Università di Pisa
Dipartimento di Informatica

Technical Report: TR-02-01

On compressing and indexing data

Paolo Ferragina Giovanni Manzini

ADDRESS: Corso Italia 40, 56125 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

On compressing and indexing data*

Paolo Ferragina† Giovanni Manzini‡

Abstract

In this paper we design two compressed data structures for the full-text indexing problem. These data structures support efficient substring searches within the indexed text T using roughly the same space required to store T in compressed form.

Our first compressed data structure retrieves the occ occurrences of a pattern $P[1,p]$ in $T[1,n]$ in $O(p + occ \log^{1+\epsilon} n)$ time and uses at most $5nH_k(T) + o(n)$ bits of storage, where $H_k(T)$ is the k-th order empirical entropy of T. This space occupancy is $\Theta(n)$ bits in the worst case and $o(n)$ bits for compressible texts. Our data structure exploits the relationship between the suffix array [20] and the Burrows-Wheeler compression algorithm [7].

Our second compressed data structure achieves $O(p+occ)$ query time and uses $O(nH_k(T)\log^{\epsilon} n) + o(n)$ bits of storage. In the worst case the space occupancy is $o(n \log n)$ bits which is asymptotically smaller than the space occupancy of suffix trees and suffix arrays. This second data structure exploits the interplay between two compressors: the Burrows-Wheeler algorithm and the `LZ78` algorithm [32].

Keywords: D.4.2 [Operating System] Storage Management - Main memory; E.1 [Data Structures] Arrays, Tables; E.4 [Coding and Information Theory] Data compaction and compression; E.5 [Files] Sorting/searching; F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical algorithms and Problems - Pattern matching; H.3 [Information Storage and Retrieval] Content Analysis and Indexing, Information Storage, Information Search and Retrieval.

General Terms: Algorithms, Design, Theory.

Additional Keywords and Phrases: Full-text indexing, Text compression, Pattern searching, Indexing data structure, Suffix tree, Suffix array, Burrows-Wheeler transform, Lempel-Ziv algorithms.

1 Introduction

Data structure is a central concept in algorithmics and computer science in general. In the last decades data structures have been investigated from different points of view and they have been enriched by new functionalities; today we have data structures which are dynamic, persistent, self-adjusting, fault-tolerant, just to cite a few examples. Recently there has been an upsurging interest in the design of *succinct* (or *implicit*) data structures (see e.g. [6, 26, 27]). Succinct data structures try to reduce as much

*Part of this work has been published in the Proceedings of the 41st IEEE Symposium on the Foundations of Computer Science, Redondo Beach, CA, 2000. Section 5 reports some of the experimental results published in the Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms, Washington DC, 2001.

†Dipartimento di Informatica, Università di Pisa, Italy. E-mail: `ferragin@di.unipi.it`. Partially supported by Italian MIUR project on "Technologies and services for enhanced content delivery", and by Italian MURST project "Algorithms for Large Data Sets: Science and Engineering".

‡Dipartimento di Scienze e Tecnologie Avanzate, Università del Piemonte Orientale, Alessandria, Italy and IMC-CNR, Pisa, Italy. E-mail `manzini@mfn.unipmn.it`. Partially supported by Italian MIUR project on "Technologies and services for enhanced content delivery".

as possible the auxiliary information stored into them without introducing any significant slowdown in the supported operations. The interest in these data structures is mainly due to the fact that processing speed is currently improving at a faster rate than disk and memory access times. This technological trend has turned space reduction into an attractive choice not only for storage saving but also for its favorable impact on algorithmic performance.

In this paper we address the issue of compressing and indexing data by studying it in a theoretical framework. We consider the following problem, classically called the *full-text indexing* problem. Given a text $T[1,n]$, drawn from a constant size alphabet Σ, the aim is to preprocess T and build an indexing data structure that occupies *small space* and supports *efficient searches* for the occurrences of an arbitrary pattern $P[1,p]$ as a substring of T. Since we are interested in small space occupancy it is natural to ask how much we can hope to squeeze a full-text index. An easy argument shows that a full-text index contains enough information to retrieve the indexed text. Hence, a natural lower bound to the size of the index is provided by the *entropy* of T. In this paper we perform the first step toward reaching this information theoretic limit by designing two compressed full-text indexes supporting pattern searches in $O(p + occ\, polylog(n))$ time and with a space occupancy bounded in terms of the *empirical entropy* of T. The model of computation is a RAM with word size $\Theta(\log n)$ bits.

Our first contribution is a data structure which indexes a text $T[1,n]$ using at most $5nH_k(T) + o(n)$ bits of storage, where $H_k(T)$ is the k-th order empirical entropy of T. This space bound holds for any $k \geq 0$. Given a pattern $P[1,p]$, our data structure allows to count the number *occ* of pattern occurrences in $O(p)$ time and allows to retrieve their positions in T in $O(p + occ \log^{1+\epsilon} n)$ time, where $\epsilon > 0$ is an arbitrary constant chosen when we build the data structure. Since the empirical entropy $H_k(T)$ is at most $\log|\Sigma|$, in the worst case our data structure occupies $\Theta(n)$ bits. However, our data structure can take advantage of the compressibility of T in order to achieve $o(n)$ space occupancy. For example, for $T = ab^{n-1}$ our data structure occupies $o(n)$ bits since $nH_k(T) = O(\log n)$.

Our indexing data structure exploits the relationship between the Burrows-Wheeler compression algorithm [7] and the suffix array [20]. The key idea is to add some auxiliary information to the compressed text, obtained via the Burrows-Wheeler algorithm, to turn it into a full-text index which supports arbitrary and efficient pattern searches. In some sense our data structure is a sort of *compressed suffix array* which encapsulates both the compressed text and the full-text indexing information.

In [13, Theorem 3] we showed how to modify our indexing data structure in order to achieve $O(p + occ \log^{\epsilon} n)$ query time using $O(nH_k(T)) + o(n)$ bits of storage. We decided not to include in this paper the description of the modified data structure because it is rather technical and does not introduce new algorithmic ideas. Moreover, in the second part of the paper we make a step further by designing a compressed index with a query time which does not increase with the length of the indexed text. We achieve this goal by exploiting the interplay between *two compressors*: the Burrows-Wheeler algorithm and the LZ78 algorithm [32]. The key idea here is to exploit text regularities not only for space reduction but also for speeding up the retrieval of the pattern occurrences. The resulting data structure achieves $O(p + occ)$ query time and occupies $O(nH_k(T) \log^{\epsilon} n) + o(n)$ bits. Note that if the text is highly compressible the space occupancy is $o(n)$ bits. Moreover, this data structure has some merit even for uncompressible texts: in the worst case it occupies $O(n \log^{\epsilon} n)$ bits and is therefore the first full-text index in the literature using $o(n \log n)$ bits of storage and achieving $O(p + occ)$ query time without any restrictions on p and *occ* (more on this in Section 1.1).

The results of this paper are theoretical in their flavor. However, some of the proposed data structures have been implemented and extensively tested in [14, 15]. The experimental results show that these data structures store both the *input text* and its *full-text index* using roughly the same space required by traditional compressors—such as gzip and bzip2—which only store the input text. Our data structures retrieve the occurrences of an arbitrary pattern in a few milliseconds, even within texts of several megabytes. This set of experiments suggests that the results of this paper might be the starting point for the design of a new family of text retrieval tools of great practical interest.

REMARK. Most of the bounds in this paper contain the parameters ϵ and k. The parameter ϵ denotes an arbitrarily small positive constant which is chosen when we build our data structures. The parameter ϵ influences both the construction of the data structures and the constants hidden in the big-O notation. Conversely, the parameter k is a positive integer that does not influence in any way the construction and use of our data structures. Nonetheless, their space occupancy can be bounded in terms of $H_k(T)$, for any $k \geq 0$. This is of course a strong property since as k increases $H_k(T)$ is a better and better approximation of the information content of text T [30]. For a discussion on $H_k(T)$ see Section 2.2 and Appendix A.

1.1 Related results

The best known data structures for the full-text indexing problem are suffix trees [23] and suffix arrays [20]. On a RAM with word-size $\Theta(\log n)$ bits, they support the pattern search in $O(\frac{p}{\log n} + occ)$ time (resp. $O(\frac{p}{\log n} + \log n + occ)$ time) and occupy $\Theta(n \log n)$ bits of storage.

Grossi and Vitter [16] were the first to break the $\Theta(n \log n)$ space barrier by proposing a full-text index with a space occupancy of $\Theta(n)$ bits and query time $O(\frac{p}{\log n} + occ \log^\epsilon n)$. More recently, another full-text index with $\Theta(n)$ space occupancy and query time $O(p \log n + occ \log^\epsilon n)$ has been proposed by Sadakane [29]. Although Sadakane's index has a higher query time, it reduces the constants hidden in the $\Theta(n)$ space bound when indexing texts defined over large alphabets.

Our first indexing data structure takes advantage of the text compressibility to break the space barrier of $\Theta(n)$ bits. Indeed it answers pattern queries in $O(p + occ \log^{1+\epsilon} n)$ time and occupies $5nH_k(T) + o(n)$ bits of storage. This is a first step toward reaching the information theoretic limit given by the entropy of the indexed text for the full-text indexing problem.

In most applications the number of occurrences occ is much larger than the pattern length p. Hence, if one is mainly interested in query time efficiency suffix trees and suffix arrays still have some merit. Grossi and Vitter extended their data structure to achieve $O(\frac{p}{\log n} + occ)$ query time when $p = \Omega(\log^3 n \log\log n)$ or $occ = \Omega(n^\epsilon)$. Our second indexing data structure guarantees $O(p + occ)$ query time independently of p and occ, and occupies $O(nH_k(T)\log^\epsilon n) + o(n)$ bits of storage. This space bound is still $o(n)$ bits for very compressible texts and is $O(n \log^\epsilon n)$ bits in the worst case. Note that this worst case space occupancy is asymptotically less than the space used by the suffix tree and suffix array.

As a final remark, we observe that the impact of compression on algorithmic performance has been investigated in many other contexts. A non exhaustive list includes: string-matching algorithms [2, 3, 9], sorting and computational geometry algorithms [8], and on-line algorithms [17, 19].

1.2 Paper structure

Section 2 contains some background material. In Section 2.1 we fix our notation. In Section 2.2 we show that the entropy of the indexed text provides a natural lower bound to the size of a full-text index. In Section 2.3 we describe the Burrows-Wheeler Transform and we review its basic properties. In Section 2.4 we describe a compression algorithm, called `BW_RLX`, which is based on the Burrows-Wheeler Transform. All of our indexing data structures consist of the compressed file Z = `BW_RLX`(T) (the output of `BW_RLX` with input T) together with some auxiliary indexing information.

Sections 3 and 4 contain the main results of the paper. In Section 3 we describe an indexing data structure which achieves a query time of $O(p + occ \log^{1+\epsilon} n)$ using at most $5nH_k(T) + o(n)$ bits of storage (Theorem 3). This index has a relatively simple structure: it consists of the compressed file Z together with two additional data structures. The first one is used to count the number of pattern occurrences, the second one is used to find the position in T of these occurrences. In Section 4 we describe a different approach to the problem of designing a compressed index. The new data structure exploits the Burrows-Wheeler Transform and the `LZ78` parsing of the input string, and achieves a query time of $O(p + occ)$ using $O(nH_k(T)\log^{\epsilon} n) + o(n)$ bits of storage (Theorem 9).

In Section 5 we briefly discuss the suitability of our compressed indexes for real-world applications, and in Section 6 we draw the conclusions of the paper. In the appendices we have confined the proof of some technical lemmas.

2 Background

In this section we fix our notation and review some known results which are used in the rest of the paper.

2.1 Basic terminology and notation

Hereafter we assume that $T[1, n]$ is the text we wish to index, compress and query. T is drawn from a constant size alphabet Σ. By $T[i]$ we denote the i-th character of T, $T[i, n]$ denotes the text suffix of length $(n - i + 1)$, and $T[1, i]$ denotes the text prefix of length i. We write $|A|$ to denote the number of elements in the set A, and we write $|w|$ to denote the length of string w.

In this paper we are interested in solving the following problem, also known as the *full-text indexing* problem. Given the text T we aim at building an indexing data structure (also called an *index*) that allows to search for the occurrences of an arbitrary pattern $P[1, p]$ as a *substring* of T. Pattern P is provided on-line whereas the text T is given to be preprocessed in advance. The number of pattern occurrences in T will be hereafter indicated with the parameter *occ*. The above index is called a *full-text index* to highlight the fact that its *substring* search operation is more powerful than the common search for a word, or for a prefix of word, usually supported by traditional *word-based* indexes [31].

In this paper we split the pattern searching process in two phases: the *counting* of the number of pattern occurrences, and the *location* (or *retrieval*) of their positions in the text T. Note that the counting process returns the single value *occ*, whereas the location process returns *occ* distinct values in the range $[1, n]$.

In the following we assume that the model of computation is a RAM with word size $\Theta(\log n)$ bits. In this model arithmetic as well as shift operations between memory words require $O(1)$ time.

2.2 Indexing and entropy

Assume we are given a full-text index $\mathcal{I}$ for a string T defined over the alphabet Σ. We can use $\mathcal{I}$ to count the number of occurrences in T of all strings in Σ^i for $i = 1, 2, \ldots$. It is easy to see that this procedure eventually leads to the identification of the text T which is indexed by $\mathcal{I}$. In other words, $\mathcal{I}$ contains enough information to retrieve T. This observation suggests that a lower bound to the size of $\mathcal{I}$ is given by the amount of data required to represent T. Following a well established practice in information theory we measure this latter quantity using the notion of *empirical entropy*. The empirical entropy is similar to the entropy defined in the probabilistic setting with the difference that it is defined in terms of the symbol frequencies observed in T rather than in terms of symbol probabilities (see Appendix A for definitions and basic properties).

The k-th order empirical entropy $H_k(T)$ is a lower bound to the output size of any compressor which encodes each character of $T[1, n]$ using a uniquely decipherable code that depends only on the character itself and on the k characters preceding it. For any $k \geq 0$ the value $H_k(T)$ is bounded above by the logarithm of the alphabet size. We point out that for strings with many regularities the value $H_k(T)$ can be $o(1)$, unlike the entropy defined in the probabilistic setting which is always a constant. As an example, for $T = (ab)^{n/2}$ we have $H_0(T) = 1$ and $H_k(T) = O((\log n)/n)$ for any $k \geq 1$.

		F		L
mississippi#		#	mississipp	i
ississippi#m		i	#mississip	p
ssissippi#mi		i	ppi#missis	s
sissippi#mis		i	ssippi#mis	s
issippi#miss		i	ssissippi#	m
ssippi#missi	$\Longrightarrow$	m	ississippi	#
sippi#missis		p	i#mississi	p
ippi#mississ		p	pi#mississ	i
ppi#mississi		s	ippi#missi	s
pi#mississip		s	issippi#mi	s
i#mississipp		s	sippi#miss	i
#mississippi		s	sissippi#m	i

Figure 1: Example of Burrows-Wheeler transform for the string $T = \texttt{mississippi}$. The matrix on the right has the rows sorted in lexicographic order. The output of the BWT is column L; in this example the string `ipssm#pissii`.

2.3 The Burrows-Wheeler transform

In [7] 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 text T into a new string which contains the same characters of T but it is usually easier to compress. The BWT consists of three basic steps (see Fig. 1): (1) append at the end of T a special character # smaller than any other text character; (2) form a *conceptual* matrix $\mathcal{M}_T$ whose rows are the cyclic shifts of the string $T\#$ sorted in lexicographic order; (3) construct the transformed text L by taking the last column of matrix $\mathcal{M}_T$. Notice that every column of $\mathcal{M}_T$, hence also the transformed text L, is a permutation of $T\#$. In particular the first column of $\mathcal{M}_T$, call it F, is obtained by lexicographically sorting the characters of $T\#$ (or, equally, the characters of L).

Note that when we sort the rows of $\mathcal{M}_T$ we are essentially sorting the suffixes of T because of the presence of the special character #. Therefore there is a strong relation between the matrix $\mathcal{M}_T$ and the suffix array [20] built on T. The matrix $\mathcal{M}_T$ has also other remarkable properties; to illustrate them we introduce the following notation:

- $C[\cdot]$ denotes the array of length $|\Sigma|$ such that $C[c]$ contains the total number of text characters which are alphabetically smaller than c.
- $\mathsf{Occ}(c, q)$ denotes the number of occurrences of character c in the prefix $L[1, q]$ of the transformed text L.

As an example, in Fig. 1 we have $C[\mathtt{s}] = 8$ and $\mathsf{Occ}(\mathtt{s}, 10) = 4$. The following properties of $\mathcal{M}_T$ have been proven in [7]:

a. Given the i-th row of $\mathcal{M}_T$, its last character $L[i]$ precedes its first character $F[i]$ in the original text T.

b. Let us define $LF(i) = C[L[i]] + \mathsf{Occ}(L[i], i)$. $LF(\cdot)$ stands for *Last-to-First column mapping* since the character $L[i]$ is located in F at position $LF(i)$. For example in Fig. 1 we have $LF(10) = C[\mathtt{s}] + \mathsf{Occ}(\mathtt{s}, 10) = 12$; and in fact $L[10]$ and $F[LF(10)] = F[12]$ both correspond to the first `s` in the string `mississippi`.

c. If $T[k]$ is the i-th character of L then $T[k-1] = L[LF(i)]$. For example in Fig. 1 we have that $T[3] = \mathtt{s}$ is the 10th character of L and we correctly have $T[2] = L[LF(10)] = L[12] = \mathtt{i}$.

Property **c.** makes it possible to reconstruct T backwards as follows. Initially set $i = 1$ and $T[n] = L[1]$ (since the first row of $\mathcal{M}_T$ is $\#T$). Then, for $k = n, \ldots, 2$ set $T[k-1] = L[LF(i)]$ and $i = LF(i)$. For example, to reconstruct the text $T = \texttt{mississippi}$ of Fig. 1 we start by setting $T[11] = L[1] = \texttt{i}$. At the first iteration (i.e. $k = 11$ and $i = 1$), we get $T[10] = L[LF(1)] = L[C[\texttt{i}] + \mathsf{Occ}(\texttt{i}, 1)] = L[2] = \texttt{p}$. At the second iteration (i.e. $k = 10$ and $i = LF(1) = 2$), we get $T[9] = L[LF(2)] = L[C[\texttt{p}] + \mathsf{Occ}(\texttt{p}, 2)] = L[7] = \texttt{p}$, and so on.

2.4 A BWT-based compression algorithm

The output of the BWT is the last column L of the matrix $\mathcal{M}_T$ (we write $L = \texttt{bwt}(T)$). The BWT by itself is not a compression algorithm since L is just a permutation of $T\#$. However, if T has some regularities the BWT will "group together" several occurrences of the same character. As a result, the transformed string L contains long runs of identical characters and turns out to be highly compressible (see [7, 10, 22] for details).

Several compression algorithms based on the BWT have been proposed since its introduction. In order to describe our results we will commit ourselves to a specific algorithm—called `BW_RLX`—which processes the string $L = \texttt{bwt}(T)$ as follows (see also [22, Section 5]).

1. Use a move-to-front encoder [5] (shortly `mtf`) to encode each character of L with the number of distinct characters seen since its previous occurrence. The encoder maintains a list, called the MTF-list, initialized with all characters of Σ ordered alphabetically. When the next character arrives, the encoder outputs its current rank in the MTF-list and moves it to the front of the list. At any instant, the MTF-list contains the characters ordered by recency of occurrence. Note that runs of identical characters in L are transformed to runs of zeroes in the resulting string $L^{mtf} = \texttt{mtf}(L)$. The string L^{mtf} is defined over the alphabet $\{0, 1, 2, \ldots, |\Sigma| - 1\}$.
2. Encode each run of zeroes in L^{mtf} using a run-length encoder (shortly `rle`). More precisely, replace any sequence 0^m with the number $(m + 1)$ written in binary, least significant bit first, discarding the most significant bit. For this encoding we use two new symbols $\mathbf{0}$ and $\mathbf{1}$. For example, 0^5 is encoded with $\mathbf{01}$, and 0^7 is encoded with $\mathbf{000}$. Note that the resulting string $L^{rle} = \texttt{rle}(L^{mtf})$ is defined over the alphabet $\{\mathbf{0}, \mathbf{1}, 1, 2, \ldots, |\Sigma| - 1\}$.
3. Compress L^{rle} by means of a variable-length prefix code (shortly `PC`) defined as follows. The symbols $\mathbf{0}$ and $\mathbf{1}$ are encoded using two bits (10 for $\mathbf{0}$, 11 for $\mathbf{1}$). For $i = 1, 2, \ldots, |\Sigma| - 1$, the symbol i is encoded using $1 + 2\lfloor \log(i + 1) \rfloor$ bits: $\lfloor \log(i + 1) \rfloor$ 0's followed by the binary representation of $i + 1$ which takes $1 + \lfloor \log(i + 1) \rfloor$ bits. The string $\texttt{PC}(L^{rle})$—which is the output of algorithm `BW_RLX`—is defined over the alphabet $\{0, 1\}$.

The algorithm `BW_RLX` is sufficiently simple so that in the following we can concentrate on the indexing problem without being distracted by the details of the compression process. Despite of its simplicity, the compression ratio of `BW_RLX` can be bounded in terms of the entropy of the input text. More precisely, in Appendix B we prove that for any $k \geq 0$ there exists a constant t_k such that for any text $T[1, n]$

$$|\texttt{BW\_RLX}(T)| \leq 5\, n\, H_k(T) + t_k \log n \tag{1}$$

where $H_k(T)$ is the k-th order empirical entropy of the text T. The possibility of bounding the size of $\texttt{BW\_RLX}(T)$ in terms of $H_k(T)$ for any $k \geq 0$, is a strong property of this algorithm and shows the effectiveness of the BWT for data compression. It is this property of `BW_RLX` which makes it possible to bound the size of our indexing data structures in terms of $H_k(T)$ for any $k \geq 0$. It goes without saying that any improvement to the constants of (1) will lead to an improvement to the space occupancy bounds of our data structures.

Algorithm get_rows($P[1,p]$)

1. $i \leftarrow p$, $c \leftarrow P[p]$, $\mathsf{First} \leftarrow C[c]+1$, $\mathsf{Last} \leftarrow C[c+1]$;
2. **while** (($\mathsf{First} \leq \mathsf{Last}$) **and** ($i \geq 2$)) **do**
3. $\quad c \leftarrow P[i-1]$;
4. $\quad \mathsf{First} \leftarrow C[c] + \mathsf{Occ}(c, \mathsf{First}-1)+1$;
5. $\quad \mathsf{Last} \leftarrow C[c] + \mathsf{Occ}(c, \mathsf{Last})$;
6. $\quad i \leftarrow i-1$;
7. **if** ($\mathsf{Last} < \mathsf{First}$) **then return** "no rows prefixed by $P[1,p]$" **else return** ($\mathsf{First}, \mathsf{Last}$).

Figure 2: Algorithm get_rows for finding the set of rows prefixed by$P[1,p]$ in $T[1,n]$. Recall that $C[c]$ is the number of text characters which are alphabetically smaller than c, and that $\mathsf{Occ}(c,q)$ denotes the number of occurrences of character c in $L[1,q]$ (see Section 2.3).

3 A compressed full-text index

We distinguish two phases in the pattern searching process: the counting of the pattern occurrences (Section 3.1), and the location of their positions in T (Section 3.2). This distinction has a twofold advantage: it simplifies the presentation and points out an important *decomposability* property of our index: the location phase builds on the top of the counting phase. Hence, the additional data structures needed by the pattern location process may be dropped if we are only interested in simpler queries like, e.g., "Does pattern P occur in T?" or "How many times does pattern P occur in T?".

3.1 Counting the pattern occurrences

In this section we design a data structure for counting the number of occurrences of a pattern $P[1,p]$ in $T[1,n]$. It consists of two basic components: the string $Z = \mathtt{BW\_RLX}(T)$ (output of `BW_RLX` on input T) and an auxiliary data structure supporting the constant-time computation of the value $\mathsf{Occ}(c,q)$ defined in Section 2.3.

We exploit two structural properties of the matrix $\mathcal{M}_T$ described in Section 2.3: (i) all suffixes of T prefixed by the searched pattern P occupy a contiguous set of rows of $\mathcal{M}_T$; (ii) this set of rows has starting position First and ending position Last, where First is the *lexicographic position* of the pattern P among the ordered rows of $\mathcal{M}_T$. Clearly, the value ($\mathsf{Last} - \mathsf{First} + 1$) accounts for the total number of pattern occurrences. For example, in Fig. 1 for the pattern $P = \mathtt{si}$ we have $\mathsf{First} = 9$ and $\mathsf{Last} = 10$ for a total of two occurrences.

The retrieval of the rows First and Last is implemented by the procedure get_rows which works in p phases numbered from p to 1 (see the pseudo-code in Fig. 2). Each phase preserves the following invariant: *At the i-th phase, the parameter* First *points to the first row of $\mathcal{M}_T$ prefixed by $P[i,p]$ and the parameter* Last *points to the last row of $\mathcal{M}_T$ prefixed by $P[i,p]$*. We start with $i = p$ (Step 1), so that First and Last can be determined via the array C defined in Section 2.3. After the final phase, First and Last will delimit the rows of $\mathcal{M}_T$ containing all the text suffixes prefixed by P. The following lemma formally proves the correctness of get_rows.

Lemma 1 *For $i = p, p-1, \ldots, 2$, let us assume that* First *(resp.* Last*) stores the position of the first (resp. last) row in $\mathcal{M}_T$ prefixed by $P[i,p]$. If $P[i-1,p]$ occurs in T, then Step 4 (resp. Step 5) of* get_rows *correctly updates the value of* First *(resp.* Last*) thus pointing to the first (resp. last) row prefixed by*

$P[i-1,p]$. *If* $P[i-1,p]$ *does not occur in* T*, after Step 5 we have* Last $<$ First *and* get_rows *correctly reports that* P *does not occur in* T*.*

Proof: We only consider the computation of First since the argument for Last is analogous. Let us inductively assume that First points to the first row of $\mathcal{M}_T$ prefixed by the string $P[i,p]$; the base case (i.e. $i=p$) is handled by Step 1 whose correctness follows from the definition of the array C.

We first consider the case that $P[i-1,p]$ does occur in T. Let $\mathcal{M}_T[s]$ be the *first* row in $\mathcal{M}_T$ prefixed by $P[i-1,p]$. We have $\mathcal{M}_T[s] = cP[i,p]w$ where $c = P[i-1]$ and w is an arbitrary string over $\Sigma \cup \{\#\}$. At the next iteration First should point to row s. Hence, to prove the correctness of Step 4 we need to show that $s = C[c] + \mathsf{Occ}(c, \mathsf{First}-1) + 1$.

We rotate cyclically to the left the row $\mathcal{M}_T[s]$ thus obtaining the row $\mathcal{M}_T[s'] = P[i,p]wc$, for a proper row index s'. By hypothesis, First is the first row of $\mathcal{M}_T$ prefixed by $P[i,p]$ and thus $\mathsf{First} \le s'$. We now show that s' is indeed the first row which follows the position First and ends with the character c (i.e. $L[s'] = c$ and $L[j] \ne c$ for $j = \mathsf{First}, \ldots, s'-1$). If this were not the case, we would have a row $\mathcal{M}_T[t']$ occurring between $\mathcal{M}_T[\mathsf{First}]$ and $\mathcal{M}_T[s']$ and ending with c. $\mathcal{M}_T[t']$ would have the form $P[i,p]w'c$ because the rows between $\mathcal{M}_T[\mathsf{First}]$ and $\mathcal{M}_T[s']$ are all prefixed by $P[i,p]$. Since $\mathcal{M}_T[t']$ would precede lexicographically $\mathcal{M}_T[s']$, w' would be lexicographically smaller than w. But then, the row $cP[i,p]w'$ would exist in $\mathcal{M}_T$ and should occur before $\mathcal{M}_T[s] = cP[i,p]w$, thus contradicting the definition of s.

By property **b.** of Section 2.3, we know that $s = LF(s')$. That is, $s = C[c] + \mathsf{Occ}(c, s')$. Since s' is the first row following First and ending with c, we have $s = C[c] + \mathsf{Occ}(c, \mathsf{First}-1) + 1$ as claimed.

To complete the proof we consider the case in which $P[i-1,p]$ does not occur in T. Since the rows between $\mathcal{M}_T[\mathsf{First}]$ and $\mathcal{M}_T[\mathsf{Last}]$ are all prefixed by $P[i,p]$ the character $c = P[i-1]$ does not appear at the end of these rows and thus it does not occur in the substring $L[\mathsf{First}, \mathsf{Last}]$. Hence, $\mathsf{Occ}(c, \mathsf{First}-1) = \mathsf{Occ}(c, \mathsf{Last})$ and after Step 5 we then have Last $<$ First as claimed. ■

It is easy to see that the running time of get_rows is dominated by the cost of the $2p$ computations of the values Occ( ). We now show how to compute $\mathsf{Occ}(c,q)$ in $O(1)$ time for any $c \in \Sigma \cup \{\#\}$ and $q \in [1, n+1]$; this implies that get_rows runs in $O(p)$ time. We compute $\mathsf{Occ}(c,q)$ using a data structure which follows, in its main ideas, the table compression scheme of [25] adapted here to work with the compressed text $Z = \mathtt{BW\_RLX}(T)$.

We logically partition L into substrings of $\ell = \Theta(\log n)$ characters[1] each, called *buckets*, and denote them by $BL_i = L[(i-1)\ell+1, i\ell]$, for $i = 1, \ldots, n/\ell$. This partition naturally induces a partition of L^{mtf} into n/ℓ buckets $BL_1^{mtf}, \ldots, BL_{n/\ell}^{mtf}$ of size ℓ too. We assume that each run of zeroes in L^{mtf} is entirely contained in a single bucket and we describe our algorithm for computing $\mathsf{Occ}(c,q)$ under this simplifying assumption. The general case in which a sequence of zeroes may span several buckets is discussed in Appendix C. Under our assumption, the buckets BL_i^{mtf}'s induce a partition of the compressed file Z into n/ℓ *compressed buckets* of variable length $BZ_1, \ldots, BZ_{n/\ell}$, defined as $BZ_i = \mathtt{PC}(\mathtt{rle}(BL_i^{mtf}))$.

The computation of $\mathsf{Occ}(c,q)$ is based on a decomposition of $L[1,q]$ into three substrings defined as follows:

(i) the longest prefix of $L[1,q]$ having length a multiple of ℓ^2;

(ii) the longest prefix of the remaining suffix having length a multiple of ℓ;

(iii) the remaining suffix of $L[1,q]$ which is indeed a prefix of the bucket BL_i, where $i = \lceil q/\ell \rceil$. Note that BL_i is the bucket containing the character $L[q]$.

[1] Since L is a permutation of $T\#$ it has length $n+1$ and is defined over an alphabet of size $|\Sigma|+1$. However, to simplify the notation, in the following we will ignore this fact and pretend that L has length n and is defined over an alphabet of size $|\Sigma|$. In addition, to avoid the use of ceilings and floors we will assume that n is a multiple of ℓ^2.

We compute $\mathsf{Occ}(c, q)$ by summing the number of occurrences of c in each one of these three substrings. As an example, if $\ell = 20$, we compute $\mathsf{Occ}(c, 1393)$ using the decomposition (i) $L[1, 1200]$, (ii) $L[1201, 1380]$, and (iii) $L[1381, 1393]$. Note that $L[1381, 1393]$ is a prefix of BL_{70} which is the bucket containing the character $L[1393]$.

The computation of the number of occurrences of c into these three substrings can be done in constant time using some auxiliary data structures which are detailed below.

- For the calculations on the substring of point (i) we build the following two data structures.
  - ◇ For $j = 1, \ldots, n/\ell^2$, the array $NO_j[1, |\Sigma|]$ stores in entry $NO_j[c]$ the number of occurrences of c in $L[1, j\ell^2]$.
  - ◇ The array $W[1, n/\ell^2]$ stores in entry $W[j]$ the value $\sum_{h=1}^{j\ell} |BZ_h|$ which is the sum of the sizes of the compressed buckets $BZ_1, \ldots, BZ_{j\ell}$. Note that $BZ_1, \ldots, BZ_{j\ell}$ is the portion of the compressed file Z corresponding to $L[1, j\ell^2]$, that is, $BZ_1 \cdots BZ_{j\ell} = \mathsf{PC}(\mathtt{rle}(\mathtt{mtf}(L[1, j\ell^2])))$.
- For the calculations on the substring of point (ii) we build the following two data structures.
  - ◇ For $j = 1, \ldots, n/\ell$, the array $NO'_j[1, |\Sigma|]$ stores in entry $NO'_j[c]$ the number of occurrences of character c in the string $L[\hat{\jmath}\ell^2 + 1, j\ell]$, where $\hat{\jmath} = \lceil j/\ell \rceil - 1$. In other words, $NO'_j[1, |\Sigma|]$ stores the number of occurrences of each character between position $j\ell$ and the closest position on its left being a multiple of ℓ^2. Note that the string $L[\hat{\jmath}\ell^2 + 1, j\ell]$ has length at most ℓ^2.
  - ◇ The array $W'[1, n/\ell]$ stores in entry $W'[j]$ the value $\sum_{h=\hat{\jmath}\ell+1}^{j} |BZ_j|$ which is the overall size of the compressed buckets $BZ_{\hat{\jmath}\ell+1}, \ldots, BZ_j$ with $\hat{\jmath} = \lceil j/\ell \rceil - 1$. Note that $BZ_{\hat{\jmath}\ell+1} \cdots BZ_j$ is the portion of the compressed file corresponding to $L[\hat{\jmath}\ell^2 + 1, j\ell]$.
- For the calculations the substring of point (iii) we build the following two data structures.
  - ◇ The array $MTF[1, n/\ell]$ stores in entry $MTF[j]$ a picture of the state of the MTF-list at the beginning of the encoding of BL_j. Note that each picture consists of $|\Sigma| \log |\Sigma|$ bits.
  - ◇ Table S stores in the entry $S[c, h, BZ_j, MTF[j]]$ the number of occurrences of the character c among the first h characters of BL_j, clearly $h \leq \ell$. This is possible since BZ_j and $MTF[j]$ completely determine BL_j. We wrote $S[c, h, BZ_j, MTF[j]]$ to make clear how we use table S. However, we emphasize that for each pair (c, h) table S contains an entry for *each* possible compressed bucket and for *each* possible picture of the state of a MTF-list. Thus, if a compressed bucket has size at most ℓ' bits, table S has overall $|\Sigma| \cdot \ell \cdot 2^{\ell'} \cdot 2^{|\Sigma| \log |\Sigma|}$ entries.

Given the above data structures we design an algorithm that computes $\mathsf{Occ}(c, q)$ using the decomposition of $L[1, q]$ described in points (i)–(iii) above. The algorithm works as follows. Initially, the bucket BL_i containing the character $c = L[q]$ is determined, by setting $i = \lceil q/\ell \rceil$, together with the position $j = q - (i-1)\ell$ of this character within BL_i and the parameter $\hat{\imath} = \lceil i/\ell \rceil - 1$. The number of occurrences of c in the prefix $L[1, \hat{\imath}\ell^2]$ (point (i)) is given by $NO_{\hat{\imath}}[c]$. Subsequently, the number of occurrences of c in the substring $L[\hat{\imath}\ell^2 + 1, (i-1)\ell]$ (point (ii)) is determined: this number is zero if $(i-1)$ is a multiple of ℓ, otherwise this number is $NO'_{i-1}[c]$. Finally, the compressed bucket BZ_i is retrieved from Z: its starting bit position in Z is $W[\hat{\imath}] + 1$ if $(i-1)$ is a multiple of ℓ, otherwise it is $W[\hat{\imath}] + W'[i-1] + 1$. Given BZ_i we can retrieve the number of occurrences of c within $L[(i-1)\ell + 1, q] = BL_i[1, j]$ (point (iii)). This number is simply the value stored in the entry $S[c, j, BZ_i, MTF[i]]$. Summing up, we have the following result.

Lemma 2 *Let Z denote the output of the algorithm* $\mathtt{BW\_RLX}$ *on input $T[1, n]$. The value* $\mathsf{Occ}(c, q)$ *can be computed in $O(1)$ time using $|Z| + O\left(n \frac{\log \log n}{\log n}\right)$ bits of storage.*

Proof: Since $BZ_i = \texttt{PC}(\texttt{rle}(BL_i^{mtf}))$ and $|BL_i^{mtf}| = \ell$, an easy calculation shows that each compressed bucket has size at most $\ell' = (1 + 2\lfloor \log |\Sigma| \rfloor)\ell$ bits. We choose the parameter ℓ so that $\ell = \Theta(\log n)$ and $\ell' = \gamma \log n$ with $\gamma < 1$. Under this assumption, every step of the algorithm computing the value $\mathsf{Occ}(c, q)$ consists of arithmetic operations or table lookups involving $O(\log n)$-bit operands. Consequently the computation of $\mathsf{Occ}(c, q)$ takes constant time.

For what concerns the space occupancy, the arrays NO and W take $O((n/\ell^2)\log n) = O(n/\log n)$ bits. The arrays NO' and W' take $O((n/\ell)\log(\ell^2)) = O((n/\log n)\log\log n)$ bits. The array MTF takes overall $O((n/\ell)|\Sigma|\log|\Sigma|) = O(n/\log n)$ bits. Table S consists of $O(\ell 2^{\ell'})$ $\log\ell$-bit entries, and thus it occupies $O(2^{\ell'}\ell\log\ell) = O(n^\gamma \log n \log\log n)$ bits, where $\gamma < 1$. We conclude that these auxiliary data structures occupy overall $O((n\log\log n)/\log n)$ bits. ∎

Combining Lemma 2 with Lemma 1 and inequality (1) we can bound the running time and space occupancy of $\mathsf{get\_rows}$.

Theorem 1 *Algorithm* $\mathsf{get\_rows}$ *computes the number of occurrences of a pattern* $P[1,p]$ *in* $T[1,n]$ *in* $O(p)$ *time using space bounded by* $5nH_k(T) + O\left(n\,\frac{\log\log n}{\log n}\right)$ *bits, for any* $k \geq 0$. ∎

The collection of data structures described in this section will play such an important role in the rest of the paper that we introduce a special notation for it. For any string T we use $\mathsf{Opp}(T)$ to denote the data structures used by algorithm $\mathsf{get\_rows}$.[2] For future reference we summarize the results of this section in terms of the data structure $\mathsf{Opp}(T)$.

Theorem 2 *For any string* T *let* $\mathsf{Opp}(T)$ *denote the collection of data structures used by* $\mathsf{get\_rows}$. *Using* $\mathsf{Opp}(T)$ *we can determine the rows of* $\mathcal{M}_T$ *prefixed by any pattern* $P[1,p]$ *in* $O(p)$ *time. The size of* $\mathsf{Opp}(T)$ *is bounded by* $5nH_k(T) + O\left(n\,\frac{\log\log n}{\log n}\right)$ *bits, for any* $k \geq 0$. ∎

3.2 Locating the pattern occurrences

Assume that using algorithm $\mathsf{get\_rows}$ we have determined the range $(\mathsf{First}, \mathsf{Last})$ of rows of the matrix $\mathcal{M}_T$ which are prefixed by the pattern P. We now consider the problem of retrieving the positions in T of these $(\mathsf{Last} - \mathsf{First} + 1)$ pattern occurrences. This means that for $i = \mathsf{First}, \mathsf{First}+1, \ldots, \mathsf{Last}$ we want to find the position in T of the suffix which prefixes the i-th row $\mathcal{M}_T[i]$. In the following we write $\mathsf{Pos}(i)$ to denote such a suffix position. For example, in Fig. 1 we have $\mathsf{Pos}(4) = 5$ because the fourth row is prefixed by the text suffix $T[5, 11] = \texttt{issippi}$.

A fundamental tool for locating the occurrences of P is the following lemma which states that, given a row index i, using $\mathsf{Opp}(T)$ we can compute in $O(1)$ time the row index j such that $\mathsf{Pos}(j) = \mathsf{Pos}(i) - 1$. In other words, even if we do not know $\mathsf{Pos}(i)$, we can "jump" from the row prefixed by $T[\mathsf{Pos}(i), n]$ to the row prefixed by $T[\mathsf{Pos}(i) - 1, n]$. Since, in some sense, we are moving backwards in T the algorithm doing this computation is called $\mathsf{backward\_step}$.

Lemma 3 *Given a row index* i, *algorithm* $\mathsf{backward\_step}$ *either establishes that* $\mathsf{Pos}(i) = 1$ *or returns the row index* j *such that* $\mathsf{Pos}(j) = \mathsf{Pos}(i) - 1$. *Algorithm* $\mathsf{backward\_step}$ *uses the data structure* $\mathsf{Opp}(T)$ *and runs in* $O(1)$ *time.*

---

[2] Opp stands for opportunistic. The concept of *opportunistic algorithm* has been introduced in [9] to denote an algorithm which takes advantage of the compressibility of the text to speed up the search operations. Here, we turn this concept into the one of *opportunistic data structure*. Our data structure is opportunistic in the sense that it takes advantage of the compressibility of the input text to reduce the space occupancy.

Algorithm $\mathsf{get\_position}(i)$

1. $i' \leftarrow i$, $t \leftarrow 0$;
2. **while** row i' is not marked **do**
3. $\quad i' \leftarrow \mathsf{backward\_step}(i')$;
4. $\quad t \leftarrow t+1$;
5. **return** $\mathsf{Pos}(i') + t$;

Figure 3: Algorithm get_position for the computation of $\mathsf{Pos}(i)$.

Proof: By properties **b.** and **c.** of Section 2.3 we know that $j = LF[i] = C[L[i]] + \mathsf{Occ}(L[i], i)$. Unfortunately $\mathsf{Occ}(L[i], i)$ cannot be directly computed because the character $L[i]$ is not available being L compressed. Thus, algorithm backward_step determines $L[i]$ by computing $\mathsf{Occ}(c, i)$ and $\mathsf{Occ}(c, i-1)$ for every $c \in \Sigma \cup \{\#\}$. Clearly, the values $\mathsf{Occ}(c, i)$ and $\mathsf{Occ}(c, i-1)$ differ only for the character $c = L[i]$. Therefore the above computation gives us both $L[i]$ and $\mathsf{Occ}(L[i], i)$. If $L[i] = \#$ then $\mathsf{Pos}(i) = 1$, otherwise backward_step returns $j = C[L[i]] + \mathsf{Occ}(L[i], i)$. By Lemma 2 we know that using $\mathsf{Opp}(T)$ each computation of $\mathsf{Occ}()$ takes constant time. Since $|\Sigma| = \Theta(1)$, backward_step takes overall constant time as claimed. ■

Let $\eta = \left\lceil \log^{1+\epsilon} n \right\rceil$. For $j = 0, 1, \ldots, \lfloor n/\eta \rfloor$, we *logically mark* the rows of $\mathcal{M}_T$ corresponding to text positions of the form $1 + j\eta$. In other words, we mark the rows r_j such that $\mathsf{Pos}(r_j) = 1 + j\eta$. For each marked row r_j we store explicitly its position $\mathsf{Pos}(r_j)$ in a data structure $\mathcal{S}$ that supports membership queries in constant time (see below). Now, given a row index i we compute $\mathsf{Pos}(i)$ as follows. If $\mathcal{M}_T[i]$ is a marked row then its position is directly available in $\mathcal{S}$ and can be determined in constant time. Otherwise, we use algorithm backward_step to find the row i' such that $\mathsf{Pos}(i') = \mathsf{Pos}(i) - 1$ (Lemma 3). We iterate this procedure t times until row $\mathcal{M}_T[i']$ is marked; then we retrieve $\mathsf{Pos}(i')$ from $\mathcal{S}$ and we compute $\mathsf{Pos}(i) = \mathsf{Pos}(i') + t$. Note that our marking strategy ensures that we find a marked row in at most η iterations. We call this algorithm get_position and its pseudo-code is shown in Fig. 3.

The crucial point of algorithm get_position is the data structure $\mathcal{S}$ which stores the positions of the marked rows. $\mathcal{S}$ must be designed carefully in order to occupy succinct space and be able to support membership queries in constant time. The design of $\mathcal{S}$ is based (again) upon a bucketing scheme. We partition the rows of $\mathcal{M}_T$ into stripes of $\Theta(\log^2 n)$ rows each. For each stripe, we take all the marked rows lying into it and store them into a Packet B-tree [4] using as a key their distance from the beginning of the stripe. Since a stripe contains at most $O(\log^2 n)$ keys, each $O(\log\log n)$ bits long, membership queries take constant time. The Packed B-tree uses space proportional to the number of stored keys, hence $\mathcal{S}$ occupies $O((n/\eta)(\log\log n + \log n))$ bits because with each marked row r_j we keep the value $\mathsf{Pos}(r_j)$ using $O(\log n)$ bits. Overall algorithm get_position uses $O(n/\log^{\epsilon} n)$ bits for storing $\mathcal{S}$ plus the space used by the data structure $\mathsf{Opp}(T)$.

For what concerns the time complexity, a single call to get_position requires at most η iterations each one taking constant time. Hence the *occ* occurrences of a pattern P in T can be retrieved in $O(occ\,\eta) = O\left(occ \log^{1+\epsilon} n\right)$ time.

The parameter ϵ introduces a time/space tradeoff in our solution: the larger is ϵ, the larger is the query time and the smaller is the space occupancy. However, since $\mathsf{Opp}(T)$ uses $5nH_k(T) + O(n(\log\log n/\log n))$ space, reducing the space occupancy of $\mathcal{S}$ below $O(n(\log\log n/\log n))$ does not reduce the overall asymptotic space occupancy; for this reason the parameter ϵ should be taken in the range $0 < \epsilon < 1$. Summing up, we have the following result.

Theorem 3 *For any text $T[1,n]$ we can build an indexing data structure such that all the occ occurrences of a pattern $P[1,p]$ in T can be retrieved in $O\left(p + occ\log^{1+\epsilon} n\right)$ time, where $0 < \epsilon < 1$ is an arbitrary constant chosen when we build the data structure. The space occupancy of this data structure is bounded by $5nH_k(T) + O\left(\frac{n}{\log^\epsilon n}\right)$ bits, for any $k \geq 0$.* ∎

Algorithm get_position can be refined in order to list all the occurrences of P in $O(p + occ\log^\epsilon n)$ time using $O\left(nH_k(T) + n\,\frac{\log\log n}{\log^\epsilon n}\right)$ bits of storage. This faster algorithm is based on two basic ingredients: (*i*) a generalization of Lemma 3 which makes it possible to move backwards in T by more than one position in $O(1)$ time, and (*ii*) a hierarchical marking of the rows of the matrix $\mathcal{M}_T$. We do not report here the details of this algorithm since they are rather technical. The interested reader can find in [13, Theorem 3] a description of the algorithm together with a sketch of the proof of its correctness.

4 Using LZ78 parsing to achieve $O(p + occ)$ query time

In this section we describe algorithms and indexing data structures for retrieving the *occ* occurrences of a pattern $P[1,p]$ in $O(p + occ)$ time using $O(nH_k(T)\log^\epsilon n) + o(n)$ bits of storage. The key idea is to replace the logical marking scheme used in Section 3.2 with a more sophisticated approach based on the LZ78 parsing of the input text T.

The LZ78 parsing of T is defined inductively as follows [32]. Assume we have parsed a prefix of T as the sequence of *words* $T_1T_2\cdots T_{i-1}$, that is, $T = T_1T_2\cdots T_{i-1}\hat{T}_i$ for some text suffix $\hat{T}_i$. The next word in the parsing, say T_i, is the longest prefix of $\hat{T}_i$ that we can obtain adding a single character to a previous word or to the empty word. Hence, either $|T_i| = 1$ or $T_i = T_j\,c$ for some $j < i$ and $c \in \Sigma$.

Let $T = T_1T_2\cdots T_d$ denote the complete LZ78 parsing of T, and let $\mathcal{D} = \{T_1, T_2, \ldots, T_d\}$. The set $\mathcal{D}$ is usually called the LZ78 *dictionary*. By construction all words in $\mathcal{D}$ are distinct, possibly with the exception of the last word in the parsing.[3] In addition, the set $\mathcal{D}$ is *prefix-complete*, that is, if $T_i \in \mathcal{D}$ then all the non-empty prefixes of T_i also belong to $\mathcal{D}$. The number of words in $\mathcal{D}$ can be bounded in terms of the text size and the k-th order empirical entropy of T. In [18] it is shown that for any string $T[1,n]$ and for any integer $k \geq 0$ we have

$$d\log d \leq nH_k(T) + O((n\log\log n)/\log n)\,. \tag{2}$$

Since the words in the LZ78 parsing are distinct (except possibly T_d), we have $d = O(n/\log n)$ and thus $d\log(n/d) = O((n\log\log n)/\log n)$. From (2) we get the following useful inequality

$$d\log n = d\log d + d\log(n/d) \leq nH_k(T) + O((n\log\log n)/\log n)\,. \tag{3}$$

Let $T = T_1T_2\cdots T_d$ denote the LZ78 parsing of T and let \$ denote a character not belonging to Σ. We introduce the new string:

$$T_\$ = T_1\$T_2\$\cdots\$T_d\$. \tag{4}$$

The reason for which we introduce $T_\$$ is the following. The characters \$'s in $T_\$$ will be used as a sort of *anchor points*. For each \$ we store explicitly its position in T, that is, for the \$ following T_i we store the value $1 + |T_1| + |T_2| + \cdots + |T_i|$. Hence, to determine the position of a pattern occurrence we only need to determine the position of P with respect to any one of the \$'s. In this respect the characters \$'s play the same role of the logically marked rows used in Section 3.2. We immediately point out that storing the positions of the anchor points requires $d\log n$ bits; by (3) this amount of space is bounded by $nH_k(T) + o(n)$ bits.

[3] To simplify the exposition in the following we will ignore this exception and assume that all words are distinct.

Given a pattern P we divide its occurrences in the text T into two subsets: the occurrences completely contained in a single word T_i, and the occurrences overlapping two or more words $T_{j-1}T_j \cdots T_{j+h}$ with $h \geq 0$. We deal with this two subsets of occurrences separately. In Section 4.1 we describe an algorithm for retrieving the occurrences contained in a single word, and in Sections 4.2 and 4.3 we deal with the overlapping occurrences. More precisely, in Section 4.2 we describe a "simple" algorithm for retrieving the overlapping occurrences which is not optimal by a factor $\log\log n$ and in Section 4.3 we show how to refine that algorithm in order to achieve the desired $O(p + occ)$ time bound.

Before proceeding further, we introduce some additional notation. Let $T_\$^R$ denote the string $T_\$$ reversed, that is, $T_\$^R = \$T_d^R\$ \cdots \$T_2^R\$T_1^R$. We write $\mathcal{M}_{T_\$^R}$ to denote the cyclic shift matrix associated with the string $T_\$^R$ (see Section 2.3). Recall that, given a pattern P' over the alphabet $\Sigma \cup \{\$\}$, using $\mathsf{Opp}(T_\$^R)$ we can find in $O(|P'|)$ time the range of rows $(\mathsf{First}, \mathsf{Last})$ of the matrix $\mathcal{M}_{T_\$^R}$ which are prefixed by P' (see Theorem 2). In Appendix D we show that the size of $\mathsf{Opp}(T_\$^R)$ is bounded by $5nH_k(T) + O\left(n\frac{\log\log n}{\log n}\right)$ bits for any $k \geq 0$.

4.1 Retrieving the internal occurrences

In this section we describe an algorithm for retrieving the internal occurrences of $P[1,p]$, that is, the occurrences which are completely contained in a single word T_j.

Since the dictionary $\mathcal{D} = \{T_1, \ldots, T_d\}$ is prefix-complete it is customary to represent it using a trie $\mathcal{T}$ with every edge labeled with a single character. Each node u of $\mathcal{T}$ corresponds to the word $s(u)$ spelled out by the root-to-u path in $\mathcal{T}$. It is easy to see that all nodes descending from u correspond to the words in $\mathcal{D}$ having $s(u)$ as a prefix. Since all words are distinct every node of $\mathcal{T}$ corresponds to a single word of $\mathcal{D}$. With a little abuse of notation, we will identify a dictionary word with its corresponding trie node. For example, we write "the subtrie rooted at T_i" to denote the subtrie rooted at the node spelling out the dictionary word T_i. Our algorithm for retrieving the internal occurrences makes use of the trie $\mathcal{T}$ represented in the usual way: every node contains a pointer to each one of its children. In addition, for $i = 1, \ldots, d$, in the node corresponding to word T_i we store the value $v_i = 1 + |T_1| + \cdots + |T_{i-1}|$ which is the starting position of T_i in T.

Our high-level strategy for finding the internal occurrences of P is the following. First we find all words $T_{i_1}, \ldots, T_{i_z}$ which have P as a suffix. Because of the prefix-completeness property of $\mathcal{D}$, every other internal occurrence will lie inside a word T_j prefixed by one of the words T_{i_k} above. Hence, we can find all the internal occurrences by simply visiting the subtries rooted at $T_{i_1}, \ldots, T_{i_z}$. This takes time proportional to the number of visited nodes, which coincides with the number of internal occurrences.

The above strategy requires that we are able to efficiently find the trie nodes corresponding to the words ending with the pattern P. To solve this problem we make use of a nice property of the matrix $\mathcal{M}_{T_\$^R}$. Since each row of this matrix contains a cyclic shift of the string $T_\$^R$, we can define a one-to-one correspondence between the words $T_1, \ldots, T_d$ and the rows of $\mathcal{M}_{T_\$^R}$ prefixed by the symbol \$. In this correspondence the word T_i is matched to the unique row of $\mathcal{M}_{T_\$^R}$ prefixed by $\$T_i^R\$$ (see Fig. 4).[4] It is easy to see that the words ending with P are matched to the rows which have $\$P^R$ as a prefix. Since the rows of $\mathcal{M}_{T_\$^R}$ are lexicographically sorted, these rows are *contiguous* in the matrix $\mathcal{M}_{T_\$^R}$ and we can find them via the procedure $\mathsf{get\_rows}(\$P^R)$ over the data structure $\mathsf{Opp}(T_\$^R)$.

This property of $\mathcal{M}_{T_\$^R}$ suggests the introduction of an array $\mathcal{N}[1,d]$ which stores in entry $\mathcal{N}[i]$ a pointer to the trie node corresponding to the i-th row of $\mathcal{M}_{T_\$^R}$ prefixed by \$. Given the pattern P we use $\mathsf{get\_rows}(\$P^R)$ over $\mathsf{Opp}(T_\$^R)$ to retrieve the range $(\mathsf{First}, \mathsf{Last})$ of rows of $\mathcal{M}_{T_\$^R}$ prefixed by $\$P^R$. These

[4]There is an exception: word T_1 is matched to the row prefixed by $\$T_1^R\#$.

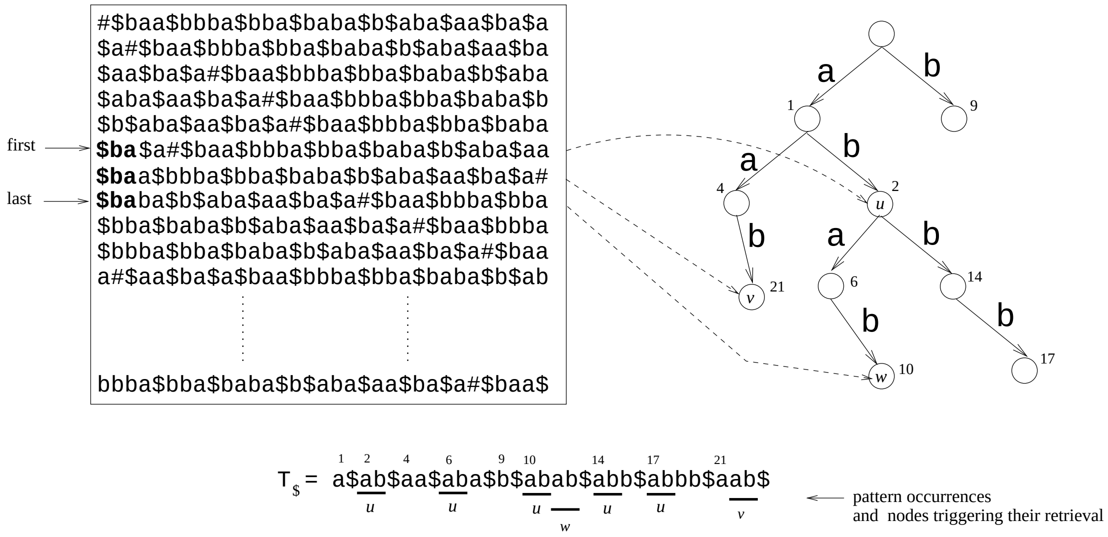

Figure 4: An example for the listing of the internal occurrences of the pattern P = ab in the text T = aabaaabababababbabbbaab. We have $T_\$$ = a\$ab\$aa\$aba\$b\$abab\$abb\$abbb\$aab\$. The figure shows a portion of the matrix $\mathcal{M}_{T^R_\$}$ on the left, and the trie $\mathcal{T}$ on the right. The value close to each trie node is the starting position in T of the corresponding word (i.e. for the word T_i the value $v_i = 1 + |T_1| + \cdots + |T_{i-1}|$). Step 1 of get_internal determines the three rows of $\mathcal{M}_{T^R_\$}$ which are prefixed by $\$P^R$ = \$ba: these are the rows 6,7 and 8. The three pointers $\mathcal{N}[6], \mathcal{N}[7], \mathcal{N}[8]$, depicted as dashed arrows, lead to the corresponding trie nodes u, v, w. These nodes spell out the three words suffixed by P, i.e. ab, aab and abab. Step 2 of get_internal visits the subtries descending from these nodes and returns all the internal occurrences. The figure also shows the string $T_\$$ with the internal occurrences underlined. For each internal occurrence it is shown the node triggering its retrieval. Notice that the word abab contains two occurrences both correctly detected by visiting the two subtries rooted one at node u and the other at node w.

rows correspond to the words $T_{i_1}, \ldots, T_{i_z}$ which have P as a suffix. The trie nodes corresponding to these words are those pointed by $\mathcal{N}[\mathsf{First}], \ldots, \mathcal{N}[\mathsf{Last}]$. See Fig. 4 for an example.

Our algorithm for retrieving the internal occurrences is summarized in Fig. 5. To prove its correctness it remains to show that the value $v_j + (|T_{i_k}| - p)$ returned at Step 2 is the correct starting position of an occurrence of P in T. To see this, recall that P is a suffix of T_{i_k}. Hence, $T_{i_k} = wP$ for some string w of length $|T_{i_k}| - p$. Assume word T_j is met during the visit of the subtrie rooted at T_{i_k}; and assume that $v_j = 1 + |T_1| + \cdots + |T_{j-1}|$ is the starting position of T_j in T (this value is kept at the trie node T_j). Then $T_j = T_{i_k} w' = wPw'$ for some string w', and hence P occurs inside T_j starting at position $v_j + |w| = v_j + (|T_{i_k}| - p)$ as claimed.

Note that it is possible that P occurs more than once in a dictionary word. For example, we could

---

Algorithm get_internal($P[1,p]$)

1. Search for $\$P^R$ in Opp($T^R_\$$) thus determining the words $T_{i_1}, \ldots, T_{i_z}$ which have P as a suffix.
2. For $k = 1, \ldots, z$, visit the subtrie of $\mathcal{T}$ rooted at node corresponding to T_{i_k}. For each word T_j encountered during this visit return the value $v_j + (|T_{i_k}| - p)$, where $v_j = 1 + |T_1| + \cdots + |T_{j-1}|$ is the starting position of T_j in T.

Figure 5: Algorithm get_internal for finding the internal occurrences of $P[1,p]$ in $T[1,n]$.

have $T_j = wPw'Pw''$. Our algorithm correctly reports both occurrences of P. The first one is reported when the algorithm visits the subtrie rooted at the word wP, the second one when the algorithm visits the subtrie rooted at the word $wPw'P$. Summing up, we have the following result.

Theorem 4 *Let occ_I denote the number of internal occurrences of $P[1,p]$ in $T[1,n]$. The algorithm* get_internal *retrieves them in $O(p + occ_I)$ time. The space occupancy of* get_internal *is $O(nH_k(T)) + O((n \log\log n)/\log n)$ bits, for any $k \geq 0$.*

Proof: We first analyze the time complexity. Step 1 is carried out using the procedure get_rows and therefore takes $O(p)$ time. At Step 2 we output the position of one internal occurrence for each word T_j encountered during the trie visit. Hence, Step 2 takes $O(occ_I)$ time overall.

To prove the bound on the space complexity, we point out that get_internal uses storage for the trie $\mathcal{T}$, the array $\mathcal{N}$, the values $v_1, \ldots, v_d$, and the data structure $\mathsf{Opp}(T_\$^R)$. The first three items all take $O(d \log n)$ bits of storage which by (3) is $O(nH_k(T)) + O((n \log\log n)/\log n)$ bits. In Appendix D we show that $\mathsf{Opp}(T_\$^R)$ takes $5nH_k(T) + O((n \log\log n)/\log n)$ bits of storage and the theorem follows. ■

4.2 Locating the overlapping occurrences: a simple algorithm

In this section we describe a first simple algorithm for retrieving the occurrences of P which overlap more than one dictionary word. The algorithm described in this section is not competitive in terms of running time. We describe it since it provides a simple introduction to the rather complex algorithm described in the next section. Our starting point is the following observation.

Property 1 *An overlapping occurrence of the pattern $P[1,p]$ starts inside the dictionary word T_{j-1}, fully overlaps $T_j \cdots T_{j+h-1}$ and ends inside T_{j+h}, for some $h \geq 0$, if and only if there exists m, $1 \leq m < p$, such that $P[1,m]$ is a suffix of T_{j-1} and $P[m+1,p]$ is a prefix of $T_j \cdots T_d$.*

Reasoning as in Section 4.1, we have that calling the procedure get_rows($\$P[1,m]^R$) over $\mathsf{Opp}(T_\$^R)$ we can retrieve the rows of the matrix $\mathcal{M}_{T_\$^R}$ prefixed by $\$P[1,m]^R$. These rows corresponds to the LZ78-words which have $P[1,m]$ as a suffix. Similarly, calling the procedure get_rows($P[m+1,p]$) over $\mathsf{Opp}(T)$ we can retrieve all suffixes of T which are prefixed by $P[m+1,p]$. By Property 1, to find the overlapping occurrences we "only" need to find which words ending with $P[1,m]$ are followed by a suffix of T prefixed by $P[m+1,p]$. We solve this problem using a geometric data structure.

For $i = 1, \ldots, d$, let x_i denote the row of the matrix $\mathcal{M}_{T_\$^R}$ prefixed by $\$T_{i-1}^R\$$. Note that x_i is the row matched to T_i in the one-to-one correspondence described in Section 4.1. Similarly, let y_i denote the row of the matrix $\mathcal{M}_T$ which is prefixed by $T_i T_{i+1} \cdots T_d\#$. We define the set of 2D-grid points $\mathcal{Q} = \{(x_1, y_1), (x_2, y_2), \ldots (x_d, y_d)\}$. Each pair (x_i, y_i) "captures" the text surrounding the i-th symbol \$. We can therefore restate Property 1 saying that there is an occurrence of P starting inside T_{j-1} and ending inside T_{j+h} iff there exists m such that $\$P[1,m]^R$ is a prefix of row x_j and $P[m+1,p]$ is a prefix of row y_j. Hence, we need to find all pairs (x_j, y_j) such that x_j is in the range of (contiguous) rows prefixed by $\$P[1,m]^R$, and y_j is in the range of (contiguous) rows prefixed by $P[m+1,p]$, for some $m < p$. We can find these pairs with $p-1$ orthogonal range queries on the set $\mathcal{Q}$. This is done by the algorithm get_overlapping shown in Fig. 6.

The correctness of algorithm get_overlapping can be proved as follows. Assume the point (x_j, y_j) is retrieved at the m-th iteration of Step 3. This means that row x_j of $\mathcal{M}_{T_\$^R}$ is prefixed by $\$P[1,m]^R$ and row y_j of $\mathcal{M}_T$ is prefixed by $P[m+1,p]$. From Property 1 it follows that there exists an occurrence of P which starts inside T_{j-1}, overlaps $T_j \cdots T_{j+h-1}$ and ends inside T_{j+h}, for some $h \geq 0$. We also know that $P[1,m]$ is a suffix of T_{j-1}, hence this pattern occurrence starts at position $v_j - m$ which is the value

Algorithm $\mathsf{get\_overlapping}(P[1,p])$

1. For $m = p, p-1, \ldots, 1$, search for $P[m,p]$ in $\mathsf{Opp}(T)$ thus retrieving the pair (f_m, ℓ_m) of positions of the first and last rows in $\mathcal{M}_T$ prefixed by $P[m,p]$.
2. For $m = 1, 2, \ldots, p$, search for $\$P[1,m]^R$ in $\mathsf{Opp}(T_\$^R)$ thus retrieving the pair (f_m^*, ℓ_m^*) of positions of the first and last rows of $\mathcal{M}_{T_\$^R}$ prefixed by $\$P[1,m]^R$.
3. For $m = 1, 2, \ldots, p-1$ use the data structure $\mathcal{RT}(\mathcal{Q})$ to retrieve the points of $\mathcal{Q}$ which lie inside the rectangle $[f_m^*, \ell_m^*] \times [f_{m+1}, \ell_{m+1}]$.
4. For each point (x_j, y_j) retrieved at the m-th iteration of Step 3 return the value $v_j - m$, where $v_j = 1 + |T_1| + \cdots + |T_{j-1}|$ is the starting position of the word T_j inside T.

Figure 6: Algorithm $\mathsf{get\_overlapping}$ for finding the overlapping occurrences of $P[1,p]$ in $T[1,n]$.

correctly returned by Step 4. With a similar reasoning one can prove that an occurrence of P which satisfies Property 1 triggers the retrieval of the point (x_j, y_j) at the m-th iteration of Step 3.

We highlight the importance of searching for $\$P[1,m]^R$ in $T_\$^R$ instead of just searching for $P[1,m]^R$ in T^R. The presence in $T_\$^R$ of the anchor \$ between adjacent words implies that, if (x_j, y_j) is retrieved at Step 3, the string $P[1,m]$ is completely contained in T_{j-1}. This ensures that a pattern overlapping more than two words is retrieved *only once* at Step 3. Assume for example that there is an overlapping occurrence such that $P[1,m]$ is a suffix of T_{j-1} and $P[1,m']$ is a suffix of $T_{j-1}T_j$ for some $m' > m$. The pair (x_{j+1}, y_{j+1}) relative to the symbol \$ which follows word T_j (it is the $(j+1)$-st \$) is not retrieved by $\mathsf{get\_overlapping}$ since the row x_{j+1} of $\mathcal{M}_{T_\$^R}$ is prefixed by $\$P[m,m']^R\$P[1,m]^R$ and not by $\$P[1,m']^R$ (this is because the pattern P does not contain the special symbol \$).

To analyze the complexity of $\mathsf{get\_overlapping}$ we make use of the following result, proven in [1].[5]

Theorem 5 *A set of 2D-grid points $\mathcal{Q}$ can be preprocessed in a data structure $\mathcal{RT}(\mathcal{Q})$ that occupies $O(|\mathcal{Q}| \log^\gamma |\mathcal{Q}|)$ memory words and supports orthogonal range queries in $O(\log\log|\mathcal{Q}| + q)$ time, where q is the number of points retrieved by a query and $\gamma > 0$ is an arbitrary constant chosen when we build the data structure.* ■

Theorem 6 *Let occ_O denote the number of overlapping occurrences of $P[1,p]$ in $T[1,n]$. Algorithm* $\mathsf{get\_overlapping}$ *retrieves them in $O(p\log\log n + occ_O)$ time. The data structures used by* $\mathsf{get\_overlapping}$ *take overall $O(nH_k(T)\log^\epsilon n) + o(n)$ bits of storage, where $\epsilon > 0$ is an arbitrary constant chosen when we build the data structures.*

Proof: We first analyze the time complexity. For what concerns Step 1, we notice that a single execution of $\mathsf{get\_rows}$ (Fig. 2) with input $P[1,p]$ computes the values $f_1, \ell_1, \ldots, f_p, \ell_p$ in $O(p)$ time. Similarly Step 2 takes $O(p)$ time to compute the values $f_1^*, \ell_1^*, \ldots, f_p^*, \ell_p^*$. By Theorem 5, Step 3 takes overall $O(p\log\log n + occ_O)$ time, where occ_O is the number of retrieved points (i.e. overlapping occurrences).

For what concerns the space occupancy, the algorithm makes use of the data structures $\mathsf{Opp}(T)$, $\mathsf{Opp}(T_\$^R)$, $\mathcal{RT}(\mathcal{Q})$, and of the values v_i for $i = 1, \ldots, d$. By Theorem 2 and Lemma 8 in Appendix D, we know that $\mathsf{Opp}(T)$ and $\mathsf{Opp}(T_\$^R)$ both uses $5nH_k(T) + O((n\log\log n)/\log n)$ bits. By Theorem 5 we know that $\mathcal{RT}(\mathcal{Q})$ takes $O(d\log^\gamma d \log n)$ bits which dominates the $O(d\log n)$ bits required for storing the values $v_1, v_2, \ldots, v_d$. By (3) and taking $\gamma = \epsilon$, we get $d(\log^\gamma d)(\log n) = O(nH_k(T)\log^\epsilon n) +$

[5] The result in [1] has been proven for a set of d points on a $d \times d$ grid. In our case, we have $d = |\mathcal{Q}|$ points defined on a $n \times n$ grid. Using standard *range reduction* techniques (see e.g. [1] Section 2.2), we can apply the result [1] with a slowdown in the query of $O(\log\log n)$ time.

$O\Big((n \log\log n)/\log^{1-\epsilon} n\Big)$. Hence the overall space occupancy of the algorithm is $O(nH_k(T)\log^{\epsilon} n) + o(n)$ bits as claimed. ∎

4.3 Locating the overlapping occurrences in $O(p + occ_O)$ time

In this section we describe an algorithm for retrieving the overlapping occurrences of a pattern $P[1,p]$ in $O(p + occ_O)$ time. We distinguish two kinds of patterns according to their length. If $p \leq (\log\log n)$ we say that the pattern is *short*, otherwise we say that the pattern is *long*. Short and long patterns are retrieved by two different algorithms which are described in the following subsections. Hereafter we assume that $\log\log n$ is an integer.

4.3.1 Case 1: long patterns

Long patterns are retrieved using a refinement of the geometric data structure described in Section 4.2. For $k < |T_j|$ let T_j^{-k} denote the prefix of the word T_j having length $|T_j| - k$ (i.e. T_j with the last k characters removed), and let T_j^{+k} denote the suffix of T_j having length k (i.e. the last k characters of T_j). Note that $T_j = T_j^{-k} T_j^{+k}$. We make use of the following property which refines Property 1.

Property 2 *An overlapping occurrence of the pattern $P[1,p]$ starts inside the dictionary word T_{j-1} and ends inside T_{j+h} (for some $h \geq 0$) if and only if there exist $i \geq 0$ and $k \in [0, \log\log n - 1]$ such that $P[1, i\log\log n + 1]$ is a suffix of T_{j-1}^{-k} and $P[i\log\log n + 2, p]$ is a prefix of $T_{j-1}^{+k}\, T_j \cdots T_d$.*

Property 2 states that within the last $\log\log n$ positions of T_{j-1} lies $P[1 + i\log\log n]$ for some $i \geq 0$. Taking advantage of Property 2 we can find the overlapping occurrences of long patterns using the data structure $\mathcal{RT}$ of Theorem 5 built over a new set of 2D-grid points $\mathcal{Q}'$. Recall that $\mathcal{Q}$ contains one grid point (x_j, y_j) for each anchor \$ in $T_\$$. The new set $\mathcal{Q}'$ contains $\log\log n$ points for each anchor. The idea is to define one grid point for each one of the $\log\log n$ positions immediately preceding an anchor.

Formally, for each word T_{j-1} and for each $k \in [0, \log\log n - 1]$, with $k < |T_{j-1}|$, we consider the pair of strings $(T_{j-1}^{-k}, T_{j-1}^{+k}\, T_j \cdots T_d)$. From the prefix-completeness property of the `LZ78` dictionary, the string T_{j-1}^{-k} is a dictionary word as well. Let x be the row of $\mathcal{M}_{T_\$^R}$ prefixed by $(\$T_{j-1}^{-k}\$)^R$, and let y be the row of $\mathcal{M}_T$ prefixed by $T_{j-1}^{+k}\, T_j \cdots T_d$. We insert in $\mathcal{Q}'$ the 2D-grid point $p_{j,k} = (x, y)$ and associate with it the value $v_{j,k} = 1 + |T_1| + \cdots + |T_{j-1}| - k$ which is the starting position of $T_{j-1}^{+k}\, T_j \cdots T_d$ in T.

Having defined $\mathcal{Q}'$ we can find the overlapping occurrences by executing $O(p/\log\log n)$ range queries on $\mathcal{RT}(\mathcal{Q}')$, instead of p queries on $\mathcal{RT}(\mathcal{Q})$. The reason is that now we can consider only the splitting of P at positions of the form $1 + i\log\log n$ for $i = 0, 1, \ldots, \lfloor p/(\log\log n)\rfloor$. The resulting algorithm, called get_overlapping_long, is shown in Fig. 7.

Theorem 7 *Let occ_O denote the number of overlapping occurrences of a long pattern $P[1,p]$ in $T[1,n]$. Algorithm* get_overlapping_long *retrieves them in $O(p + occ_O)$ time using $O(nH_k(T)\log^{\epsilon} n) + o(n)$ bits of storage, where $\epsilon > 0$ is an arbitrary constant chosen when we build the data structures.*

Proof: We first prove the correctness of the algorithm. Assume that an occurrence of P overlaps two or more dictionary words. By Property 2 there exist i, j, and $k \in [0, \log\log n - 1]$ such that $P[1, i\log\log n + 1]$ is a suffix of T_{j-1}^{-k} and $P[i\log\log n + 2, p]$ is a prefix of $T_{j-1}^{+k}\, T_j \cdots T_d$. By construction the set $\mathcal{Q}'$ contains a point $p_{j,k}$ corresponding to the pair of strings $(T_{j-1}^{-k}, T_{j-1}^{+k}\, T_j \cdots T_d)$. Since $\$P[1, i\log\log n + 1]^R$ is a prefix of $(\$T_{j-1}^{-k}\$)^R$, and $P[i\log\log n + 2, p]$ is a prefix of $T_{j-1}^{+k}\, T_j \cdots T_d$, point $p_{j,k}$ will be retrieved at Step 3 of algorithm get_overlapping_long when $h = i\log\log n + 1$. It is easy to see that the value $v_{j,k} - (i\log\log n + 1)$ returned at Step 4 is the correct starting position of the occurrence.

Algorithm $\mathsf{get\_overlapping\_long}(P[1,p])$

1. For $m = p, p-1, \ldots, 1$, search for $P[m,p]$ in $\mathsf{Opp}(T)$ thus retrieving the pair (f_m, ℓ_m) of positions of the first and last rows in $\mathcal{M}_T$ prefixed by $P[m,p]$.
2. For $m = 1, 2, \ldots, p$, search for $\$P[1,m]^R$ in $\mathsf{Opp}(T_\$^R)$ thus retrieving the pair (f_m^*, ℓ_m^*) of positions of the first and last rows of $\mathcal{M}_{T_\$^R}$ prefixed by $\$P[1,m]^R$.
3. For $i = 0, 1, \ldots, \lfloor p/(\log\log n)\rfloor$, set $h = i\log\log n + 1$ and use the data structure $\mathcal{RT}(\mathcal{Q}')$ to retrieve the points of $\mathcal{Q}'$ which lie inside the rectangle $[f_h^*, \ell_h^*] \times [f_{h+1}, \ell_{h+1}]$.
4. For each point (x, y) retrieved at the i-th iteration of Step 3 return the value $v_{j,k} - (i\log\log n + 1)$, where $v_{j,k}$ is the value associated with (x, y).

Figure 7: Algorithm $\mathsf{get\_overlapping\_long}$ for retrieving the overlapping occurrences of long patterns.

Conversely, assume that the point (x, y) is retrieved at the i-th iteration of Step 3. This means that row x of $\mathcal{M}_{T_\$^R}$ is prefixed by the string $\$P[1,h]^R$ and row y of $\mathcal{M}_T$ is prefixed by $P[h+1,p]$, where $h = i\log\log n + 1$. In addition, there exist j, k such that row x of $\mathcal{M}_{T_\$^R}$ is prefixed by $(\$T_{j-1}^{-k}\$)^R$, and row y of $\mathcal{M}_T$ is prefixed by $T_{j-1}^{+k}\, T_j \cdots T_d$. We conclude that $P[1,h]$ is a suffix of T_{j-1}^{-k} and $P[h+1,p]$ is a prefix $T_{j-1}^{+k}\, T_j \cdots T_d$. Hence (x, y) corresponds to an occurrence of P whose starting position is correctly reported at Step 4.

To analyze the time and space complexity of $\mathsf{get\_overlapping\_long}$ we first observe that $\mathcal{Q}'$ contains $O(d\log\log n)$ points. From Theorem 5 we get that $\mathcal{RT}(\mathcal{Q}')$ occupies $O(d(\log^\gamma n)\log\log n)$ memory-words and answers orthogonal range-queries in $O(\log\log n + q)$ time, where q is the number of retrieved points. Reasoning as in the proof of Theorem 6 we get that the space occupancy of $\mathsf{get\_overlapping\_long}$ is dominated by $\mathcal{RT}(\mathcal{Q}')$ which uses $O(d(\log^\gamma n)(\log n)\log\log n)$ bits of storage. By (3) and taking $\gamma < \epsilon$, we have $d(\log^\gamma n)(\log n)\log\log n = O(nH_k(T)\log^\epsilon n) + o(n)$ as claimed.

For what concerns the time complexity, the first two steps of $\mathsf{get\_overlapping\_long}$ take $O(p)$ time. Step 3 takes overall $O\left(\left\lceil \frac{p}{\log\log n}\right\rceil \log\log n + occ_O\right) = O(p + occ_O)$ time. Since Step 4 takes $O(occ_O)$ time, the algorithm takes $O(p + occ_O)$ time as claimed. ∎

4.3.2 Case 2: Short patterns

We now consider the retrieval of the overlapping occurrences when the pattern $P[1,p]$ is *short*, that is, $p < (\log\log n)$. The key observation here is that there cannot be too many of these occurrences so that we can store explicitly their positions in T.

Let $S = \{p_1, p_2, \ldots, p_\ell\}$ denote a lexicographically ordered list of all short patterns which have overlapping occurrences in T. We build an array $\mathcal{A}_S$ containing the positions in T of all overlapping occurrences of p_1, followed by the positions of all overlapping occurrences of p_2, and so on. Note that at most $O((\log\log n)^2)$ occurrences of short patterns may overlap any given word boundary since there are $O(\log\log n)$ possible starting positions before the boundary and $O(\log\log n)$ possible ending positions after the boundary. Since there are $d-1$ word boundaries the array $\mathcal{A}_S$ contains $O(d(\log\log n)^2)$ entries each requiring $O(\log n)$ bits. Hence the array $\mathcal{A}_S$ takes overall $O(d\log n(\log\log n)^2)$ bits of storage. By (3) this amount of space is bounded by $O(n\,H_k(T)\,(\log\log n)^2) + o(n)$ bits.

To access the array $\mathcal{A}_S$ we build a table $\mathcal{T}_S$ indexed by all possible short patterns, which are $O(|\Sigma|^{\log\log n})$ in total. Table $\mathcal{T}_S$ is defined as follows. Given a short pattern P:

1. if $P \in S$ then $\mathcal{T}_S[P] = (b, e)$, where $\mathcal{A}_S[b], \mathcal{A}_S[b+1], \ldots, \mathcal{A}_S[e]$ are the positions of the overlapping occurrences of P in T;

2. if $P \notin S$ then $\mathcal{T}_S[P] = (0,0)$.

Note that $\mathcal{T}_S$ requires $O(|\Sigma|^{\log\log n} \log n) = o(n)$ bits of storage. Now, given a short pattern P, we can list its overlapping occurrences in T by performing one access to $\mathcal{T}_S$ and occ_O accesses to $\mathcal{A}_S$. Since these operations involve $O(\log n)$ bits operands, the running time of this procedure is proportional to the number of overlapping occurrences.

Theorem 8 *If the pattern $P[1,p]$ is short, the occ_O overlapping occurrences of P in $T[1,n]$ can be retrieved in $O(p + occ_O)$ time. The data structures supporting the retrieval of short patterns takes overall $O(n\, H_k(T)\, (\log\log n)^2) + o(n)$ bits of storage.* ∎

Finally, combining Theorem 4 (for the retrieval of the internal occurrences) with Theorems 7 and 8 (for the retrieval of the overlapping occurrences) we get the following result.

Theorem 9 *For any text $T[1,n]$ we can build an indexing data structure such that all the occ occurrences of a pattern $P[1,p]$ in T can be retrieved in $O(p + occ)$ time. For any $k \geq 0$ the space required by this data structure is $O(n\, H_k(T)\, \log^\epsilon n) + o(n)$ bits, where $\epsilon > 0$ is an arbitrary constant chosen when we build the data structure.* ∎

5 Theory or practice?

The results described in the previous sections are certainly theoretical in their flavor. However, the possibility of storing in compressed form both the input file and its full-text index has obviously a strong practical interest. We believe that our results could be the starting point for the design of a new family of text retrieval tools. In this section we back up this statement by briefly describing two results which show the flexibility of our compressed indexes and their suitability for real-world applications.

The first result concerns the possibility of building a compressed index for handling a collection of texts which changes over time, that is, new texts can be inserted or old texts can be deleted from the collection. It is important to be able to handle efficiently this dynamic situation since we do not want to rebuild the whole index each time that a single text is inserted, deleted, or modified. Starting from the data structures described in Section 3 it is possible to build a dynamic compressed full-text index whose performances are summarized by the following theorem proven in [13].

Theorem 10 *Let Δ be a dynamic collection of texts $\{T_1, T_2, \ldots, T_m\}$ having total length u. All the occ occurrences of a pattern $P[1,p]$ in the texts of Δ can be listed in $O(p \log^3 u + occ \log^{1+\epsilon} u)$ time in the worst case, where ϵ is a positive constant. We can insert a new text $T[1,t]$ in Δ in $O(t \log u)$ amortized time. We can delete a text $T[1,t]$ from Δ in $O(t \log^2 u)$ amortized time. The space occupancy is $O(uH_k(\Delta) + m \log u) + o(u)$ bits, where $H_k(\Delta)$ is the k-th order entropy of the concatenation of the texts $T_1, \ldots, T_m$.* ∎

The data structure which realizes the bound of Theorem 10 is based on the results of Section 3 combined with an elegant "dynamization" technique proposed in [24, 28]. This technique allows to make dynamic any static data structure at the cost of some slowdown in the query operation. However, in Theorem 10 the items to be indexed are not atomic so that this technique could not be used in [13] as a black-box but it had to be adapted to manage items (texts) of variable lengths.

The second result that we discuss in this section is a "practical" implementation of the compressed index described in Section 3. This implementation, proposed in [14], is not a verbatim translation of the data structures of Section 3. In several points it uses simpler techniques which work reasonably well in practice rather than more cumbersome techniques with guaranteed good asymptotic worst case behavior. In [14] two different versions of our compressed index have been implemented:

Name	Size	Content	Name	Size	Content
bible	4,047,392	King James Bible	*cantrbry*	2,821,120	Canterbury corpus
e.coli	4,638,690	DNA sequence	*ap90*	67,108,864	SGML-tagged text
world192	2,473,400	1992 CIA world fact book	*jdk13*	69,728,899	`html` and `java` sources

Table 1: Files used in the experiments in [14].

	File	*bible*	*e.coli*	*world*	*cantbry*	*jdk13*	*ap90*
tiny index	Compr. ratio	21.09	26.92	19.62	24.02	5.87	22.14
	Construction time	2.24	2.19	2.26	2.21	3.43	3.04
	Decompression time	0.45	0.49	0.44	0.38	0.42	0.57
	Ave. count time	4.3	12.3	4.7	8.1	3.2	5.6
fat index	Compr. ratio	32.28	33.61	33.23	46.10	17.02	35.49
	Construction time	2.28	2.17	2.33	2.39	3.50	3.10
	Decompression time	0.46	0.51	0.46	0.41	0.43	0.59
	Ave. count time	1.0	2.3	1.5	2.7	1.3	1.6
	Ave. locate time	7.5	7.6	9.4	7.1	21.7	5.3
`bzip2`	Compression ratio	20.90	26.97	19.79	20.24	7.03	27.36
	Compression time	1.16	1.28	1.17	0.89	1.52	1.16
	Decompression time	0.39	0.48	0.39	0.31	0.28	0.43
`gzip`	Compr. ratio	29.07	28.00	29.17	26.10	10.79	37.35
	Compression time	1.74	10.48	0.87	5.04	0.39	0.97
	Decompression time	0.07	0.07	0.06	0.06	0.04	0.07

Table 2: Compression ratio (percentage) and compression/decompression speed (microseconds per input byte) of our indexes compared with those of `gzip` (with option `-9` for maximum compression) and `bzip2` (version 1.0.1). For our indexes we also reports the average time (in milliseconds) for the count and locate operations. The average was computed searching 1,000 randomly chosen English words of length between 4 and 8 (for *e.coli* we used random DNA sequences of length between 8 and 15). The experiments were run on a machine equipped with a 600Mhz Pentium III processor, 512Kb L2 cache, and 1 Gb RAM. The operating system was Gnu/Linux Debian 2.2.

- A *tiny* index designed to achieve high compression. The tiny index only supports the get_rows procedure described in Section 3.1, and for this reason it can only compute the number of occurrences of a pattern P in the input text.
- A *fat* index designed to support both the counting and the retrieval of the pattern occurrences. The fat index loosely corresponds to the data structure described in Section 3.2.

Both the tiny and the fat indexes consist of a compressed version of the input file plus additional information used for pattern searching. The details of the implementation and a complete experimental analysis are described in [14]. In the following we only report a comparison between our indexes, `gzip` (the standard Unix compressor) and `bzip2` (the best known compressor based on the BWT).

Table 1 reports the files used in the experiments and Table 2 summarizes the results of our comparison. The experiments show that the tiny index takes significantly less space than the corresponding `gzip`-compressed file. In addition, for all files except *bible* and *cantrbry*, the tiny index takes less space than `bzip2`. This may appear surprising since `bzip2` is also based on the BWT. The explanation is that our index computes the BWT for the entire file whereas `bzip2` splits the input in 900Kb blocks. This compression improvement is payed in terms of speed; the construction of the tiny index takes more time that compressing the file with `bzip2`, the difference being more noticeable for larger files. The experiments also show that the fat index takes slightly more space than the corresponding `gzip`-compressed file. For what concerns the query time we have that both the tiny and the fat index compute the number of occurrences of a pattern in a few milliseconds, independently of the size of the searched file. Using the fat index one can also compute the position of each occurrence. This takes a few milliseconds per occurrence, independently of the size of the searched file. The reader should refer to [14] for more details and comments on these experiments. Here we only point out that the amount of additional information required for efficiently searching patterns in a compressed file is surprisingly small. This is especially true if one is only interested in counting the number of occurrences. We believe that a compressed data structure with this "pattern counting" feature could be an extremely useful tool in applications such as virus detectors, spell checkers, personal search engines, etc.

6 Conclusions and open problems

In this paper we have studied the interplay between compression and indexing of data. Our first result is a data structure which indexes a text $T[1,n]$ using at most $5nH_k(T) + o(n)$ bits of storage, where $H_k(T)$ is the k-th order empirical entropy of T. Our data structure supports arbitrary substring searches in $O(p + occ \log^{1+\epsilon} n)$ time, where p is the length of the queried substring, occ the number of occurrences, and $\epsilon > 0$ an arbitrarily small constant chosen when we build the data structure. Our second result builds upon the previous one and achieves $O(p + occ)$ query time using $O(nH_k(T) \log^{\epsilon} n) + o(n)$ bits of storage.

As a main open problem we ask for a data structure which achieves the best of the previous bounds: $O(p + occ)$ query time and $O(nH_k(T)) + o(n)$ bits of storage occupancy. If such a result is possible is yet open. Another challenging question is if, and how, locality of reference may be introduced in our data structures to take advantage of the blocked access to a disk memory when indexing very large text collections. We aim at achieving $O(occ/B)$ I/O's for the location of the pattern occurrences, where B is the disk-page size. Such an index might compete successfully against the String B-tree data structure [12] which is actually the best solution known for the full-text indexing problem on large text collections. String B-trees require more than $12n$ bytes of storage [11].

Our compressed index can be used as a building block for more complex Information Retrieval applications. As an example in [13, Section 5] we have described the use of our index within the block addressing scheme introduced by Glimpse [21].

References

[1] S. Alstrup, G. Stolting Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *Proc. of the 41st Annual IEEE Symposium on Foundations of Computer Science*, pages 198–207, 2000.

[2] A. Amir and G. Benson. Efficient two-dimensional compressed matching. *Proc. of IEEE Data Compression Conference*, pages 279–288, 1992.

[3] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52(2):299–307, 1996.

[4] A. Andersson. Sorting and searching revisited. In *Proc. of the 5th Scandinavian Workshop on Algorithm Theory*, pages 185–197. Springer-Verlag LNCS n. 1097, 1996.

[5] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive compression scheme. *Communication of the ACM*, 29(4):320–330, 1986.

[6] A. Brodnik and I. Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999.

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

[8] S. Chen and J. Reif. Using difficulty of prediction to decrease computation: Fast sort, priority queue and convex hull on entropy bounded inputs. In *Proc. of the 34th IEEE Symposium on Foundations of Computer Science*, pages 104–112, 1993.

[9] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998.

[10] P. Fenwick. The Burrows-Wheeler transform for block sorting text compression: principles and improvements. *The Computer Journal*, 39(9):731–740, 1996.

[11] P. Ferragina and R. Grossi. Fast string searching in secondary storage: Theoretical developments and experimental results. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 373–382, 1996.

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

[13] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. of the 41st IEEE Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[14] P. Ferragina and G. Manzini. An experimental study of a compressed index. *Information Sciences: special issue on "Dictionary Based Compression"*, 135:13–28, 2001.

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

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

[17] A. Karlin, S. Phillips, and P. Raghavan. Markov paging (extended abstract). In *Proc. of the 33rd IEEE Symposium on Foundations of Computer Science*, pages 208–217, 1992.

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

[19] P. Krishnan and J. Vitter. Optimal prediction for prefetching in the worst case. *SIAM Journal on Computing*, 27(6):1617–1636, 1998.

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

[21] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. of the USENIX Winter 1994 Technical Conference*, pages 23–32, 1994.

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

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

[24] K. Mehlhorn and M. H. Overmars. Optimal dynamization of decomposable searching problems. *Information Processing Letters*, 12(2):93–98, 1981.

[25] I. Munro. Tables. In *Proceeding of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 37–42. Springer-Verlag LNCS n. 1180, 1996.

[26] I. Munro. Succinct data structures. In *Proc. of the Workshop on Data Structures,* within the 19th Conference on Foundations of Software Technology and Theoretical Computer Science, pages 1–6, 1999.

[27] I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science*, pages 118–126, 1997.

[28] M. H. Overmars and J. van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Information Processing Letters*, 12(4):168–173, 1981.

[29] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proceeding of the 11th International Symposium on Algorithms and Computation*, pages 410–421. Springer-Verlag, LNCS n. 1969, 2000.

[30] C. E. Shannon and W. Weaver. *The Mathematical Theory of Communication.* The University of Illinois Press, Urbana, 11. edition, 1964.

[31] 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.

[32] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Transaction on Information Theory*, 24:530–536, 1978.

A The empirical entropy of a string

In this appendix we review the definition of empirical entropy and we prove some of its properties that are used elsewhere in the paper. Let $T[1,n]$ be a string over the alphabet $\Sigma = \{\alpha_1, \ldots, \alpha_h\}$, and let n_i denote the number of occurrences of the symbol α_i in T. The zeroth order empirical entropy of the string T is defined as

$$H_0(T) = -\sum_{i=1}^{h} \frac{n_i}{n} \log\left(\frac{n_i}{n}\right), \tag{5}$$

(in the following all logarithms are taken to the base 2 and we assume $0 \log 0 = 0$). The value $nH_0(T)$, represents the output size of an ideal compressor which uses $-\log \frac{n_i}{n}$ bits for coding the symbol α_i. It is well known that this is the maximum compression we can achieve using a uniquely decodable code in which a fixed codeword is assigned to each alphabet symbol. We can achieve a greater compression if the codeword we use for each symbol depends on the k symbols preceding it. For any length-k string $w \in \Sigma^k$ and $\alpha_i \in \Sigma$, let $n_{w\alpha_i}$ denote the number of occurrences in T of the string w followed by α_i (that is, the number of occurrences of the string $w\alpha_i$ in T). Let $n_w = \sum_i n_{w\alpha_i}$. The value

$$H_k(T) = -\frac{1}{n} \sum_{w \in \Sigma^k} n_w \left[\sum_{i=1}^{h} \frac{n_{w\alpha_i}}{n_w} \log\Bigl(\frac{n_{w\alpha_i}}{n_w}\Bigr)\right] \tag{6}$$

is called the k-th order empirical entropy of the string T. The value $nH_k(T)$ represents a lower bound to the output size of any uniquely decodable encoder which uses for each symbol a code which depends only on the symbol itself and on the k most recently seen symbols. Not surprisingly, for any string T and $k \geq 0$, we have $H_k(T) \geq H_{k+1}(T)$. The empirical entropy resembles the entropy defined in the probabilistic setting (for example, when the input comes from a Markov source). However, the empirical entropy is defined for any string and can be used to measure the performance of compression algorithms without any assumption on the input source.

Let T^R denote the string T reversed. In general $H_k(T)$ is different from $H_k(T^R)$; however these two values cannot be too far away: we now show that their difference is bounded by $O((\log n)/n)$.

For any string w of length $k > 0$, we define m_w as the number of occurrences of w in $T[1,n]T[1,k-1]$. The value m_w can be seen also as the number of occurrences of w in T counting also the occurrences which wrap around the boundaries of T. For example, for $T = \mathtt{aabaa}$ we have $m_{\mathtt{aa}} = 3$, and $m_{\mathtt{aaa}} = 2$. Note that $\sum_{w\in\Sigma^k} m_w = n$. We define the value $\hat{H}_k(T)$ as

$$\hat{H}_k(T) = -\frac{1}{n} \sum_{w\in\Sigma^k} m_w \left[\sum_{i=1}^{h} \frac{m_{w\alpha_i}}{m_w} \log\Big(\frac{m_{w\alpha_i}}{m_w}\Big) \right]. \tag{7}$$

The reason for which we introduce $\hat{H}_k(T)$ is that its value does not change if we reverse the input string.

Lemma 4 *For any $k > 0$ and for any string T, $\hat{H}_k(T) = \hat{H}_k(T^R)$.*

Proof: Let $F_k(T) = \sum_{w\in\Sigma^k} m_w \log m_w$. A simple algebraic calculation shows that $\hat{H}_k(T) = F_{k+1}(T) - F_k(T)$. Since m_w is equal to the number of occurrences of w^R in T^R we have $F_k(T) = F_k(T^R)$. Hence

$$\hat{H}_k(T) = F_{k+1}(T) - F_k(T) = F_{k+1}(T^R) - F_k(T^R) = \hat{H}_k(T^R).$$

■

Lemma 5 *For any $k > 0$ and for any string $T[1,n]$ we have*

$$H_k(T) \leq \hat{H}_k(T) \leq H_k(T) + (k/n)\log(ne).$$

Proof: Let $t_1, t_2, \ldots, t_h$ be non negative values and let $t = \sum_{i=1}^h t_i$. We define

$$G(t_1, t_2, \ldots, t_h) = -\sum_{i=1}^{h} t_i \log\left(\frac{t_i}{t}\right). \tag{8}$$

Clearly $G(t_1, \ldots, t_h) \geq 0$ and the value $G(t_1, \ldots, t_h)$ does not change if we permute the positions of the variables t_i's. Moreover, using elementary calculus one can prove that

$$G(t_1, t_2, \ldots, t_h) \;\leq\; G(t_1 + 1, t_2, \ldots, t_h) \;\leq\; G(t_1, t_2, \ldots, t_h) + \log(1 + t_1 + \cdots + t_h) + \log e. \tag{9}$$

By (6) and (7) we have that using G we can rewrite the values $H_k(T)$ and $\hat{H}_k(T)$ as

$$H_k(T) = \frac{1}{n} \sum_{w\in\Sigma^k} G(n_{w\alpha_1}, \ldots, n_{w\alpha_h}), \qquad \hat{H}_k(T) = \frac{1}{n} \sum_{w\in\Sigma^k} G(m_{w\alpha_1}, \ldots, m_{w\alpha_h}). \tag{10}$$

By construction, for any string w and for any character α_i we have $m_{w\alpha_i} \geq n_{w\alpha_i}$. We also have

$$\sum_{w\in\Sigma^k} \sum_{\alpha_i\in\Sigma} n_{w\alpha_i} = n - k, \qquad \sum_{w\in\Sigma^k} \sum_{\alpha_i\in\Sigma} m_{w\alpha_i} = n.$$

This means that starting from $H_k(T)$ we can get the value $\hat{H}_k(T)$ increasing k terms $n_{w\alpha_i}$ by one. These terms correspond to the strings which wrap around T's boundaries (these strings are counted in $\hat{H}_k(T)$ but not in $H_k(T)$). By (9) we see that increasing a term $n_{w\alpha_j}$ by one never decreases G and increases it by at most $\log(1+\sum_i n_{w\alpha_i})+\log e$. Since $(1+\sum_i n_{w\alpha_i}) \leq n$ we conclude that k increments can increase G by at most $k\log(ne)$ and the lemma follows. ■

Theorem 11 *For any fixed $k \geq 0$ and for any string $T[1,n]$ we have*

$$nH_k(T) - O(\log n) \leq nH_k(T^R) \leq nH_k(T) + O(\log n).$$

Proof: For $k=0$ we have $H_0(T) = H_0(T^R)$. For $k>0$, using Lemmas 4 and 5 we get

$$nH_k(T^R) \leq n\hat{H}_k(T^R) = n\hat{H}_k(T) \leq nH_k(T) + O(\log n).$$

To complete the proof we simply replace T^R with T. ■

We conclude this section giving a bound for the *modified empirical entropy* of a string. Recalling the function G defined by (8) we define

$$G^*(t_1,\ldots,t_h) = \begin{cases} 1+\lfloor\log(t_1+\cdots+t_h)\rfloor & \text{if } \exists i \text{ such that } t_i \neq 0 \text{ and } t_j = 0 \text{ for } j \neq i, \\ G(t_1,\ldots,t_h) & \text{otherwise.} \end{cases}$$

The k-th order modified empirical entropy of a string $T[1,n]$ is defined as

$$H^*_k(T) = \frac{1}{n}\sum_{w\in\Sigma^k} G^*(n_{w\alpha_1},\ldots,n_{w\alpha_h}). \tag{11}$$

The value $H^*_k(T)$ represents the maximum compression we can achieve using for each symbol a code which only depends on the symbol itself and on the previous k symbols, with the additional requirement that the coding of a string takes at least enough bits to write down its length in binary. The modified empirical entropy $H^*_k(T)$ has been used in [22] for the analysis of BWT-based compression algorithms. The following lemma bounds the entropy $H^*_k(T)$ in terms of $H_k(T)$.

Lemma 6 *For any fixed $k \geq 0$ and for any string $T[1,n]$*

$$nH_k(T) \leq nH^*_k(T) \leq nH_k(T) + O(\log n).$$

Proof: Comparing (10) with (11) we see that

$$nH^*_k(T) - nH_k(T) = \sum_{w\in\Sigma^k}[G^*(n_{w\alpha_1},\ldots,n_{w\alpha_h}) - G(n_{w\alpha_1},\ldots,n_{w\alpha_h})].$$

Since $G^*(t_1,\ldots,t_h) \geq G(t_1,\ldots,t_h)$ and $G^*(t_1,\ldots,t_h) > G(t_1,\ldots,t_h)$ implies $G(t_1,\ldots,t_h) = 0$ we have

$$0 \leq nH^*_k(T) - nH_k(T) \;\leq\; \sum_{w\in\Sigma^k}(1+\lfloor\log n\rfloor) \;\leq\; |\Sigma|^k(1+\lfloor\log n\rfloor)$$

and the lemma follows since k and $|\Sigma|$ are constants. ■

B The compression ratio of algorithm BW_RLX

In this appendix we prove the bound (1) on the compression achieved by the algorithm BW_RLX. Our proof is based on some results about BWT-based compressors given in [22].

Following the notation in [22, Section 5], we denote by $\mathtt{mtf}_\pi$ the move-to-front encoder in which the recency list initially contains the alphabet symbols ordered according to the permutation π. We define the algorithm $\mathtt{A}_\pi = \mathtt{mtf}_\pi + \mathtt{rle} + \mathtt{PC}$ as the algorithm that encodes a given string s using $\mathtt{mtf}_\pi$ followed by rle followed by PC. For any string s, we denote with $\pi(s)$ the ordering which *maximizes* the output size of $\mathtt{A}_\pi$ and define $\mathtt{A}^* = \mathtt{mtf}_{\pi(s)} + \mathtt{rle} + \mathtt{PC}$. Note that algorithm $\mathtt{A}^*$ first computes the worst case ordering $\pi(s)$ and then uses it as the initial ordering for move-to-front encoding. Although the definition of $\mathtt{A}^*$ may appear rather unnatural, its worst-case structure makes it a very useful tool.

By Lemma 5.7 in [22] we know that for every k there exists a constant g_k such that for any string $T[1,n]$ the output size $|\mathtt{A}^*(\mathtt{bwt}(T))|$ is bounded by $5nH_k^*(T) + g_k$.[6] Since $\mathtt{A}^*$ uses the worst possible ordering, the algorithm BW_RLX will certainly produce an output of smaller size. Hence, for any string $T[1,n]$ we have

$$|\mathtt{BW\_RLX}(T)| \leq |\mathtt{A}^*(\mathtt{bwt}(T))| \leq 5nH_k^*(T) + g_k.$$

Using Lemma 6 we get the desired bound

$$|\mathtt{BW\_RLX}(T)| \leq 5nH_k(T) + O(\log n)\,.$$

C Managing long runs of zeroes

In this appendix we discuss the computation of $\mathsf{Occ}(c,q)$ in the general case in which L^{mtf} contains one or more runs of zeroes which cross the border of the buckets BL_i^{mtf}'s. Recall that rle replaces each (maximal) run of zeroes 0^m in L^{mtf} with the string $\mathtt{bin}(m)$ defined as the binary representation of $(m+1)$ in reverse order (least significant bit first) and with the most significant bit discarded. This encoding is also known as 1/2 encoding since if $\mathtt{bin}(m) = b_0 b_1 \cdots b_k$, with $b_i \in \{0,1\}$, then $m = \sum_{j=0}^{k}(b_j+1)2^j$. Because of this property, from $\mathtt{bin}(m)$ we can retrieve 0^m by simply replacing each bit b_j with a sequence of $(b_j+1)2^j$ zeroes.

Suppose now that a sequence of $a+b$ zeroes is split between two buckets: let say BL_{i-1}^{mtf} ends with a zeroes and BL_i^{mtf} starts with b zeroes (a similar argument holds for a sequence overlapping an entire bucket). We define the compressed buckets BZ_{i-1} and BZ_i as follows. We know that rle transforms the sequence 0^{a+b} into the string $\sigma = \mathtt{bin}(a+b)$. We interpret every digit of σ according to the 1/2 decoding scheme mentioned above and we assign to BZ_{i-1} the shortest prefix of σ whose 1/2 decoding is *greater or equal* than a. The remaining digits of σ are assigned to BZ_i. For example, if $a=3$ and $b=14$, we have $\sigma = \mathtt{bin}(17) = \mathbf{0100}$. We assign to BZ_{i-1} the symbols **01** (whose 1/2 decoding is 0^5), and to BZ_i the symbols **00** (whose 1/2 decoding is 0^3).

It should be clear that table S introduced in Section 3.1 can be used to count the number of occurrences of any character in a prefix of BL_{i-1} but the same table does not work for BL_i. Indeed, BZ_{i-1} is a faithful[7] encoding of BL_{i-1}^{mtf} whereas the leading b zeroes of BL_i^{mtf} are usually not faithfully encoded by the digits of σ assigned to BZ_i. In the above example we have $b=14$ and we have assigned to BZ_i the symbols **00** whose 1/2 decoding is 0^3; hence there are 11 missing zeroes. It is easy to prove that the

[6]Notice that Lemma 5.7 in [22] establishes that $\mathtt{A}^*$ is locally 5-optimal. The inequality $|\mathtt{A}^*(\mathtt{bwt}(T))| \leq 5nH_k^*(T) + g_k$ follows immediately from the discussion in Section 3 of [22].

[7]Note that BZ_{i-1} encodes BL_{i-1}^{mtf} possibly followed by some additional zeroes. However these additional zeroes do not cause any problem in the use of table S because S is queried with positions lying within the bucket and therefore to the left of the additional zeroes.

1. **if** c is the first character in $MTF[i]$ **then**
2. **if** $j \le MZ[i]$ **then return** j
3. **else return** $MZ[i] + S[c, j - MZ[i], BZ_i, MTF[i]]$
4. **else**
5. **if** $j \le MZ[i]$ **then return** 0
6. **else return** $S[c, j - MZ[i], BZ_i, MTF[i]]$

Figure 8: Procedure for computing the number of occurrences of a character c among the first j characters of the i-th bucket BL_i.

digits of σ assigned to BZ_i never encode more than b zeroes and therefore there is always a non-negative number of missing zeroes.

These observations suggest the introduction of an array $MZ[1, n/\ell]$ such that $MZ[i]$ contains the number of leading zeroes of BL_i^{mtf} not encoded by BZ_i. In the above example we would have $MZ[i] = 11$. Note that, if α is the first symbol in $MTF[i]$ (that is, the first character in the MTF-list when the i-th bucket is encoded), then BL_i consists of $MZ[i]$ copies of α followed by the decoding of BZ_i. It follows that we can compute the number of occurrences of any character c among the first j characters of BL_i using the procedure described in Fig. 8. Since the array MZ occupies $O((n/\ell)\log \ell) = O((n/\log n)\log\log n)$ bits, its introduction does not change the asymptotic space occupancy of the procedure get_rows.

D Size of $\mathsf{Opp}(T_\$^R)$

Let $T[1, n]$ be a string over the alphabet $\Sigma = \{\alpha_1, \ldots, \alpha_h\}$, and let $T = T_1 T_2 \cdots T_d$ denote its LZ78 parsing. We recall that the string $T_\$^R$ is defined as $T_\$^R = \$T_d^R\$T_{d-1}^R\$\cdots\$T_1^R$ where \$ is a character not belonging to Σ. In order to prove an upper bound on the size of $\mathsf{Opp}(T_\$^R)$, we first need a preliminary lemma relating the entropies of T and $T_\$$.

Lemma 7 *For any fixed $k \ge 0$ and for any string $T[1, n]$ we have*

$$|T_\$| H_k(T_\$) \le n H_k(T) + O(n/\log n)\,.$$

Proof: We recall that $|T_\$| H_k(T_\$)$ is a lower bound to the output size of any uniquely decodable encoder in which the code for each symbol only depends on the symbol itself and on the k preceding symbols. We now define an encoder, called `Aux`, which has this property and therefore produces an output $\texttt{Aux}(T_\$)$ whose size is greater than $|T_\$| H_k(T_\$)$. We prove the lemma by showing that $|\texttt{Aux}(T_\$)| \le nH_k(T) + O(n/\log n)$.

For any $w \in \Sigma^k$ and for any $\alpha_i \in \Sigma$, let $t_{w\alpha_i}$ denote the number of occurrences of the string $w\alpha_i$ in $T_\$$, thus $t_w = \sum_i t_{w\alpha_i}$. The algorithm `Aux` encodes each character $\alpha \in T_\$$ as follows:

1. if both α and the k characters preceding it are in Σ then α is encoded using $-\log(t_{w\alpha}/t_w)$ bits, where w is the length-k string preceding α;
2. otherwise α is encoded using $\log(|\Sigma| + 1)$ bits.

In other words, if α and the k characters preceding it are in Σ we use the best possible conditional code, otherwise (that is, if α or one of the k characters preceding it is the \$ character) we use a code of length equal to the logarithm of the alphabet size (which means no compression at all). Note that `Aux` is only an ideal compressor since a code may consists of a fractional number of bits. However, it is still true that $|\mathtt{Aux}(T_\$)| \geq |T_\$| H_k(T_\$)$.

From the definition of Aux we get

$$|\mathtt{Aux}(T_\$)| \leq -\left[\sum_{w\in\Sigma^k}\sum_{i=1}^{h} t_{w\alpha_i}\log\left(\frac{t_{w\alpha_i}}{t_w}\right)\right] \; + \; (k+1)d\log(|\Sigma|+1).$$

Recalling that $d = O(n/\log n)$, and using the function G defined by (8) we have

$$|\mathtt{Aux}(T_\$)| \leq \left[\sum_{w\in\Sigma^k} G(t_{w\alpha_1},\ldots,t_{w\alpha_h})\right] + O(n/\log n)$$

We complete the proof showing that

$$\sum_{w\in\Sigma^k} G(t_{w\alpha_1},\ldots,t_{w\alpha_h}) \leq nH_k(T).$$

Let $n_{w\alpha_i}$ denote the number of occurrences of the string $w\alpha_i$ in T (by construction $w\alpha_i$ does not contain the special character \$). Since every occurrence of $w\alpha_i$ in $T_\$$ corresponds to an occurrence in T we have $n_{w\alpha_i} \geq t_{w\alpha_i}$. By the proof of Lemma 5 we know that G is monotonically increasing in each of its components. Hence we have

$$\sum_{w\in\Sigma^k} G(t_{w\alpha_1},\ldots,t_{w\alpha_h}) \leq \sum_{w\in\Sigma^k} G(n_{w\alpha_1},\ldots,n_{w\alpha_h}) = nH_k(T).$$

∎

Lemma 8 *For any $k \geq 0$ and for any text $T[1,n]$, the space occupancy of the data structure $\mathsf{Opp}(T_\$^R)$ is bounded by $5nH_k(T) + O\left(n\frac{\log\log n}{\log n}\right)$ bits.*

Proof: Since $|T_\$^R| = n + d$ and $d = O(n/\log n)$, by Theorem 2 we get

$$|\mathsf{Opp}(T_\$^R)| \leq 5|T_\$^R| H_k(T_\$^R) + O\left(n\frac{\log\log n}{\log n}\right).$$

Using Theorem 11 and Lemma 7 we get

$$\begin{aligned}|\mathsf{Opp}(T_\$^R)| &\leq 5|T_\$| H_k(T_\$) + O\left(n\frac{\log\log n}{\log n}\right)\\ &\leq 5nH_k(T) + O\left(n\frac{\log\log n}{\log n}\right)\end{aligned}$$

as claimed. ∎