An Alphabet-Friendly FM-index

Paolo Ferragina* Veli Mäkinen† Giovanni Manzini‡ Gonzalo Navarro§

Abstract

We show that, by combining an existing compression boosting technique with the wavelet tree data structure, we are able to design a variant of the FM-index which scales well with the size of the input alphabet Σ. The size of the new index built on a string $T[1,n]$ is bounded by $nH_k(T) + O\big((n \log\log n)/\log_{|\Sigma|} n\big)$ bits, where $H_k(T)$ is the k-th order empirical entropy of T.

The above bound holds simultaneously for all $k \le \alpha \log_{|\Sigma|} n$ and $0 < \alpha < 1$. Moreover, the index does not depend on the parameter k, which plays a role only in analysis of the space occupancy.

Using our index, counting the occurrences of an arbitrary pattern $P[1,p]$ as a substring of T takes $O(p \log |\Sigma|)$ time. Locating each pattern occurrence takes $O(\log|\Sigma| \; \log^{1+\epsilon} n)$ time. Reporting a text substring of length ℓ takes $O((\ell + \log^{1+\epsilon} n)\log|\Sigma|)$ time.

1 Introduction

A *full-text index* is a data structure built over a text string $T[1,n]$ that supports the efficient search for an arbitrary pattern as a *substring* of the indexed text. A *self-index* is a full-text index that encapsulates the indexed text T, without hence requiring its explicit storage.

The FM-index [3] has been the first self-index in the literature to achieve a space occupancy close to the k-th order entropy of T—hereafter denoted by $H_k(T)$ (see Section 2.1). Precisely, the FM-index occupies at most $5nH_k(T) + o(n)$ bits of storage, and allows the search for the *occ* occurrences of a pattern $P[1,p]$ within T in $O(p + occ \log^{1+\epsilon} n)$ time, where $\epsilon > 0$ is an arbitrary constant fixed in advance. It can display any text substring of length ℓ in $O(\ell + \log^{1+\epsilon} n)$ time. The design of the FM-index is based upon the relationship between the Burrows-Wheeler compression algorithm [1] and the suffix array data structure [13]. It is therefore a sort of *compressed suffix array* that takes advantage of the compressibility of the indexed text in order to achieve space occupancy close to the Information Theoretic minimum. Indeed, the design of the FM-index does not depend on the parameter k and its space bound holds *simultaneously* over all $k \ge 0$. These remarkable theoretical properties have been validated by experimental results [4, 5] and applications [11, 18].

The above bounds on the FM-index space occupancy and query time have been obtained assuming that the size of the input alphabet is a constant. Hidden in the big-O notation there is an exponential dependency on the alphabet size in the space bound, and a linear dependency on the alphabet size in the time bounds. More specifically, the search time is $O(p + occ\,|\Sigma| \; \log^{1+\epsilon} n)$ and the time to display a text substring is $O((\ell + \log^{1+\epsilon} n)\;|\Sigma|)$.

Although in practical implementations of the FM-index [4, 5] these dependencies are removed with only a small penalty in the query time, it is worthwhile to investigate whether it is possible to build a more "alphabet-friendly" FM-index.

*Dipartimento di Informatica, University of Pisa, Italy.
†Department of Computer Science, University of Helsinki, Finland.
‡Dipartimento di Informatica, University of Piemonte Orientale, Italy.
§Department of Computer Science, University of Chile, Chile.

In this paper we use the compression boosting technique [2, 6] and the wavelet tree data structure [8] to design a version of the FM-index which scales well with the size of the alphabet. Compression boosting permits partitioning the Burrows-Wheeler transformed text into contiguous areas in order to maximize the performance of zero-order compression over each area. The wavelet tree achieves such a zero-order compression and at the same time permits answering some simple queries over the indexed area.

The resulting data structure indexes a string $T[1,n]$ drawn from an alphabet Σ using $nH_k(T) + O((n \log\log n)/\log_{|\Sigma|} n)$ bits of storage. The above bound holds simultaneously for all $k \leq \alpha \log_{|\Sigma|} n$ and $0 < \alpha < 1$. The structure of our index is extremely simple and does not depend on the parameter k, which plays a role only in the analysis of the space occupancy.

Using our index, counting the occurrences of an arbitrary pattern $P[1,p]$ as a substring of T takes $O(p \log|\Sigma|)$ time. Locating each pattern occurrence takes $O(\log|\Sigma| \;\; \log^{1+\epsilon})$ time. Displaying a text substring of length ℓ takes $O((\ell + \log^{1+\epsilon} n)\log|\Sigma|)$.

Compared to the original FM-index, we note that the new version scales better with the alphabet size in all aspects. Albeit the time to count pattern occurrences has increased, that of locating occurrences and displaying text substrings has decreased. The latter complexities were a weak point in the FM-index. Furthermore, our space complexity does not hide large alphabet-dependent factors in the sublinear term of the space complexity.

Recently, various compressed full-text indexes have been proposed in the literature achieving several time/space trade-offs [10, 17, 15, 8, 9]. Among them, the one with the smallest space occupancy is the data structure described in [8] (Theorem 5.2). Such data structure achieves $O(p \log|\Sigma| \; + \mathrm{polylog}(n))$ time to count the pattern occurrences and uses $nH_k(T) + O((n \log\log n)/\log_{|\Sigma|} n)$ bits of storage. The space bound holds for *a fixed* k which must be chosen in advance, i.e., when the index is built. The parameter k must satisfied the constraint $k \leq \alpha \log_{|\Sigma|} n$ with $0 < \alpha < 1$, which is the same limitation that we have for our space bound.

To summarize, our data structure is extremely simple, has the smallest known space occupancy, and counts occurrences faster than the data structure in [8], which is the only other compressed index known to date with a $nH_k(T) + o(n)$ space occupancy.

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 an alphabet Σ of size $|\Sigma|$. By $T[i]$ we denote the i-th character of T, $T[i,n]$ denotes the ith text suffix, and $T[1,i]$ denotes the ith text prefix. We write $|w|$ to denote the length of string w.

2.1 The k-th order empirical entropy

Following a well established practice in Information Theory, we lower bound the space needed to store a string T by using the notion of *empirical entropy*. The empirical entropy is similar to the entropy defined in the probabilistic setting with the difference that it is defined in terms of the character frequencies observed in T rather than in terms of character probabilities. The key property of empirical entropy is that it is defined *pointwise* for *any* string T 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*.

Formally, the *zero-th order* empirical entropy of T is defined as $H_0(T) = -\sum_i (n_i/n)\log(n_i/n)$, where n_i is the number of occurrences of the i-th alphabet character in T, $n = \sum_i n_i = |T|$, and all logarithms are taken to the base 2 (with $0 \log 0 = 0$). To introduce the concept of *k-th order* empirical entropy we need to define what is a *context*. A length-k context w in T is one of its substrings of length k. Given w, we denote by $\vec{w}_T$ the string formed by concatenating all the symbols following the occurrences of w

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

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

in T, taken from left to right. For example, if $T = \texttt{mississippi}$ then $\vec{\texttt{s}}_T = \texttt{sisi}$ and $\vec{\texttt{si}}_T = \texttt{sp}$. The k-th order empirical entropy of T is defined as:

$$H_k(T) = \frac{1}{n} \sum_{w \in \Sigma^k} |\, \vec{w}_T \,|\; H_0(\vec{w}_T). \tag{1}$$

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

2.2 The Burrows-Wheeler transform

In [1] Burrows and Wheeler introduced a new compression algorithm based on a reversible transformation now called the *Burrows-Wheeler Transform* (BWT from now on). The BWT 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 of the string $T\#$ sorted in lexicographic order; (3) construct the transformed text T^{bwt} by taking the last column of matrix $\mathcal{M}_T$. Notice that every column of $\mathcal{M}_T$, hence also the transformed text T^{bwt}, 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 T^{bwt}).

We remark that the BWT by itself is not a compression algorithm since T^{bwt} is just a permutation of $T\#$. However, if T has some regularities the BWT will "group together" several occurrences of the same character. As a result, the transformed string T^{bwt} contains long runs of identical characters and turns out to be highly compressible (see e.g. [1, 14] for details).

Because of the special character #, when we sort the rows of $\mathcal{M}_T$ we are essentially sorting the suffixes of T. Therefore there is a strong relation between the matrix $\mathcal{M}_T$ 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:

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

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

Figure 2: Algorithm get_rows for finding the set of rows prefixed by $P[1,p]$, and thus for counting the pattern's occurrences $occ = \mathsf{Last} - \mathsf{First} + 1$. 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 $T^{bwt}[1,q]$.

- Let $C[\cdot]$ denote the array of length $|\Sigma|$ such that $C[c]$ contains the total number of text characters which are alphabetically smaller than c.
- Let $\mathsf{Occ}(c,q)$ denote the number of occurrences of character c in the prefix $T^{bwt}[1,q]$.
- Let $LF(i) = C[T^{bwt}[i]] + \mathsf{Occ}(T^{bwt}[i], i)$.

$LF(\cdot)$ stands for *Last-to-First* column mapping since the character $T^{bwt}[i]$, in the last column of $\mathcal{M}_T$, is located in the first column F at position $LF(i)$. For example in Figure 1 we have $LF(10) = C[\mathtt{s}] + \mathsf{Occ}(\mathtt{s}, 10) = 12$; and in fact $T^{bwt}[10]$ and $F[LF(10)] = F[12]$ both correspond to the first s in the string `mississippi`.

The $LF(\cdot)$ mapping allows us to scan the text T backward. Namely, if $T[k] = T^{bwt}[i]$ then $T[k-1] = T^{bwt}[LF(i)]$. For example in Fig. 1 we have that $T[3] = \mathtt{s}$ is the 10th character of T^{bwt} and we correctly have $T[2] = T^{bwt}[LF(10)] = T^{bwt}[12] = \mathtt{i}$ (see [3] for details).

2.3 The FM-index

The FM-index is a *self-index* that allows to efficiently search for the occurrences of an arbitrary pattern $P[1,p]$ as a *substring* of the text $T[1,n]$. Pattern P is provided on-line whereas the text T is given to be preprocessed in advance. The number of pattern occurrences in T is hereafter indicated with *occ*. The term *self-index* highlights the fact that T is not stored explicitly but it can be derived from the FM-index.

The FM-index consists of a compressed representation of T^{bwt} together with some auxiliary information which makes it possible to compute in $O(1)$ time the value $\mathsf{Occ}(c,q)$ for any character c and for any q, $0 \leq q \leq n$. The two key procedures to operate on the FM-index are: the *counting* of the number of pattern occurrences (shortly get_rows), and the *location* of their positions in the text T (shortly get_position). Note that the counting process returns the value *occ*, whereas the location process returns *occ* distinct integers in the range $[1,n]$.

Figure 2 sketches the pseudocode of the counting operation that works in p phases, numbered from p to 1. The i-th phase preserves the following invariant: *The parameter* First *points to the first row of the BWT matrix $\mathcal{M}_T$ prefixed by $P[i,p]$, and the parameter* Last *points to the last row of $\mathcal{M}_T$ prefixed by $P[i,p]$.* After the final phase, P prefixes the rows between First and Last and thus, according to the properties of matrix $\mathcal{M}_T$ (see Section 2.2), we have $occ = \mathsf{Last} - \mathsf{First} + 1$. It is easy to see that the running time of get_rows is dominated by the cost of the $2p$ computations of the values Occ( ).

Algorithm get_position(i)

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

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

Given the range $(\mathsf{First}, \mathsf{Last})$, we now consider the problem of retrieving the positions in T of these pattern occurrences. We notice that every row in $\mathcal{M}_T$ is prefixed by some suffix of T. For example, in Fig. 1 the fourth row of $\mathcal{M}_T$ is prefixed by the text suffix $T[5, 11] = \mathtt{issippi}$. Then, for $i = \mathsf{First}, \mathsf{First}+1, \ldots, \mathsf{Last}$ we use procedure get_position(i) to find the position in T of the suffix that prefixes the i-th row $\mathcal{M}_T[i]$. Such a position is denoted hereafter by $\mathsf{Pos}(i)$, and the pseudocode of get_position is given in Figure 3. The intuition underlying its functioning is simple. We scan backward the text T using the $LF(\cdot)$ mapping (see Section 2.2) until a *marked* position is met. If we mark one text position every $\Theta(\log^{1+\epsilon} n)$ the `while` loop is executed $O(\log^{1+\epsilon} n)$ times. Since the computation of $LF(i)$ can be done via at most $|\Sigma|$ computations of $\mathsf{Occ}(\cdot, \cdot)$, we have that locating a single execution of get_position takes $O(|\Sigma|\ \log^{1+\epsilon} n)$ time. Finally, we observe that marking one position position every $\Theta(\log^{1+\epsilon} n)$ takes $\Theta(n/\log^{\epsilon} n)$ bits overall. Combining the observations on get_position with the ones for get_rows, we get a more refined version of the result proved in [3].

Theorem 1 *For any string $T[1, n]$ drawn from a constant-sized alphabet Σ, the FM-index counts the occurrences of any pattern $P[1, p]$ within T taking $O(p)$ time. The location of each pattern occurrence takes $O(|\Sigma|\ \log^{1+\epsilon} n)$ time. The size of the FM-index is bounded by $5nH_k(T) + o(n)$ bits, for any $k \geq 0$.* ■

In order to retrieve the content of $T[l, r]$, we must first find the row in $\mathcal{M}_T$ that corresponds to r, and then issue $\ell = r - l + 1$ backward steps in T, using the $LF(\cdot)$ mapping. Starting at the lowest marked text position that follows r, we perform $O(\log^{1+\epsilon} n)$ steps until reaching r. Then we perform ℓ additional steps, this time collecting the text characters that are discovered as a part of the process of computing $LF(\cdot)$. The resulting complexity is $O((\ell + \log^{1+\epsilon} n)\ |\Sigma|)$.

As we mentioned in the Introduction, the main drawback of the FM-index is that hidden in the $o(n)$ term of the space bound there are constants which depend exponentially on the alphabet size $|\Sigma|$. In Section 3 we describe a simple alternative representation of T^{bwt} which takes $nH_k(T) + O(\log|\Sigma| \frac{n \log\log n}{\log n})$ bits and still allows the computation of $\mathsf{Occ}(c, q)$ and $T^{bwt}[i]$ in $O(\log|\Sigma|)$ time.

2.4 Compression boosting

The concept of *compression boosting* has been recently introduced in [2, 6, 7] opening the door to a new approach to data compression. The key idea is that one can take an algorithm whose performance can be bounded in terms of the 0-th order entropy and obtain, via the booster, a new compressor whose performance can be bounded in terms of the k-th order entropy, *simultaneously for all k*. Putting it another way, one can take a compression algorithm that uses no context information at all and, via the boosting process, obtain an algorithm that automatically uses the "best possible" contexts.

To simplify the exposition, we now state a boosting theorem in a form which is slightly different from the version described in [2, 6]. However, the proof of Theorem 2 can be obtained by a straightforward modification of the proof of Theorem 4.1 in [6].

Theorem 2 *Let $\mathcal{A}$ be an algorithm which compresses any string s in less than $|s|H_0(s)+f(|s|)$ bits, where $f(\cdot)$ is a non decreasing concave function. Given $T[1,n]$ there is a $O(n)$ time procedure that computes a partition $s_1, s_2, \ldots, s_z$ of T^{bwt} such that for any $k \geq 0$ we have*

$$\sum_{i=1}^{z} |\mathcal{A}(s_i)| \;\leq\; \sum_{i=1}^{z} (|s_i|H_0(s_i) + f(|s_i|)) \;\leq\; nH_k(T) + |\Sigma|^k f(n/|\Sigma|^k).$$

Proof: (Sketch). According to Theorem 4.1 in [6], the booster computes the partition that minimizes the function $\sum_{i=1}^{z} |s|H_0(s_i) + f(|s_i|)$. To determine the right side of the above inequality, we consider the partition $\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_m$ induced by the contexts of length k in T. For such partition we have $m \leq |\Sigma|^k$ and $\sum_{i=1}^{m} |\hat{s}|H_0(\hat{s}_i) = nH_k(T)$. The hypothesis on f implies that $\sum_{i=1}^{m} f(|\hat{s}_i|) \leq |\Sigma|^k f(n/|\Sigma|^k)$ and the theorem follows. ■

To understand the relevance of this result suppose that we want to compress $T[1,n]$ and that we wish to exploit the zero-th order compressor $\mathcal{A}$. Using the booster we can first compute the partition $s_1, s_2, \ldots, s_z$ of T^{bwt}, and then compress each s_i using $\mathcal{A}$. By the above theorem, the overall space occupancy would be bounded by $\sum_i |\mathcal{A}(s_i)| \leq nH_k(T) + |\Sigma|^k f(n/|\Sigma|^k) + \log n$. Note that the process is reversible, because the decompression of each s_i retrieves T^{bwt}, and from T^{bwt} we can retrieve T using the inverse BWT. Summing up, the booster allows us to compress T up to its k-th order entropy using only the zero-th order compressor $\mathcal{A}$. Note that the parameter k is neither known to $\mathcal{A}$ nor to the booster, it comes into play only in the space complexity analysis. Additionally, the space bound in Theorem 2 holds *simultaneously* for all $k \geq 0$. The only information that is required by the booster is the function $f(n)$ such that $|s|H_0(s) + f(|s|)$ is an upper bound on the size of the output produced by $\mathcal{A}$ on input s.

2.5 The wavelet tree

Given a binary sequence $S[1,m]$ and $b \in \{0,1\}$, consider the following operations: $\mathsf{Rank}_b(S,i)$ computes the number of b's in $S[1,i]$, and $\mathsf{Select}_b(S,i)$ computes the position of the i-th b in $S[1,i]$. In [16] it has been proven the following:

Theorem 3 *Let $S[1,m]$ be a binary sequence containing t 1's. There exists a data structure (called* FID*) that supports $\mathsf{Rank}_b(S,i)$ and $\mathsf{Select}_b(S,i)$ in constant time, and uses $\lceil \log \binom{m}{t} \rceil + O((m \log\log m)/\log m) = mH_0(S) + O((m \log\log m)/\log m)$ bits of space.* ■

If, instead of a binary sequence, we have a sequence $W[1,w]$ over an arbitrary alphabet Σ, a compressed and indexable representation of W is provided by the wavelet tree [8] which is a clever generalization of the FID data structure.

Theorem 4 *Let $W[1,w]$ denote a string over an arbitrary alphabet Σ. The wavelet tree built on W uses $wH_0(W) + O(\log|\Sigma|\ (w \log\log w)/\log w)$ bits of storage and supports in $O(\log|\Sigma|)$ time the following operations:*

- *given q, $1 \leq q \leq w$, the retrieval of the character $W[q]$;*
- *given $c \in \Sigma$ and q, $1 \leq q \leq w$, the computation of the number of occurrences $\mathsf{Occ}_W(c,q)$ of c in $W[1,q]$.* ■

1. Use Theorem 2 to determine the optimal partition $s_1, s_2, \ldots, s_z$ of T^{bwt} with respect to $f(t) = (Kt \log|\Sigma| \log\log t)/\log t + (1+|\Sigma|)\log n$, where K is such that $(Kt\log|\Sigma|\log\log t)/\log t$ is larger than the $O((t\log|\Sigma|\log\log t)/\log t)$ term in Theorem 4.
2. Build a binary string $\mathcal{B}$ that keeps track of the starting positions in T^{bwt} of the s_i's. The entries of $\mathcal{B}$ are all zeroes except for the bits at positions $\sum_{j=1}^{i}|s_j|$ for $i = 1, \ldots, z$ which are set to 1. Construct the data structure of Theorem 3 over the binary string $\mathcal{B}$.
3. For each string s_i, $i = 1, \ldots, z$ build:
   (a) the array $\mathcal{C}_i[1, |\Sigma|]$ such that $\mathcal{C}_i[c]$ stores the occurrences of character c within $s_1 s_2 \cdots s_{i-1}$;
   (b) the wavelet tree $\mathcal{T}_i$.

Figure 4: Construction of an alphabet-friendly FM-index.

To make the paper more self-contained we recall the basic ideas underlying the wavelet tree. Consider a balanced binary tree $\mathcal{T}$ whose leaves contain the characters of the alphabet Σ. $\mathcal{T}$ has depth $O(\log|\Sigma|)$. Each node u of $\mathcal{T}$ is associated with a string W_u that represents the subsequence of W containing *only* the characters that descend from u. The root is thus associated with the entire W. To save space and be alphabet-friendly, the wavelet tree does not store W_u but a binary image of it, denoted by B_u, that is computed as follows: $B_u[i] = 0$ if the character $W_u[i]$ descends from the left child of u, otherwise $B_u[i] = 1$. Assume now that every binary sequence B_u is implemented with the data structure of Theorem 3; then it is an exercise to derive the given space bounds and to implement $\mathsf{Occ}_W(c, q)$ and retrieve $W[q]$ in $O(\log|\Sigma|)$ time.

3 Alphabet-friendly FM-index

We now have all the tools we need in order to build a version of the FM-index that scales well with the alphabet size. The crucial observation is the following. To build the FM-index we need to solve two problems: a) to compress T^{bwt} up to $H_k(T)$, and b) to compute $\mathsf{Occ}(c, q)$ in time independent of n. We use the boosting technique to transform problem a) into the problem of compressing the strings $s_1, s_2, \ldots, s_z$ up to their zero-th order entropy, and we use the wavelet tree to create a compressed (up to H_0) and indexable representation of each s_i thus solving simultaneously problems a) and b). The details of the construction are given in Figure 4.

We now show that the data structures described in Figure 4 allow us to compute $T^{bwt}[q]$ and $\mathsf{Occ}(c, q)$ in $O(\log|\Sigma|)$ time. To compute $T^{bwt}[q]$, we first determine the substring s_y containing the q-th character of T^{bwt} by computing $y = \mathsf{Rank}_1(\mathcal{B}, q)$. Then we exploit the wavelet tree $\mathcal{T}_y$ to determine $T^{bwt}[q]$. By Theorem 3 the former step takes $O(1)$ time, and by Theorem 4 the latter step takes $O(\log|\Sigma|)$ time.

To compute $\mathsf{Occ}(c, q)$, we initially determine the substring s_y where the row q occurs, $y = \mathsf{Rank}_1(\mathcal{B}, q)$. Then we exploit the wavelet tree $\mathcal{T}_y$ and the array $\mathcal{C}_y[c]$ to compute $\mathsf{Occ}(c, q) = \mathsf{Occ}_{s_y}(c, q) + \mathcal{C}_y[c]$. Again, by Theorems 3 and 4 this computation takes overall $O(\log|\Sigma|)$ time.

Combining these bounds with the results stated in Section 2.3, we obtain that the alphabet-friendly FM-index takes $O(p\log|\Sigma|)$ time to count the occurrences of a pattern $P[1, p]$ and $O(\log|\Sigma| \log^{1+\epsilon} n)$ time to retrieve the position of each occurrence.

Concerning the space occupancy of our data structure we observe that by Theorem 3, the storage of $\mathcal{B}$ takes $\lceil \log\binom{n}{z}\rceil + O((n\log\log n)/\log n)$ bits. Each array $\mathcal{C}_i$ takes $O(|\Sigma|\log n)$ bits, and each wavelet tree $\mathcal{T}_i$ occupies $|s_i|H_0(s_i) + O\left(|s_i|\frac{\log|\Sigma|\log\log|s_i|}{\log|s_i|}\right)$ bits (Theorem 4). Since $\log\binom{n}{z} \le z\log n$, the total occupancy is bounded by

$$\sum_{i=1}^{z}\Big(|s_i|H_0(s_i)+K|s_i|\frac{\log|\Sigma|\log\log|s_i|}{\log|s_i|}+(1+|\Sigma|)\log n\Big)+O((n\log\log n)/\log n)\,.$$

Function $f(t)$ defined at Step 1 of Figure 4 was built to match exactly the overhead space bound we get for each partition, so the partitioning was optimally built for that overhead. Hence we can apply Theorem 2 to get that the above summation is bounded by

$$nH_k(T)+O\Big(n\frac{\log|\Sigma|\log\log n}{\log(n/|\Sigma|^k)}\Big)+O\Big(|\Sigma|^{k+1}\log n\Big)\,. \tag{2}$$

Note that (2) bounds the space occupancy of the data structure supporting the counting of the occurrences of a pattern in T. If we want to locate each occurrence in $O(\log^{1+\epsilon} n)$ time, we need additional $O(n/\log^{\epsilon} n)$ bits (see Section 2.3). This does not affect, however, the $O(n\log\log n/\log n)$ term above.

In the *logarithmic-context* setting, we are interested in bounding the space occupancy in terms of H_k only for $k\le\alpha\log_{|\Sigma|}n$ for some $\alpha<1$. In this case we have $|\Sigma|^k\le n^{\alpha}$ and (2) becomes

$$nH_k(T)+O(\log|\Sigma|(n\log\log n)/\log n)\,. \tag{3}$$

We achieve the following result.

Theorem 5 *The data structure described in Figure 4 indexes a string $T[1,n]$ over an arbitrary alphabet $|\Sigma|$, using a storage bounded by*

$$nH_k(T)+O(\log|\Sigma|(n\log\log n)/\log n)$$

bits for any $k\le\alpha\log_{|\Sigma|}n$, $0<\alpha<1$. Using the data structure of Figure 4 we can count the number of occurrences of a pattern $P[1,p]$ in T in $O(p\log|\Sigma|)$ time, locate each occurrence in $O(\log^{1+\epsilon} n\;\log|\Sigma|)$ time, and display a text substring of length ℓ in $O((\ell+\log^{1+\epsilon} n)\log|\Sigma|)$ time. ■

It is natural to ask whether a more sophisticated data structure can achieve a $nH_k(T)+o(n)$ space bound without any restriction on the alphabet size or context length. The answer to this question is negative. To see this, consider the extreme case in which $|\Sigma|=n$, that is, the input string consists of a permutation of n distinct characters. In this case we have $H_k(T)=0$ for $k\ge1$. Since the representation of such string requires $\Theta(n\log n)$ bits, a self index of size $nH_k(T)+o(n)$ bits cannot exist.

Finally, we note that the wavelet tree alone, over the full BWT transformed text T^{bwt}, would be enough to obtain the time bounds we achieved. However, the resulting structure size would depend on $H_0(T)$ rather than $H_k(T)$. The partitioning of the text into areas is crucial to obtain the latter space bounds. A previous technique combining wavelet trees with text partitioning [12] takes each run of equal letters in T^{bwt} as an area. It requires $2n(H_k\log\sigma+1+o(1))$ bits of space and counts pattern occurrences in the optimal $O(p)$ time. It would be interesting to retain the optimal space complexity obtained in this work and also the optimal search time $O(p)$.

References

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

[2] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. In *Technical Report*, 2004.

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

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

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

[6] P. Ferragina and G. Manzini. Compression boosting in optimal linear time using the Burrows-Wheeler transform. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms (SODA '04)*, 2004.

[7] R. Giancarlo and M. Sciortino. Optimal partitions of strings: A new class of Burrows-Wheeler compression algorithms. In *Combinatorial Pattern Matching Conference (CPM '03)*, pages 129–143, 2003.

[8] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA '03)*, pages 841–850, 2003.

[9] R. Grossi, A. Gupta, and J. Vitter. Indexing equals compression: Experiments on suffix arrays and trees. In *Proc. 15th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA '04)*, 2004. This volume.

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

[11] J. Healy, E. E. Thomas, J. T. Schwartz, and M. Wigler. Annotating large genomes with exact word matches. *Genome Research*, 13:2306–2315, 2003.

[12] V. Mäkinen and G. Navarro. *New Search Algorithms and Time/Space Tradeoffs for Succinct Suffix Arrays.* Technical Report C-2004-20, University of Helsinki, Finland, April 2004.

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

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

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

[16] R. Raman, V. Raman, and S.Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 233–242, 2002.

[17] K. Sadakane. Succinct representations of LCP information and improvements in the compressed suffix arrays. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 225–232, 2002.

[18] K. Sadakane and T. Shibuya. Indexing huge genome sequences for solving various problems. *Genome Informatics*, 12:175–183, 2001.