Indexing Compressed Text

PAOLO FERRAGINA

Università di Pisa, Pisa, Italy

AND

GIOVANNI MANZINI

Università del Piemonte Orientale, Alessandria, Italy

Abstract. We design two compressed data structures for the full-text indexing problem that support efficient substring searches using roughly the space required for storing the text in compressed form.

Our first compressed data structure retrieves the *occ* occurrences of a pattern $P[1, p]$ within a text $T[1, n]$ in $O(p + occ \log^{1+\epsilon} n)$ time for any chosen ϵ, $0 < \epsilon < 1$. This data structure uses at most $5nH_k(T) + o(n)$ bits of storage, where $H_k(T)$ is the kth order empirical entropy of T. The space usage is $\Theta(n)$ bits in the worst case and $o(n)$ bits for compressible texts. This data structure exploits the relationship between suffix arrays and the Burrows–Wheeler Transform, and can be regarded as a *compressed suffix array*.

Our second compressed data structure achieves $O(p + occ)$ query time using $O(nH_k(T) \log^{\epsilon} n) + o(n)$ bits of storage for any chosen ϵ, $0 < \epsilon < 1$. Therefore, it provides optimal *output-sensitive* query time using $o(n \log n)$ bits in the worst case. This second data structure builds upon the first one and exploits the interplay between two compressors: the Burrows–Wheeler Transform and the `LZ78` algorithm.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management; 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.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing; H.3.2 [**Information Storage and Retrieval**]: Information Storage; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

Part of this work was published in *Proceedings of the 41st IEEE Symposium on the Foundations of Computer Science* (Redondo Beach, Calif.), IEEE Computer Society Press, Los Alamitos, Calif., 2000.

The work of P. Ferragina was partially supported by the Italian MIUR projects "Algorithmics for Internet and the Web (ALINWEB)", "Algorithms for the Next Generation Internet and Web (ALGO-NEXT)", and "Enhanced Content Delivery (ECD)".

The work of G. Manzini was partially supported by the Italian MIUR projects "Algorithmics for Internet and the Web (ALINWEB)" and "Enhanced Content Delivery (ECD)".

Authors' addresses: P. Ferragina, Dipartimento di Informatica, Largo B. Pontecorvo 3, I-56127 Pisa, Italy, e-mail: ferragina@di.unipi.it; G. Manzini, Dipartimento di Informatica, Via Bellini 25g, I-15100 Alessandria, Italy, e-mail: manzini@mfn.unipmn.it.

General Terms: Algorithms, Design, Theory

Additional Key Words and Phrases: Burrows–Wheeler transform, full-text indexing, indexing data structure, Lempel–Ziv compressor, pattern searching, suffix tree, suffix array, text compression

1. *Introduction*

We address the issue of *simultaneously* compressing and indexing data. We consider the full-text indexing problem: Given a text T we want to build a data structure which supports the efficient retrieval of the occurrences of an arbitrary pattern P. We describe two full-text indexes that use space bounded in terms of the empirical entropy of T, exploiting for the first time the regularity of the *indexed text* to reduce the size of the *index*. This result is achieved studying the interplay between the suffix array data structure [Manber and Myers 1993; Gonnet et al. 1992] and two compressors: the Burrows–Wheeler Transform [Burrows and Wheeler 1994] and the LZ78 algorithm [Ziv and Lempel 1978].

Let $T[1, n]$ denote a text drawn from a constant-size alphabet Σ. Let $P[1, p]$ denote an arbitrary pattern, and let occ be the number of occurrences of P as a substring of T. Our first data structure computes the number of occurrences in $O(p)$ time and reports the position of the occurrences in $O(p + occ \log^{1+\epsilon} n)$ time, where $0 < \epsilon < 1$ is an arbitrary constant chosen when we build the index. The model of computation is the RAM with a word size of $\Theta(\log n)$ bits. This data structure uses at most $5nH_k(T) + O(n/\log^{\epsilon} n)$ bits, where $H_k(T)$ is the kth order empirical entropy of T. This bound holds *simultaneously* for any $k \geq 0$ and we assume that k is a constant with respect to n since there is a $|\Sigma|^k$ term hidden in the big-O notation (the significance of the $|\Sigma|^k$ term is discussed in Section 2.1). Note that the kth order empirical entropy is defined for any text T and depends only on the *text structure*, that is, we are not assuming that the text is generated by an information source. We point out that the parameter k does not influence either the design or the usage of our data structure, but it plays a role only in the analysis of the space complexity. Since the empirical entropy $H_k(T)$ is at most $\log |\Sigma|$, our data structure uses $\Theta(n)$ bits in the worst case. However, the data structure can take advantage of the compressibility of T in order to use $o(n)$ space. For example, for $T = ab^{n-1}$ we have $nH_k(T) = O(\log n)$ and the data structure uses $o(n)$ bits. Note that in this extreme case the term $O(n/\log^{\epsilon} n)$ dominates the term $5nH_k(T)$.

Our index[1] exploits the relationship between the Burrows–Wheeler Transform [Burrows and Wheeler 1994] and the suffix array data structure [Manber and Myers 1993; Gonnet et al. 1992]. The key idea is to add some auxiliary information to a text compressed via the Burrows–Wheeler Transform to turn the compressed text into a full-text index. Our data structure is a *compressed suffix array* that encapsulates both the compressed text and the full-text indexing information. In Ferragina and Manzini [2000, Theorem 3] we have shown how to modify our index 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 here the description of the modified data structure because it is rather technical and does not introduce new algorithmic ideas.

[1] In the current literature (see, e.g., Ferragina et al. [2004], Grabowski et al. [2004], Grossi et al. [2003, 2004], and Hon et al. [2004b]) this first index is usually referred to as the *FM-index*, which stands for Full-text index in Minute space.

Our second contribution is a full-text index which reports the position of the occurrences in $O(p + occ)$ time using $O(nH_k(T)\log^{\epsilon} n) + O(n/\log^{1-\epsilon} n)$ bits of storage, where $0 < \epsilon < 1$ is an arbitrary constant chosen when we build the index. The space bound holds *simultaneously* for any $k \geq 0$; it is $o(n)$ bits for highly compressible texts and $O(n\log^{\epsilon} n)$ bits in the worst case. Hence, this is the first data structure in the literature to achieve $o(n\log n)$ space and optimal *output-sensitive* query time without any restrictions on p and occ (more on this in Section 1.1). The design of this second compressed index builds upon the first one and the interplay between two compressors: the Burrows-Wheeler Transform and the LZ78 algorithm [Ziv and Lempel 1978]. The key idea is to exploit text regularities not only for space reduction but also for speeding up the retrieval of the pattern occurrences.

1.1. RELATED RESULTS. The results of this article are similar in spirit to recent results on succinct representation of dictionaries. A dictionary over a universe of size n can be seen as a binary sequence B of length n. In Brodnik and Munro [1999] and Pagh [2001], the authors describe data structures that represent B in $nH_0(B) + o(n)$ bits and support membership queries in constant time. In Raman et al. [2002], these results have been extended to support also *Rank* and *Select* queries in constant time still using $nH_0(B) + o(n)$ bits. These results cannot be directly compared with ours since they refer to a different problem. We point out that, in the field of text indexing, achieving H_k rather than H_0 can save a significant amount of space. For example, for $T = (ab)^{n/2}$ we have $nH_0(T) = n$ and $nH_k(T) = 0$ for $k \geq 1$.

For the full-text indexing problem, the best-known data structures are suffix trees [McCreight 1976] and suffix arrays [Manber and Myers 1993; Gonnet et al. 1992]. They support the pattern search in $O(p + occ)$ time (respectively, $O(p + \log n + occ)$ time) using $\Theta(n\log n)$ bits of storage. The major drawback of these data structures is their large space usage and several authors have addressed the problem of reducing the constants hidden in the $\Theta(n\log n)$ term [Kurtz 1999; Clark and Munro 1996; Munro et al. 2001; Colussi and De Col 1996]. Other authors achieved asymptotically smaller space usage at the cost of either limiting the search to q-grams [Kärkkäinen and Sutinen 1998] or worsening significantly the query performance [Kärkkäinen and Ukkonen 1996].

Grossi and Vitter [2000] were the first to break the space barrier of $\Theta(n\log n)$ bits for a full-text index by proposing a succinct representation of the suffix array that uses $\Theta(n)$ bits. Their data structure supports the *lookup* of the ith entry of the suffix array in $O(\log^{\epsilon} n)$ time. Using this representation, the authors built a compressed full-text index that computes the number of occurrences in $O(p/\log n + \log^{\epsilon} n)$ time and reports the positions of the occurrences in additional $O(occ\log^{\epsilon} n)$ time (these results hold on a RAM with a word size of $\Theta(\log n)$ bits). This data structure can be further enriched to achieve $O(\frac{p}{\log n} + occ)$ query time when $p = \Omega(\log^3 n\log\log n)$ or $occ = \Omega(n^{\epsilon})$.

The index in Grossi and Vitter [2000] is not a self-index since it requires the text T in raw form. Sadakane [2000] proposed a variant of the index in Grossi and Vitter [2000] that also encapsulates the text. Sadakane's variant uses $\Theta(n)$ space and answers queries in $O(p\ \log n + occ\log^{\epsilon} n)$ time. Although this variant has a higher query time, the constants hidden in the $\Theta(n)$ space bound are smaller than in the index in Grossi and Vitter [2000].

The results of our article are theoretical in flavor. However, the compressed index described in Section 3 has been implemented and extensively tested in Ferragina and Manzini [2001]. The experiments show that our index, which also encapsulates the input text, takes roughly the same space used by the traditional compressors `gzip` and `bzip2` that store the input text *only*. The retrieval of the occurrences of an arbitrary pattern with our index takes a few milliseconds even within texts of several megabytes on a desktop PC. Finally, we point out that recent experimental studies have provided further evidence that compressed indexes are a valid alternative to suffix trees and suffix arrays in several application fields [Grossi et al. 2004; Hon et al. 2004b; Healy et al. 2003; Sadakane and Shibuya 2001].

1.2. STRUCTURE OF THE ARTICLE. Section 2 contains some background material. In Section 2.1, we discuss the notion of empirical entropy and its differences with respect to the Shannon's entropy. In Section 2.2, we describe the Burrows–Wheeler Transform and we review its basic properties. In Section 2.3, we introduce the compression algorithm `BW_RLX` that is based on the Burrows–Wheeler Transform and is at the hearth of our indexing data structures. In Section 3, we describe our first compressed index. This index has a relatively simple structure, as it consists of the compressed file returned by `BW_RLX` combined with two auxiliary data structures. The first one is used to *count* the number of pattern occurrences, the second one is used to *locate* the position in the indexed text of these occurrences. In Section 4, we describe our second compressed index that builds upon the first one and uses the `LZ78` parsing of the indexed text to speed up the retrieval of the pattern occurrences. In Section 5, we draw the conclusions of the article and discuss the current research on compressed full-text indexes. The appendices contain the proofs of some technical lemmas.

2. *Background 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 ith 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 article, we are interested in solving the *full-text indexing* problem. Given the text T we aim at building an indexing data structure (also called an *index*) that supports the 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 that its *substring* search operation is more powerful than the search for a word, or for a prefix of word, usually supported by traditional *word-based* indexes [Witten et al. 1999].

In this article, we split the pattern-searching process into two phases: *counting* the number of pattern occurrences, and *locating* their positions. The counting phase returns the single value *occ*, while the locating phase returns *occ* distinct values in the range $[1, n]$.

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

2.1. INDEXING AND ENTROPY. Let $\mathcal{I}$ denote a full-text index for a text 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 $i = 1, 2, \ldots$ This procedure eventually leads to the identification of the text T that is indexed by $\mathcal{I}$. In other words, $\mathcal{I}$ contains enough information to determine 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 the 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 symbol frequencies observed in T rather than in terms of symbol probabilities. Let $\Sigma = \{\alpha_1, \ldots, \alpha_h\}$ and let n_i denote the number of occurrences of the symbol α_i in T. The zeroth order empirical entropy of T is defined as

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

(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 symbol. We can achieve a greater compression if the codeword also depends on the k symbols immediately preceding the one to be encoded. For any $w \in \Sigma^k$ 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 (i.e., the number of occurrences of the string $w\alpha_i$ in T). Let $n_w = \sum_i n_{w\alpha_i}$. The value

$$H_k(T) = -\frac{1}{n} \sum_{w \in \Sigma^k} n_w \left[\sum_{i=1}^{h} \frac{n_{w\alpha_i}}{n_w} \log\left(\frac{n_{w\alpha_i}}{n_w}\right) \right] \tag{2}$$

is called the kth order empirical entropy of T. The value $nH_k(T)$ is a lower bound to the output size of any compressor that encodes each symbol with a code that only depends on the symbol itself and on the k immediately preceding symbols. Not surprisingly, for any $k \geq 0$ we have $H_k(T) \geq H_{k+1}(T)$. Note however that $nH_k(T)$ does not include the cost of describing the codewords. Hence, a more realistic lower bound would be the quantity $nH_k(T) + \Omega(|\Sigma|^k)$ that also accounts for the codewords description. We follow the established practice in Information Theory and we only use $nH_k(T)$.

We point out one important difference between empirical entropy and the Shannon's entropy defined in the probabilistic setting. Shannon's entropy is an expected value taken on an ensemble of strings, while empirical entropy is defined *pointwise* for any string and can be used to measure the performance of compression algorithms as a function of the *string structure*, thus without any assumption on the input source. In a sense, compression bounds produced in terms of empirical entropy are worst-case measures.

Another important difference is that for strings with many regularities the value $H_k(T)$ can be $o(1)$ whereas Shannon's entropy 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     ==>      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
```

FIG. 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`.

2.2. THE BURROWS–WHEELER TRANSFORM. Burrows and Wheeler [1994] 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 that is usually easier to compress. The BWT consists of three basic steps (see Figure 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 $T\#$ 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 also 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 #. This shows that there is a strong relation between $\mathcal{M}_T$ and the suffix array built on T. The matrix $\mathcal{M}_T$ has also other remarkable properties. To illustrate them, we introduce the following notation:

—$C[\cdot]$ denotes an 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 Figure 1, we have $C[\texttt{s}] = 8$ and $\mathsf{Occ}(\texttt{s}, 10) = 4$. The following properties of $\mathcal{M}_T$ have been proven in Burrows and Wheeler [1994]:

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

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

(c) If $T[k]$ is the ith character of L, then $T[k-1] = L[LF(i)]$. For example, in Figure 1 $T[3] = \mathtt{s}$ 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 retrieve T from L 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 $T = \mathtt{mississippi}$ of Figure 1 we start by setting $T[11] = L[1] = \mathtt{i}$. At the first iteration (i.e., $k = 11$ and $i = 1$), we get $T[10] = L[LF(1)] = L[C[\mathtt{i}] + \mathsf{Occ}(\mathtt{i}, 1)] = L[2] = \mathtt{p}$. At the second iteration (i.e., $k = 10$ and $i = LF(1) = 2$), we get $T[9] = L[LF(2)] = L[C[\mathtt{p}] + \mathsf{Occ}(\mathtt{p}, 2)] = L[7] = \mathtt{p}$, and so on.

2.3. A BWT-BASED COMPRESSION ALGORITHM. The output of the BWT is the last column L of the matrix $\mathcal{M}_T$. The BWT by itself is not a compression algorithm since L is just a permutation of $T\#$. However, consecutive characters in L are adjacent to similar strings in T. Hence, L usually contains long runs of identical characters and turns out to be highly compressible (see Burrows and Wheeler [1994], Fenwick [1996], and Manzini [2001] for details).

Several compression algorithms based on the BWT have been proposed since its introduction. To describe our results, we commit ourselves to a specific algorithm—called BW_RLX—which processes the string L in three steps (see also Manzini [2001, Sect. 5]):

(1) Use move-to-front encoding [Bentley et al. 1986] to replace 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 into runs of zeroes in the resulting string $L^{mtf} = \mathtt{mtf}(L)$. 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. 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 $L^{rle} = \mathtt{rle}(L^{mtf})$ is defined over the alphabet $\{\mathbf{0}, \mathbf{1}, 1, 2, \ldots, |\Sigma| - 1\}$.

(3) Compress L^{rle} with the following variable-length prefix code. Encode $\mathbf{0}$ and $\mathbf{1}$ using two bits (10 for $\mathbf{0}$, 11 for $\mathbf{1}$). For $i = 1, 2, \ldots, |\Sigma| - 1$, encode the symbol i using $1 + 2\lfloor \log(i+1) \rfloor$ bits: $\lfloor \log(i+1) \rfloor$ 0's followed by the binary representation of $i + 1$ that takes $1 + \lfloor \log(i+1) \rfloor$ bits. The resulting string is the final output of algorithm BW_RLX and is defined over the alphabet $\{0, 1\}$.

Despite the obvious inefficiencies of the above algorithm, we can bound its compression ratio in terms of the empirical entropy of the input text. In Appendix B, we show that for any $k \geq 0$ and for any text $T[1, n]$

$$|\mathtt{BW\_RLX}(T)| \leq 5\, n\, H_k(T) + O(\log n) \tag{3}$$

where $H_k(T)$ is the kth order empirical entropy of T. The bound (3) holds assuming that k and the alphabet size $|\Sigma|$ are constant with respect to the text size n since

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

FIG. 2. Algorithm backward_search 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 $L[1, q]$.

there is a $|\Sigma|^k$ term hidden in the big-O notation. As we pointed out in Section 2.1, the term $|\Sigma|^k$ represents an additional overhead that is naturally associated to the kth order entropy $H_k(T)$.

The possibility of bounding the size of BW_RLX(T) in terms of $H_k(T)$ simultaneously for any $k \geq 0$, is a remarkable property of this algorithm and shows the effectiveness of the BWT for data compression. The algorithm BW_RLX is at the hearth of our indexing data structures. This means that any improvement to the bound (3) automatically leads to an improvement to the space bounds for our compressed indexes.

3. *A Compressed Full-Text Index*

We distinguish two phases in the pattern searching process: counting the pattern occurrences (Sections 3.1 and 3.2), and locating their positions in T (Section 3.3). This distinction has a twofold advantage: It simplifies the presentation and shows that the locating phase builds on the top of the counting phase. The additional data structures needed for locating the pattern occurrences may be dropped if we are only interested in simple queries like, for example, "Does P occur in T?" or "How many times does P occur in T?".

3.1. THE BACKWARD_SEARCH PROCEDURE. In this section, we describe a simple algorithm for counting the number of occurrences of P in T using *only* the matrix $\mathcal{M}_T$ introduced in Section 2.2. Our algorithm exploits the following two properties of the matrix $\mathcal{M}_T$: (i) all suffixes of T prefixed by the 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 (Last $-$ First $+ 1$) gives the total number of pattern occurrences in T. For example, in Figure 1, for $P = \texttt{si}$, we have First $= 9$ and Last $= 10$ for a total of two occurrences.

The procedure backward_search works in p phases numbered from p to 1 (see the pseudo-code in Figure 2). Each phase preserves the following invariant: *At the end of the ith 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 are determined via the array C defined in Section 2.2. After the final phase, First and Last delimit the rows of $\mathcal{M}_T$ containing all the text suffixes prefixed by P. The following lemma proves the correctness of backward_search.

LEMMA 3.1. *For $i = p, p-1, \ldots, 2$, let us assume that* First *(respectively,* Last*) stores the position of the first (respectively, last) row in $\mathcal{M}_T$ prefixed by $P[i, p]$. If $P[i-1, p]$ occurs in T, then Step* (4) *(respectively, Step* (5)*) of* backward_search *correctly updates the value of* First *(resp.* Last*) thus pointing to the first (respectively, 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* backward_search *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 $s = C[c] + \mathrm{Occ}(c, \mathrm{First} - 1) + 1$.

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 First $\leq s'$. We now show that s' is indeed the first row that follows the position First and ends with the character c (i.e. $L[s'] = c$ and $L[j] \neq c$ for $j = \mathrm{First}, \ldots, s'-1$). If not, we would have a row $\mathcal{M}_T[t']$ occurring between $\mathcal{M}_T[\mathrm{First}]$ 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 $\mathcal{M}_T[\mathrm{First}]$ 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 2.2, we know that $s = LF(s')$. That is, $s = C[c] + \mathrm{Occ}(c, s')$. Since s' is the first row following First and ending with c, we have $s = C[c] + \mathrm{Occ}(c, \mathrm{First} - 1) + 1$ 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 $\mathcal{M}_T[\mathrm{First}]$ and $\mathcal{M}_T[\mathrm{Last}]$ 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 $L[\mathrm{First}, \mathrm{Last}]$. Hence, $\mathrm{Occ}(c, \mathrm{First} - 1) = \mathrm{Occ}(c, \mathrm{Last})$ and after Step (5) we then have Last $<$ First as claimed. □

The running time of the procedure backward_search is dominated by the cost of evaluating $\mathrm{Occ}(c, q)$. Therefore, if we build a two-dimensional array OCC such that $OCC[c][q] = \mathrm{Occ}(c, q)$ the procedure backward_search runs in $O(p)$ time. Note that OCC requires $O(|\Sigma|\, n \log n) = O(n \log n)$ bits. Hence, combining OCC with the suffix array of T, we get a data structure of size $O(n \log n)$ bits with query time $O(p + occ)$. The use of OCC is a valid alternative to the classical approach of enriching the suffix array with the LCP array. We summarize these observations with the following corollary.

COROLLARY 3.2. *Combining the suffix array of $T[1, n]$ with the array OCC and using the* backward_search *procedure, we get a full-text index of size $O(n \log n)$ bits and query time $O(p + occ)$.* □

3.2. COUNTING THE PATTERN OCCURRENCES IN $O(p)$ TIME. In this section, we describe an implementation of the backward_search procedure that runs in $O(p)$ time and uses $5nH_k(T) + o(n)$ bits. Our solution makes use of the string $Z = \texttt{BW\_RLX}(T)$ (the output of BW_RLX on input T) and of an auxiliary data structure of size $o(n)$ supporting the computation of $\mathsf{Occ}(c, q)$ in $O(1)$ time. We compute $\mathsf{Occ}(c, q)$ using the technique called *word-size truncated recursion* (also known as the *four Russians trick*, see, e.g., Brodnik and Munro [1999]). The idea is to split the problem into "small enough" subproblems that are solved by indexing into a table of all solutions.

We logically partition L into substrings of $\ell = \Theta(\log n)$ characters[2] 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. 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 variable-length *compressed buckets* $BZ_1, \ldots, BZ_{n/\ell}$, where each BZ_i is obtained from BL_i^{mtf} applying run-length encoding and the prefix free code described in Section 2.3.

To compute $\mathsf{Occ}(c, q)$ we logically split $L[1, q]$ into the following substrings:

(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 $L[1, q]$. Note that this last substring is a prefix of the bucket containing the character $L[q]$.

We compute $\mathsf{Occ}(c, q)$ by summing the number of occurrences of c in each one of these substrings. For example, if $\ell = 20$, we compute $\mathsf{Occ}(c, 1393)$ summing the number of occurrences of c in $L[1, 1200]$, $L[1201, 1380]$, and $L[1381, 1393]$. To compute the number of occurrences of c into these substrings, we use the following arrays and tables.

—For the substring of point (i):
- For $j = 1, \ldots, n/\ell^2$, the array $NO_j[1, |\Sigma|]$ stores in $NO_j[c]$ the number of occurrences of c in $L[1, j\,\ell^2]$.
- The array $W[1, n/\ell^2]$ stores in $W[j]$ 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 $L[1, j\,\ell^2]$.

—For the substring of point (ii):
- For $j = 1, \ldots, n/\ell$, the array $NO'_j[1, |\Sigma|]$ stores in $NO'_j[c]$ the number of occurrences of c in the string $L[\hat{\jmath}\ \ell^2 + 1, j\,\ell]$, where $\hat{\jmath} = \lceil j/\ell \rceil - 1$. In

[2] 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.

other words, $NO'_j[1, |\Sigma|]$ stores the number of occurrences of each character between position $j\ell$ and the closest position to the left that is 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 $W'[j]$ 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 substring of point (iii):

- The array $MTF[1, n/\ell]$ stores in $MTF[j]$ 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 c among the first h characters of BL_j, where $h \le \ell$. 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.

To compute $\mathsf{Occ}(c, q)$, we first determine the bucket BL_i containing the character $c = L[q]$ by setting $i = \lceil q/\ell \rceil$. Then, we compute the position $h = q - (i-1)\ell$ of this character within BL_i and the parameter $t = \lceil i/\ell \rceil - 1$. The number of occurrences of c in the prefix $L[1, t\ell^2]$ (point (i)) is given by $NO_t[c]$. Subsequently, we compute the number of occurrences of c in the substring $L[t\ell^2 + 1, (i-1)\ell]$ (point (ii)): this number is zero if $(i-1)$ is a multiple of ℓ; otherwise, it is $NO'_{i-1}[c]$. Finally, we retrieve the compressed bucket BZ_i from Z. The starting position of BZ_i in Z is $W[t]+1$ if $(i-1)$ is a multiple of ℓ, otherwise it is $W[t]+W'[i-1]+1$. Given BZ_i we retrieve the number of occurrences of c within $L[(i-1)\ell+1, q] = BL_i[1, h]$ (point (iii)). This number is the value stored in $S[c, h, BZ_i, MTF[i]]$. Summing up, we have the following result.

LEMMA 3.3. *We can compute* $\mathsf{Occ}(c, q)$ *in* $O(1)$ *time using* $|Z| + O(n \frac{\log \log n}{\log n})$ *bits of storage, where* Z *is the output of the algorithm* BW_RLX *on input* $T[1, n]$.

PROOF. An easy computation shows that each compressed bucket BZ_i 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 above algorithm consists of arithmetic operations or table lookups involving $O(\log n)$-bit operands. Consequently, the computation of $\mathsf{Occ}(c, q)$ takes constant time.

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 $O((n/\ell)|\Sigma| \log |\Sigma|) = O(n/\log n)$ bits. Table S consists of $O(\ell 2^{\ell'})$ $\log \ell$-bit entries, and thus it uses $O(2^{\ell'} \ell \log \ell) = O(n^\gamma \log n \log \log n)$ bits, where $\gamma < 1$. Hence, the data structures for computing $\mathsf{Occ}(c, q)$ takes $O((n \log \log n)/\log n)$ bits and the lemma follows. □

Combining Lemma 3.3 with Lemma 3.1 and inequality (3) we get the main result of this section.

THEOREM 3.4. *Using the* backward_search *procedure we can compute the number of occurrences of a pattern* $P[1, p]$ *in* $T[1, n]$ *in* $O(p)$ *time using space bounded by* $5nH_k(T) + O(n\,\frac{\log\log n}{\log n})$ *bits, for any* $k \geq 0$.

In the remainder of the article, we use $\mathsf{Opp}(T)$ to denote the collection of data structures used by algorithm backward_search.[3] For future reference, we summarize the results of this section in terms of the data structure $\mathsf{Opp}(T)$.

THEOREM 3.5. *For any string* T *let* $\mathsf{Opp}(T)$ *denote the collection of data structures used by* backward_search. *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* $5nH_k(T) + O(n\,\frac{\log\log n}{\log n})$ *bits, for any* $k \geq 0$.

3.3. LOCATING THE PATTERN OCCURRENCES. Assume that, using algorithm backward_search, we determined the range $\langle \mathsf{First}, \mathsf{Last}\rangle$ of rows of the matrix $\mathcal{M}_T$ 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 ith row $\mathcal{M}_T[i]$. In the following, we write $\mathsf{Pos}(i)$ to denote such a suffix position. For example, in Figure 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 shows that, given a row index i, we can compute in constant 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 backward_step (see Figure 3).

LEMMA 3.6. *Given a row index* i*, algorithm* 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* backward_step *uses the data structure* $\mathsf{Opp}(T)$ *and runs in* $O(1)$ *time.*

PROOF. By properties **b.** and **c.** of Section 2.2 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 since L is compressed. Thus, algorithm backward_step determines $L[i]$ by comparing $\mathsf{Occ}(c, i)$ and $\mathsf{Occ}(c, i - 1)$ for every $c \in \Sigma \cup \{\#\}$ (Step (1)). Clearly, the values $\mathsf{Occ}(c, i)$ and $\mathsf{Occ}(c, i - 1)$ differ only for $c = L[i]$. Therefore, the above computation gives us both $L[i]$ and $\mathsf{Occ}(L[i], i)$. If $L[i] = \#$ then backward_step reports that $\mathsf{Pos}(i) = 1$ (Step (2)). Otherwise, backward_step returns $j = C[L[i]] + \mathsf{Occ}(L[i], i)$ (Step (3)). By Lemma 3.3, we know that using $\mathsf{Opp}(T)$ each computation of $\mathsf{Occ}()$ takes constant time. Since $|\Sigma| = \Theta(1)$, backward_step takes $O(1)$ time as claimed. □

[3] Opp stands for opportunistic. The concept of *opportunistic algorithm* has been introduced in Farach and Thorup [1998] 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 its space usage.

```
Algorithm backward_step($i$)
Compute $L[i]$ comparing $\mathsf{Occ}(c, i)$ with $\mathsf{Occ}(c, i-1)$ for every $c \in \Sigma \cup \{\#\}$.
if ($L[i] = \#$) then return "$\mathsf{Pos}(i) = 1$";
else return $C[L[i]] + \mathsf{Occ}(L[i], i)$;

Algorithm get_position($i$)
$i' \leftarrow i$, $t \leftarrow 0$;
while row $i'$ is not marked do
    $i' \leftarrow$ backward_step($i'$);
    $t \leftarrow t + 1$;
return $\mathsf{Pos}(i') + t$;
```

FIG. 3. Algorithms backward_step and get_position.

Let $\eta = \lceil \log^{1+\epsilon} n \rceil$. We *logically mark* the rows of $\mathcal{M}_T$ corresponding to text positions of the form $1 + j\eta$, for $j = 0, 1, \ldots, \lfloor n/\eta \rfloor$. In other words, we mark each row 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 3.6). 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 $\mathsf{Pos}(i) = \mathsf{Pos}(i') + t$. We call this algorithm get_position and its pseudo-code is shown in Figure 3.

Note that our marking strategy ensures that a marked row is found in at most η iterations. Since each iteration takes constant time, a single call to get_position takes $O(\eta) = O\big(\log^{1+\epsilon} n\big)$ time. Retrieving the *occ* occurrences of P in T takes $O\big(occ \log^{1+\epsilon} n\big)$ time.

The crucial point of algorithm get_position is the data structure $\mathcal{S}$ storing the positions of the marked rows. $\mathcal{S}$ must be designed carefully in order to use succinct space and 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 in it and store them in a Packet B-tree [Andersson 1996] using their distance from the beginning of the stripe as the key. Since a stripe contains at most $O(\log^2 n)$ keys, each $O(\log \log n)$ bits long, membership queries take constant time. Each Packed B-tree uses space proportional to the number of stored keys. $\mathcal{S}$ uses overall $O((n/\eta)(\log \log n + \log n))$ bits since with each marked row r_j we also keep the value $\mathsf{Pos}(r_j)$ using $O(\log n)$ bits. We conclude that algorithm get_position uses $O(n/\log^{\epsilon} n)$ bits for $\mathcal{S}$ plus the space used by the data structure $\mathsf{Opp}(T)$.

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

THEOREM 3.7. *For any text $T[1, n]$ we can build a compressed index such that all the occ occurrences of a pattern $P[1, p]$ in T can be retrieved in $O\big(p + occ \log^{1+\epsilon} n\big)$ time, where $0 < \epsilon < 1$ is an arbitrary constant chosen when we build the data structure. The space usage of this data structure is bounded by $5nH_k(T) + (\frac{n}{\log^\epsilon n})$ bits, for any $k \geq 0$.*

Algorithm get_position can be improved in order to list all the occurrences of P in $O(p + occ \log^\epsilon n)$ time using $(nH_k(T) + n\,\frac{\log\log n}{\log^\epsilon n})$ bits of storage. The improvement is based on two basic ingredients: (i) a generalization of Lemma 3.6 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 the improved algorithm since they are rather technical. The interested reader can find in Ferragina and Manzini [2000, 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 3.3 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 [Ziv and Lempel 1978]. 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, with the possible exception of the last word in the parsing.[4] In addition, the set $\mathcal{D}$ is *prefix-complete*, that is, if $T_i \in \mathcal{D}$ then all the nonempty 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 kth order empirical entropy of T. In Kosaraju and Manzini [1999], it is shown that, for any string $T[1, n]$ and for any $k \geq 0$, we have

$$d\log d \leq nH_k(T) + O((n\log\log n)/\log n)\,. \tag{4}$$

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 (4), we get the following useful inequality

$$d\log n = d\log d + d\log(n/d) \leq nH_k(T) + O((n\log\log n)/\log n)\,. \tag{5}$$

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:

$$T_\$ = T_1\$T_2\$\cdots\$T_d\$. \tag{6}$$

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

The reason for which we introduce $T_\$$ is the following. The characters \$'s in $T_\$$ will be *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 3.3. We immediately point out that storing the positions of the anchor points requires $d \log n$ bits; by (5), this amount of space is bounded by $nH_k(T) + o(n)$ bits.

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 2.2). 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 $\langle \mathsf{First}, \mathsf{Last} \rangle$ of the matrix $\mathcal{M}_{T_\$^R}$ which are prefixed by P' (see Theorem 3.5). In Appendix D, we show that the size of $\mathsf{Opp}(T_\$^R)$ is bounded by $5nH_k(T) + O(n\frac{\log\log n}{\log n})$ 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 that 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 labelled 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 LZ78-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.

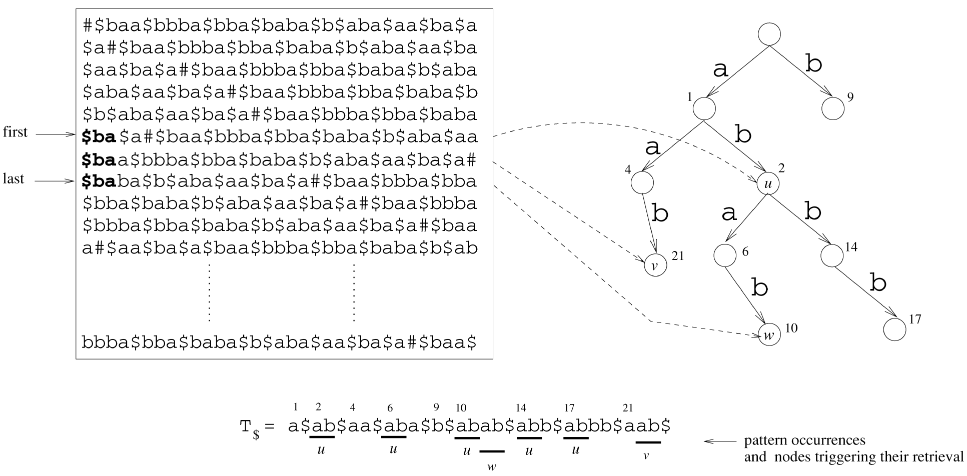
#$baa$bbba$bba$baba$b$aba$aa$ba$a
$a#$baa$bbba$bba$baba$b$aba$aa$ba
$aa$ba$a#$baa$bbba$bba$baba$b$aba
$aba$aabaa#baabbbabbababa$b
$b$aba$aa$ba$a#$baa$bbba$bba$baba
first
baa#baabbbabbababa$b$aba$aa
$baa$bbba$bba$baba$b$aba$aa$ba$a#
last
$baba$b$aba$aabaa#baabbba$bba
bbababa$b$aba$aa$ba$a#$baa$bbba
$bbba$bba$baba$b$aba$aabaa#$baa
a#$aa$ba$a$baa$bbba$bba$baba$b$ab
bbbabbababa$b$aba$aa$ba$a#$baa$
a
b
1
9
a
b
4
2
u
b
a
b
21
v
6
14
b
b
w
10
17
1 2 4 6 9 10 14 17 21
T$ = a$ab$aa$aba$b$abab$abb$abbb$aab$
u u u w u u v
pattern occurrences
and nodes triggering their retrieval

FIG. 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 LZ78-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 rows 6, 7, and 8. The three pointers $\mathcal{N}[6], \mathcal{N}[7], \mathcal{N}[8]$, shown as dashed arrows, lead to the corresponding trie nodes u, v, w. These nodes spell out the three words suffixed by P, that is, 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 we show the node triggering its retrieval. Notice that the LZ78-word abab contains two occurrences both correctly detected by visiting the two subtries rooted one at node u and the other at node w.

We now show how to efficiently implement the above strategy. The first step consists in locating the trie nodes corresponding to the words which have P as a suffix. We solve this problem using the matrix $\mathcal{M}_{T_\$^R}$. Since each row of this matrix contains a cyclic shift of the string $T_\$^R$, there is 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 Figure 4).[5] 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 $\mathcal{M}_{T_\$^R}$.

This property of $\mathcal{M}_{T_\$^R}$ suggests the introduction of an array $\mathcal{N}[1, d]$ which stores in $\mathcal{N}[i]$ a pointer to the trie node corresponding to the ith row of $\mathcal{M}_{T_\$^R}$ prefixed by \$. Given the pattern P we use the backward_search procedure of Section 3.1 to retrieve the range $\langle$First, Last$\rangle$ of rows of $\mathcal{M}_{T_\$^R}$ prefixed by $\$P^R$. These rows correspond to the words $T_{i_1}, \ldots, T_{i_z}$ ending with P. The trie nodes corresponding to these words are those pointed by $\mathcal{N}$[First], ..., $\mathcal{N}$[Last]. See Figure 4 for an example.

Our algorithm for retrieving the internal occurrences is summarized in Figure 5. To prove its correctness, it remains to be shown that each value $v_j + (|T_{i_k}| - p)$

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

```
Algorithm get_internal($P[1, p]$)
(1) Search for $\$P^R$ in $\mathcal{M}_{T^R_\$}$ 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 visited word $T_j$ return the value $v_j + (|T_{i_k}| - p)$, where $v_j$ is the starting position of $T_j$ in $T$.
```

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

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 now that the visit of the subtrie rooted at T_{i_k} reaches the word T_j, and let v_j denote the starting position of T_j in T (recall that v_j is stored into the trie node T_j). We have $T_j = T_{i_k}w' = wPw'$ for some string w'. Hence, the occurrence of P contained in T_j starts at position $v_j + |w| = v_j + (|T_{i_k}| - p)$ as claimed.

Note that P may occur more than once in a dictionary word. For example, we could 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 4.1. *Let occ_I denote the number of internal occurrences of $P[1, p]$ in $T[1, n]$. The algorithm* get_internal *retrieves the internal occurrences in $O(p + occ_I)$ time.* get_internal *uses space bounded by $O(nH_k(T)) + O((n \log \log n)/ \log n)$ bits for any $k \geq 0$.*

PROOF. We first analyze the time complexity. In Step (1), we use the backward_search procedure with the Opp($T^R_\$$) data structure. By Theorem 3.5, this takes $O(p)$ time. At Step (2), we output the position of one internal occurrence for each visited word. Hence, Step (2) takes $O(occ_I)$ time overall.

To prove the bound on the space complexity, we observe 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 Opp($T^R_\$$). The first three items all take $O(d \log n)$ bits which is $O(nH_k(T)) + O((n \log \log n)/ \log n)$ bits by (5). In Appendix D, we show that 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 4.2. *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 $T_j \cdots T_d$.*

```
Algorithm get_overlapping($P[1,p]$)
(1) For $m = p, p-1, \dots, 1$, search for $P[m,p]$ in Opp($T$) thus retrieving the range $\langle f_m, \ell_m \rangle$
    of rows in $\mathcal{M}_T$ prefixed by $P[m,p]$.
(2) For $m = 1, 2, \dots, p$, search for $\$P[1,m]^R$ in Opp($T_\$^R$) thus retrieving the range
    $\langle f_m^*, \ell_m^* \rangle$ of rows in $\mathcal{M}_{T_\$^R}$ prefixed by $\$P[1,m]^R$.
(3) For $m = 1, 2, \dots, p-1$ 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 $T$.
```

FIG. 6. Algorithm get_overlapping for finding the overlapping occurrences of $P[1, p]$ in $T[1, n]$.

Reasoning as in Section 4.1, we call procedure backward_search($\$P[1, m]^R$) over Opp($T_\$^R$) and thus retrieve the rows of the matrix $\mathcal{M}_{T_\$^R}$ prefixed by $\$P[1, m]^R$ for any $1 \leq m < p$. These rows corresponds to the LZ78-words which have $P[1, m]$ as a suffix. Similarly, calling the procedure backward_search($P[m + 1, p]$) over Opp(T), we can retrieve all suffixes of T which are prefixed by $P[m + 1, p]$. By Property 4.2, to find the overlapping occurrences, we 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 $i = 1, \dots, d$, 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 $T_i T_{i+1} \cdots T_d\#$. Define the set of 2D-grid points $\mathcal{Q} = \{(x_1, y_1), (x_2, y_2), \dots, (x_d, y_d)\}$. Each pair (x_i, y_i) "captures" the text surrounding the ith symbol \$. We can therefore restate Property 4.2 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 Figure 6.

To prove the correctness of get_overlapping, assume the point (x_j, y_j) is retrieved at the mth 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 4.2 it follows that there exists an occurrence of P that starts inside T_{j-1}, overlaps $T_j \cdots T_{j+h-1}$ and ends inside T_{j+h}, for some $h \geq 0$. Since $P[1, m]$ is a suffix of T_{j-1}, this occurrence starts at position $v_j - m$, which is the value correctly returned at Step (4). With a similar argument one can prove that an occurrence of P that satisfies Property 4.2 triggers the retrieval of the point (x_j, y_j) at the mth 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*. Assume for example that there is an 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 \$ that follows T_j is not retrieved by 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 cannot contain the special symbol \$).

To analyze the complexity of get_overlapping we make use of the following result, proven in Alstrup et al. [2000].[6]

THEOREM 4.3. *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 4.4. *Let occ_O denote the number of overlapping occurrences of $P[1, p]$ in $T[1, n]$. Algorithm* get_overlapping *retrieves them in $O(p \log\log n + occ_O)$ time. The data structures used by* get_overlapping *take $O(nH_k(T)\log^\epsilon n) + O(n/\log^{1-\epsilon} n)$ bits of storage overall, where $0 < \epsilon < 1$ is an arbitrary constant chosen when we build the data structures.*

PROOF. We first analyze the time complexity. At Step (1) with a single execution of backward_search with input $P[1, p]$ we compute the values $f_1, \ell_1, \ldots, f_p, \ell_p$ in $O(p)$ time. Similarly at Step (2), we compute the values $f_1^*, \ell_1^*, \ldots, f_p^*, \ell_p^*$ in $O(p)$ time. By Theorem 4.3, 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). We conclude that the running time is $O(p \log\log n + occ_O)$.

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 3.5 and Lemma D.2 in Appendix D, we know that $\mathsf{Opp}(T)$ and $\mathsf{Opp}(T_\$^R)$ both use $5nH_k(T) + O((n \log\log n)/\log n)$ bits. By Theorem 4.3, 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 (5) and taking $\gamma < \epsilon$, we get $d(\log^\gamma d)(\log n) = O(nH_k(T)\log^\epsilon n) + O(n/\log^{1-\epsilon} n)$ 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 $p \le (\log\log n)$, 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.*

We retrieve long patterns 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

[6] The result in Alstrup et al. [2000] 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., Alstrup et al. [2000, Sect. 2.2]), we can apply the result of Alstrup et al. [2000] with a slowdown in the query of $O(\log\log n)$ time.

```
Algorithm get_overlapping_long($P[1,p]$)
(1) For $m = p, p-1, \dots, 1$, search for $P[m,p]$ in $\mathsf{Opp}(T)$ thus retrieving the range $\langle f_m, \ell_m \rangle$
    of rows in $\mathcal{M}_T$ prefixed by $P[m,p]$.
(2) For $m = 1, 2, \dots, p$, search for $\$P[1,m]^R$ in $\mathsf{Opp}(T_\$^R)$ thus retrieving the range
    $\langle f_m^*, \ell_m^* \rangle$ of rows in $\mathcal{M}_{T_\$^R}$ prefixed by $\$P[1,m]^R$.
(3) For $i = 0, 1, \dots, \lfloor p/(\log\log n) \rfloor$, 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 $(x,y)$.
```

FIG. 7. Algorithm get_overlapping_long for retrieving the overlapping occurrences of long patterns.

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 4.2:

PROPERTY 4.5. *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 4.5 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 4.5, we can find the overlapping occurrences of long patterns using the data structure $\mathcal{RT}$ of Theorem 4.3 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 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 $i = 0, 1, \dots, \lfloor p/(\log\log n) \rfloor$. The resulting algorithm, called get_overlapping_long, is shown in Figure 7.

THEOREM 4.6. *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 $O(nH_k(T) \log^\epsilon n) + O(n/\log^{1-\epsilon} n)$ bits of storage, where $0 < \epsilon < 1$ 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 4.5 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.

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 get_overlapping_long, we first observe that $\mathcal{Q}'$ contains $O(d\log\log n)$ points. From Theorem 4.3, we get that $\mathcal{RT}(\mathcal{Q}')$ uses $O(d(\log^\gamma n)\log\log n)$ memory-words and answers orthogonal range-queries in $O(\log\log n + q)$ time, where q is the number of retrieved points. Hence, Step (3) takes overall $O(\lceil \frac{p}{\log\log n} \rceil \log\log n + occ_O) = O(p + occ_O)$ time. Since Steps (1) and (2) take $O(p)$ time, and Step (4) takes $O(occ_O)$ time, we conclude that the running time of get_overlapping_long is $O(p + occ_O)$ as claimed.

Reasoning as in the proof of Theorem 4.4 we get that the space complexity of the algorithm is dominated by the storage of the data structure $\mathcal{RT}(\mathcal{Q}')$ which takes overall $O(d(\log^\gamma n)(\log n)\log\log n)$ bits. Using (5) and taking $\epsilon > \gamma$, we get $d(\log^\gamma n)(\log n)\log\log n = O(nH_k(T)\log^\epsilon n) + O(n/\log^{1-\epsilon} n)$ 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 such occurrences and it is therefore feasible to 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 overlap two or more words. 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 $(d(\log\log n)^2)$ 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 (5), this amount is bounded by $O(n\, H_k(T)\, (\log\log n)^2) + O(n(\log\log n)^3/\log n)$ 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 $O(|\Sigma|^{\log\log n} \log n) = O(\text{polylog}(n))$ 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 4.7. *If the pattern $P[1, p]$ is short, we can retrieve the occ_O overlapping occurrences of P in $T[1, n]$ in $O(p + occ_O)$ time. The data structures supporting the retrieval of short patterns take overall $O(nH_k(T)(\log\log n)^2) + O(n(\log\log n)^3/\log n)$ bits of storage.* □

Finally, combining Theorem 4.1 (for the retrieval of the internal occurrences) with Theorems 4.6 and 4.7 (for the retrieval of the overlapping occurrences), we get the following result.

THEOREM 4.8. *For any text $T[1, n]$ we can build a compressed full-text index with $O(p + occ)$ query time. The size of the index is bounded by $O(nH_k(T) \log^\epsilon n) + O(n/\log^{1-\epsilon} n)$ bits for any $k \geq 0$, where $0 < \epsilon < 1$ is an arbitrary positive constant chosen when we build the index.* □

5. *Conclusions and Open Problems*

In this article, we described two full-text indexes whose size is bounded in terms of the kth order empirical entropy of the indexed text. This result shows that it is possible to simultaneously compress and index textual data. Subsequent to our work, there have been several new results on compressed full-text indexes. A new ingredient of recent research has been a greater attention to the influence of the alphabet size $|\Sigma|$ on the space and time bounds. The bounds given in this article assume that the alphabet size is constant with respect to n. Hidden in the big-O notation, there is an exponential dependence on the alphabet size for the space bounds, and a linear dependence on the alphabet size for the time bounds. The recent research on compressed indexes has produced data structures that are more "alphabet-friendly" and achieve various tradeoffs between space usage and query time [Grossi et al. 2003; Rao 2002; Sadakane 2002, 2003; Grabowski et al. 2004; Navarro 2004; Mäkinen et al. 2004; Mäkinen and Navarro 2004].

Currently, the most space economical compressed indexes [Grossi et al. 2003; Ferragina et al. 2004] take $nH_k(T) + o(n)$ bits for $k < \alpha \log_{|\Sigma|} n$ with $\alpha < 1$. These data structures report the pattern occurrences in $O(\log|\Sigma|(p + \text{polylog}(n) + occ \log^2 n/\log\log n))$ time. These "second generation" compressed indexes make use of new algorithmic tools such as succinct dictionaries [Raman et al. 2002], wavelet trees [Grossi et al. 2003] and compression boosting [Ferragina et al. 2005].

Despite these recent developments, the field of compressed indexes still offers many challenging open problems. For example, it is still open whether we can achieve $O(p + occ)$ query time using $O(nH_k(T)) + o(n)$ bits of storage. Another challenging question is how *locality of reference* may be introduced in compressed indexes to take advantage of the blocked access to disk memory. We aim at achieving

$O(p + occ/B)$ I/Os for the pattern search, where B is the disk-page size. Such an index might compete successfully with the String B-tree [Ferragina and Grossi 1999] that is the I/O-fastest full-text index but uses $\Theta(n \log n)$ bits of storage. Finally, it would be interesting to extend the operations supported by compressed full-text indexes to include text updates, regex, and approximate searches. Some preliminary results in this direction are reported in Ferragina and Manzini [2000], Chan et al. [2004], Hon et al. [2004a], and Huynh et al. [2004].

Appendix

A. *Some Properties of the Empirical Entropy $H_k(T)$*

A.1. $H_k(T)$ VERSUS $H_k(T^R)$. 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: In this section, we 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 $T = \texttt{aabaa}$, we have $m_{\texttt{aa}} = 3$, and $m_{\texttt{aaa}} = 2$. Note that $\sum_{w\in\Sigma^k} m_w = n$. We define the value $\hat{H}_k(T)$ as

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

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

LEMMA A.1. *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,

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

LEMMA A.2. *For any $k > 0$ and for any string $T[1, n]$, we have*

$$H_k(T) \le \hat{H}_k(T) \le H_k(T) + \left(\frac{k}{n}\right) \log(ne).$$

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 $G(t_1, \ldots, t_h) \ge 0$ 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

$$\begin{aligned} G(t_1, t_2, \ldots, t_h) &\le G(t_1 + 1, t_2, \ldots, t_h) & (9)\\ &\le G(t_1, t_2, \ldots, t_h) + \log(1 + t_1 + \cdots + t_h) + \log e. & (10) \end{aligned}$$

By (2) and (7) we have that using G we can rewrite the values $H_k(T)$ and $\hat{H}_k(T)$ as

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

$$\hat{H}_k(T) = \frac{1}{n} \sum_{w \in \Sigma^k} G(m_{w\alpha_1}, \ldots, m_{w\alpha_h}). \tag{12}$$

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 A.3. *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 A.1 and A.2, 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. □

A.2. $H_k(T)$ VERSUS $H_k^*(T)$. The empirical entropy H_k is not the only reasonable measure of the compressibility of a string. Recalling the function G defined by (8), we define

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

The kth order *modified* empirical entropy $H_k^*(T)$ was defined in Manzini [2001] as

$$H_k^*(T) = \frac{1}{n} \sum_{w \in \Sigma^k} G^*(n_{w\alpha_1}, \ldots, n_{w\alpha_h}). \tag{13}$$

The value $H_k^*(T)$ represents the maximum compression we can achieve using for each symbol a code which only depends the k immediately preceding k symbols, with the additional assumption that the coding of a string takes at least enough bits to write down its length in binary. This additional assumption makes $nH_k^*(T)$ a more realistic lower bound than $nH_k^*(T)$. The modified empirical entropy $H_k^*(T)$ has been used in Manzini [2001] and Ferragina et al. [2005] for the analysis of BWT-based compression algorithms. The following lemma bounds the entropy $H_k^*(T)$ in terms of $H_k(T)$.

LEMMA A.4. *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 (11)–(12) with (13), 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

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

B. *The Compression Ratio of Algorithm* BW_RLX

In this appendix, we prove the bound (3) on the compression achieved by the algorithm BW_RLX. Our proof is based on some results about BWT-based compressors given in Manzini [2001].

Following the notation in Manzini [2001, Sec. 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 (run-length encoding) followed by PC (the variable length prefix code described in Section 2.3). 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 Manzini [2001], we know that for every k, the output size $|\mathtt{A}^*(\mathtt{bwt}(T))|$ is bounded by $5nH_k^*(T) + g_k$, where $H_k^*(T)$ is the modified empirical entropy defined in Section A.2 and $g_k = |\Sigma|^{k+1} \log |\Sigma|$. 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 A.4, 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 $\mathtt{Occ}(c, q)$ in the general case in which L^{mtf} contains one or more runs of zeroes that cross the border of the buckets BL_i^{mtf}'s. Recall that run-length encoding 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.

```
if $c$ is the first character in $MTF[i]$ then
    if $j \le MZ[i]$ then return $j$
    else return $MZ[i] + S[c, j - MZ[i], BZ_i, MTF[i]]$
else
    if $j \le MZ[i]$ then return $0$
    else return $S[c, j - MZ[i], BZ_i, MTF[i]]$
```

FIG. 8. Procedure for computing the number of occurrences of a character c among the first j characters of the ith bucket BL_i.

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 run-length encoding 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 3.2 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 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 $MTF[i]$ (i.e., the first character in the MTF-list when the ith 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 Figure 8. Since the array MZ takes $O((n/\ell)\log \ell) = O((n/\log n)\log\log n)$ bits, its introduction does not change the asymptotic space used by the procedure backward_search.

D. *Size of* Opp($T_\$^R$)

Let $T[1, n]$ be a string over the alphabet $\Sigma = \{\alpha_1, \dots, \alpha_h\}$, and let $T = T_1 T_2 \cdots T_d$ denote its LZ78 parsing. We recall that $T_\$^R$ 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 upper bound the size of Opp($T_\$^R$), we first need a preliminary lemma relating the entropies of T and $T_\$$.

[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.

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

$$|T_\$|H_k(T_\$) \leq nH_k(T) + O(n/\log n).$$

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 $|\texttt{Aux}(T_\$)| \leq nH_k(T) + O(n/\log n)$.

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 $|\texttt{Aux}(T_\$)| \geq |T_\$|H_k(T_\$)$.

From the definition of Aux, we get

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

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

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

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 A.2, we know that G is monotonically increasing in each of its components. Hence, we have

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

LEMMA D.2. *For any $k \geq 0$ and for any text $T[1, n]$, the space used by the data structure $\mathsf{Opp}(T_\$^R)$ is bounded by $5nH_k(T) + (n\frac{\log\log n}{\log n})$ bits.*

PROOF. Since $|T_\$^R| = n + d$ and $d = O(n/\log n)$, by Theorem 3.5, we get

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

Using Theorem A.3 and Lemma D.1, we get

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

as claimed. □

ACKNOWLEDGMENTS. We thank the anonymous referees for their valuable comments, and Fabrizio Luccio for suggestions on an early version of the article.

REFERENCES

ALSTRUP, S., BRODAL, G. S., AND RAUHE, T. 2000. New data structures for orthogonal range searching. In *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., 198–207.

ANDERSSON, A. 1996. Sorting and searching revisited. In *Proceedings of the 5th Scandinavian Workshop on Algorithm Theory*. Lecture Notes in Computer Science, vol. 1097. Springer-Verlag, New York, 185–197.

BENTLEY, J., SLEATOR, D., TARJAN, R., AND WEI, V. 1986. A locally adaptive compression scheme. *Commun. ACM 29*, 4, 320–330.

BRODNIK, A., AND MUNRO, I. 1999. Membership in constant time and almost-minimum space. *SIAM J. Comput. 28*, 5, 1627–1640.

BURROWS, M., AND WHEELER, D. 1994. A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation.

CHAN, H., HON, W., AND LAM, T. 2004. Compressed index for a dynamic collection of texts. In *Proceedings of the 15th Symposium on Combinatorial Pattern Matching*. Lecture Notes in Computer Science, vol. 3109. Springer-Verlag, New York, 445–456.

CLARK, D. R., AND MUNRO, I. 1996. Efficient suffix trees on secondary storage. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*. ACM, New York, 383–391.

COLUSSI, L., AND DE COL, A. 1996. A time and space efficient data structure for string searching on large texts. *Info. Proc. Lett. 58*, 5, 217–222.

FARACH, M., AND THORUP, M. 1998. String matching in Lempel–Ziv compressed strings. *Algorithmica 20*, 4, 388–404.

FENWICK, P. 1996. The Burrows-Wheeler transform for block sorting text compression: principles and improvements. *The Computer J. 39*, 9, 731–740.

FERRAGINA, P., GIANCARLO, R., MANZINI, G., AND SCIORTINO, M. 2005. Boosting textual compression in optimal linear time. *J. ACM 52*, 4 (July), 688–713.

FERRAGINA, P., AND GROSSI, R. 1999. The string B-tree: A new data structure for string search in external memory and its applications. *J. ACM 46*, 2, 236–280.

FERRAGINA, P., AND MANZINI, G. 2000. Opportunistic data structures with applications. In *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., 390–398.

FERRAGINA, P., AND MANZINI, G. 2001. An experimental study of a compressed index. *Information Sciences: Special issue on "Dictionary Based Compression" 135*, 13–28.

FERRAGINA, P., MANZINI, G., MÄKINEN, V., AND NAVARRO, G. 2004. An alphabet-friendly FM-index. In *Proceedings of the 11th International Symposium on String Processing and Information Retrieval*. Lecture Notes in Computer Science, vol. 3246. Springer-Verlag, New York, 150–160.

GONNET, G. H., BAEZA-YATES, R. A., AND SNIDER, T. 1992. New indices for text: PAT trees and PAT arrays. In *Information Retrieval: Data Structures and Algorithms*, B. Frakes and R. A. Baeza-Yates Eds. Prentice-Hall, Englewood Cliffs, N.J., Chapter 5, 66–82.

GRABOWSKI, S., MÄKINEN, V., AND NAVARRO, G. 2004. First Huffman, then Burrows-Wheeler: An alphabet-independent FM-index. In *Proceedings of the 11th International Symposium on String Processing and Information Retrieval*. Lecture Notes in Computer Science, vol. 3246. Springer-Verlag, New York, 210–211.

GROSSI, R., GUPTA, A., AND VITTER, J. 2003. High-order entropy-compressed text indexes. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM Press, New York, 841–850.

GROSSI, R., GUPTA, A., AND VITTER, J. 2004. When indexing equals compression: Experiments on compressing suffix arrays and applications. In *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM Press, New York, 636–645.

GROSSI, R., AND VITTER, J. 2000. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32nd ACM Symposium on Theory of Computing*. ACM, New York, 397–406.

HEALY, J., THOMAS, E. E., SCHWARTZ, J. T., AND WIGLER, M. 2003. Annotating large genomes with exact word matches. *Genome Research 13*, 2306–2315.

HON, W., LAM, T., SADAKANE, K., SUNG, W., AND YIU, S. 2004a. Compressed index for dynamic text. In *Proceedings of the IEEE Data Compression Conference*. IEEE Computer Society Press, Los Alamitos, Calif., 102–111.

HON, W., LAM, T., SUNG, W., TSE, W., WONG, C., AND YIU, S. 2004b. Practical aspects of compressed suffix arrays and FM-index in searching DNA sequences. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments*. SIAM Press, Philadelphia, Pa., 31–38.

HUYNH, N., HON, W., LAM, T., AND SUNG, W. 2004. Approximate string matching using compressed suffix arrays. In *Proceedings of the 15th Symposium on Combinatorial Pattern Matching*. Lecture Notes in Computer Science, vol. 3109. Springer-Verlag, New York, 434–444.

KÄRKKÄINEN, J., AND SUTINEN, E. 1998. Lempel-Ziv index for q-grams. *Algorithmica 21*, 137–154.

KÄRKKÄINEN, J., AND UKKONEN, E. 1996. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proceedings of the 3rd South American Workshop on String Processing*, N. Ziviani, R. Baeza-Yates, and K. Guimarães, Eds. Carleton University Press, 141–155.

KOSARAJU, R., AND MANZINI, G. 1999. Compression of low entropy strings with Lempel–Ziv algorithms. *SIAM J. Comput. 29*, 3, 893–911.

KURTZ, S. 1999. Reducing the space requirement of suffix trees. *Software—Practice and Experience 29*, 13, 1149–1171.

MÄKINEN, V., AND NAVARRO, G. 2004. Compressed compact suffix arrays. In *Proceedings of the 15th Symposium on Combinatorial Pattern Matching*. Lecture Notes in Computer Science, vol. 3109. Springer-Verlag, New York, 420–433.

MÄKINEN, V., NAVARRO, G., AND SADAKANE, K. 2004. Advantages of backward searching—efficient secondary memory and distributed implementation of compressed suffix arrays. In *Proceedings of the 15th International Symposium on Algorithms and Computation*. Lecture Notes in Computer Science, Springer-Verlag, New York.

MANBER, U., AND MYERS, G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput. 22*, 5, 935–948.

MANZINI, G. 2001. An analysis of the Burrows-Wheeler transform. *J. ACM 48*, 3, 407–430.

MCCREIGHT, E. M. 1976. A space-economical suffix tree construction algorithm. *J. ACM 23*, 2, 262–272.

MUNRO, J. I., RAMAN, V., AND RAO, S. S. 2001. Space efficient suffix trees. *J. Algor. 39*, 2, 205–222.

NAVARRO, G. 2004. Indexing text using the Ziv-Lempel trie. *J. Discr. Algor. 2*, 1, 87–114.

PAGH, R. 2001. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput. 31*, 2, 353–363.

RAMAN, R., RAMAN, V., AND RAO, S. S. 2002. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM Press, New York, 233–242.

RAO, S. S. 2002. Time-space trade-offs for compressed suffix arrays. *Inf. Proc. Lett. 82*, 6, 307–311.

SADAKANE, K. 2000. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proceeding of the 11th International Symposium on Algorithms and Computation*. Lecture Notes in Computer Science, vol. 1969. Springer-Verlag, New York, 410–421.

SADAKANE, K. 2002. Succinct representations of LCP information and improvements in the compressed suffix arrays. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM Press, New York, 225–232.

SADAKANE, K. 2003. New text indexing functionalities of compressed suffix arrays. *J. Algor. 48*, 2, 294–313.

SADAKANE, K., AND SHIBUYA, T. 2001. Indexing huge genome sequences for solving various problems. *Genome Informatics 12*, 175–183.

WITTEN, I. H., MOFFAT, A., AND BELL, T. C. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images*, Second ed., Morgan Kaufmann Publishers, Los Altos, Calif.

ZIV, J., AND LEMPEL, A. 1978. Compression of individual sequences via variable length coding. *IEEE Trans. Info. Theory 24*, 530–536.

RECEIVED JANUARY 2002; REVISED APRIL 2005; ACCEPTED APRIL 2005