m$ , we have

$$\begin{aligned} nH_1(L) &= (m-1)H_0(a^{m-1}) + 2mH_0(b^{2m-1}a) \\ &= -(2m-1) \log \frac{2m-1}{2m} - \log \frac{1}{2m} \\ &= 2m \log 2m - (2m-1) \log(2m-1) \\ &= O(\log n) \end{aligned}$$

which is exponentially smaller than  $nH_1(T)$ , for any  $m > 1$ . On the other side, we show an example in which  $nH_1(L) > nH_1(T)$ . Let  $T = (a_1 a_2 \dots a_m)^m$  and  $k = 1$ . We have  $nH_1(T) = 0$ . Since  $L = \mathbf{Bwt}(T) = a_m^m a_1^m \dots a_{m-1}^m$ , we have

$$nH_1(L) = -(m-1)((m-1) \log \frac{m-1}{m} + \log \frac{1}{m}) = \Theta(\sqrt{n} \log \sqrt{n})$$





## Experiments on compressed full-text indexing

Most of the manipulations required over texts involve, sooner or later, *searching* those (usually long) sequences for (usually short) pattern sequences. Not surprisingly, text searching and processing has been a central issue in Computer Science research since its beginnings.

Despite the increase in processing speeds, sequential text searching long ago ceased to be a viable alternative for many applications, and indexed text searching has become mandatory. A *text index* is a data structure built over a text which significantly speeds up searches for arbitrary patterns, at the cost of some additional space. The inverted list structure (see e.g. [164]) is an extremely popular index to handle so-called “natural language” texts, due to its simplicity, low space requirements, and fast query execution. An inverted list is essentially a table recording the positions of the occurrences of every distinct word in the indexed text. Thus every word-based query is already pre-computed, and phrase queries are carried out via list intersections. These design features made inverted lists the *de facto* choice for the implementation of Web search engines and IR systems (see e.g. [164], [168], and references therein).

There are contexts, however, in which either the texts or the queries cannot be factored out into sequences of words, or it is the case that the number of distinct words is so large that indexing all of them in a table would be too much space consuming. Typical examples include bio-informatics, computational linguistics, multimedia databases, and search engines for agglutinating

and Far East languages. In these cases texts and queries must be modeled as arbitrarily long sequences of symbols and an index for these types of texts, in order to be efficient, must be able to search and retrieve any *substring* of *any* length. These are nowadays the so called *full-text indexes*.

As we already discussed in Section 2.3, classical full-text indexes wasted a lot of space: Data structures like suffix trees and suffix arrays require at the very least four times the text size (plus text) to achieve reasonable efficiency [2, 83]. Several engineered versions achieved relevant, yet not spectacular, reductions in space [5, 74, 95, 99, 100]. For example, space consumptions like 2.5 times the text size, plus text, were reported (see survey by [133]).

Although space consumption by itself is not usually a problem today given the availability of cheap massive storage, the access speed of that storage has not improved much, while CPU speeds have been doubling every 24 months, as well the sizes of the various (internal) memory levels. Given that nowadays an access to the disk can be up to one million times slower than main memory, it is often mandatory to fit the index in internal memory and leave as few data as possible onto disk.

A folklore alternative way to further reduce the space of full-text indexes are the so-called *q-gram* indexes [93, 112, 132, 140, 155, 158, 163] (more references in [133]). This can be seen as an adaptation of the inverted-list scheme that takes as a “word” any *q-gram* occurring in the indexed text (i.e., any substring of length  $q$ ), and stores all occurrences of these *q-grams* within a table. Queries are solved by intersecting/joining lists of *q-gram* occurrences depending on whether the query pattern is longer/shorter than  $q$ . In principle this index can take as much as four times the text size, as each text position starts a *q-gram* and thus spends one integer in some list. The space can be alleviated by several means, already explored in the cited papers: (1) compressing the inverted lists, since they contain increasing numbers, as done for classical inverted indexes [164]; (2) indexing spaced text *q-grams*, while still finding all the occurrences, e.g. [155]; (3) using *block-addressing*, first introduced to reduce the space of natural-language indexes [120] and later extended to *q-gram* indexes, e.g. [112, 140]. In block-addressing the text is divided into blocks and the index only points to the blocks where the *q-grams* appear. This permits achieving very little index space at the price of having to scan the candidate text blocks. Hence the text must be separately available in a form that permits fast scanning. Block-addressing has been successfully combined with the compression of the text in the case of natural language [135], but not in general text as far as we know. Besides needing the text in plain form, these indexes do not offer relevant worst-case search time guarantees.

---

This situation is drastically changed in the last decade [134]. Starting in the year 2000, a rapid sequence of achievements showed how to relate information theory with string matching concepts. The regularities in compressible texts were exploited to reduce index occupancy without impairing the query efficiency. 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 — hence the name *self-indexes*. This way *compressed full-text self-indexes* (compressed indexes, for short) allow one to add search and random access functionalities to compressed data with a negligible penalty in time and space performance. For example, it is feasible today to index the 3 GB Human genome on a 1 GB RAM desktop PC.

The content of this chapter is primarily devoted to a practical study of this novel technology. Although a comprehensive survey of theoretical aspects has recently appeared [134], the algorithmics underlying these compressed indexes require for their implementation a significant programming skill, a deep engineering effort, and a strong algorithmic background. To date only isolated implementations and focused comparisons of compressed indexes have been reported, and they missed a common API, which prevented their re-use or deploy within other applications. The present work has therefore a threefold purpose:

**Algorithmic Engineering.** 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 as on their limitations. We think that this point of view complements [134] 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 two novel implementations of compressed indexes. These correspond to new versions of the FM-Index data structure, one of which combines the best existing theoretical guarantees with a competitive space/time tradeoff in practice.

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

**Technology Transfer.** We introduce the *Pizza&Chili* site<sup>1</sup>, which was devel-

---

<sup>1</sup>Available at two mirrors: `pizzachili.dcc.uchile.cl` and `pizzachili.di.unipi.it`

oped 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, in our intention, 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. Every time one needs to store a set of strings which must be subsequently accessed for query-driven or id-driven string retrieval, one can use a compressed index with the goal of squeezing the dictionary space without slowing down the query performance. This is the subtle need that any programmer faces when implementing hash tables, tries or other indexing data structures. Actually, the use of compressed indexes has been successfully extended to handle several other more sophisticated data structures, such as dictionary indexes [56, 59] (see Chapter 7), labeled trees [48, 50], graphs [38], etc. Dealing with all those applications is out of the scope of this chapter, whose main goal is to address the above three issues, and comment on the experimental behavior of this new algorithmic technology.

This chapter is organized as follows. Section 6.1 explains the key conceptual ideas underlying the most relevant compressed indexes. Section 6.2 describes how the indexes implement those basic ideas. Section 6.3 presents the *Pizza&Chili* site, and the next Section 6.4 comments on a large suite of experiments aimed at comparing the most successful implementations of the compressed indexes present in this web site.

## 6.1 Basics and background

We recall from Section 2.3 that the *text searching problem* is then stated as follows. Given a *text* string  $T[1, n]$  and a *pattern*  $P[1, p]$ , we wish to answer the following queries: (1) *count* the number of occurrences (*occ*) of  $P$  in  $T$ ; (2) *locate* the *occ* positions in  $T$  where  $P$  occurs. In this chapter we assume that  $T$  can be preprocessed, and an *index* is built on it, in order to speed up the execution of subsequent queries. We assume that the cost of index construction is amortized over sufficiently many searches, as otherwise sequential searching is preferable.

In the case of self-indexes, which replace the text, a third operation of interest is (3) *extract* the substring  $T[l : r]$ , given positions  $l$  and  $r$  in  $T$ .

In Section 2.3 we described suffix trees and suffix arrays which are the most important classical full-text indexes whose main drawback is their space occupancy (namely,  $\Theta(n \log n)$  bits).

### 6.1.1 Backward Search

In Section 2.3.2 we described the classical binary-search method over suffix arrays. Here we review an alternative approach which has been recently proposed in [52], hereafter named *backward search*. For any  $i = p, p - 1, \dots, 1$ , this search algorithm keeps the interval  $SA[\text{First}_i : \text{Last}_i]$  storing all text suffixes which are prefixed by  $P[i : p]$ . This is done via two main steps:

**Initial step.** We have  $i = p$ , so that it suffices to access a precomputed table that stores the pair  $[\text{First}_p, \text{Last}_p]$  for all possible symbols  $P[p] \in \Sigma$ .

**Inductive step.** Let us assume to have computed the interval  $SA[\text{First}_{i+1} : \text{Last}_{i+1}]$ , whose suffixes are prefixed by  $P[i + 1 : p]$ . The present step determines the next interval  $SA[\text{First}_i : \text{Last}_i]$  for  $P[i : p]$  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.

The backward-search algorithm is executed by decreasing  $i$  until either an empty interval is found (i.e.  $\text{First}_i > \text{Last}_i$ ), or  $SA[\text{First}_1 : \text{Last}_1]$  contains all pattern occurrences. In the former case no pattern occurrences are found; in the latter case the algorithm has found  $\text{occ} = \text{Last}_1 - \text{First}_1 + 1$  pattern occurrences.

### 6.1.2 Rank Query

Given a string  $S[1, n]$ , function  $\text{RANK}_x(S, i)$  returns the number of times symbol  $x$  appears in the prefix  $S[1 : i]$ . Rank queries are central to compressed indexing, so it is important to understand how they are implemented and how much space/time they need. We have two cases depending on the alphabet of  $S$ . For the aims of this chapter it suffices to pose our attention on practical implementations and defer the discussion on theoretical results to next chapter which is more focused on theoretical aspects.

**Rank over Binary Sequences.** In this case there exist simple and practical constant-time solutions using  $o(n)$  bits of space in addition to  $S$  [128]. We cover only  $\text{RANK}_1$  as  $\text{RANK}_0(S, i) = i - \text{RANK}_1(S, i)$ . The solution partitions  $S$  into blocks of size  $s$ , and stores explicit answers for rank-queries done at block

beginnings. One of the best practical implementations of the idea [78] solves  $\text{RANK}_1(S, i)$  by summing two quantities: (1) the pre-computed answer for the prefix of  $S$  which ends at the beginning of the block enclosing  $S[i]$ , plus (2) the *relative* rank of  $S[i]$  within its block. The latter is computed via a byte-wise scanning of the block, using small precomputed tables. This solution involves a space/time tradeoff related to  $s$ , but nonetheless its query-time performance is rather satisfactory already with 5% space overhead on top of  $S$ .

**Rank over General Sequences.** Given a sequence  $S[1, n]$  over an alphabet of size  $\sigma$ , the *wavelet tree* [62, 81] 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 node  $v$  the subsequence  $S^v$  of  $S$  formed by the symbols in  $\Sigma^v$ . Sequence  $S^v$  is not really stored at the node, but it is replaced by a bit sequence  $B^v$  such that  $B^v[i] = 0$  iff  $S^v[i]$  is a symbol whose leaf resides in the left subtree of  $v$ . Otherwise,  $B^v[i]$  is set to 1.

The power of the wavelet tree is to reduce rank operations over general alphabets to rank operations over a binary alphabet, so that the rank-machinery above can be used in each wavelet-tree node. Precisely, let us answer the query  $\text{RANK}_c(S, i)$ . 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  $i = \text{RANK}_1(B^v, i)$  and move to the right child of  $v$ . Similarly, if  $c$  belongs to the left subtree, we set  $i = \text{RANK}_0(B^v, i)$  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  $\text{RANK}_c(S, i)$ . Since any binary-rank takes  $O(1)$  time, the overall rank 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  $i = \text{RANK}_0(B^v, i)$  and go to the left child. Similarly, if  $B^v[i] = 1$ , then we set  $i = \text{RANK}_1(B^v, i)$  and go to the right child. We repeat this 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 requires comparable space to the original sequence, as it requires  $n \log \sigma (1 + o(1))$  bits of space. A practical way to reduce the space occupancy to the zero-order entropy of  $S$  is to replace the balanced tree structure by the Huffman tree of  $S$ . Now we have to follow the binary Huffman code of a symbol to find its place in the tree. It is not hard to see 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 and access operations is  $O(H_0(S))$ , where  $H_0$

is the zero-th order empirical entropy of  $S$  (see next section). This structure is the key tool in our implementation of *SSA* or *AF-index* (Section 6.4).

## 6.2 Compressed Indexes

Compressed indexes provide a viable alternative to classical indexes that are parsimonious in space and efficient in query time. 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 [134]. In theoretical terms, the most succinct indexes achieve  $nH_k(T) + o(n \log \sigma)$  bits of space, and for a fixed  $\epsilon > 0$ , require  $O(p \log \sigma)$  counting time,  $O(\log^{1+\epsilon} n)$  time per located occurrence, and  $O(\ell \log \sigma + \log^{1+\epsilon} n)$  time to extract a substring of  $T$  of length  $\ell$ .<sup>2</sup> This is a surprising result because it 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 time.

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

### 6.2.1 The FM-index Family

The FM-index is composed of a compressed representation of  $\text{Bwt}(T)$  plus auxiliary structures for efficiently computing generalized rank queries on it. The main idea [52] is to obtain a text index from the BWT and then use backward searching for identifying the pattern occurrences (Sections 6.1.1 and 2.4.2). Several variants of this algorithmic scheme do exist [51, 52, 53, 114] 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  $SA[\text{First} : \text{Last}]$  of text suffixes prefixed by it (or, which is equivalent, the interval of rows of the matrix  $\mathcal{M}_T$  prefixed by  $P$ , see Section 2.4.2). Figure 6.1 gives

<sup>2</sup>These locating and extracting complexities are better than those reported in [53], and can be obtained by setting the sampling step to  $\frac{\log^{1+\epsilon} n}{\log \sigma}$ .



the pseudocode to compute First and Last.

---

**Algorithm** FM-count( $P[1, p]$ )

1.  $i = p$ , First = 1, Last =  $n$ ;
  2. **while** ((First  $\leq$  Last) **and** ( $i \geq 1$ )) **do**
  3.    $c = P[i]$ ;
  4.   First =  $C[c] + \text{RANK}_c(L, \text{First} - 1) + 1$ ;
  5.   Last =  $C[c] + \text{RANK}_c(L, \text{Last})$ ;
  6.    $i = i - 1$ ;
  7. **if** (Last < First) **then return** “no rows prefixed by  $P$ ” **else return** [First, Last];
- 

**Figure 6.1:** Algorithm to get the interval  $SA[\text{First} : \text{Last}]$  of text suffixes prefixed by  $P$ , using an FM-index.

The algorithm is correct: Let  $[\text{First}_{i+1}, \text{Last}_{i+1}]$  be the range of rows in  $\mathcal{M}_T$  that start with  $P[i + 1 : p]$ , and we wish to know which of those rows are preceded by  $P[i]$ . These correspond precisely to the occurrences of  $P[i]$  in  $\text{Bwt}(T)[\text{First}_{i+1} : \text{Last}_{i+1}]$ . Those occurrences, mapped to the first column of  $\mathcal{M}_T$ , form a (contiguous) range that is computed with a rationale similar to that for  $LF(\cdot)$  in Section 2.4.2, and thus via a just two rank operations on  $\text{Bwt}(T)$ .

**Locating.** Algorithm in Figure 6.2 obtains the position of the suffix that prefixes the  $i$ -th row of  $\mathcal{M}_T$ . The basic idea is to logically mark a suitable set of rows of  $\mathcal{M}_T$ , and keep for each of them their position in  $T$  (that is, we store the corresponding  $SA$  values). Then,  $\text{FM-locate}(i)$  scans the text  $T$  backwards using the LF-mapping until a marked row  $i'$  is found, and then it reports  $SA[i'] + t$ , where  $t$  is the number of backward steps used to find such  $i'$ . To compute the positions of all occurrences of a pattern  $P$ , it is thus enough to call  $\text{FM-locate}(i)$  for every  $\text{First} \leq i \leq \text{Last}$ .

The sampling rate of  $\mathcal{M}_T$ 's rows, hereafter denoted by  $s_{SA}$ , is a crucial parameter that trades space for query time. Most FM-index implementations mark all the  $SA[i]$  that are a multiple of  $s_{SA}$ , via a bitmap  $B[1, n]$ . All the marked  $SA[i]$ s are stored contiguously in suffix array order, so that if  $B[i] = 1$  then one finds the corresponding  $SA[i]$  at position  $\text{RANK}_1(B, i)$  in that contiguous storage. This guarantees that at most  $s_{SA}$  LF-steps are necessary for locating the text position of any occurrence. The extra space is  $\frac{n \log n}{s_{SA}} + n + o(n)$  bits.

A way to avoid the need of bitmap  $B$  is to choose a symbol  $c$  having some

**Algorithm FM-locate( $i$ )**

1.  $i' = i, t = 0;$
2. **while**  $SA[i']$  is not explicitly stored **do**
3.    $i' = LF(i');$
4.    $t = t + 1;$
5. **return**  $SA[i'] + t;$

---

**Figure 6.2:** Algorithm to obtain  $SA[i]$  using an FM-index.

---

suitable frequency in  $T$ , and then store  $SA[i]$  if  $\mathbf{Bwt}(T)[i] = c$  [51]. Then the position of  $SA[i]$  in the contiguous storage is  $\text{RANK}_c(\mathbf{Bwt}(T), i)$ , so no extra space is needed other than  $\mathbf{Bwt}(T)$ . 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  $s_{SA}$ , we store the positions  $i$  such that  $SA[i]$  is a multiple of  $s_{SA}$  now in the text order (previously we followed the  $SA$ -driven order). To extract  $T[l : r]$ , we start from the first sample that follows the area of interest, that is, sample number  $d = \lceil (r + 1) / s_{SA} \rceil$ . From it we obtain the desired text backwards with the same mechanism for inverting the BWT (see Section 2.4.2), here starting with the value  $i$  stored for the  $d$ -th sample. We need at most  $s_{SA} + r - l + 1$  applications of the LF-step.

## 6.2.2 Implementing the FM-index

All the query complexities are governed by the time required to obtain  $C[c]$ ,  $\mathbf{Bwt}(T)[i]$ , and  $\text{RANK}_c(\mathbf{Bwt}(T), i)$  (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  $2p$  calls to  $\text{RANK}_c$ , locating requires  $s_{SA}$  calls to  $\text{RANK}_c$  and  $\mathbf{Bwt}(T)$ , and extracting  $\ell$  symbols requires  $s_{SA} + \ell$  calls to  $\text{RANK}_c$  and  $\mathbf{Bwt}(T)$ . In what follows we briefly comment on the solutions adopted to implement those basic operations.

The original FM-index implementation (*FM-index* [51]) compressed  $\mathbf{Bwt}(T)$  by splitting it into blocks and using independent zero-order compression on each block. Values of  $\text{RANK}_c$  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  $\mathbf{Bwt}(T)[i]$ . 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  $\text{Bwt}(T)$ . On the other hand, the time to decompress the block makes computation of  $\text{RANK}_c$  relatively expensive. For locating, this implementation marks the BWT positions where some chosen symbol  $c$  occurs, as explained above.

A very simple and effective alternative to represent  $\text{Bwt}(T)$  has been proposed with the *Succinct Suffix Array (SSA)* [53, 114]. It uses a Huffman-shaped wavelet tree, plus the marking of one out-of- $s_{SA}$  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  $\text{RANK}_c(\text{Bwt}(T), i)$  and  $\text{Bwt}(T)[i]$  is  $O(H_0(T) + 1)$ . The space bound is not 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 too.

The *Run-Length FM-index (RLFM)* [114] has been introduced to achieve  $k$ -th order compression by applying *run-length compression* to  $\text{Bwt}(T)$  prior to building a wavelet tree on it. The BWT generates long runs of identical symbols on compressible texts, 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 a bit, in comparison to the SSA.

### 6.2.3 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 [82]. Sadakane [144, 145] then proposed a variant which is a self-index and achieves space bound in term of 0-order entropy. The result in [81] described the first index that achieves high-order compression.

The CSA represents the suffix array  $SA[1, n]$  by a sequence of numbers  $\Psi(i)$ , such that  $SA[\Psi(i)] = SA[i] + 1$ . It is not hard to see [82] that  $\Psi$  is piecewise monotone increasing in the areas of  $SA$  where the suffixes start with the same symbol. In addition, there are long runs where  $\Psi(i + 1) = \Psi(i) + 1$ , and these runs can be mapped one-to-one to the runs in  $\text{Bwt}(T)$  [134]. These properties permit a compact representation of  $\Psi$  and its fast access. Essentially, we differentially encode  $\Psi(i) - \Psi(i - 1)$ , run-length encode the long runs of 1's occurring over those differences, and for the rest use an encoding favoring small numbers. Absolute samples are stored at regular intervals to permit the

efficient decoding of any  $\Psi(i)$ . The sampling rate (hereafter denoted by  $s_\Psi$ ) gives a space/time tradeoff for accessing and storing  $\Psi$ . In [145] 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 [134] to  $nH_k(T) + O(n \log \log \sigma)$  for any  $k \leq \alpha \log_\sigma n$  and constant  $0 < \alpha < 1$ .

**Counting.** The CSA [145] used the classical binary searching to count the number of pattern occurrences in  $T$ . The actual implementation, proposed in [144], uses backward searching (Section 6.1.1):  $\Psi$  is used to obtain  $[\text{First}_i, \text{Last}_i]$  from  $[\text{First}_{i+1}, \text{Last}_{i+1}]$  in  $O(\log n)$  time, for a total of  $O(p \log n)$  counting time. Precisely, let  $SA[\text{First}_i : \text{Last}_i]$  be the range of suffixes  $SA[j]$  that start with  $P[i]$  and such that  $SA[j] + 1 (= SA[\Psi(j)])$  starts with  $P[i+1 : p]$ . The former is equivalent to the condition  $[\text{First}_i, \text{Last}_i] \subseteq [C[P[i]] + 1, C[P[i] + 1]]$ . The latter is equivalent to saying that  $\text{First}_{i+1} \leq \Psi(j) \leq \text{Last}_{i+1}$ . Since  $\Psi(i)$  is monotonically increasing in the range  $C[P[i]] < j \leq C[P[i] + 1]$  (since the first symbols of suffixes in  $SA[\text{First}_i : \text{Last}_i]$  are the same), we can binary search this interval to find the range  $[\text{First}_i, \text{Last}_i]$ . Fig. 6.3 shows the pseudocode for counting using the CSA.

---

**Algorithm CSA-count( $P[1 : p]$ )**

1.  $i = p, \text{First} = 1, \text{Last} = n;$
2. **while**  $((\text{First} \leq \text{Last}) \text{ and } (i \geq 1))$  **do**
3.    $c = P[i];$
4.    $[\text{First}, \text{Last}] = [\text{min}, \text{max}] \{j \in [C[c] + 1, C[c + 1]], \Psi(j) \in [\text{First}, \text{Last}]\};$
5.    $i = i - 1;$
6. **if**  $(\text{Last} < \text{First})$  **then return** “no rows prefixed by  $P$ ” **else return**  $[\text{First}, \text{Last}];$

---

**Figure 6.3:** Algorithm to get the interval  $SA[\text{First}, \text{Last}]$  prefixed by  $P$ , using the CSA. The  $[\text{min}, \text{max}]$  interval is obtained via binary search.

**Locating.** Locating is similar to the FM-index, in that the suffix array is sampled at regular intervals of size  $s_{SA}$ . However, instead of using the LF-mapping to traverse the text backwards, this time we use  $\Psi$  to traverse the text forward, given that  $SA[\Psi(i)] = SA[i] + 1$ . 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  $\text{Bwt}(T)$ , 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  $\Psi$ , we can obtain  $T[SA[i] : n]$  symbolwise from  $i$ , as follows. The first symbol of the suffix pointed to by  $SA[i]$ , namely,  $T[SA[i]]$ , is the symbol  $c$  such that  $C[c] < i \leq C[c + 1]$ , because all the suffixes  $SA[C[c] + 1], \dots, SA[C[c + 1]]$  start with symbol  $c$ . Now, in order to obtain the next symbol,  $T[SA[i] + 1]$ , we compute  $i' = \Psi(i)$  and use the same procedure above to obtain  $T[SA[i']] = T[SA[i] + 1]$ , 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  $c = \text{RANK}_1(D, i)$ .

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

## 6.2.4 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 (e.g., [8, 52, 131]), we focus on the versions described in [8, 131] and available in the Pizza&Chili site. This index uses LZ78 parsing [167] to generate a partitioning of  $T[1 : n]$  into  $n'$  phrases,  $T = Z_1, \dots, Z_{n'}$ . 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 [131] is formed by (1) the LZ78 trie; (2) a trie formed with the reverse phrases  $Z_i'$ , 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  $Z_i'$  in the reverse phrases. The tree shapes in (1) and (2) are represented using parentheses and the encoding proposed in [130] 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  $k = o(\log_\sigma n)$  [134]. This can be reduced to  $(2 + \epsilon)nH_k(T) + o(n \log \sigma)$  by noticing that the mapping (3) is essentially the inverse permutation of the sequence of phrase identifiers in (1), and similarly (4) with (2) [7]. 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 [129].

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  $P[j + 1 : p]$  is a prefix of  $Z_{i+1}$ . For each  $j$ , we search for  $P^r[1 : j]$  in the reverse trie and  $P[j + 1 : p]$  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  $P[j + 1 : p]$  in the LZ78 trie. This can be done in constant time by comparing preorder numbers.
3.  $P$  spans three or more nodes. This implies that some phrase is completely contained in  $P$ , and since all phrases are different, there are only  $O(p^2)$  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 the LZ-index 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.

### 6.2.5 Novel Implementations

We introduce two novel compressed index implementations in this chapter. Both are variants of the FM-index family. The first one is interesting because it is a re-engineering of the first reported implementation of a self-index [51]. The second is relevant because it implements the self-index offering the best current theoretical space/time guarantees. It is fortunate, as it does not always happen, that theory and practice marry well and this second index is also relevant in the *practical* space/time tradeoff map.

## The FMI-2

As the original FM-index [51], the *FMI-2* adopts a two-level bucketing scheme for implementing efficient *rank* and *access* operations onto  $\text{Bwt}(T)$ . In detail, string  $\text{Bwt}(T)$  is partitioned into *buckets* and *superbuckets*: a bucket consists of  $lb$  symbols, a superbucket consists of  $lsb$  buckets. Additionally, the *FMI-2* maintains two tables: Table  $T_{sb}$  stores, for each superbucket and for each symbol  $c$ , the number of occurrences of  $c$  before that superbucket in  $\text{Bwt}(T)$ ; table  $T_b$  stores, for each bucket and for each symbol  $c$ , the number of occurrences of  $c$  before that bucket and up to the beginning of its superbucket. In other words,  $T_{sb}$  stores the value of the ranking function up to the beginning of superbuckets; whereas  $T_b$  stores the ranking function up to the beginning of buckets and *relative* to their enclosing superbuckets. Finally, every bucket is individually compressed using the sequence of zero-order compressors: MTF, RLE, Huffman (as in `bzip2`). This compression strategy does not guarantee that the space of *FMI-2* is bounded by the  $k$ th order entropy of  $T$ . Nevertheless, the practical performance is close to the one achievable by the best known compressors, and can be traded by tuning parameters  $lb$  and  $lsb$ .

The main difference between the original FM-index and the novel *FMI-2* lies in the strategy adopted to select the rows/positions of  $T$  which are explicitly stored. The *FMI-2* marks logically and uniformly the text  $T$  by adding a special symbol every  $s_{SA}$  symbols of the original text. This way, all of the  $\mathcal{M}_T$ 's rows that start with that special symbol are contiguous, and thus their positions can be stored and accessed easily.

The count algorithm is essentially a backward search (Algorithm 6.1), modified to take into account the presence of special symbols added to the indexed text. To search for a pattern  $P[1 : p]$ , the *FMI-2* actually searches for  $\min\{p - 1, s_{SA}\}$  patterns obtained by inserting the special symbols in  $P$  at each  $s_{SA}$ -th position, and searches for the pattern  $P$  itself. This search is implemented *in parallel* over all patterns above by exploiting the fact that, at any step  $i$ , we have to search either for  $P[p - i]$  or for the special symbol. As a result, the overall search cost is quadratic in the pattern length, and the output is now a set of at most  $p$  ranges of rows.

Therefore, the *FMI-2* is slower in counting than the original FM-index, but locating is faster, and this is crucial because this latter operation is usually the bottleneck of compressed indexes. Indeed the locate algorithm proceeds for at most  $s_{SA}$  phases. Let  $S_0$  be the range of rows to be located, eventually identified via a count operation. At a generic phase  $k$ ,  $S_k$  contains the rows that may be reached in  $k$  backward steps from the rows in  $S_0$ .  $S_k$  consists of a *set* of ranges of rows, rather than a single range. To maintain the invariant, the

algorithm picks up a range of  $S_k$ , say  $[a, b]$ , and determines the  $z \leq \sigma$  distinct symbols that occur in the substring  $\mathbf{Bwt}(T)[a : b]$  via two bucket scans and some accesses to tables  $T_{sb}$  and  $T_b$ . Then it executes  $z$  backward steps, one per such symbols, thus determining  $z$  new ranges of rows (to be inserted in  $S_{k+1}$ ) which are at distance  $k + 1$  from the rows in  $S_0$ . The algorithm cycles over all ranges of  $S_k$  to form the new set  $S_{k+1}$ . Notice that if the rows of a range start with the special symbol, their positions in the indexed text are explicitly stored, and can be accessed in constant time. Then, the position of the corresponding rows in  $S_0$  can be inferred by summing  $k$  to those values. Notice that this range can be dropped from  $S_k$ . After no more than  $s_{SA}$  phases the set  $S_k$  will be empty.

### The Alphabet-Friendly FM-index

The *Alphabet-Friendly FM-index* (AF-index) [53] resorts to the definition of  $k$ -th order entropy (see Section 2.4.1) by encoding each substring  $w_T$  up to its zero-order entropy. Since all the  $w_T$  are contiguous in  $\mathbf{Bwt}(T)$  (regardless of which  $k$  value we are considering), it suffices to split  $\mathbf{Bwt}(T)$  into blocks given by the  $k$ -th order contexts, for any desired  $k$ , and to use a Huffman-shaped wavelet tree (see Section 6.1.2) to represent each such block. In addition, we need all  $\text{RANK}_c$  values precomputed for every block beginning, as the local wavelet trees can only answer  $\text{RANK}_c$  within their blocks. In total, this achieves  $nH_k(T) + o(n \log \sigma)$  bits, for moderate and fixed  $k \leq \alpha \log_\sigma n$  and  $0 < \alpha < 1$ . Actually the AF-index does better, by splitting  $\mathbf{Bwt}(T)$  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* [43].

The compression booster finds the optimal partitioning of  $\mathbf{Bwt}(T)$  into  $t$  non-empty blocks,  $s_1, \dots, s_t$ , 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  $k$ .

As explained, 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  $\text{RANK}_c$  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  $\mathbf{Bwt}(T)$ . 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,  $j = \text{RANK}_1(R, i)$ , and the starting position of that block,  $i' = \text{SELECT}_1(R, j)$ . (This tells the position of the  $j$ -th 1 in  $R$ . As it is a sort of inverse of  $\text{RANK}$ , it is computed by binary search over  $\text{RANK}$  values.) Hence  $\text{Bwt}(T)[i] = s_j[i'']$ , 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 Fig. 6.1. In this case, we have  $\text{RANK}_c(\text{Bwt}(T), i) = C_j[c] + \text{RANK}_c(s_j, i'')$ , where the latter is computed using the wavelet tree  $wt_j$  of  $s_j$ .
- For locating, we use the algorithm of Fig. 6.2. In this case, we have  $c = \text{Bwt}(T)[i] = s_j[i'']$ . 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 6.2.1.

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/select operations into this representation.

### 6.3 The Pizza&Chili Site

The *Pizza&Chili* site has two mirrors: one in Chile and one in Italy<sup>3</sup>. Its ultimate goal is to push towards the technology transfer of this fascinating algorithmic technology lying at the crossing point of data compression and data structure design. 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, in our intention, allow any programmer to plug easily 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:

<sup>3</sup><http://pizzachili.dcc.uchile.cl> and <http://pizzachili.di.unipi.it>.

- A set of compressed indexes which are able to support the search functionalities of classical full-text indexes (e.g., substring searches), but requiring 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, or not, 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 applicative 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 developing the compressed-indexing technology. The API we propose 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.

### 6.3.1 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 [124] which constructs the Suffix Array data structure using little extra space and is fast in practice.

- The Suffix Array [119] is a plain implementation of the classical index (see Section 2.3.2), using either  $n \log n$  bits of space or simply  $n$  computer

integers, depending on the version. This was implemented by Rodrigo González.

- The SSA [53, 114] uses a Huffman-based wavelet tree over the string  $\text{Bwt}(T)$  (Section 6.2.1). It achieves zero-order entropy in space with little extra overhead and striking simplicity. It was implemented by Veli Mäkinen and Rodrigo González.
- The AF-index [53] combines compression boosting [43] with the above wavelet tree data structure (Section 6.2.5). It achieves high-order compression, at the cost of being more complex than SSA. It was implemented by Rodrigo González.
- The RLFM [114] is an improvement over the SSA (Section 6.2.1), 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 by Veli Mäkinen and Rodrigo González.
- The FMI-2 (Section 6.2.5) is an engineered implementation of the original FM-index [51], 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 [144, 145] is the variant using backward search (Section 6.2.3). It achieves high-order compression and is robust for large alphabets. It was implemented by Kunihiko Sadakane and adapted by Rodrigo González to adhere the API of the *Pizza&Chili* site. To construct the suffix array, it uses the *qsufsort* by Jesper Larsson and Kunihiko Sadakane [111].
- The LZ-index [8, 131] is a compressed index based on LZ78 compression (Section 6.2.4), 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 “\$”.

### 6.3.2 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 applica-

tion 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 in order to avoid unreadable tables of many results. Third, we have chosen the size of the texts to be large enough in order 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: 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` available at the Gutenberg Project site. 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 [113]. 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`, obtained from the Swissprot database (`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` (`ftp.kernel.org`) and `gcc-4.0.0` (`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 `dblp.uni-trier.de`.

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 6.1:** General statistics for our indexed texts.

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 6.2:** 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	$\log \sigma$	$H_0$	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	2222
english	7.814	4.525	3.620	225	2.948	10829	2.422	102666	2.063	589230
pitches	7.055	5.633	4.734	133	4.139	10946	3.457	345078	2.334	3845792
proteins	4.644	4.201	4.178	25	4.156	607	4.066	11607	3.826	224132
sources	7.845	5.465	4.077	230	3.102	9525	2.337	253831	1.852	1719387
xml	6.585	5.257	3.480	96	2.170	7049	1.434	141736	1.045	907678

Table 6.1 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 6.2 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 6.3 shows the performance of three well-known compressors (sources available in the site): `gzip` (Lempel-Ziv-based compressor), `bzip2` (BWT-based compressor), and `ppmd` ( $k$ -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. Typical values of  $k$  for `ppmd` are around 5 or 6. It is interesting to note in Table 6.3 that the compression ratios achievable by the tested compressors may be superior to  $H_4$ , because they use (explicitly or implicitly) longer contexts.

**Table 6.3:** Real compressibility of our indexed texts, as achieved by the best-known compressors: `gzip` (option `-9`), `bzip2` (option `-9`), and `ppmd` (option `-l 9`).

Text	$H_4$	gzip	bzip2	ppmd
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

## 6.4 Experimental Results

In this section we report experimental results from a subset of the compressed indexes available at the *Pizza&Chili* site. We restricted our experiments to a few indexes: Succinct Suffix Array (version `SSA_v2` in *Pizza&Chili*), Alphabet-Friendly FM-index (version `AF-index_v2` in *Pizza&Chili*), Compressed Suffix Array (`CSA` in *Pizza&Chili*), and LZ-index (version `LZ-index4` in *Pizza&Chili*), because they are the best representatives of the three classes of compressed indexes we discussed in Section 6.2. This small number will provide us with a succinct, yet significant, picture of the performance of all known compressed indexes [134].

There is no need to say that further algorithmic engineering of the indexes experimented in this chapter, as well of the other indexes available in the *Pizza&Chili* site, could possibly change the charts and tables 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 following list of experimental results has a twofold goal: on one hand, to quantify the space and time performance of compressed indexes over real datasets, and on the other hand, to motivate further algorithmic research by highlighting the limitations of the present indexes and their implementations.

Table 6.4 shows the parameters used to construct the indexes in our experiments. The SSA and AF-index have a sampling rate parameter  $s_{SA}$  that trades locating and extracting time for space. More precisely, they need  $O(s_{SA})$  accesses to the wavelet tree for locating, and  $O(s_{SA} + r - l + 1)$  accesses to extract  $T_{l,r}$ , in exchange for  $\frac{n \log n}{s_{SA}}$  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,  $s_{\Psi}$ , governs the access

**Table 6.4:** Parameters used for the different indexes in our experiments. The cases of multiple values correspond to space/time tradeoff curves.

Index	count	locate / extract
AF-index	—	$s_{SA} = \{4, 16, 32, 64, 128, 256\}$
CSA	$s_{\Psi} = \{128\}$	$s_{SA} = \{4, 16, 32, 64, 128, 256\}; s_{\Psi} = \{128\}$
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	—	$s_{SA} = \{4, 16, 32, 64, 128, 256\}$

time to  $\Psi$ , which is  $O(s_{\Psi})$  in exchange for  $\frac{n \log n}{s_{\Psi}}$  bits of space required by the samples. The second,  $s_{SA}$ , affects locating and extracting time just as above. For pure counting we can remove the sampling related to  $s_{SA}$ , 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  $\epsilon$  which trades counting/locating time for 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).

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.

### 6.4.1 Construction

Table 6.5 shows construction time and peak of memory usage during construction for one collection, namely *english*, as all the others give roughly similar results. In order to fairly evaluate the time and space consumption of the algorithm needed to construct the suffix array underlying the CSA implementation, we replaced the construction algorithm proposed by [111] and used in the original implementation by Sadakane (see Section 6.3.1), with the faster algorithm proposed by [124] and used by all other compressed SA-based indexes.<sup>4</sup> The bulk of the time of SSA and CSA is that of suffix array construction (prior to its compression). The AF-index takes much more time because it needs to run the compression boosting algorithm over the suffix array. The

<sup>4</sup>We note that this change was done just for timing measurements, the code available on the Pizza&Chili site still uses Larsson-Sadakane's algorithm because this was the choice of the CSA's implementor.

**Table 6.5:** 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,  $s_{SA} = 64$  for AF-index and SSA;  $s_{SA} = 64$  and  $s_{\Psi} = 128$  for CSA; and  $\epsilon = \frac{1}{4}$  for the LZ-index).

Index	Build Time (sec)	Main Memory Usage (MB)
AF-index	772	1,751
CSA	233	1,801
LZ-index	198	1,037
SSA	217	1,251

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<sup>5</sup>. This is a problem of compressed indexes, which is attracting a lot of practical and theoretical research [6, 89, 110, 117].

Note we have given construction time and space for just one parameter setting per index. The reason is that time and space for construction is mostly insensitive to these parameters. They imply sparser or denser sampling of suffix arrays and permutations, but those sampling times are negligible compared to suffix array and trie construction times, and sampling does not affect peak memory consumption either.

## 6.4.2 Counting

We searched for 50,000 patterns of length  $p = 20$ , randomly chosen from the indexed texts. The average counting time was then divided by  $p$  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  $p$ , and not competitive at all for this task.

Table 6.6 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

<sup>5</sup>In particular, this limited us to indexing up to 200 MB of text in our machine.



**Table 6.6:** 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.

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

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 predictably loses in counting time on smaller alphabets (dna, proteins), due to its  $O(p \log n)$  rather than  $O(p \log \sigma)$  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 6.6) is 2–6 times faster than any compressed index, but its space occupancy can be up to 18 times larger.

### 6.4.3 Locate

We locate sufficient random patterns of length 5 to obtain a total of 2–3 million occurrences per text (see Table 6.7). This way we are able to evaluate the average cost of a single locate operation, by making the impact of the counting cost negligible. Figures 6.4 and 6.5 report 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 effi-

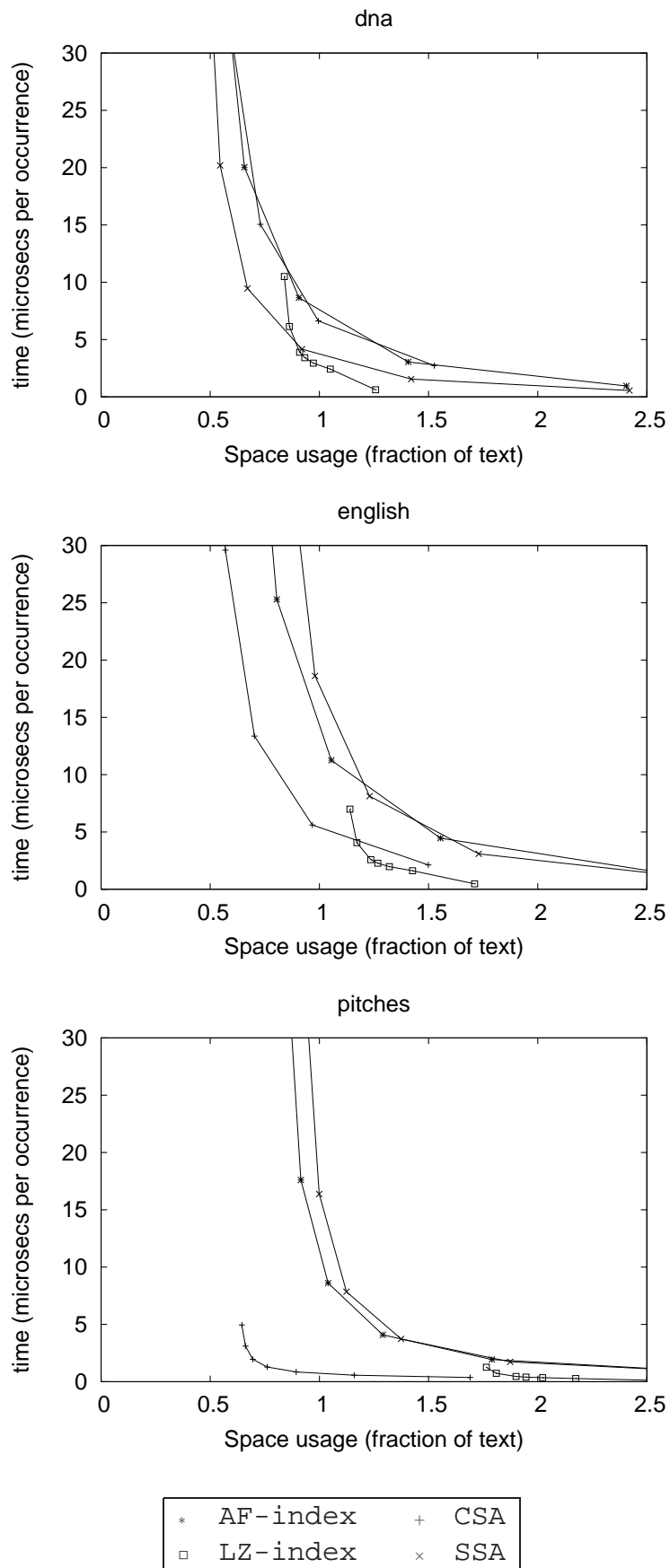


Figure 6.4: Space-time tradeoffs for locating occurrences of patterns of length 5.

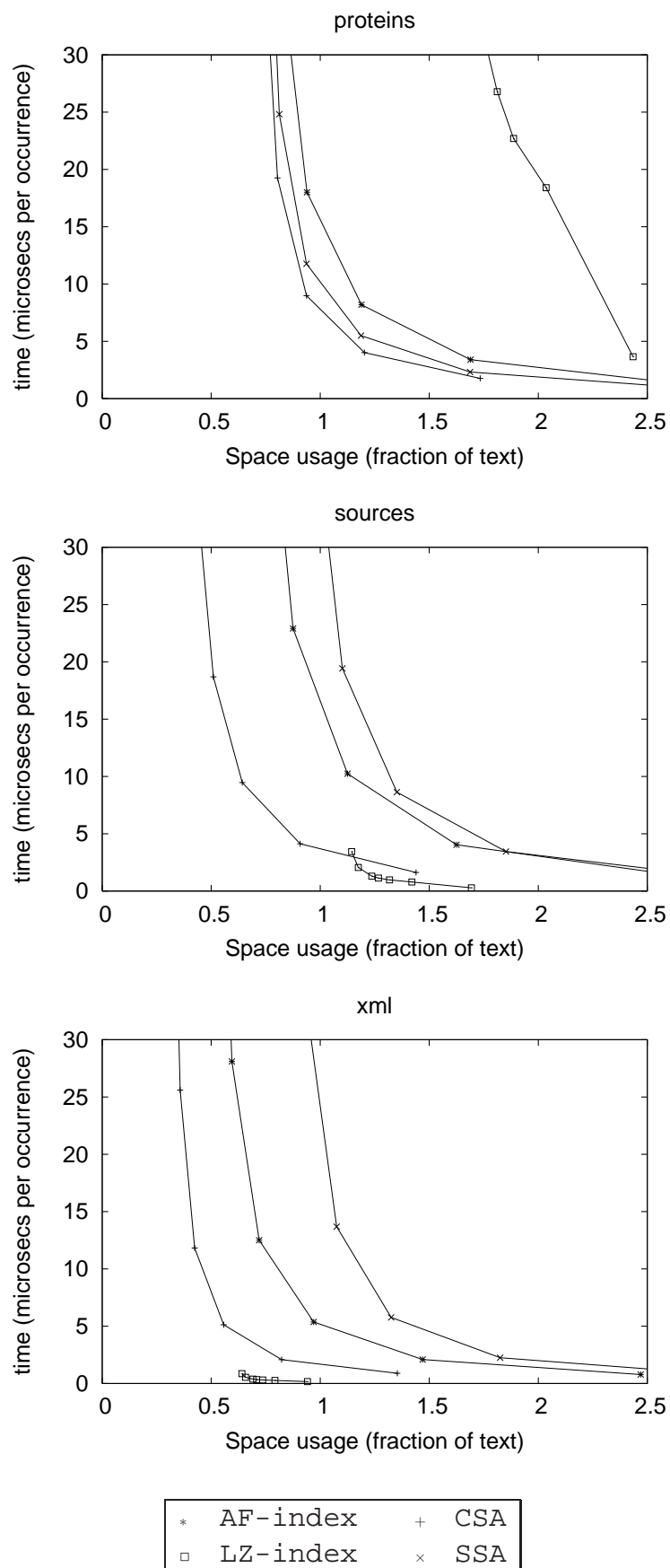


Figure 6.5: Space-time tradeoffs for locating occurrences of patterns of length 5.

**Table 6.7:** Number of searched patterns of length 5 and total number of located occurrences.

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

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

cient extraction should always be included.<sup>6</sup>

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 Section 6.2.2), and on `proteins` for which the suffix-array-based indexes perform similarly (and the LZ-index does much worse). The CSA is also the most attractive alternative if we fix that the space of the index should be equal to that of the text (recall that it includes the text), `dna` and `xml` being the exceptions, 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 6.7), as the counting is expensive on the LZ-index.

Table 6.8 shows the locate 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

<sup>6</sup>Of course, we could have a sparser sampling for extraction, but we did not want to complicate the evaluation more than necessary.

indexes are currently very efficient in case of selective queries, but traditional indexes become more effective when locating many occurrences. This fact has triggered recent research activity on this subject but a deeper understanding on index performance on hierarchical memories is still needed.

#### 6.4.4 Extract

We extracted substrings of length 512 from random text positions, for a total of 5 MB of extracted text. Figures 6.6 and 6.7 report 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 `itches` 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 some additional space.<sup>7</sup>

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 rather good: 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.

#### 6.4.5 Final comparison

In Table 6.9 we summarize our experimental results by showing the most promising compressed index(es) depending on the text type and task.

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  $O(p(H_0(T) + 1))$ . The SSA has the advantage of a simpler search mechanism, but the AF-index is superior

---

<sup>7</sup>Actually the LZ-index is not plotted for `itches` and `proteins` because it needs more than 1.5 times the text size.

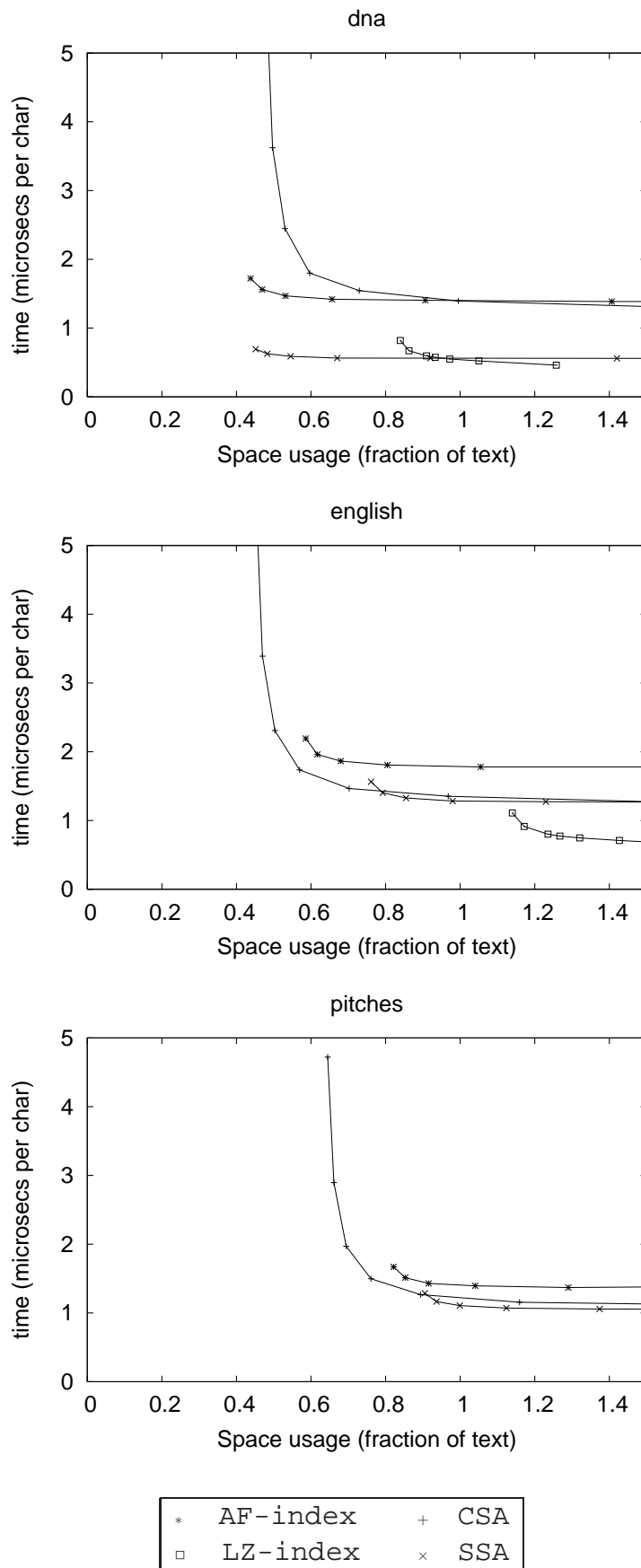


Figure 6.6: Space-time tradeoffs for extracting text symbols.

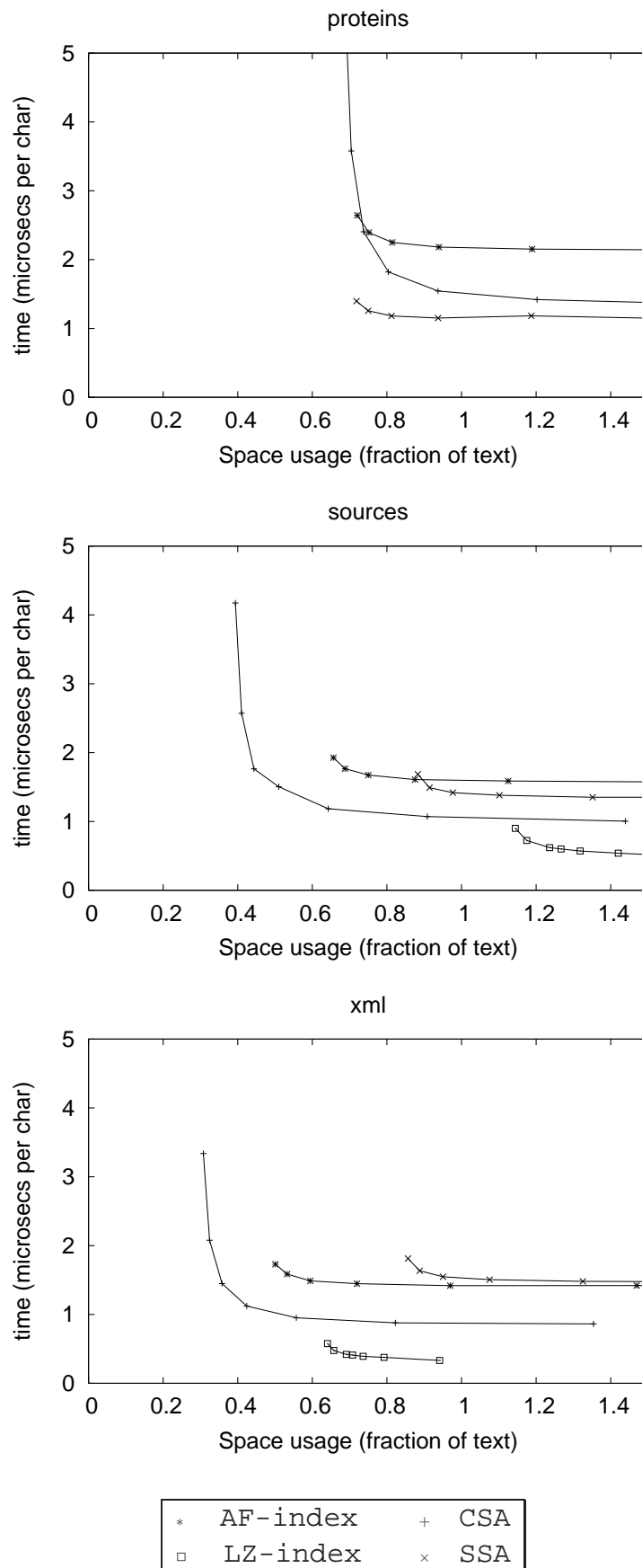


Figure 6.7: Space-time tradeoffs for extracting text symbols.

**Table 6.9:** The most promising indexes given the size and time they obtain for each operation/text.

	dna	english	pitches	proteins	sources	xml
COUNT	SSA -	SSA AF-index	AF-index SSA	SSA -	CSA AF-index	AF-index -
LOCATE	LZ-index SSA	CSA LZ-index	CSA -	SSA -	CSA LZ-index	CSA LZ-index
EXTRACT	SSA -	CSA LZ-index	CSA -	SSA -	CSA LZ-index	CSA LZ-index

for texts with small high-order entropy (i.e. `xml`, `sources`, `english`). The CSA usually loses because of its  $O(p \log n)$  counting complexity.

For locating and extracting, which are LF-computation intensive, `AF-index` is hardly better than 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 is better than 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, and that this is not to be taken as a static, final result, because the area is developing fast. Nonetheless, our current 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 show that the compressed full-text indexes 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. Nonetheless compressed indexes achieve a (search/extract) throughput which is significant and may match the efficiency specifications of most software tools which run on a commodity PC.





---

CHAPTER  
SEVEN

---

## Dictionary indexes

String processing and searching tasks are at the core of modern Web search, information retrieval and data mining applications. Most of such tasks boil down to some basic algorithmic primitives which involve a large dictionary of strings having variable length. Typical examples include: pattern matching (exact, approximate, with wild-cards,...), the ranking of a string in a sorted dictionary, or the selection of the  $i$ -th string from it. While it is easy to imagine uses of pattern matching primitives in real applications, such as search engines and text mining tools, rank/select operations appear uncommon. However they are quite often used (probably, unconsciously!) by programmers to replace long strings with unique IDs which are easier and faster to be processed and compressed. In this context ranking a string means mapping it to its unique ID, whereas selecting the  $i$ -th string means retrieving it from its ID (i.e. its ranked position  $i$ ).

As strings are getting longer and longer, and dictionaries of strings are getting larger and larger, it becomes crucial to devise implementations for the above primitives which are fast and work in compressed space. This is the topic of the present chapter that actually addresses the design of compressed data structures for the so called *tolerant retrieval* problem, defined as follows [121]. Let  $\mathcal{S}$  be a sorted dictionary of  $m$  strings having total length  $n$  and drawn from an arbitrary alphabet  $\Sigma$  of size  $\sigma$ . The *tolerant retrieval* problem consists of preprocessing  $\mathcal{S}$  in order to efficiently support the following WILDCARD( $P$ ) query operation: *search for the strings in  $\mathcal{S}$  which match the pattern  $P \in (\Sigma \cup \{*\})^+$* . Symbol  $*$  is the so called *wild-card* symbol, and matches any substring of  $\Sigma^*$ . In principle, the pattern  $P$  might contain several occurrences of  $*$ ; however, for

practical reasons, it is common to restrict the attention to the following significant cases:

- MEMBERSHIP query determines whether a pattern  $P \in \Sigma^+$  occurs in  $\mathcal{S}$ . Here  $P$  does not include wild-cards.
- PREFIX query determines all strings in  $\mathcal{S}$  which are prefixed by string  $\alpha$ . Here  $P = \alpha*$  with  $\alpha \in \Sigma^+$ .
- SUFFIX query determines all strings in  $\mathcal{S}$  which are suffixed by string  $\beta$ . Here  $P = *\beta$  with  $\beta \in \Sigma^+$ .
- SUBSTRING query determines all strings in  $\mathcal{S}$  which have  $\gamma$  as a substring. Here  $P = *\gamma*$  with  $\gamma \in \Sigma^+$ .
- PREFIXSUFFIX query is the most sophisticated one and asks for all strings in  $\mathcal{S}$  that are prefixed by  $\alpha$  and suffixed by  $\beta$ . Here  $P = \alpha * \beta$  with  $\alpha, \beta \in \Sigma^+$ .

We extend the tolerant retrieval problem to include the following two basic primitives:

- RANKSTRING( $P$ ) computes the rank of string  $P \in \Sigma^+$  within the (sorted) dictionary  $\mathcal{S}$ .
- SELECTSTRING( $i$ ) retrieves the  $i$ -th string of the (sorted) dictionary  $\mathcal{S}$ .

There are two classical approaches to string searching: Hashing and Tries [10]. Hashing supports only the exact MEMBERSHIP query; its more sophisticated variant called *minimal ordered perfect* hashing [164] supports also the RANKSTRING operation but only on strings of  $\mathcal{S}$ . All other queries need however the inefficient scan of the whole dictionary!

Tries are more powerful in searching than hashing, but they introduce extra space and fail to provide an efficient solution to the PREFIXSUFFIX query. In fact, the search for  $P = \alpha * \beta$  needs to visit the subtree descending from the trie-path labeled  $\alpha$ , in order to find the strings that are suffixed by  $\beta$ . Such a brute-force visit may cost  $\Theta(|\mathcal{S}|)$  time independently of the number of query answers (cfr [9]). We can circumvent this limitation by using the sophisticated approach proposed in [47] which builds two tries, one storing the strings of  $\mathcal{S}$  and the other storing their reversals, and then *reduce* the PREFIXSUFFIX query to a geometric 2D-range query, which is eventually solved via a proper efficient geometric data structure in  $O(|\alpha| + |\beta| + \text{polylog}(n))$  time. The overall space

occupancy would be  $\Theta(n \log n)$  bits, with a large constant hidden in the big-O notation due to the presence of the two tries and the geometric data structure.

Recently Manning et al. [121] resorted the *Permuterm index* of Garfield [68] as a time-efficient and elegant solution to the tolerant retrieval problem above. The idea is to take every string  $s \in \mathcal{S}$ , append a special symbol  $\$$ , and then consider all the cyclic rotations of  $s\$$ . The dictionary of all *rotated* strings is called the *permuterm dictionary*, and is then indexed via any data structure that supports prefix searches, e.g. the trie. The key to solve the PREFIXSUFFIX query is to rotate the query string  $\alpha * \beta\$$  so that the wild-card symbol appears at the end, namely  $\beta\$ \alpha *$ . Finally, it suffices to perform a prefix-query for  $\beta\$ \alpha$  over the permuterm dictionary. As a result, the Permuterm index allows to *reduce any query of the Tolerant Retrieval problem on the dictionary  $\mathcal{S}$  to a prefix query over its permuterm dictionary*. The limitation of this elegant approach relies in its space occupancy, as “its dictionary becomes quite large, including as it does all rotations of each term.” [121]. In practice, one memory word per rotated string (and thus 4 bytes per symbol) is needed to index it, for a total of  $\Omega(n \log n)$  bits.

In this chapter we propose the *Compressed Permuterm Index* which solves the tolerant retrieval problem in time proportional to the length of the queried string  $P$ , and space close to the  $k$ -th order empirical entropy of the dictionary  $\mathcal{S}$ . The time complexity matches the one achieved by the (uncompressed) Permuterm index. The space complexity is close to the  $k$ -th order empirical entropy of the dictionary  $\mathcal{S}$ . In addition, we devise a *dynamic Compressed Permuterm Index* that is able to maintain the dictionary  $\mathcal{S}$  under insertions and deletions of an individual string  $s$  in  $O(|s|(1 + \log \sigma / \log \log n) \log n)$  time. All query operations are slowed down by a multiplicative factor of at most  $O((1 + \log \sigma / \log \log n) \log n)$ . The space occupancy is still close to the  $k$ -th order empirical entropy of the dictionary  $\mathcal{S}$ .

Our result is based on a variant of the Burrows-Wheeler Transform here extended to work on a dictionary of strings of variable length. We prove new properties of such BWT, and show that known (dynamic) compressed indexes may be easily adapted to solve efficiently the (dynamic) Tolerant Retrieval problem.

We finally complement our theoretical study with a significant set of experiments over large dictionaries of URLs, hosts and terms, and compare our Compressed Permuterm index against some classical approaches to the Tolerant Retrieval problem mentioned in [121, 164] such as tries and front-coded dictionaries. Experiments will show that tries are fast but much space consuming; conversely our compressed permuterm index allows to trade query time by space occupancy, resulting as fast as Front-Coding in searching the

---

**Algorithm** Backward\_search( $Q[1, q]$ )

1.  $i = q, c = Q[q], \text{First} = C[c] + 1, \text{Last} = C[c + 1]$ ;
  2. **while**  $((\text{First} \leq \text{Last}) \text{ and } (i \geq 2))$  **do**
  3.    $c = Q[i - 1]$ ;
  4.    $\text{First} = C[c] + \text{RANK}_c(L, \text{First} - 1) + 1$ ;
  5.    $\text{Last} = C[c] + \text{RANK}_c(L, \text{Last})$ ;
  6.    $i = i - 1$ ;
  7. **if**  $(\text{Last} < \text{First})$  **then return** “no rows prefixed by  $Q$ ” **else return**  $[\text{First}, \text{Last}]$ .
- 

**Figure 7.1:** The algorithm to find the range  $[\text{First}, \text{Last}]$  of rows of  $\mathcal{M}_T$  prefixed by  $Q[1, q]$ .

dictionary but more than 50% smaller in space occupancy— thus being close to `gzip`, `bzip2` and `ppmd`. This way the compressed permuterm index offers a plethora of solutions for the Tolerant Retrieval problem which may well adapt to different applicative scenarios.

## 7.1 Background

The Backward Search algorithm of the FM-index family has been already intensively discussed in previous chapter (see Section 6.2.1). Since our solution is based on it, we report here the pseudocode of this algorithm (Algorithm 7.1). Chapter 6 was primarily focused on practical solutions, thus, we presented only practical solutions to provide RANK and SELECT queries. The literature offers also many theoretical solutions for this problem having often better time/space bounds (see e.g. [11, 134] and references therein). We do not want to enter into details on this topic and, thus, we just summarize below the ones having best bounds.

**Lemma 9.** *Let  $T[1, n]$  be a string over alphabet  $\Sigma$  of size  $\sigma$  and let  $L = \text{Bwt}(T)$  be its BW-transform.*

1. *For  $\sigma = O(\text{polylog}(n))$ , there exists a data structure which supports RANK queries and the retrieval of any symbol of  $L$  in constant time, by using  $nH_k(T) + o(n)$  bits of space, for any  $k \leq \alpha \log_\sigma n$  and  $0 < \alpha < 1$  [53, Theorem 5].*
2. *For general  $\Sigma$ , there exists a data structure which supports RANK queries and*

the retrieval of any symbol of  $L$  in  $O(\log \log \sigma)$  time, by using  $nH_k(T) + n \cdot o(\log \sigma)$  bits of space, for any  $k \leq \alpha \log_\sigma n$  and  $0 < \alpha < 1$  [11, Theorem 4.2].

By plugging Lemma 9 into `Backward_search`, the authors of [11, 53] obtained:

**Theorem 12.** *Given a text  $T[1, n]$  drawn from an alphabet  $\Sigma$  of size  $\sigma$ , there exists a compressed index that takes  $q \times t_{\text{rank}}$  time to support `Backward_search(Q[1, q])`, where  $t_{\text{rank}}$  is the time cost of a single RANK operation over  $L = \text{Bwt}(T)$ . The space usage is bounded by  $nH_k(T) + l_{\text{space}}$  bits, for any  $k \leq \alpha \log_\sigma n$  and  $0 < \alpha < 1$ , where  $l_{\text{space}}$  is  $o(n)$  when  $\sigma = O(\text{polylog}(n))$  and  $n \cdot o(\log \sigma)$  otherwise. ■*

As we have already seen in previous chapter, compressed indexes support also other operations, like locate and display of pattern occurrences, which are slower than `Backward_search` in that they require  $\text{polylog}(n)$  time per occurrence. One positive feature of our compressed permuterm index is that it will not need these (sophisticated) data structures, and thus it will not incur in this  $\text{polylog}$ -slowdown.

## 7.2 Compressed Permuterm Index

The way in which the Permuterm dictionary is computed, immediately suggests that there *should be* a relation between the BWT and the Permuterm dictionary of the string set  $\mathcal{S}$ . In both cases we talk about *cyclic rotations* of strings, but in the former we refer to just one string, whereas in the latter we refer to a dictionary of strings of possibly different lengths. The notion of BWT for a set of strings has been considered in [122] for the purpose of string compression and comparison. Here, we are interested in the compressed *indexing* of the string dictionary  $\mathcal{S}$ , which introduces more challenges. Surprisingly enough the solution we propose is novel, simple, and efficient in time and space; furthermore, it admits an effective dynamization.

### 7.2.1 A simple, but inefficient solution

Let  $\mathcal{S} = \{s_1, s_2, \dots, s_m\}$  be the lexicographically sorted dictionary of strings to be indexed. Let  $\$$  (resp.  $\#$ ) be a symbol smaller (resp. larger) than any other symbol of  $\Sigma$ . We consider the *doubled* strings  $\hat{s}_i = s_i\$s_i$ . It is easy to note that any pattern searched by `PREFIXSUFFIX(P)` matches  $s_i$  if, and only if, the rotation of  $P$  mentioned in the Introduction is a substring of  $\hat{s}_i$ . For example, the query `PREFIXSUFFIX( $\alpha * \beta$ )` matches  $s_i$  iff the rotated string  $\beta\$ \alpha$  occurs as a substring of  $\hat{s}_i$ .

Consequently, the simplest approach to solve the Tolerant Retrieval problem with compressed indexes seems to boil down to the indexing of the string  $\widehat{\mathcal{S}}_{\mathcal{D}} = \#\widehat{s}_1\#\widehat{s}_2\cdots\#\widehat{s}_m\#$  by means of the data structure of Theorem 12. Unfortunately, this approach suffers of subtle inefficiencies in the indexing and searching steps. To see them, let us “compare” string  $\widehat{\mathcal{S}}_{\mathcal{D}}$  against string  $\mathcal{S}_{\mathcal{D}} = \$s_1\$s_2\$ \dots \$s_{m-1}\$s_m\#\#$ , which is a *serialization* of the dictionary  $\mathcal{S}$  (and it will be at the core of our approach, see below). We note that the “duplication” of  $s_i$  within  $\widehat{s}_i$ : (1) doubles the string to be indexed, because  $|\widehat{\mathcal{S}}_{\mathcal{D}}| = 2|\mathcal{S}_{\mathcal{D}}| - 1$ ; and (2) doubles the space bound of compressed indexes evaluated in Theorem 12, because  $|\widehat{\mathcal{S}}_{\mathcal{D}}|H_k(\widehat{\mathcal{S}}_{\mathcal{D}}) \cong 2|\mathcal{S}_{\mathcal{D}}|H_k(\mathcal{S}_{\mathcal{D}}) \pm m(k \log \sigma + 2)$ , where the second term comes from the presence of symbol # which introduces new  $k$ -long substrings in the computation of  $H_k(\widehat{\mathcal{S}}_{\mathcal{D}})$ . Point (1) is a limitation for building large *static* compressed indexes in practice, being their construction space a primary concern [139]; point (2) will be experimentally investigated in Section 7.4 where we show that a compressed index built on  $\widehat{\mathcal{S}}_{\mathcal{D}}$  may be up to 1.9 times larger than a compressed index built on  $\mathcal{S}_{\mathcal{D}}$ .

## 7.2.2 A simple and efficient solution

Unlike the previous solution, our Compressed Permuterm index works on the plain string  $\mathcal{S}_{\mathcal{D}}$ , and is built in three steps (see Figure 7.2):

1. Build the string  $\mathcal{S}_{\mathcal{D}} = \$s_1\$s_2\$ \dots \$s_{m-1}\$s_m\#\#$ . Recall that the dictionary strings are lexicographically ordered, and that symbol \$ (resp. #) is assumed to be smaller (resp. larger) than any other symbol of  $\Sigma$ .
2. Compute  $L = \text{Bwt}(\mathcal{S}_{\mathcal{D}})$ .
3. Build a compressed data structure to support RANK queries over the string  $L$  (Lemma 9).

Our goal is to turn every wild-card search over the dictionary  $\mathcal{S}$  into a substring search over the string  $\mathcal{S}_{\mathcal{D}}$ . Some of the required queries are immediately implementable as *substring searches* over  $\mathcal{S}_{\mathcal{D}}$  (and thus they can be supported by procedure `Backward_search` and the RANKSTRING data structure built on  $L$ ). But the sophisticated PREFIXSUFFIX query needs a different approach because it requires to *simultaneously match* a prefix and a suffix of a dictionary string, which are possibly far apart from each other in  $\mathcal{S}_{\mathcal{D}}$ . In order to circumvent this limitation, we prove a novel property of  $\text{Bwt}(\mathcal{S}_{\mathcal{D}})$  and

F	L	jump2end
\$ hat\$hip\$hope\$hot\$ #		↓
\$ hip\$hope\$hot\$#\$ha t		↓
\$ hope\$hot\$#\$hat\$hi p		↓
\$ hot\$#\$hat\$hip\$hop e		↓
\$ #\$hat\$hip\$hope\$ho t		
a t\$hip\$hope\$hot\$#\$ h		
e \$hot\$#\$hat\$hip\$ho p		
h at\$hip\$hope\$hot\$# \$		
h ip\$hope\$hot\$#\$hat \$		
h ope\$hot\$#\$hat\$hip \$		
h ot\$#\$hat\$hip\$hope \$		
i p\$hope\$hot\$#\$hat\$ h		
o pe\$hot\$#\$hat\$hip\$ h		
o t\$#\$hat\$hip\$hope\$ h		
p \$hope\$hot\$#\$hat\$h i		
p e\$hot\$#\$hat\$hip\$h o		
t \$hip\$hope\$hot\$#\$h a		
t #\$hat\$hip\$hope\$h o		
# \$hat\$hip\$hope\$hot \$		

**Figure 7.2:** Given the dictionary  $\mathcal{S} = \{\text{hat, hip, hope, hot}\}$ , we build the string  $\mathcal{S}_{\mathcal{D}} = \$\text{hat}\$\text{hip}\$\text{hope}\$\text{hot}\$ \#$ , and then compute its BW-transform. Arrows denote the positions incremented by the function `jump2end`.

deploy it to design a function, called `jump2end`, that allows to modify the procedure `Backward_search` of Figure 7.1 in a way that is suitable to support efficiently the PREFIXSUFFIX query. The main idea is that when `Backward_search` reaches the beginning of some dictionary string, say  $s_i$ , then it “jumps” to the last symbol of  $s_i$  rather than continuing onto the last symbol of its previous string in  $\mathcal{S}$ , i.e. the last symbol of  $s_{i-1}$ . Surprisingly enough, function `jump2end(i)` consists of one line of code:

```
if 1 ≤ i ≤ m then return(i+1) else return(i)
```

and its correctness derives from the following two Lemmas. (Refer to Figure 7.2 for an illustrative example.)

**Lemma 10.** *Given the sorted dictionary  $\mathcal{S}$ , and the way string  $\mathcal{S}_{\mathcal{D}}$  is built, matrix  $\mathcal{M}_{\mathcal{S}_{\mathcal{D}}}$  satisfies the following properties:*

- The first row of  $\mathcal{M}_{\mathcal{S}_{\mathcal{D}}}$  is prefixed by  $\$s_1\$$ , thus it ends with symbol  $L[1] = \#$ .
- For any  $2 \leq i \leq m$ , the  $i$ -th row of  $\mathcal{M}_{\mathcal{S}_{\mathcal{D}}}$  is prefixed by  $\$s_i\$$  and thus it ends with the last symbol of  $s_{i-1}$ , i.e.  $L[i] = s_{i-1}[|s_{i-1}|]$ .



- The  $(m + 1)$ -th row of  $\mathcal{M}_{\mathcal{S}_{\mathcal{D}}}$  is prefixed by  $\#s_1$ , and thus it ends with the last symbol of  $s_m$ , i.e.  $L[m + 1] = s_m[|s_m|]$ .

**Proof.** The three properties come from the sorted ordering of the dictionary strings in  $\mathcal{S}_{\mathcal{D}}$ , from the fact that symbol  $\$$  (resp.  $\#$ ) is the smallest (resp. largest) alphabet symbol, from the cyclic rotation of the rows in  $\mathcal{M}_{\mathcal{S}_{\mathcal{D}}}$ , and from their lexicographic ordering. ■

The previous Lemma immediately implies the “locality” property deployed by function `jump2end(i)`:

**Lemma 11.** Any row  $i \in [1, m]$  is prefixed by  $s_i$  and the next row  $(i + 1)$  ends with the last symbol of  $s_i$ .

We are now ready to design the procedures for pattern searching and for displaying the strings of  $\mathcal{S}$ . As we anticipated above the main search procedure, called `BackPerm_search`, is derived from the original `Backward_search` of Figure 7.1 by adding one step which makes proper use of `jump2end`:

3': `First = jump2end(First); Last = jump2end(Last);`

It is remarkable that the change is minimal (just one line of code!) and takes constant time, because `jump2end` takes  $O(1)$  time. Let us now comment on the correctness of the new procedure `BackPerm_search( $\beta\$\alpha$ )` in solving the sophisticated query `PREFIXSUFFIX( $\alpha * \beta$ )`. We note that `BackPerm_search` proceeds as the standard `Backward_search` for all symbols  $Q[i] \neq \$$ . In fact, the rows involved in these search steps do not belong to the range  $[1, m]$ , and thus `jump2end` is ineffective. When  $Q[i] = \$$ , the range  $[\text{First}, \text{Last}]$  is formed by rows which are prefixed by  $\$\alpha$ . By Lemma 11 we know that these rows are actually prefixed by strings  $s_j$ , with  $j \in [\text{First}, \text{Last}]$ , and thus these strings are in turn prefixed by  $\$\alpha$ . Given that  $[\text{First}, \text{Last}] \subset [1, m]$ , Step 3' moves this range of rows to  $[\text{First} + 1, \text{Last} + 1]$ , and thus identifies the new block of rows which are ended by the last symbols of those strings  $s_j$  (Lemma 11). After that, `BackPerm_search` continues by scanning backward the symbols of  $\beta$  (no other  $\$$  symbol is involved), thus eventually finding the rows prefixed by  $\beta\$\alpha$ .

Figure 7.3 shows the pseudo-code of two other basic procedures: `Back_step(i)` and `Display_string(i)`. The former procedure is a slight variation of the *backward step* implemented by any current compressed index based on BWT (see Chapter 6), here modified to support a leftward *cyclic* scan of every dictionary string. Precisely, if  $F[i]$  is the  $j$ -th symbol of some dictionary string  $s$ , then `Back_step(i)` returns the row prefixed by the  $(j - 1)$ -th symbol of that string if  $j > 1$  (this is a standard backward step), otherwise it returns the row prefixed by the last

---

**Algorithm** Back\_step( $i$ )

1. Compute  $L[i]$ ;
2. **return**  $C[L[i]] + \text{RANK}_{L[i]}(L, i)$ ;

**Algorithm** Display\_string( $i$ )

1. // Go back to the preceding \$, let it be at row  $k_i$   
**while** ( $F[i] \neq \$$ ) **do**  $i = \text{Back\_step}(i)$ ;
2.  $i = \text{jump2end}(i)$ ;
3.  $s = \text{empty string}$ ;
4. // Construct  $s = s_{k_i}$   
**while** ( $L[i] \neq \$$ ) {  $s = L[i] \cdot s$ ;  $i = \text{Back\_step}(i)$ ; };
5. **return**( $s$ );

---

**Figure 7.3:** Algorithm Back\_step is the one devised in [52] for standard compressed indexes. Algorithm Display\_string( $i$ ) retrieves the string containing the symbol  $F[i]$ .

symbol of  $s$  (by means of `jump2end`). Procedure `Display_string( $i$ )` builds upon `Back_step( $i$ )` and retrieves the string  $s$ , namely the dictionary string that contains the symbol  $F[i]$ .

Using the data structures of Lemma 9 for supporting RANK queries over the string  $L = \text{Bwt}(\mathcal{S}_{\mathcal{D}})$ , we obtain:

**Theorem 13.** *Let  $\mathcal{S}_{\mathcal{D}}$  be the string built upon a dictionary  $\mathcal{S}$  of  $m$  strings having total length  $n$  and drawn from an alphabet  $\Sigma$  of size  $\sigma$ , such that  $\sigma = \text{polylog}(n)$ . We can design a Compressed Permuterm index such that:*

- Procedure `Back_step( $i$ )` takes  $O(1)$  time.
- Procedure `BackPerm_search( $Q[1, q]$ )` takes  $O(q)$  time.
- Procedure `Display_string( $i$ )` takes  $O(|s|)$  time, if  $s$  is the string containing symbol  $F[i]$ .

Space occupancy is bounded by  $nH_k(\mathcal{S}_{\mathcal{D}}) + o(n)$  bits, for any  $k \leq \alpha \log_{\sigma} n$  and  $0 < \alpha < 1$ .

**Proof.** For the time complexity, we observe that function `jump2end` takes constant time, and it is invoked  $O(1)$  times at each possible iteration of proce-

dures `BackPerm_search` and `Display_string`. Moreover, `Back_step` takes constant time, by Lemma 9. For the space complexity, we use the data structure of Lemma 9 (case 1) to support RANK queries on the string  $L = \text{Bwt}(\mathcal{S}_{\mathcal{D}})$ . ■

If  $\sigma = \Omega(\text{polylog}(n))$ , the above time bounds must be multiplied by a factor  $O(\log \log \sigma)$  and the space bound has an additive term of  $n \cdot o(\log \sigma)$  bits (Lemma 9, case 2).

We are left with detailing the implementation of WILDCARD, RANKSTRING and SELECTSTRING queries for the Tolerant Retrieval problem. As it is standard in the Compressed Indexing literature we distinguish between two sub-problems: *counting* the number of dictionary strings that match the given wildcard query  $P$ , and *retrieving* these strings. Based on the Compressed Permuterm index of Theorem 13 we have:

- MEMBERSHIP query invokes procedure `BackPerm_search($P$)`, then simply checks if `First < Last`.
- PREFIX query invokes procedure `BackPerm_search($\alpha$)` and returns the value `Last - First + 1` as the number of dictionary strings prefixed by  $\alpha$ . These strings can be retrieved by applying `Display_string(i)`, for each  $i \in [\text{First}, \text{Last}]$ .
- SUFFIX query invokes procedure `BackPerm_search($\beta$)` and returns the value `Last - First + 1` as the number of dictionary strings suffixed by  $\beta$ . These strings can be retrieved by applying `Display_string(i)`, for each  $i \in [\text{First}, \text{Last}]$ .
- SUBSTRING query invokes procedure `BackPerm_search($\gamma$)` and returns the value `Last - First + 1` as the number of occurrences of  $\gamma$  as a substring of  $\mathcal{S}$ 's strings.<sup>1</sup> Unfortunately, the efficient retrieval of these strings cannot be through the execution of `Display_string`, as we did for the queries above. A dictionary string  $s$  may now be retrieved multiple times if  $\gamma$  occurs many times as a substring of  $s$ . To circumvent this problem we design a simple time-optimal retrieval, as follows. We use a bit vector  $V$  of size `Last - First + 1`, initialized to 0. The execution of `Display_string` is modified so that  $V[j - \text{First}]$  is set to 1 when a row  $j$  within the range  $[\text{First}, \text{Last}]$  is visited during its execution. In order to retrieve once all dictionary strings that contain  $\gamma$ , we scan through  $i \in [\text{First}, \text{Last}]$  and invoke the *modified* `Display_string(i)` only if  $V[i - \text{First}] = 0$ . It is easy

<sup>1</sup>This is different from the problem of efficiently counting the number of strings containing  $\gamma$ . Our index does not solve this interesting problem (cfr. [147] and references therein).

to see that if  $i_1, i_2, \dots, i_k \in [\text{First}, \text{Last}]$  are the rows of  $\mathcal{M}_{\mathcal{S}_D}$  denoting the occurrences of  $\gamma$  in some dictionary string  $s$  (i.e.  $F[i_j]$  is a symbol of  $s$ ), only `Display_string( $i_1$ )` is fully executed, thus taking  $O(|s|)$  time. For all the other rows  $i_j$ , with  $j > 1$ , we find  $V[i_j - \text{First}] = 1$  and thus `Display_string( $i_j$ )` is not invoked.

- `PREFIXSUFFIX` query invokes `BackPerm_search( $\beta\$ \alpha$ )` and returns the value  $\text{Last} - \text{First} + 1$  as the number of dictionary strings which are prefixed by  $\alpha$  and suffixed  $\beta$ . These strings can be retrieved by applying `Display_string( $i$ )`, for each  $i \in [\text{First}, \text{Last}]$ .
- `RANKSTRING( $P$ )` invokes `BackPerm_search( $\$P\$$ )` and returns the value  $\text{First}$ , if  $\text{First} < \text{Last}$ , otherwise  $P \notin \mathcal{S}$  (see Lemma 10) and thus the lexicographic position of  $P$  in  $\mathcal{S}$  can be discovered by means of a slight variant of `Backward_search` whose details are given in Figure 7.5 (see Section 7.3.2 for further comments).
- `SELECTSTRING( $i$ )` invokes `Display_string( $i$ )` provided that  $1 \leq i \leq m$  (see Lemma 10).

**Theorem 14.** *Let  $\mathcal{S}$  be a dictionary of  $m$  strings having total length  $n$ , drawn from an alphabet  $\Sigma$  of size  $\sigma$  such that  $\sigma = \text{polylog}(n)$ . Our Compressed Permuterm index ensures that:*

- If  $P[1, p]$  is a pattern with one-single wild-card, the query `WILDCARD( $P$ )` takes  $O(p)$  time to count the number of occurrences of  $P$  in  $\mathcal{S}$ , and  $O(L_{\text{occ}})$  time to retrieve the dictionary strings matching  $P$ , where  $L_{\text{occ}}$  is their total length.
- `SUBSTRING( $\gamma$ )` takes  $O(|\gamma|)$  time to count the number of occurrences of  $\gamma$  as a substring of  $\mathcal{S}$ 's strings, and  $O(L_{\text{occ}})$  time to retrieve the dictionary strings having  $\gamma$  as a substring, where  $L_{\text{occ}}$  is their total length.
- `RANKSTRING( $P[1, p]$ )` takes  $O(p)$  time.
- `SELECTSTRING( $i$ )` takes  $O(|s_i|)$  time.

The space occupancy is bounded by  $nH_k(\mathcal{S}_D) + o(n)$  bits, for any  $k \leq \alpha \log_\sigma n$  and  $0 < \alpha < 1$ .

According to Lemma 9 (case 2), if  $\sigma = \Omega(\text{polylog}(n))$  the above time bounds must be multiplied by  $O(\log \log \sigma)$  and the space bound has an additive term of  $n \cdot o(\log \sigma)$  bits. We remark that our Compressed Permuterm index can

support all wild-card searches *without* using any locate-data structure, which is known to be the main bottleneck of current compressed indexes [134]: it implies the polylog-term in their query bounds and most of the  $o(n \log \sigma)$  term of their space cost. The net result is that our Compressed Permuterm index achieves in practice space occupancy much closer to known compressors and very fast queries, as we will experimentally show in Section 7.4.

A comment is in order at this point. Instead of introducing function `jump2end` and then modify the `Backward_search` procedure, we could have modified  $L = \text{Bwt}(\mathcal{S}_D)$  just as follows: cyclically rotate the prefix  $L[1, m + 1]$  of one single step (i.e. move  $L[1] = \#$  to position  $L[m + 1]$ ). This way, we are actually *plugging* Lemma 11 directly into the string  $L$ . It is thus possible to show that the compressed index of Theorem 12 applied on the *rotated*  $L$ , is equivalent to the compressed permuterm index introduced above. The performance in practice of this variation are slightly better since the computation of `jump2end` is no longer required. This is the implementation we used in the experiments of Section 7.4.

### 7.3 Dynamic Compressed Permuterm Index

In this section we deal with the *dynamic* Tolerant Retrieval problem in which the dictionary  $\mathcal{S}$  changes over the time under two update operations:

- `INSERTSTRING(W)` inserts the string  $W$  in  $\mathcal{S}$ .
- `DELETESTRING(j)` removes the  $j$ -th lexicographically smallest string  $s_j$  from  $\mathcal{S}$ .

The problem of maintaining a compressed index over a dynamically changing collection of strings, has been addressed in e.g. [25, 52, 116]. In those papers the design of dynamic Compressed Indexes boils down to the design of dynamic compressed data structures for supporting `Rank/Select` operations. Here we adapt those solutions to the design of our dynamic Compressed Permuterm Index by showing that the insertion/deletion of an individual string  $s$  in/from  $\mathcal{S}$  can be implemented via an *optimal* number  $O(|s|)$  of basic insert/delete operations of *single* symbols in the compressed `RANK/SELECT` data structure built on  $L = \text{Bwt}(\mathcal{S}_D)$ . Precisely, we will consider the following two basic update operations:

- `INSERT(L, i, c)` inserts symbol  $c$  between symbols  $L[i]$  and  $L[i + 1]$ .

- $\text{DELETE}(L, i)$  removes the  $i$ th symbol  $L[i]$ .

The literature provides several dynamic data structures for supporting RANK queries and the above two update operations, with various time/space trade-offs. The best known results are currently due to [80]:

**Lemma 12.** *Let  $S[1, s]$  be a string drawn from an alphabet  $\Sigma$  of size  $\sigma$  and let  $L = \text{Bwt}(S)$  be its BW-Transform. There exists a dynamic data structure that supports RANK, SELECT and ACCESS operations in  $L$  taking  $O((1 + \log \sigma / \log \log s) \log s)$  time, and maintains  $L$  under insert and delete operations of single symbols in  $O((1 + \log \sigma / \log \log s) \log s)$  time. The space required by this data structure is  $nH_k(S) + o(n \log \sigma)$  bits, for any  $k < \alpha \log_\sigma s$  and constant  $0 < \alpha < 1$ .*

Our dynamic Compressed Permuterm Index is designed upon the above dynamic data structures, in a way that any improvement to Lemma 12 will positively reflect onto an improvement to our bounds. Therefore we will indicate the time complexities of our algorithms as a function of the number of INSERT and DELETE operations executed onto the changing string  $L = \text{Bwt}(\mathcal{S}_{\mathcal{D}})$ . We also notice that these operations will change not only  $L$  but also the string  $F$  (which is the lexicographically sorted version of  $L$ , see Section 2.4.2). The maintenance of  $L$  will be discussed in the next subsections; while for  $F$  we will make use of the solution proposed in [116, Section 7] that takes  $\sigma \log s + o(\sigma \log s)$  bits and implements in  $O(\log s)$  time the following query and update operations:  $C[c]$  returns the number of symbols in  $F$  smaller than  $c$ ;  $\text{deleteF}(c)$  removes from  $F$  an occurrence of symbol  $c$ ; and  $\text{insertF}(c)$  adds an occurrence of symbol  $c$  in  $F$ .

The next two subsections detail our implementations of INSERTSTRING and DELETESTRING. The former is a slight modified version of the algorithm introduced in [25], here adapted to deal with the specialties of our dictionary problem: namely, the dictionary strings forming  $\mathcal{S}_{\mathcal{D}}$  must be kept in lexicographic order. The latter coincides with the algorithm presented in [116] for which we prove an additional property (Lemma 13) which is a key for using this result as is in our context.

### 7.3.1 Deleting one dictionary string

The operation  $\text{DELETESTRING}(j)$  requires to delete the string  $s_j$  from the dictionary  $\mathcal{S}$ , and thus recompute the BW-transform  $L'$  of the new string  $\mathcal{S}_{\mathcal{D}}' = s_1 \dots s_{j-1} s_{j+1} \dots s_m \#$ . The key property we deploy next is that this removal does not impact on the ordering of the rows of  $\mathcal{M}_{\mathcal{S}_{\mathcal{D}}}$  which do not refer to suffixes of  $s_j$ .

---

**Algorithm** DELETESTRING( $j$ )

1.  $prev = j + 1; next = n; c = \$;$
  2. **while** ( $next \neq j$ ) **do**
  3.      $next = \text{Back\_step}(prev);$
  4.      $\text{deleteF}(c); c = L[prev]; \text{DELETE}(L, prev);$
  5.     **if**  $prev < next$  **then**  $next = next - 1;$
  6.      $prev = next;$
- 

**Figure 7.4:** Algorithm to delete the string  $\$s_j$  from  $\mathcal{S}_{\mathcal{D}}$ .

**Lemma 13.** *The removal from  $L$  of the symbols of  $\$s_j$  gives the correct string  $\text{Bwt}(\mathcal{S}_{\mathcal{D}}')$ .*

**Proof.** It is enough to prove that the removal of  $\$s_j$  will not influence the order between any pair of rows  $i' < i''$  in  $\mathcal{M}_{\mathcal{S}_{\mathcal{D}}}$ . Take  $i', i''$  as two rows which are not deleted from  $\mathcal{M}_{\mathcal{S}_{\mathcal{D}}}$ , and thus do not start/end with symbols of  $\$s_j$ . We compare the suffix of  $\mathcal{S}_{\mathcal{D}}$  corresponding to the  $i'$ -th row, say  $S_{i'}$ , and the suffix of  $\mathcal{S}_{\mathcal{D}}$  corresponding to the  $i''$ -th row, say  $S_{i''}$ . We recall that these are *increasing* strings, in that they are composed by the dictionary strings which are arranged in increasing lexicographic order and they are separated by the special symbol  $\$$  (see Section 7.2.2). Since all dictionary strings are distinct, the mismatch between  $S_{i'}$  and  $S_{i''}$  occurs before the second occurrence of  $\$$  in them. Let us denote the prefix of  $S_{i'}$  and  $S_{i''}$  preceding the second occurrence of  $\$$  with  $\alpha' \$ s' \$$  and  $\alpha'' \$ s'' \$$ , respectively, where  $\alpha', \alpha''$  are (possibly empty) suffixes of dictionary strings, and  $s', s''$  are dictionary strings. If the mismatch occurs in  $\alpha'$  or  $\alpha''$  we are done, because they are not suffixes of  $\$s_j$  (by the assumption), and therefore they are not interested by the deletion process. If the mismatch occurs in  $s'$  or  $s''$  and they are both different of  $s_j$ , we are also done. The trouble is when  $s' = s_j$  or  $s'' = s_j$ . We consider the first case, because the second is similar. This case occurs when  $|\alpha'| = |\alpha''|$ , so that the order between  $S_{i'}$  and  $S_{i''}$  is given by the order of  $s'$  vs  $s''$ . If  $s' = s_j$ , then the order of the two rows is then given by comparing  $s_{j+1}$  and  $s''$ . Since  $s' < s''$  (because  $S_{i'} < S_{i''}$ ) and  $s_{j+1}$  is the smallest dictionary string greater than  $s'$ , we have that  $s_{j+1} \leq s''$ , and the thesis follows. ■

Given this property, we can use the same string-deletion algorithm of [116] to remove all symbols of  $\$s_j$  from  $L$  and  $F$ . (Figure 7.4 reports the pseudo-code of this algorithm, for the sake of completeness.)

---

**Algorithm**  $\text{LexOrder}(W[1, w])$

1.  $i = w, c = W[w], \text{First} = C[c] + 1;$
2. **while**  $(i \geq 1)$  **do**
3.    $c = W[i - 1];$
4.    $\text{First} = C[c] + \text{RANK}_c(L, \text{First} - 1) + 1;$
5.    $i = i - 1;$
6. **return**  $\text{RANK}_\$(L, \text{First} - 1) + 1$

---

**Figure 7.5:** Algorithm  $\text{LexOrder}(W[1, w])$  returns the lexicographic position of  $W$  in  $\mathcal{S}$ .

### 7.3.2 Inserting one dictionary string

An implementation of  $\text{INSERTSTRING}(W)$  for standard compressed indexes was described in [25]. Here we present a slightly modified version of that algorithm which correctly deals with the maintenance of the lexicographic ordering of the dictionary strings in  $\mathcal{S}_{\mathcal{D}}$ , and the re-computation of its BW-transform. We recall that this order is crucial for the correctness of most of our query operations.

Let  $j$  be the lexicographic position of the string  $W$  in  $\mathcal{S}$ .  $\text{INSERTSTRING}(W)$  requires to recompute the BWT  $L'$  of the new string  $\mathcal{S}_{\mathcal{D}}' = \$s_1\$ \dots \$s_{j-1}\$W\$s_j\$s_{j+1}\$ \dots \$s_m\$\#$ . For this purpose, we can use the reverse of Lemma 13 in order to infer that this insertion does not affect the ordering of the rows already in  $\mathcal{M}_{\mathcal{S}_{\mathcal{D}}}$ . Thus  $\text{INSERTSTRING}(W)$  boils down to insert just the symbols of  $W$  in their correct positions within  $L$  (and, accordingly, in  $F$ ). This is implemented in two main steps: first, we find the lexicographic position of  $W$  in  $\mathcal{S}$  (Algorithm  $\text{LexOrder}(W)$ ); and then, we deploy this position to infer the positions in  $L$  where all symbols of  $W$  have to be inserted (Algorithm  $\text{INSERTSTRING}$ ).

The pseudo-code in Figure 7.5 details algorithm  $\text{LexOrder}(W)$  which assumes that any symbol of  $W$  already occurs in the dictionary strings. If this is not the case, we set  $c = W[x]$  as the leftmost symbol of  $W$  which does not occur in any string of  $\mathcal{S}$ , and set  $c'$  as the smallest symbol which is lexicographically greater than  $c$  and occurs in  $\mathcal{S}$ . If  $\text{LexOrder}$  is correct, then  $\text{LexOrder}(W[1, x - 1]c')$  returns the lexicographic position of  $W$  in  $\mathcal{S}$ .

**Lemma 14.** *Given a string  $W[1, w]$  whose symbols occurs in  $\mathcal{S}$ ,  $\text{LexOrder}(W)$  returns the lexicographic position of  $W$  among the strings in  $\mathcal{S}$ .*



---

**Algorithm** INSERTSTRING( $W[1, w], j$ )

1.  $i = w, \text{First} = j + 1, f = \$$ ;
  2. **while** ( $i \geq 1$ ) **do**
  3.    $c = W[i]$ ;
  4.   INSERT( $L, \text{First}, c$ ); insertF( $f$ );
  5.    $\text{First} = C[c] + \text{RANK}_c(L, \text{First} - 1) + 1$ ;
  6.    $f = c, i = i - 1$ ;
  7. INSERT( $L, \text{First}, \$$ ); insertF( $f$ );
- 

**Figure 7.6:** Algorithm to insert string  $\$W[1, w]$  by knowing its lexicographically order  $j$  among the strings in  $\mathcal{S}$ .

**Proof.** Its correctness derives from the correctness of `Backward_search`. At any step  $i$ , `First` points to the first row of  $\mathcal{M}_{\mathcal{S}_D}$  which is prefixed by the suffix  $W[w - i, w]\$$ . If such a row does not exist, `First` points to the first row of  $\mathcal{M}_{\mathcal{S}_D}$  which is lexicographically greater than  $W[w - i, w]\$$ . ■

Now we have all the ingredients to describe algorithm `INSERTSTRING(W)`. Suppose that  $j$  is the value returned by `LexOrder(W[1, w])`. We have to insert the symbol  $W[i]$  preceding any suffix  $W[i + 1, w]$  in its correct position of  $L' = \text{Bwt}(\mathcal{S}_D')$  and update the string  $F$  too. The algorithm in Figure 7.6 starts from the last symbol  $W[w]$ , and inserts it at the  $(j + 1)$ -th position of `Bwt`( $\mathcal{S}_D$ ) (by Lemma 11). It also inserts the symbol  $\$$  in  $F$ , since it is the first symbol of the  $(j + 1)$ -th row. After that, the algorithm performs a backward step from the  $(j + 1)$ -th row with the symbol  $W[w]$  in order to find the position in  $L$  where  $W[w - 1]$  should be inserted. Accordingly, the symbol  $W[w]$  is inserted in  $F$  too. These insertions are executed in  $L$  and  $F$  until all positions of  $W$  are processed. Step 7 completes the process by inserting the special symbol  $\$$ . Overall, `INSERTSTRING` executes an optimal number of inserts of single symbols in  $L$  and  $F$ . We then use the dynamic data structures of Lemma 12 to dynamically maintain  $L$ , and the solution of [116, Section 7] to maintain  $F$ , thus obtaining:

**Theorem 15.** *Let  $\mathcal{S}$  be a dynamic dictionary of  $m$  strings having total length  $n$ , drawn from an alphabet  $\Sigma$  of size  $\sigma$ . The Dynamic Compressed Permuterm index supports all queries of the Tolerant Retrieval problem with a slowdown factor of  $O((1 + \log \sigma / \log \log n) \log n)$  with respect to its static counterpart (see Theorem 14). Additionally, it can support `INSERTSTRING(W)` in  $O(|W|(1 + \log \sigma / \log \log n) \log n)$*

time; and `DELETESTRING(j)` in  $O(|s_j|(1 + \log \sigma / \log \log n) \log n)$  time.

The space occupancy is bounded by  $nH_k(\mathcal{S}_D) + o(n \log \sigma)$  bits, for any  $k \leq \alpha \log_\sigma n$  and  $0 < \alpha < 1$ .

We point out again that any improvement to Lemma 12 will positively affect the dynamic bounds above.

## 7.4 Experimental Results

We downloaded from <http://law.dsi.unimi.it/> various crawls of the web—namely, `arabic-2005`, `indocina-2004`, `it-2004`, `uk-2005`, `web-base-2001` [19]. We extracted from `uk-2005` about 190Mb of distinct urls, and we derived from all crawls about 34Mb of distinct host-names. The dictionary of urls and hosts have been lexicographically sorted by *reversed host names* in order to maximize the longest common-prefix (shortly, `lcp`) shared by strings adjacent in the lexicographic order. We have also built a dictionary of (alphanumeric) terms by parsing the TREC collection `WT10G` and by dropping (spurious) terms longer than 50 symbols. These three dictionaries are representatives of string sets usually manipulated in Web search and mining engines.

Table 7.1 reports some statistics on these three dictionaries: `DictUrl` (the dictionary of urls), `DictHost` (the dictionary of hosts), and `DictTerm` (the dictionary of terms). In particular lines 3-5 describe the composition of the dictionaries at the *string level*, lines 6-8 account for the repetitiveness in the dictionaries at the *string-prefix level* (which affects the performance of front-coding and trie, see below), and the last three lines account for the repetitiveness in the dictionaries at the *sub-string level* (which affects the performance of compressed indexes). It is interesting to note that the `TotalLcp` varies between 55–69% of the dictionary size, whereas the amount of compression achieved by `gzip`, `bzip2` and `ppmdi` is superior and reaches 67–92%. This proves that there is much repetitiveness in these dictionaries not only at the string-prefix level but also *within* the strings. The net consequence is that compressed indexes, which are based on the Burrows-Wheeler Transform (and thus have the same `bzip2-core`), should achieve on these dictionaries significant compression, much better than the one achieved by front-coding based schemes!

In Tables 7.2 and 7.3 we test the time and space performance of three (compressed) solutions to the Tolerant Retrieval problem:

**CPI** is our Compressed Permuterm Index of Section 7.2.2. In order to com-

Statistics	DictUrl	DictHost	DictTerm
Size (Mb)	190	34	118
$\sigma$	95	52	36
# strings	3,034,144	1,778,927	10,707,681
Avg_len strings	64.92	18.91	10.64
Max_len strings	1,138	180	50
Avg_lcp	45.85	11.25	6.81
Max_lcp	720	69	49
Total_lcp	68.81%	55.27%	58.50%
gzip -9	11.49%	23.77%	29.50%
bzip2 -9	10.86%	24.03%	32.58%
ppmdi -19	8.32%	19.08%	29.06%

**Table 7.1:** Statistics on our three dictionaries.

Method	DictUrl	DictHost	DictTerm
Trie	1374.29%	1793.19%	1727.93%
FC-32	109.95%	113.22%	106.45%
FC-128	107.41%	109.91%	102.10%
FC-1024	106.67%	108.94%	100.84%
CPI-AFI	49.72%	47.48%	52.24%
CPI-CSA-64	37.82%	56.36%	73.98%
CPI-CSA-128	31.57%	50.11%	67.73%
CPI-CSA-256	28.45%	46.99%	64.61%
CPI-FMI-256	24.27%	40.68%	55.41%
CPI-FMI-512	18.94%	34.58%	47.80%
CPI-FMI-1024	16.12%	31.45%	44.13%

**Table 7.2:** Space occupancy is reported as a percentage of the original dictionary size. Recall that Trie and Fc are built on both the dictionary strings and their reversals, in order to support PREFIXSUFFIX queries.

Method	DictUrl		DictHost		DictTerm	
	10	60	5	15	5	10
Trie	0.1	0.2	0.4	0.5	1.2	0.9
FC-32	1.3	0.4	1.5	1	2.5	1.7
FC-128	3.2	1.0	3.4	1.8	4.6	2.8
FC-1024	26.6	5.2	24.6	11.0	25.0	14.6
CPI-AFI	1.8	2.9	1.6	2.5	2.9	3.0
CPI-CSA-64	4.9	5.6	4.3	5.2	5.4	5.7
CPI-CSA-128	7.3	8.0	6.9	7.6	7.6	8.3
CPI-CSA-256	11.8	14.1	11.8	12.5	12.8	13.2
CPI-FMI-256	11.9	9.8	19.3	15.5	22.5	20.1
CPI-FMI-512	16.2	13.4	28.4	23.1	34.2	30.3
CPI-FMI-1024	24.1	20.7	46.4	38.4	57.6	50.1

**Table 7.3:** Timings are given in  $\mu\text{secs}/\text{char}$  averaged over one million of searched patterns, whose length is reported at the top of each column. Value  $b$  denotes in CPI-FMI- $b$  the bucket size of the FM-index, in CPI-CSA- $b$  the sample rate of the function  $\Psi$  [44], and in FC- $b$  the bucket size of the front-coding scheme. We recall that  $b$  allows in all these solutions to trade space occupancy per query time.

press the string  $\mathcal{S}_D$  and implement procedures `BackPerm_search` and `Display_string`, we modified three types of compressed indexes available under the `Pizza&Chili` site [44] and discussed in Chapter 6, which represent the best choices in this setting. Namely `CSA`, `FM-index v2` (shortly `FMI`), and the alphabet-friendly `FM-index` (shortly `AFI`). We tested three variants of `CSA` and `FMI` by properly setting their parameter which allows to trade space occupancy by query performance.

**FC** data structure applies *front-coding* to groups of  $b$  adjacent strings in the sorted dictionary, and then keeps explicit pointers to the beginners of every group [164].

**Trie** is the ternary search tree of Bentley and Sedgewick which “combines the time efficiency of digital tries with the space efficiency of binary search trees” [17].<sup>2</sup>

Theorem 14 showed that `Cpi` supports efficiently all queries of the Tolerant Retrieval problem. The same positive feature does not hold for the other two data structures. In fact `Fc` and `Trie` support only prefix searches over the indexed strings. Therefore, in order to implement the `PREFIXSUFFIX` query, we need to build these data structures *twice*— one on the strings of  $\mathcal{S}$  and the other on their reversals. This *doubles* the space occupancy, and slows down the search performance because we need to first make two prefix-searches, one for  $P$ 's prefix  $\alpha$  and the other for  $P$ 's suffix  $\beta$ , and then we need to *intersect* the two candidate lists of answers. If we wish to also support the rank/select primitives, we need to add some auxiliary data that keep information about the left-to-right numbering of trie leaves, thus further increasing the space occupancy of the trie-based solution. In Table 7.2 we account for such “space doubling”, but not for the auxiliary data, thus giving an advantage in space to these data structures wrt `Cpi`. It is evident the large space occupancy of ternary search trees because of the use of pointers and the explicit storage of the dictionary strings (without any compression). As predicted from the statistics of Table 7.1, `Fc` achieves a compression ratio of about 40% on the original dictionaries, but more than 60% on their reversal. Further, we note that `Fc` space improves negligibly if we vary the bucket size  $b$  from 32 to 1024 strings, and achieves the best space/time trade-off when  $b = 32$ .<sup>3</sup> In summary, the space occupancy of the `Fc` solution is more than the original dictionary size, if we wish to support all queries of the Tolerant Retrieval problem! As far as

<sup>2</sup>Code at <http://www.cs.princeton.edu/~rs/strings/>.

<sup>3</sup>A smaller  $b$  would enlarge the extra-space dedicated to pointers, a larger  $b$  would impact seriously on the time efficiency of the prefix searches.

the variants of `Cpi` are concerned, we note that their space improvement is significant: a multiplicative factor from 2 to 7 wrt `Fc`, and from 40 to 86 wrt `Trie`.

In Section 7.2.1 we mentioned another simple solution to the Tolerant Retrieval problem which was based on the compressed indexing of the string  $\widehat{\mathcal{S}}_{\mathcal{D}}$ , built by juxtaposing *twice* every string of  $\mathcal{S}$ . In that section we argued that this solution is *inefficient* in indexing time and compressed-space occupancy because of this “string duplication” process. Here we investigate experimentally our conjecture by computing and comparing the  $k$ -th order empirical entropy of the two strings  $\widehat{\mathcal{S}}_{\mathcal{D}}$  and  $\mathcal{S}_{\mathcal{D}}$ . As predicted theoretically, the two entropy values are close for all three dictionaries, thus implying that the compressed indexing of  $\widehat{\mathcal{S}}_{\mathcal{D}}$  should require about twice the compressed indexing of  $\mathcal{S}_{\mathcal{D}}$  (recall that  $|\widehat{\mathcal{S}}_{\mathcal{D}}| = 2|\mathcal{S}_{\mathcal{D}}| - 1$ ). To check this, we have then built two FM-indexes: one on  $\widehat{\mathcal{S}}_{\mathcal{D}}$  and the other on  $\mathcal{S}_{\mathcal{D}}$ , by varying  $\mathcal{S}$  over the three dictionaries. We found that the space occupancy of the FM-index built on  $\widehat{\mathcal{S}}_{\mathcal{D}}$  is a factor 1.6–1.9 worse than our `Cpi-Fmi` built on  $\mathcal{S}_{\mathcal{D}}$ . So we were right when in Section 7.2.1 we conjectured the inefficiency of the compressed indexing of  $\widehat{\mathcal{S}}_{\mathcal{D}}$ .

We have finally tested the time efficiency of the above indexing data structures over a P4 2.6 GHz machine, with 1.5 Gb of internal memory and running Linux kernel 2.4.20. We executed a large set of experiments by varying the searched-pattern length, and by searching one million patterns per length. Since the results were stable over all these timings, we report in Table 7.3 only the most significant ones by using the notation *microsecs per searched symbol* (shortly  $\mu\text{s}/\text{char}$ ): this is obtained by dividing the overall time of an experiment by the total length of the searched patterns. We remark that the timings in Table 7.3 account for the cost of searching a pattern prefix and a pattern suffix of the specified length. While this is the total time taken by our `Cpi` to solve a `PREFIXSUFFIX` query, the timings for `Fc` and `Trie` are *optimistic* evaluations because they should also take into account the time needed to intersect the candidate list of answers returned by the prefix/suffix queries! Keeping this in mind, we look at Table 7.3 and note that `Cpi` allows to trade space occupancy per query time: we can go from a space close to `gzip-ppmdi` and access time of 20–57  $\mu\text{s}/\text{char}$  (i.e. `CPI-FMI-1024`), to an access time similar to `Fc` of few  $\mu\text{s}/\text{char}$  but using less than half of its space (i.e. `CPI-AFI`). Which variant of `Cpi` to choose depends on the application for which the Tolerant Retrieval problem must be solved.

We finally notice that, of course, any improvement to compressed indexes [134] will immediately and positively impact onto our `Cpi`, both in theory and in practice. Overall our experiments show that `Cpi` is a *novel compressed*

*storage scheme for string dictionaries* which is fast in supporting the sophisticated searches of the Tolerant Retrieval problem, and is as compact as the best known compressors!

## 7.5 Further considerations

In [121] the more sophisticated wild-card query  $P = \alpha * \beta * \gamma$  is also considered and implemented by intersecting the set of strings containing  $\gamma\$ \alpha$  with the set of strings containing  $\beta$ . Our compressed permuterm index allows to avoid the *materialization* of these two sets by working only on the compressed index built on the string  $\mathcal{S}_{\mathcal{D}}$ . The basic idea consists of the following steps:

- Compute  $[\text{First}', \text{Last}'] = \text{BackPerm\_search}(\gamma\$ \alpha)$ ;
- Compute  $[\text{First}'', \text{Last}''] = \text{BackPerm\_search}(\beta)$ ;
- For each  $r \in [\text{First}', \text{Last}']$  repeatedly apply **Back\_step** of Figure 7.2 until it finds a row which either belongs to  $[\text{First}'', \text{Last}'']$  or to  $[1, m]$  (i.e. starts with \$).
- In the former case  $r$  is an answer to  $\text{WILDCARD}(P)$ , in the latter case it is not.

The number of **Back\_step**'s invocations depends on the length of the dictionary strings which match the query  $\text{PREFIXSUFFIX}(\alpha * \gamma)$ . In practice, it is possible to engineer this paradigm to reduce the total number of **Back\_steps** (see Chapter 6, FM-indexV2). The above scheme can be also used to answer more complex queries as  $P = \alpha * \beta_1 * \beta_2 * \dots * \beta_k * \gamma$ , with possibly empty  $\alpha$  and  $\gamma$ . The efficiency depends on the *selectivity* of the individual queries  $\text{PREFIXSUFFIX}(\alpha * \gamma)$  and  $\text{SUBSTRING}(\beta_i)$ , for  $i = 1, \dots, k$ .

It would be then interesting to extend our results in two directions, either by proving guaranteed and efficient worst-case bounds for queries with *multiple* wild-card symbols, or by turning our Compressed Permuterm index in a I/O-conscious or, even better, cache-oblivious compressed data structure. This latter issue actually falls in the key challenge of current data structural design: does it exist a cache-oblivious compressed index?

## Future directions of research

We conclude the thesis by presenting some of the most important open problems of this fascinating field of research.

**Exact optimal partitioning.** In Chapter 3 we have investigated the problem of partitioning an input string  $T$  in such a way that compressing individually its parts via a base-compressor  $\mathcal{C}$  gets a compressed output that is shorter than applying  $\mathcal{C}$  over the entire  $T$  at once. We provide an algorithm which is guaranteed to compute in  $O(n \log_{1+\epsilon} n)$  time a partition of  $T$  whose compressed output is guaranteed to be no more than  $(1 + \epsilon)$ -worse the optimal one, where  $\epsilon$  may be any positive constant. An interesting open question consists in understanding if it is possible to design an  $o(n^2)$  time solution for computing the *exact* optimal partition.

**Speeding up solutions of dynamic programming recurrences.** Many applications require to solve efficiently dynamic programming recurrences (see [71] and references therein). The simplest type of recurrence, called 1D/1D, have the form  $E[j] = \min_{i < j} (E[i] + c(i, j))$  and is used as a building block to solve more sophisticated recurrences. The trivial algorithm solves such recurrences in quadratic time and, by simple argumentations, it has been shown that this algorithm is optimal for general cost functions  $c()$ . Nevertheless, subquadratic solutions can be obtained whenever the cost function  $c()$  has particular properties that can be exploited. For example, in literature are known efficient algorithms that permit to solve these recurrences even in linear time provided that the cost function  $c()$  satisfies a property called quadrangle inequality or some of its variants. However, properties related to quadrangle inequality



may not hold in many contexts. As an example consider the recurrences that come out in the two problems addressed in chapters 3 and 4. Our efficient algorithms have been obtained by showing that dynamic programming recurrences can be solved/approximated in subquadratic time even when function  $c()$  is increasing without requiring properties related to quadrangle inequality. We believe that this research can be refined and integrated in order to better understand which are necessary and/or sufficient properties that function  $c()$  must be satisfied in order to guarantee subquadratic solutions.

**Random Access in compressed strings in RAM model.** The scheme presented in Chapter 5 is very simple and, as deeply illustrated in [148], may find successful applications into many other interesting contexts (see [12, 15, 27, 39, 61, 87, 91, 92, 136, 148] and references therein). However, all known solutions are far from being usable in practice because of the additive term  $o(n \log \sigma)$  which usually dominates the  $k$ -th order entropy term. More research is still needed to either to reduce the lower order term as much as possible (e.g., removing the factor  $k \log \sigma$  in the term  $O(\frac{n}{\log_\sigma n} k \log \sigma)$  would be a valuable improvement), or to show a lower bound related to  $k$ -th order entropy. Since our storage scheme, unlike [79, 148], does not use any sophisticated data compression machinery, we are led to think that there is room for improvement.

**Random Access in compressed string on External-Memory model.** This problem extends the one stated in the previous point to the External-Memory model, for which the complexity of retrieving a substring from  $T$  is measured in terms of I/Os. In this scenario a time optimal solution is one that retrieves any  $B \log_\sigma n$  consecutive symbols of  $T$  with  $O(1)$  disk accesses. The scheme we presented in Chapter 5 might work well in the external-memory model too, except for three main inefficiencies: (1) its blocking approach reduces the overall compression ratio by posing a limit to the value  $k$  which is too small for common alphabet sizes, since it must be  $o(\log_\sigma n)$ ; (2) it does not completely exploit the fact that internal-memory operations have no cost; (3) the table used for block-compression may not fit in the internal memory, and thus may force us to reduce furthermore the block size. Following the definition of  $k$ th order empirical entropy (see Section 2.4.1), we can obtain an I/O conscious and more space-efficient scheme by storing in internal memory the *model* corresponding to all contexts occurring in  $T$ , and then compressing any symbol of  $T$  according to its preceding context and its number of occurrences. Clearly, this model could be very large and thus not fit in internal memory. An interesting open problem is therefore how to *prune* the whole model in order to fit it in internal memory and achieve the *maximum* compression among all models which satisfy that space-bound. As proved in [43], a further difficulty

arises from the fact that the best possible compression ratio is not necessarily achieved by considering contexts of the same length.

**Faster COUNT queries.** The time complexity of COUNT procedure of (compressed) full-text indexes is non optimal in the RAM model. An interesting open problem concerns the possibility of designing a compressed full-text index that supports  $\text{COUNT}(P[1, p])$  in (close to) optimal time (namely,  $O(\frac{p}{\log_{\sigma} n})$ ). The first result in this direction appeared in [82] in which it has been shown how to build a non compressed full-text index on  $T$  that requires  $(\epsilon^{-1} + O(1))n \log \sigma$  bits of space and performs counting operations in  $O(1)$  time if  $p = o(\log_{\sigma} n)$  or  $O(\frac{p}{\log_{\sigma} n} + \log_{\sigma}^{\epsilon} n)$  time otherwise, for any fixed value of  $0 < \epsilon < 1$ . A step forward has been done in [81] that proposes a compressed full-text index that occupies  $\epsilon^{-1}nH_k + O(n\frac{\log \log n}{\log_{\sigma}^{\epsilon} n})$  bits of space keeping a similar time complexity, for any fixed value of  $0 < \epsilon < \frac{1}{3}$ .

**Faster LOCATE queries.** Another open challenge concerning compressed indexes is to fasten their locate queries in order to achieve the optimal  $O(occ)$  time bound. The best known results are two indexes due to He *et al.* [85] and Ferragina and Manzini [52]. The former considers only binary texts and locates the  $occ$  occurrences of a pattern  $P[1, p]$  in  $O(p + occ)$  time for large enough  $p$  and  $2n + o(n)$  bits of space (so it is not compressed); while the latter has no restriction on  $p$  but requires space  $O(nH_k(T) \log^{\epsilon} n) + o(n \log \sigma)$  bits (which has the extra log-factor in front of the optimal  $nH_k(T)$  term). Is it possible to achieve  $O(p + occ)$  (or even better  $O(\frac{p}{\log_{\sigma} n} + occ)$ ) searching time and  $O(nH_k(T)) + o(n \log \sigma)$  bits of space in the worst case? This result would be provably better than classical data structures.

**Compressed indexes on External-Memory model.** The memory of current PCs is hierarchical so that, in order to achieve effective algorithmic performance, cache-aware or even cache-oblivious solutions should be designed. Although their small space requirements might permit compressed indexes to fit in main memory, there will always be cases where they have to operate on external memory. The most attractive full-text indexes for secondary memory are the String B-tree [45], the Self-adjusting Suffix Tree [106] and the two cache-oblivious String B-trees [16, 46]. Unfortunately these data structures do not achieve higher order entropy. A recent result [88] shows that a text  $T$  can be indexed in  $O(n(H_k(T) + 1)) + o(n \log \sigma)$  bits and such that all occurrences of a pattern  $P[1, p]$  in  $T$  can be reported in  $O(p/(B \log_{\sigma} n) + \frac{\log^4 n}{\log \log n} + occ \log_B n)$  I/Os where  $occ$  is the number of occurrences of  $P$  in  $T$ . Thus, the

index achieves optimal query I/O performance with respect to the length  $P$  (namely,  $O(p/B)$  I/Os but not with respect to the number of its occurrences. Unfortunately, [27] noted that lower bounds in range searching data structures suggest that the last term  $O(occ \log_B n)$  cannot be improved to  $O(occ/B)$  by occupying  $O(n \log \sigma)$  bits of space. This implies that there are only two ways to improve the above index: 1) reduce the middle polylogarithmic term, and 2) reduce the space term from  $O(n(H_k(T) + 1))$  to exactly  $nH_k(T)$ .

**Space efficient Bwt construction.** An interesting problem of great practical significance concerns the auxiliary space needed to build the compressed indexes. We require a space efficient computation of the **Bwt**, since many such indexes are based on it. The **Bwt** of a string  $T$  can be stored using  $n \log \sigma$  bits but the linear time algorithms used to construct it make use of auxiliary arrays (i.e. suffix array) whose storage takes  $\Theta(n \log n)$  bits. This poses a serious limitation to the size of the largest **Bwt** that can be computed efficiently in internal memory. The problem of space and time efficient computation of large **Bwt** is still open even if interesting preliminary results are proposed in [89, 96]. Other approaches (e.g., the one described in Chapter 7) process the text from left to right by adding one symbol at a time to the partial **Bwt**. The space required by such solutions is  $nH_k(T) + o(n \log \sigma)$  bits but their time complexity is not linear (namely,  $O(n(1 + \log \sigma / \log \log n) \log n)$ ). We also mention the result in [40] that proposes an algorithm that requires just  $o(n)$  bits of auxiliary space (here original text is replaced with its **Bwt**) and computes the **Bwt** in  $O(n \log^{1+\epsilon} n)$  time, for any  $\epsilon > 0$ . A natural question arises: Is it possible to build the **Bwt** of a string  $T$  in linear time using  $O(n \log \sigma)$  (or better  $O(nH_k(T))$ ) bits of auxiliary space?

# References

- [1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
- [2] S. Aluru and P. Ko. *Encyclopedia of Algorithms*, chapter on “Lookup Tables, Suffix Trees and Suffix Arrays”. Springer, 2008.
- [3] A. Amir, G. M. Landau, and E. Ukkonen. Online timestamped text indexing. *Information Processing Letters*, 82(5):253–259, 2002.
- [4] A. Amir, G.M. Landau, M. Lewenstein, and D. Sokol. Dynamic text and static pattern matching. *ACM Transactions on Algorithms*, 3(2), 2007.
- [5] A. Andersson and S. Nilsson. Efficient implementation of suffix trees. *Software Practice & Experience*, 25(2):129–141, 1995.
- [6] D. Arroyuelo and G. Navarro. Space-efficient construction of LZ-index. In *Proceedings of the 16th Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS vol. 3827, pages 1143–1152. Springer, 2005.
- [7] D. Arroyuelo and G. Navarro. Practical approaches to reduce the space requirement of Lempel-Ziv-based compressed text indices. Technical Report TR/DCC-2008-9, Dept. of Computer Science, Univ. of Chile, 2008.
- [8] D. Arroyuelo, G. Navarro, and K. Sadakane. Reducing the space requirement of LZ-index. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS vol. 4009, pages 319–330. Springer, 2006.

- 
- [9] R.A. Baeza-Yates and G. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM*, 43(6):915–936, 1996.
- [10] R.A. Baeza-Yates and B.A. Ribeiro-Neto. *Modern Information Retrieval*. ACM/Addison-Wesley, 1999.
- [11] J. Barbay, M. He, I. J. Munro, and S. Srinivasa Rao. Succinct indexes for string, binary relations and multi-labeled trees. In *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 680–689, 2007.
- [12] J. Barbay and J. I. Munro. Succinct encoding of permutations: Applications to text indexing. In *Encyclopedia of Algorithms*. 2008.
- [13] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem. In *Proceedings of the 31st ACM Symposium on Theory of Computing (STOC)*, pages 295–304, 1999.
- [14] J. Békési, G. Galambos, U. Pferschy, and G.J. Woeginger. Greedy algorithms for on-line data compression. *Journal of Algorithms*, 25(2):274–289, 1997.
- [15] D. Belazzougui, F. C. Botelho, and M. Dietzfelbinger. Hash, displace, and compress. In *Proceedings of the 17th Annual European Symposium on Algorithms (ESA)*, pages 682–693, 2009.
- [16] M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. Cache-oblivious string b-trees. In *Proceedings of the 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 233–242, 2006.
- [17] J. L. Bentley and R. Sedgwick. Fast algorithms for sorting and searching strings. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 360–369, 1997.
- [18] J.L. Bentley and M.D. McIlroy. Data compression with long repeated strings. *Information Sciences*, 135(1-2):1–11, 2001.
- [19] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. UbiCrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.
- [20] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web (WWW)*, pages 595–602, 2004.

- 
- [21] A. L. Buchsbaum, D. F. Caldwell, K. W. Church, G. S. Fowler, and S. Muthukrishnan. Engineering the compression of massive tables: an experimental approach. In *Proceedings 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 175–184, 2000.
- [22] A. L. Buchsbaum, G. S. Fowler, and R. Giancarlo. Improving table compression with combinatorial optimization. *Journal of the ACM*, 50(6):825–851, 2003.
- [23] A.L. Buchsbaum and R. Giancarlo. *Encyclopedia of Algorithms*, chapter Table Compression, pages 939–942. Springer, 2008.
- [24] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
- [25] H.L. Chan, W.K. Hon, T.W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2), 2007.
- [26] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2), 2008.
- [27] Y.-F. Chien, W.-K. Hon, R. Shah, and J. S. Vitter. Geometric burrows-wheeler transform: Linking range searching and text indexing. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 252–261, 2008.
- [28] R. Cilibrasi and P. M. B. Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.
- [29] M. Cohn and R. Khazan. Parsing with prefix and suffix dictionaries. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 180–189, 1996.
- [30] R. Cole, T. Kopelowitz, and M. Lewenstein. Suffix trays and suffix trists: Structures for faster text indexing. In *Proceedings of the 33th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 358–369, 2006.
- [31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.

- [32] G. Cormode and S. Muthukrishnan. Substring compression problems. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 321–330, 2005.
- [33] M. Crochemore, L. Ilie, and W. F. Smyth. A simple algorithm for computing the Lempel-Ziv factorization. In *Proceedings of the IEEE Data Compression Conference (DCC)*, pages 482–488, 2008.
- [34] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific Publishing Company, 2003.
- [35] S. Dasgupta, C. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw-Hill Science/Engineering/Math, 2006.
- [36] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.
- [37] M. Farach-Colton. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium Foundation Computer Science (FOCS)*, pages 137–143, 1997.
- [38] A. Farzan and I. J. Munro. Succinct representations of arbitrary graphs. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA)*, pages 393–404, 2008.
- [39] P. Ferragina and J. Fischer. Suffix arrays on words. In *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 328–339, 2007.
- [40] P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. In *Proceedings of the 10th Latin American Symposium on Theoretical Informatics (LATIN)*, 2010 (to appear).
- [41] P. Ferragina, R. Giancarlo, and G. Manzini. The engineering of a compression boosting library: Theory vs practice in BWT compression. In *Proceedings of the 14th European Symposium on Algorithms (ESA)*, pages 756–767. LNCS vol. 4168, 2006.
- [42] P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. *Information and Computation*, 207:849–866, 2009.
- [43] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52:688–713, 2005.

- 
- [44] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13, 2008.
- [45] P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of ACM*, 46(2):236–280, 1999.
- [46] P. Ferragina, R. Grossi, A. Gupta, R. Shah, and J. S. Vitter. On searching compressed string collections cache-obliviously. In *Proceedings of the 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 181–190, 2008.
- [47] P. Ferragina, N. Koudas, S. Muthukrishnan, and D. Srivastava. Two-dimensional substring indexing. *Journal of Computer System Science*, 66(4):763–774, 2003.
- [48] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proceedings of the 46th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 184–193, 2005.
- [49] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and searching XML data via two zips. In *Proceedings of the 15th International World Wide Web Conference (WWW)*, pages 751–760, 2006.
- [50] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and searching XML data via two zips. In *Proceedings of the 15th World Wide Web Conference (WWW)*, pages 751–760, 2006.
- [51] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 269–278, 2001.
- [52] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.
- [53] 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.
- [54] P. Ferragina, I. Nitto, and R. Venturini. On optimally partitioning a text to improve its compression. In *Proceedings of the 17th Annual European Symposium on Algorithms (ESA)*, pages 420–431, 2009.



- 
- [55] P. Ferragina, I. Nitto, and R. Venturini. On the bit-complexity of Lempel-Ziv compression. In *Proceedings of the 20th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 768–777, 2009.
- [56] P. Ferragina and R. Venturini. Compressed permuterm index. In *Proceedings of the 30th Annual International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 535–542, 2007.
- [57] P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. *Theoretical Computer Science*, 372(1):115–121, 2007.
- [58] P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. In *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 690–696, 2007.
- [59] P. Ferragina and R. Venturini. The compressed permuterm index. *ACM Transactions on Algorithms (to appear)*, 2009.
- [60] E. R. Fiala and D. H. Greene. Data compression with finite windows. *Communications of the ACM*, 32(4):490–505, 1989.
- [61] J. Fischer. Short labels for lowest common ancestors in trees. In *Proceedings of the 17th Annual European Symposium on Algorithms (ESA)*, pages 752–763, 2009.
- [62] L. Foschini, R. Grossi, A. Gupta, and J. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms*, 2(4):611–639, 2006.
- [63] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.
- [64] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994.
- [65] T. Gagie. Compressing probability distributions. *Information Processing Letters*, 97(4):133–137, 2006.
- [66] T. Gagie and G. Manzini. Space-conscious compression. In *Proceedings of the 32th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 206–217, 2007.

- 
- [67] T. Gagie and G. Manzini. *Encyclopedia of Algorithms*, chapter Dictionary-Based Data Compression. Springer, 2008.
- [68] E. Garfield. The permuterm subject index: An autobiographical review. *Journal of the ACM*, 27:288–291, 1976.
- [69] R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms*, 2(4):510–534, 2006.
- [70] D. Geer. Reducing the storage burden via data deduplication. *Computer*, 41(12):15–17, 2008.
- [71] R. Giancarlo. Dynamic programming: special cases. In Alberto Apostolico and Zvi Galil, editors, *Pattern Matching Algorithms*, pages 201–236. Oxford Univ. Press, 2nd edition, 1997.
- [72] R. Giancarlo, A. Restivo, and M. Sciortino. From first principles to the Burrows and Wheeler transform and beyond, via combinatorial optimization. *Theoretical Computer Science*, 387(3):236–248, 2007.
- [73] R. Giancarlo and M. Sciortino. Optimal partitions of strings: A new class of Burrows-Wheeler compression algorithms. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 129–143. LNCS vol. 2676, 2003.
- [74] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. *Software Practice & Experience*, 33(11):1035–1049, 2003.
- [75] A. Golynski. Optimal lower bounds for rank and select indexes. In *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming (ICALP)*, 2006.
- [76] A. Golynski. Cell probe lower bounds for succinct data structures. In *Proceedings of the 20th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 625–634, 2009.
- [77] A. Golynski, R. Raman, and S. S. Rao. On the redundancy of succinct data structures. In *Proceedings of the 11th Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 148–159. LNCS vol. 5124, 2008.
- [78] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005.

- [79] R. González and G. Navarro. Statistical encoding of succinct data structures. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 294–305, 2006.
- [80] R. González and G. Navarro. Improved dynamic rank-select entropy-bound structures. In *Proceedings of the 8th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 4957, pages 374–386, 2008.
- [81] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.
- [82] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.
- [83] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
- [84] T. Hagerup and T. Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Proceedings of the 18th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 317–326, 2001.
- [85] M. He, I. J. Munro, and S. S. Rao. A categorization theorem on suffix arrays with applications to space efficient text indexes. In *Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 23–32, 2005.
- [86] Gzip home page. <http://www.gzip.org>.
- [87] W.-K. Hon, T. W. Lam, R. Shah, S.-L. Tam, and J. S. Vitter. Compressed index for dictionary matching. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 23–32, 2008.
- [88] W.-K. Hon, R. Shah, S. V. Thankachan, and J. S. Vitter. On entropy-compressed text indexing in external memory. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 75–89, 2009.
- [89] W.K. Hon, K. Sadakane, and W. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 251–260. IEEE Computer Society, 2003.

- [90] P. G. Howard and J. S. Vitter. Analysis of arithmetic coding for data compression. *Information Processing Management*, 28(6):749–764, 1992.
- [91] Fischer J, V. Heun, and H. M. Stühler. Practical entropy-bounded schemes for  $o(1)$ -range minimum queries. In *Proceedings of IEEE Data Compression Conference (DCC)*, 2008.
- [92] J. Jansson, K. Sadakane, and W.K. Sung. Ultra-succinct representation of ordered trees. In *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 575–584, 2007.
- [93] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proceedings of the 16th Annual Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 16, pages 240–248, 1991.
- [94] H. Kaplan, S. Landau, and E. Verbin. A simpler analysis of Burrows-Wheeler-based compression. *Theoretical Computer Science*, 387(3):220–235, 2007.
- [95] J. Kärkkäinen. Suffix cactus: a cross between suffix tree and suffix array. In *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS vol. 937, pages 191–204. Springer, 1995.
- [96] J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387(3):249–257, 2007.
- [97] J. Kärkkäinen and S. S. Rao. Full-text indexes in external memory. In *Algorithms for Memory Hierarchies*, pages 149–170, 2002.
- [98] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of ACM*, 53(6):918–936, 2006.
- [99] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proceedings of the 3rd South American Workshop on String Processing (WSP)*, pages 141–155. Carleton University Press, 1996.
- [100] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proceedings of the 2nd Annual International Conference on Computing and Combinatorics (COCOON)*, LNCS vol. 1090, pages 219–230. Springer, 1996.
- [101] J. Katajainen and T. Raita. An approximation algorithm for space-optimal encoding of a text. *Computer Journal*, 32(3):228–237, 1989.

- [102] J. Katajainen and T. Raita. An analysis of the longest match and the greedy heuristics in text encoding. *Journal of the ACM*, 39(2):281–294, 1992.
- [103] S. T. Klein. Efficient optimal recompression. *Computer Journal*, 40(2/3):117–126, 1997.
- [104] D. E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.
- [105] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005.
- [106] P. Ko and S. Aluru. Optimal self-adjusting trees for dynamic string data in secondary storage. In *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 184–194, 2007.
- [107] R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.
- [108] P. Kulkarni, F. Douglis, J.D. LaVoie, and J.M. Tracey. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference*, pages 59–72, 2004.
- [109] S. Kurtz. Reducing the space requirement of suffix trees. *Software—Practice and Experience*, 29(13):1149–1171, 1999.
- [110] T.-W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. In *Proceedings of the 8th Conference on Computing and Combinatorics (COCOON)*, LNCS vol. 2387, pages 401–410. Springer, 2002.
- [111] N. J. Larsson and K. Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007.
- [112] O. Lehtinen, E. Sutinen, and J. Tarhio. Experiments on block indexing. In *Proceedings of the 3rd South American Workshop on String Processing (WSP)*, pages 183–193. Carleton University Press, 1996.
- [113] K. Lemström and S. Perttu. SEMEX – an efficient music retrieval prototype. In *Proceedings of the 1st International Symposium on Music Information Retrieval (ISMIR)*, 2000.

- 
- [114] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.
- [115] V. Mäkinen and G. Navarro. Position-restricted substring searching. In *Proceedings of the 7th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 703–714. LNCS vol. 3887, 2006.
- [116] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 229–241. LNCS col. 4726, 2007.
- [117] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3):article 32, 2008.
- [118] V. Mäkinen and G. Navarro. On self-indexing images – image compression with added value. In *Proceedings of the Data Compression Conference (DCC)*, pages 422–431, 2008.
- [119] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 22:935–948, 1993.
- [120] U. Manber and S. Wu. Glimpse: a tool to search through entire file systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 4–4. USENIX Association, 1994.
- [121] C. D. Manning, P. Raghavan, and H. Schülze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.
- [122] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the Burrows-Wheeler transform and applications to sequence comparison and data compression. In *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 178–189, 2005.
- [123] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
- [124] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.
- [125] Y. Matias and S.C. Şahinalp. On the optimality of parsing in dynamic dictionary based data compression. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 943–944, 1999.

- [126] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.
- [127] A. Moffat and R.Y. Isal. Word-based text compression using the Burrows-Wheeler transform. *Information Processing Management*, 41(5):1175–1192, 2005.
- [128] I. J. Munro. Tables. In *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS vol. 1180, pages 37–42. Springer, 1996.
- [129] I. J. Munro, R. Raman, V. Raman, and S. Rao. Succinct representations of permutations. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS vol. 2719, pages 345–356. Springer, 2003.
- [130] I. J. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 118–126. IEEE Computer Society, 1997.
- [131] G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.
- [132] G. Navarro and R. Baeza-Yates. A practical  $q$ -gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2), 1998.
- [133] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.
- [134] G. Navarro and V. Mäkinen. Compressed full text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
- [135] G. Navarro, E. Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.
- [136] I. Nitto and R. Venturini. On compact representations of all-pairs-shortest-path-distance matrices. In *Proceedings of the 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 166–177, 2008.
- [137] Z. Ouyang, N.D. Memon, T. Suel, and D. Trendafilov. Cluster-based delta compression of a collection of files. In *Proceedings of the 3rd Conference on Web Information Systems Engineering (WISE)*, pages 257–268. IEEE Computer Society, 2002.

- 
- [138] M. Patrascu. Succincter. In *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 305–313, 2008.
- [139] S. Puglisi, W. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2), 2007.
- [140] S. J. Puglisi, W. F. Smyth, and A. Turpin. Inverted files versus suffix arrays for locating patterns in primary memory. In *Proceedings of the 13th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS vol. 4209, pages 122–133. Springer, 2006.
- [141] N. Rajpoot and C. Sahinalp. *Handbook of Lossless Data Compression*, chapter Dictionary-based data compression, pages 153–167. Academic Press, 2002.
- [142] N. Rajpoot and C. Sahinalp. *Handbook of Lossless Data Compression*, chapter Dictionary-based data compression, pages 153–167. Academic Press, 2002.
- [143] R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding  $\alpha$ -ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.
- [144] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 225–232, 2002.
- [145] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
- [146] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.
- [147] K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.
- [148] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1230–1239, 2006.
- [149] D. Salomon. *Data Compression: the Complete Reference, 3rd Edition*. Springer Verlag, 2004.
- [150] D. Salomon. *Variable-length Codes for Data Compression*. Springer, 2007.



- [151] E. J. Schuegraf and H. S. Heaps. A comparison of algorithms for data base compression by use of fragments as language elements. *Information Storage and Retrieval*, 10(9-10):309319, 1974.
- [152] D. Shkarin. PPM: One step to practicality. In *Proceedings of the IEEE Data Compression Conference (DCC)*, pages 202–211, 2002.
- [153] M. E. Gonzalez Smith and J. A. Storer. Parallel algorithms for data compression. *Journal of the ACM*, 32(2):344–373, 1985.
- [154] T. Suel and N. Memon. Algorithms for delta compression and remote file synchronization. In Khalid Sayood, editor, *Lossless Compression Handbook*. Academic Press, 2002.
- [155] E. Sutinen and J. Tarhio. Filtration with  $q$ -samples in approximate string matching. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS vol. 1075, pages 50–61. Springer, 1996.
- [156] W. Szpankowski. Asymptotic properties of data compression and suffix trees. *IEEE Transactions on Information Theory*, 39(5):1647–1659, 1993.
- [157] D. Trendafilov, N. Memon, and T. Suel. Compressing file collections with a TSP-based approach. Technical report, Technical Report TR-CIS-2004-02, Polytechnic University, 2004.
- [158] J. Ullman. A binary  $n$ -gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. *The Computer Journal*, 10:141–147, 1977.
- [159] J. S. Vitter and P. Krishnan. Optimal prefetching via data compression. *Journal of the ACM*, 43(5):771–793, 1996.
- [160] B.D. Vo and K.-P. Vo. Compressing table data with column dependency. *Theoretical Computer Science*, 387(3):273–283, 2007.
- [161] R. Wan and A. Moffat. Block merging for off-line compression. *Journal of the American Society for Information Science and Technology*, 58(1):3–14, 2007.
- [162] D. E. Willard. Examining computational geometry, Van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal on Computing*, 29(3), 2000.
- [163] H. E. Williams and J. Zobel. Indexing and retrieval for genomic databases. *IEEE Transaction on Knowledge and Data Engineering*, 14(1):63–78, 2002.

- 
- [164] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.
- [165] J. Ziv. Classification with finite memory revisited. *IEEE Transactions on Information Theory*, 53(12):4413–4421, 2007.
- [166] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transaction on Information Theory*, 23:337–343, 1977.
- [167] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Transaction on Information Theory*, 24:530–536, 1978.
- [168] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):6, 2006.