Nordic Journal of Computing

SUCCINCT SUFFIX ARRAYS BASED ON RUN-LENGTH ENCODING *

VELI MÄKINEN†
Dept. of Computer Science, University of Helsinki
Gustaf Hällströmin katu 2b, 00014 University of Helsinki, Finland
vmakinen@cs.helsinki.fi

GONZALO NAVARRO‡
Dept. of Computer Science, University of Chile
Blanco Encalada 2120, Santiago, Chile
gnavarro@dcc.uchile.cl

Abstract. A succinct full-text self-index is a data structure built on a text $T = t_1 t_2 \dots t_n$, which takes little space (ideally close to that of the compressed text), permits efficient search for the occurrences of a pattern $P = p_1 p_2 \dots p_m$ in T, and is able to reproduce any text substring, so the self-index replaces the text.

Several remarkable self-indexes have been developed in recent years. Many of those take space proportional to nH_0 or nH_k bits, where H_k is the kth order empirical entropy of T. The time to count how many times does P occur in T ranges from $O(m)$ to $O(m \log n)$.

In this paper we present a new self-index, called RLFM index for "run-length FM-index", that counts the occurrences of P in T in $O(m)$ time when the alphabet size is $\sigma = O(\mathrm{polylog}(n))$. The RLFM index requires $nH_k \log \sigma + O(n)$ bits of space, for any $k \le \alpha \log_\sigma n$ and constant $0 < \alpha < 1$. Previous indexes that achieve $O(m)$ counting time either require more than nH_0 bits of space or require that $\sigma = O(1)$. We also show that the RLFM index can be enhanced to locate occurrences in the text and display text substrings in time independent of σ.

In addition, we prove a close relationship between the kth order entropy of the text and some regularities that show up in their suffix arrays and in the Burrows-Wheeler transform of T. This relationship is of independent interest and permits bounding the space occupancy of the RLFM index, as well as that of other existing compressed indexes.

Finally, we present some practical considerations in order to implement the RLFM index. We empirically compare our index against the best existing implementations and show that it is practical and competitive against those. In passing, we obtain a competitive implementation of an existing theoretical proposal that can be seen as a simplified RLFM index, and explore other practical ideas such as Huffman-shaped wavelet trees.

ACM CCS Categories and Subject Descriptors: E.1. Data structures; F.2.2. Nonnumerical Algorithms and Problems — Computations on discrete structures, Pattern matching, Sorting and searching; H.3. Information Storage and Retrieval

*Parts of this work have appeared in [29, 31, 30, 32].

†Part of the work was conducted during a postdoctoral visit to Bielefeld University funded by the Deutsche Forschungsgemeinschaft (BO 1910/1-3) within the Computer Science Action Program. Also partially funded by the Academy of Finland.

‡Funded by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.

Received ...; revised ...; accepted ....

Key words: Compressed full-text self-indexes, empirical k-th order entropy, indexed string matching, text retrieval, Burrows-Wheeler transform, suffix arrays, run-length compression.

1. Introduction

The classical problem in string matching is to determine the *occ* occurrences of a short pattern $P = p_1 p_2 \dots p_m$ in a large text $T = t_1 t_2 \dots t_n$. Text and pattern are sequences of characters over an alphabet Σ of size σ. Actually one may want to know the number *occ* of occurrences (this is called a *counting query*), the text positions of those *occ* occurrences (a *locating query*), or also a text context around them (a *context query*). Usually the same text is queried several times with different patterns, and therefore it is worthwhile to preprocess it in order to speed up the searches. The preprocessing builds an *index* structure on the text.

To allow fast searches for patterns of any size, the index must allow access to all *suffixes* of the text (the ith suffix of T is $t_i t_{i+1} \dots t_n$). These kind of indexes are called *full-text indexes*. The *suffix tree* [49, 36, 48, 2] is the best-known full-text index, requiring $O(m)$ time for counting and $O(occ)$ for locating queries.

The suffix tree takes much more memory than the text. In general, it takes $O(n \log n)$ bits, while the text takes $n \log \sigma$ bits[1]. In practice the suffix tree requires about 20 times the text size. A smaller constant factor, close to 4 in practice, is achieved by the *suffix array* [34]. Still, the space complexity of $O(n \log n)$ bits does not change. Moreover, counting queries take $O(m \log n)$ time with the suffix array. This can be improved to $O(m + \log n)$ by using twice the original amount of space [34].

Since the last decade, several attempts to reduce the space of the suffix trees or arrays have been made [23, 25, 26, 27, 1, 28], and other structures have been proposed as well [4, 47, 24]. In some cases these attempts have obtained remarkable results in space (for example, 1.6 times the text size in practice) at a small price in query time. Some of those structures [25, 28] have the interesting property of requiring less space when the text is compressible.

In parallel, intensive work on *succinct data structures* focused on the representation of basic structures such as sequences and trees [22, 37, 38, 42, 43]. Those representations were able to approach the information theoretic minimum space required to store the structures. Based on those results, new succinct representations of suffix trees and arrays were proposed [6, 7, 39, 18, 44]. Yet, all of them still required the text separately available to answer queries.

This trend evolved into the concept of *self-indexing*. A self-index is a succinct index that contains enough information to reproduce any text substring. Hence a self-index that implements such functionality can *replace* the text. The exciting possibility of an index that requires space proportional to the *compressed* text, and yet replaces it, has been explored in recent years [8, 9, 10, 45, 46, 41, 16, 17, 33, 14, 11, 12].

[1] By log we mean $\log_2$ in this paper.

Table I compares the space requirements, counting time, and restrictions on the alphabet size for those self-indexes. In the table, H_0 stands for the zero-order entropy of T, while H_k stands for the kth order entropy of T, for any $k \leq \alpha \log_\sigma n$, where $0 < \alpha < 1$ is any constant.[2] The number of bits of space required by the indexes ranges from proportional to nH_0 to proportional to nH_k. A few of those require exactly $nH_k + o(n)$ bits, which is currently the lowest asymptotic space requirement that has been achieved. Counting time ranges from $O(m)$ to $O(m \log n)$. Finally, some indexes achieve their results only on constant-size alphabets, while some others require that $\sigma = O(\text{polylog}(n))$, and the rest works well for any $\sigma = o(n/\log n)$.[3]

We also point out that $O(m/\log_\sigma n + \text{polylog}(n))$ time has been achieved on a succinct (non-self) index [18]. This time is optimal on the RAM model if the pattern is long enough compared to the text (that is, the polylog(n) term has to be $O(m/\log_\sigma n)$), yet not $O(m)$ in general.

T I: Comparison of space, counting time, and restrictions on the alphabet size for the existing self-indexes. We show the contributions ordered by the time when they first appeared.

Reference	Space in bits	Counting time	Works for $\sigma =$
[8, 9]	$5nH_k + o(n)$	$O(m)$	$O(1)$
[45]	$nH_0 + O(n \log\log\sigma)$	$O(m \log n)$	$o(n/\log n)$
[46]	$nH_0 + O(n)$	$O(m)$	$O(\text{polylog}(n))$
[41]	$4nH_k + o(n)$	$O(m^3 \log\sigma + m \log n)$	$o(n/\log n)$
[16, 17]	$nH_k + o(n \log\sigma)$	$O(m \log\sigma + \text{polylog}(n))$	$o(n/\log n)$
This paper	$nH_k \log\sigma + O(n)$	$O(m)$	$O(\text{polylog}(n))$
[33]	$nH_k(\log\sigma + \log\log n) + O(n)$	$O(m \log n)$	$o(n/\log n)$
[14, 15]	$2n(H_0 + 1)(1 + o(1))$	$O(m \log\sigma)$	$o(n/\log n)$
[11]	$nH_k + o(n \log\sigma)$	$O(m \log\sigma)$	$o(n/\log n)$
[12]	$nH_k + o(n)$	$O(m)$	$O(\text{polylog}(n))$

Table I shows the indexes in chronological order of their first publication, including the RLFM index presented in this paper. This permits distinguishing the indexes that existed when the RLFM index first appeared [29, 31, 30], from those that appeared later, most of them deriving in some aspect from the RLFM index ideas. The RLFM index was the first in obtaining $O(m)$ counting time and space proportional to nH_k for any $\sigma = O(\text{polylog}(n))$. Inspired by the results of this paper [31], we started a joint work with the authors of the FM-index [8, 9] that ended up in new indexes [11] that very recently superseded our original results [12]. Similarly, we applied our ideas in joint work with the author of the CSA index [45, 46] and obtained new CSA variants [33]. Yet, our index is still unique in its run-length-

[2] It also holds for any constant k as long as $\sigma = O(\text{polylog}(n))$.

[3] This restriction is usually not explicit in those publications, but they store at least an array of integers in the range $[1, n]$ indexed by characters of Σ. Unless $\sigma = o(n/\log n)$, that array alone requires $\sigma \log n = \Omega(n)$ bits.

based approach to obtain space proportional to nH_k, and still represents a relevant space-time tradeoff among existing implementations.

Precisely, this paper presents the following contributions:

(1) We show that there is a close relationship between the kth order entropy of T and (i) the zones in the suffix array that appear replicated in another area with the values incremented by 1, and (ii) the runs of equal letters that appear once T undergoes the Burrows-Wheeler transform [5]. This proof has independent interest and not only permits analyzing some existing succinct indexes [28, 29, 33], but also gives the baseline to develop new indexes of size proportional to nH_k.

(2) We use the previous idea to develop a new self-index based on the runs of equal letters that result from the Burrows-Wheeler transform of T. The index, called *run-length FM-index (RLFM)*, requires $nH_k \log \sigma + O(n)$ bits of space, and it is able to answer counting queries in $O(m)$ time, for any $\sigma = O(\text{polylog}(n))$. We show different ways to answer locating queries and displaying text.

(3) We focus on a practical RLFM implementation, and develop simpler versions that perform better in practice. In passing we obtain an implementation of an existing proposal [46], that we call SSA for "succinct suffix array". We compare our implementations against the best existing ones for alternative indexes and show that the RLFM and SSA are competitive in practice and achieve space-time tradeoffs that are not reached by others.

2. Basic concepts

Let us first recall some definitions. A *string* $S = s_1 s_2 \ldots s_n$ is a sequence of *characters* (also called *symbols* or *letters*) from an alphabet Σ. The size of the alphabet is σ, and for clarity of exposition, we sometimes assume that $\Sigma = \{1, \ldots, \sigma\}$. The *length* of S is $|S| = n$, and its individual characters are $S[i] = s_i$. A *substring* of S is denoted by $S[i, j] = s_i s_{i+1} \ldots s_j$. An *empty string* is denoted ϵ. If $i > j$, then $S[i, j] = \epsilon$. A *suffix* of S is any substring $S[i, n]$. A *prefix* of S is any substring $S[1, i]$. A *cyclic shift* of S is any string $s_i s_{i+1} \ldots s_n s_1 s_2 \ldots s_{i-1}$. The *lexicographic order* of two strings is the natural order induced by the alphabet order: If two strings have the same first k letters, then their order depends on the order of their $(k+1)$th letter. We denote by $T = t_1 t_2 \ldots t_n$ our *text* string. We assume that a special *endmarker* $t_n = \$$ has been appended to T, such that the endmarker is smaller than any other text character. We denote by $P = p_1 p_2 \ldots p_n$ our *pattern* string, and seek to find the *occurrences* of P in T, that is, the positions in T where P appears as a substring of T.

We now survey some existing results used in our paper.

2.1 Empirical kth order entropy

We recall some basic facts and definitions related to the empirical entropy of texts [35]. Let n_c denote the number of occurrences in T of symbol $c \in \Sigma$. The zero-order empirical entropy of string T is

$$H_0(T) = -\sum_{c\in\Sigma} \frac{n_c}{n} \log \frac{n_c}{n},$$

where $0 \log 0 = 0$. If we use a fixed codeword for each symbol in the alphabet, then $nH_0(T)$ bits is the smallest encoding we can achieve for T.

If the codeword is not fixed, but it depends on the k symbols that follow the character in T, then the smallest encoding one can achieve for T is $nH_k(T)$ bits, where $H_k(T)$ is the kth order empirical entropy of T. This is defined as

$$H_k(T) = \frac{1}{n} \sum_{W\in\Sigma^k} |W_T| H_0(W_T), \tag{1}$$

where W_T is the concatenation of all symbols t_j (in arbitrary order) such that $t_j W$ is a substring of T. String W is the *k-context* of each such t_j.[4] Note that the order in which the symbols t_j are permuted in W_T does not affect $H_0(W_T)$.

We use H_0 and H_k as shorthands for $H_0(T)$ and $H_k(T)$ in this paper. We note that empirical entropies can be $o(n)$ for compressible texts. As an extreme example, consider the family $\{(ab)^n,\ n \geq 0\}$, where $H_0 = 1$ and $H_1 = O(\log n/n)$. This is in contrast with the classical notion of entropy, which is applied to infinite streams and is always constant [3]. By expressing the space requirement of indexes in terms of empirical entropies, we relate their size with compressibility bounds of each particular text.

2.2 *The Burrows-Wheeler transform*

The *Burrows-Wheeler transform (BWT)* [5] of a text T produces a permutation of T, denoted by T^{bwt}. Recall that T is assumed to be terminated by the endmarker "\$". String T^{bwt} is the result of the following transformation: (1) Form a *conceptual* matrix $\mathcal{M}$ whose rows are the cyclic shifts of the string T, call F its first column and L its last column; (2) sort the rows of $\mathcal{M}$ in lexicographic order; (3) the transformed text is $T^{bwt} = L$.

The BWT is reversible, that is, given T^{bwt} we can obtain T. Note the following properties [5]:

a. Given the ith row of $\mathcal{M}$, its last character $L[i]$ precedes its first character $F[i]$ in the original text T, that is, $T = \ldots L[i]F[i] \ldots$.

b. Let $L[i] = c$ and let r_i be the number of occurrences of c in $L[1, i]$. Take the row $\mathcal{M}[j]$ as the r_ith row of $\mathcal{M}$ starting with c. Then the character corresponding to $L[i]$ in the first column F is located at $F[j]$ (this is called the *LF mapping*: $LF(i) = j$). This is because the occurrences of character c are sorted both in F and L using the same criterion: by the text following the occurrences.

[4] Note that our contexts are the characters *following* each text position t_j. This has been chosen for technical convenience. Alternatively one can choose Wt_j, that is, the characters *preceding* t_j. Should this be problematic for the generality of our results, we can index the reversed text and search it for the reversed patterns to obtain the other definition.

The BWT can then be reversed as follows:

(1) Compute the array $C[1, \sigma]$ storing in $C[c]$ the number of occurrences of characters $\{\$, 1, \ldots, c-1\}$ in the text T. Notice that $C[c]+1$ is the position of the first occurrence of c in F (if any).

(2) Define the *LF mapping* as follows: $LF(i) = C[L[i]] + Occ(L, L[i], i)$, where $Occ(L, c, i)$ is the number of occurrences of character c in the prefix $L[1, i]$.

(3) Reconstruct T backwards as follows: set $s = 1$ (since $\mathcal{M}[1] = \$t_1t_2 \ldots t_{n-1}$) and, for each $n-1, \ldots, 1$ do $T[i] \leftarrow L[s]$ and $s \leftarrow LF[s]$. Finally put the endmarker $T[n] = \$$.

The BWT transform by itself does not compress T, it just permutes its characters. However, this permutation is more compressible than the original T. Actually, it is not hard to compress T^{bwt} to $O(nH_k + \sigma^k)$ bits, for any $k \geq 0$ [35].

2.3 *Suffix arrays*

The *suffix array* $\mathcal{A}[1, n]$ of text T is an array of pointers to all the suffixes of T in lexicographic order. Since T is terminated by the endmarker "\$", all lexicographic comparisons are well defined. The ith entry of $\mathcal{A}$ points to text suffix $T[\mathcal{A}[i], n] = t_{\mathcal{A}[i]}t_{\mathcal{A}[i]+1} \ldots t_n$, and it holds $T[\mathcal{A}[i], n] < T[\mathcal{A}[i+1], n]$ in lexicographic order.

Given the suffix array, the occurrences of the pattern $P = p_1p_2 \ldots p_m$ can be counted in $O(m \log n)$ time. The occurrences form an interval $\mathcal{A}[sp, ep]$ such that suffixes $t_{\mathcal{A}[i]}t_{\mathcal{A}[i]+1} \ldots t_n$, for all $sp \leq i \leq ep$, contain the pattern P as a prefix. This interval can be searched for using two binary searches in time $O(m \log n)$. Once the interval is obtained, a locating query is solved simply by listing all its pointers in $O(occ)$ time.

We note that the suffix array $\mathcal{A}$ is essentially the matrix $\mathcal{M}$ of the BWT (Section 2.2), as sorting the cyclic shifts of T is the same as sorting its suffixes given the endmarker "\$": $\mathcal{A}[i] = j$ if and only if the ith row of $\mathcal{M}$ contains the string $t_jt_{j+1} \ldots t_{n-1}\$t_1 \ldots t_{j-1}$.

A feature of suffix arrays that is essential for their compression is that they (may) contain *self-repetitions*. A self-repetition in $\mathcal{A}$ is an interval $\mathcal{A}[j \ldots j+\ell]$ that appears elsewhere, say in $\mathcal{A}[i \ldots i+\ell]$, so that all values are displaced by 1. This is, for any $0 \leq r \leq \ell$, it holds $\mathcal{A}[j+r] = \mathcal{A}[i+r]+1$. Self-repetitions were one of the first tools used to compact suffix arrays [28].

2.4 *The FM-index*

The FM-index [8, 9] is a self-index based on the Burrows-Wheeler transform. It solves counting queries by finding the interval of $\mathcal{A}$ that contains the occurrences of pattern P. The FM-index uses the array C and function $Occ(L, c, i)$ defined in Section 2.2. Fig. 1 shows the counting algorithm. Using the properties of the BWT, it is easy to see that the algorithm maintains the following invariant [8]: At the ith phase, variables sp and ep point, respectively, to the first and last row of $\mathcal{M}$ prefixed by $P[i, m]$. The correctness of the algorithm follows from this observation. Note that P is processed backwards, from p_m to p_1.

Algorithm FMcount($P[1,m]$,$T^{bwt}[1,n]$)
$i \leftarrow m$;
$sp \leftarrow 1$; $ep \leftarrow n$;
while $(sp \leq ep)$ **and** $(i \geq 1)$ **do**
    $c \leftarrow P[i]$;
    $sp \leftarrow C[c] + Occ(T^{bwt}, c, sp - 1)+1$;
    $ep \leftarrow C[c] + Occ(T^{bwt}, c, ep)$;
    $i \leftarrow i - 1$;
if $(ep < sp)$ **then return** "not found" **else return** "found $(ep - sp + 1)$ occurrences".

Fig. 1: FM-index algorithm for counting the number of occurrences of $P[1,m]$ in $T[1,n]$.

Note that while array C can be explicitly stored in little space, implementing $Occ(T^{bwt}, c, i)$ is problematic. The first solution [8] implemented $Occ(T^{bwt}, c, i)$ by storing a compressed representation of T^{bwt} plus some additional tables. With this representation, $Occ(T^{bwt}, c, i)$ could be computed in constant time and therefore the counting algorithm required $O(m)$ time.

The representation of T^{bwt} required $O(nH_k)$ bits of space, while the additional tables required space exponential in σ. Assuming that σ is constant, the space requirement of the FM-index is $5nH_k + o(n)$. In a practical implementation [9] this exponential dependence on σ was avoided, but the constant time guarantee for answering $Occ(T^{bwt}, c, i)$ was no longer valid.

Let us now consider how to locate the positions in $\mathcal{A}[sp, ep]$. The idea is that T is sampled at regular intervals, so that we explicitly store the positions in $\mathcal{A}$ pointing to the sampled positions in T (note that the sampling is not regular in $\mathcal{A}$). Hence, using the LF mapping, we move backward in T until finding a position that is known in $\mathcal{A}$. Then it is easy to infer our original text position. Fig. 2 shows the pseudocode.

Algorithm FMlocate(i,$T^{bwt}[1,n]$)
$i' \leftarrow i$, $t \leftarrow 0$;
while $\mathcal{A}[i']$ is not known **do**
    $i' \leftarrow LF(i') = C[T^{bwt}[i']] + Occ(T^{bwt}, T^{bwt}[i'], i')$;
    $t \leftarrow t + 1$;
return "text position is $\mathcal{A}[i'] + t$".

Fig. 2: FM-index algorithm for locating the occurrence $\mathcal{A}[i]$ in T.

We note that, in addition to C and Occ, we need access to characters $T^{bwt}[i']$ as well. In the original paper [8] this is computed in $O(\sigma)$ time by linearly looking for the character c such that $Occ(T^{bwt}, c, i') \neq Occ(T^{bwt}, c, i' - 1)$. Finally, if we sample one out of $\log^{1+\varepsilon} n$ positions in T, for any constant $\varepsilon > 0$, and use $\log n$ bits to represent each corresponding known $\mathcal{A}$ value, we require $O(n/\log^{\varepsilon} n) = o(n)$ additional bits of space and can locate the *occ* occurrences of P in $O(occ\ \sigma\ \log^{1+\varepsilon} n)$ time.[5]

Finally, let us consider displaying text contexts. To retrieve $T[l_1, l_2]$, we first find the position in $\mathcal{A}$ that points to l_2, and then issue $\ell = l_2 - l_1$ backward steps in T, using the LF mapping. Starting at the lowest marked text position that follows l_2, we perform $O(\log^{1+\varepsilon} n)$ steps until reaching l_2. Then we perform ℓ additional LF steps to collect the text characters. The resulting complexity is $O(\sigma\ (\ell + \log^{1+\varepsilon} n))$.

2.5 The compressed suffix array (CSA)

The *compressed suffix array (CSA)* [45] is a self-index based on an earlier succinct data structure [18]. In the CSA, the suffix array $\mathcal{A}[1, n]$ is represented by a sequence of numbers $\Psi(i)$, such that $\mathcal{A}[\Psi(i)] = \mathcal{A}[i] + 1$.[6] The sequence Ψ is differentially encoded, $\Psi(i + 1) - \Psi(i)$.

Note that if there is a self-repetition $\mathcal{A}[j \ldots j + \ell] = \mathcal{A}[i \ldots i + \ell] + 1$ (recall Section 2.3), then $\Psi(i \ldots i + \ell) = j \ldots j + \ell$, and thus $\Psi(i + 1) - \Psi(i) = 1$ in all that area. This property was used to represent Ψ using run-length compression in space proportional to nH_k [30, 29, 33], using ideas from this paper.

Yet, the original CSA achieved space proportional to nH_0 by different means. Note that the Ψ values are increasing in the areas of $\mathcal{A}$ where the suffixes start with the same character c, because $cX < cY$ if and only if $X < Y$ in lexicographic order. It is enough to store those increasing values differentially with a method like Elias coding to achieve $O(nH_0)$ overall space [45]. Some additional information is stored to permit constant time access to Ψ. This includes the same C array used by the FM-index. Considering all the structures, the CSA takes $n(H_0 + O(\log \log \sigma))$ bits of space.

A binary search on suffix array $\mathcal{A}$ is simulated by extracting strings of the form $t_{\mathcal{A}[i]} t_{\mathcal{A}[i]+1} t_{\mathcal{A}[i]+2} \ldots$ from the CSA, for any index i required by the binary search. The first character $t_{\mathcal{A}[i]}$ is easy to obtain because all the first characters of suffixes appear in order when pointed from $\mathcal{A}$, so $t_{\mathcal{A}[i]}$ is the character c such that $C[c] < i \leq C[c + 1]$. This is found in constant time by using small additional structures. Once the first character is obtained, we move to $i' \leftarrow \Psi(i)$ and go on with $t_{\mathcal{A}[i']} = t_{\mathcal{A}[i]+1}$. We continue until the result of the lexicographical comparison against the pattern P is clear. The overall search complexity is the same as with the original suffix array, $O(m \log n)$.

The method to locate occurrences could have been the same as for the FM-index (Section 2.4), using Ψ to move forward in the text instead of using the LF mapping to move backward. Note that the times are not multiplied by σ, so they can locate the *occ* occurrences in $O(occ\ \log^{1+\varepsilon} n)$ time and display a text substring of length ℓ in $O(\ell + \log^{1+\varepsilon} n)$ time, for any constant $\varepsilon > 0$. The reason behind the independence

[5] Actually, if one insists in that $\sigma = O(1)$, and thus the locate time is $O(occ\ \log^{1+\varepsilon} n)$, then it is possible to achieve $O(occ\ \log^{\varepsilon} n)$ time by enlarging the alphabet. This is not a choice if $\sigma = \omega(1)$.

[6] Since $\mathcal{A}[1] = n$ because $T[n, n] = \$$ is the smallest suffix, it should hold $\mathcal{A}[\Psi(1)] = n + 1$. For technical convenience we set $\Psi(1)$ so that $\mathcal{A}[\Psi(1)] = 1$, which makes Ψ a permutation of $[1, n]$.

of σ is that the CSA encodes Ψ explicitly (albeit compressed), whereas the FM-index does not encode the LF mapping but it needs to compute it using $T^{bwt}[i]$, so it needs to know the current character in order to move.

Yet, the CSA locates even faster (in $O(\log^{\varepsilon} n)$ steps) with a more complicated structure: the inverse of $\mathcal{A}$. This inverse permits moving by more than one text position at a time, and is implemented in succinct space using ideas in previous work [18]. The price is that the main term of the space complexity is actually $nH_0(1 + 1/\varepsilon)$.

A more recent variant of the CSA [46] achieves $O(m)$ counting time if $\sigma = O(\mathrm{polylog}(n))$, by means of simulating an FM-index-like backward search (Section 2.4). This is interesting because it shows a deep connection between the FM-index and the CSA structures. Even more important for this paper is that they solve the problem of computing $Occ(T^{bwt}, c, i)$ of the FM-index in constant time using $|T|H_0(T) + O(|T|)$ bits of space, provided the alphabet size of T is $\sigma = O(\mathrm{polylog}(|T|))$. This is done by storing σ bit arrays B_c such that $B_c[i] = 1$ if and only if $T^{bwt}[i] = c$, and thus $Occ(T^{bwt}, c, i) = rank_1(B_c, i)$ (Section 2.6). They manage to use a succinct representation for the B_c arrays [43] so as to get the desired space bounds.

2.6 *Succinct data structures for binary sequences*

Binary sequences are among the most basic data structures, and they are intensively used by succinct full-text indexes. Hence their succinct representation is of interest for these applications. In particular, *rank* and *select* queries over the compressed sequence representations are the most interesting ones.

Given a binary sequence $B = b_1 b_2 \ldots b_n$, we denote by $rank_b(B, i)$ the number of times bit b appears in the prefix $B[1, i]$, and by $select_b(B, i)$ the position in B of the ith occurrence of bit b. By default we assume $rank(B, i) = rank_1(B, i)$ and $select(B, i) = select_1(B, i)$.

There are several already classical results [22, 37, 6] that show how B can be represented using $n + o(n)$ bits so as to answer *rank* and *select* queries in constant time. The best current results [42, 43] are able to answer those queries in constant time, yet using only $nH_0(B) + o(n)$ bits of space. More precisely, the former [42] uses $nH_0(B) + O(n \log\log n / \log n)$ bits, and it answers in constant time all *rank* queries, also retrieving any bit b_i. The latter [43], on one hand, adds support for all *select* queries within the same $nH_0(B) + O(n \log\log n / \log n)$ bits of space used by the former [42]. On the other hand, it presents a more limited structure answering $rank_0(B, i)$ and $rank_1(B, i)$ only if $b_i = 1$, only $select_1(B, i)$ but not $select_0(B, i)$, and retrieving any bit b_i. In exchange, this last structure needs less space on sparse bit vectors: $nH_0(B) + o(\ell) + O(\log\log n)$ bits, where ℓ is the number of bits set in B.

We remark that these space bounds include that for representing B itself, so the binary sequence is being compressed, yet it allows those queries to be answered in optimal time.

2.7 Wavelet trees

Sequences $S = s_1 s_2 \ldots s_n$ on general alphabets of size σ can also be represented using $nH_0(S)+o(n\log\sigma)$ bits by using a *wavelet tree* [16]. Queries *rank* and *select* can be defined equivalently on general sequences. The wavelet tree takes $O(\log\sigma)$ time to answer those queries, as well as to retrieve character s_i.

The wavelet tree is a perfectly balanced binary tree where each node corresponds to a subset of the alphabet. The children of each node partition the node subset into two. A bitmap at the node indicates to which children does each sequence position belong. Each child then handles the subsequence of the parent's sequence corresponding to its alphabet subset. The leaves of the tree handle a single letter of the alphabet and require no space.

More formally, the root partition puts characters in $[1, \lfloor\sigma/2\rfloor]$ on the left child, and characters in $[\lfloor\sigma/2\rfloor + 1, \sigma]$ on the right child. A bitmap $B_{root}[1, n]$ is stored at the root node, so that $B[i] = 0$ if and only if $1 \leq S[i] \leq \lfloor\sigma/2\rfloor$ (that is, if the ith character of S belongs to the left child) and 1 otherwise. The two children are processed recursively. However, each of them considers the text positions whose character belongs to their subset. That is, the bitmap of the left child of the root will have only $n_1 + \ldots + n_{\lfloor\sigma/2\rfloor}$ bits and that of the right child only $n_{\lfloor\sigma/2\rfloor+1} + \ldots + n_\sigma$, where n_c is the number of occurrences of c in S.

To answer query $rank_c(S, i)$, we first determine to which branch of the root does c belong. If it belongs to the left, then we recursively continue at the left subtree with $i \leftarrow rank_0(B_{root}, i)$. Otherwise we recursively continue at the right subtree with $i \leftarrow rank_1(B_{root}, i)$. The value reached by i when we arrive at the leaf that corresponds to c is $rank_c(S, i)$. To answer $select_c(S, i)$ the process is bottom-up, starting at the leaf that corresponds to c and updating $i \leftarrow select_0(B_{node}, i)$ and $i \leftarrow select_1(B_{node}, i)$ depending on whether the current node is a left or right child. Finally, to find out s_i we go left or right in the tree depending on whether $B_{root}[i] =$ 0 or 1, and we end up at the leaf that corresponds to s_i. All those queries take $O(\log\sigma)$ time.

If every bitmap in the wavelet tree is represented using a data structure that takes space proportional to its zero-order entropy (Section 2.6), then it can be shown that the whole wavelet tree requires $nH_0(S) + o(n\log\sigma)$ bits of space [16].

When $\sigma = O(\text{polylog}(n))$, a generalization of wavelet trees takes $nH_0(S) + o(n)$ bits and answers all those queries in constant time [12].

3. Relating the *k*th order entropy with self-repetitions

In this section we prove a relation between the kth order entropy of a text T and both the number of self-repetitions in its suffix array (Section 2.3) and the number of runs of equal letters in the Burrows-Wheeler transformed text T^{bwt} (Section 2.2). In this proof we use some techniques already presented in a much more complete analysis [35]. Our analysis can be regarded as a simplified version that turns out to be enough for our purposes.

The concept of self-repetition has been used [28] to compact suffix arrays, essentially by replacing the areas $\mathcal{A}[i \ldots i + \ell]$ that appear elsewhere as $\mathcal{A}[j \ldots j + \ell] =$

$\mathcal{A}[i \ldots i+\ell]+1$, by pointers of the form (j,ℓ). Let us define the minimum number of self-repetitions necessary to cover a suffix array.

D 1. *Given a suffix array $\mathcal{A}$, n_{sr} is the minimum number of self-repetitions necessary to cover the whole $\mathcal{A}$. This is the minimum number of nonoverlapping intervals $[i_s, i_s+\ell_s]$ that cover the interval $[1,n]$ such that, for any s, there exists $[j_s, j_s+\ell_s]$ such that $\mathcal{A}[j_s+r] = \mathcal{A}[i_s+r]+1$ for all $0 \le r \le \ell_s$. (Note that $\Psi(i_s+r) = j_s+r$ for $0 \le r \le \ell_s$.)*

We show now that, in a cover of minimum size of self-repetitions, these have to be maximal, and that if self-repetitions are maximal, then the cover is of minimum size.

L 1. *Let $[i_s, i_s+\ell_s]$ be a cover of $[1,n]$ using nonoverlapping self-repetitions. Assume them to be sorted, thus $i_{s+1} = i_s+\ell_s+1$. If some self-repetition $[i_s, i_s+\ell_s]$ is not maximal, then the cover is not of minimum size.*

P . Let j_s be such that $\mathcal{A}[j_s] = \mathcal{A}[i_s]+1$. Assume that the interval $[i_s, i_s+\ell_s]$ can be extended to the right, that is, $\mathcal{A}[j_s+\ell_s+1] = \mathcal{A}[i_s+\ell_s+1]+1$. Then, since j_s is unique for each i_s (actually $j_s = \Psi(i_s)$), and since $i_s+\ell_s+1 = i_{s+1}$, we have $j_{s+1} = \Psi(i_{s+1}) = j_s+\ell_s+1$. Moreover, $\mathcal{A}[j_s+\ell_s+1+r] = \mathcal{A}[j_{s+1}+r] = \mathcal{A}[i_{s+1}+r]+1 = \mathcal{A}[i_s+\ell_s+1+r]+1$ for $0 \le r < \ell_{s+1}$. Thus, intervals $[i_s, i_s+\ell_s]$ and $[i_{s+1}, i_{s+1}+\ell_{s+1}]$ can be merged into one. The argument is similar if $[i_s, i_s+\ell_s]$ can be extended to the left. □

L 2. *Let $[i_s, i_s+\ell_s]$ be a cover of $[1,n]$ using nonoverlapping maximal self-repetitions. Then the cover is of minimum size.*

P . We simply note that there is only one possible cover where self-repetitions are maximal. Consider again that the intervals are sorted. Thus $i_1 = 1$ and ℓ_1 is maximal. Thus i_2 is fixed at $i_2 = i_1+\ell_1+1$ and ℓ_2 is maximal, and so on. □

The size of the *compact suffix array* [28] is actually $O(n_{sr} \log n)$. No useful bound on n_{sr} was obtained before. Our results in this section will permit bounding the size of the compact suffix array in terms of the kth order entropy of T.

Let us first define more conveniently the number of self-repetitions n_{sr} in a suffix array $\mathcal{A}$. As explained in Section 2.5, a self-repetition $\mathcal{A}[j \ldots j+\ell] = \mathcal{A}[i \ldots i+\ell]+1$ translates into the condition $\Psi(i \ldots i+\ell) = j \ldots j+\ell$. The following definition is convenient.

D 2. *A* run *in Ψ is any maximal interval $[i, i+\ell]$ in sequence Ψ such that $\Psi(r+1) - \Psi(r) = 1$ for all $i \le r < i+\ell$. Note that the number of runs in Ψ is n minus the number of positions r such that $\Psi(r+1) - \Psi(r) = 1$.*

The following lemma gives us an alternative definition of self-repetitions, which will be more convenient for us and is interesting in its own right to analyze the CSA.

L 3. *The number of self-repetitions n_{sr} to cover $\mathcal{A}$ is equal to the number of runs in Ψ.*

P . As explained in Section 2.5, there exists a self-repetition $\mathcal{A}[j \ldots j+\ell] = \mathcal{A}[i \ldots i+\ell]+1$ if and only if $\Psi(i \ldots i+\ell) = j \ldots j+\ell$, that is, if $\Psi(r+1)-\Psi(r) = 1$ for all $i \le r < i+\ell$. Therefore, each maximal self-repetition is also a (maximal) run in Ψ and vice versa. □

Let us now consider the number of equal-letter runs in $T^{bwt} = L$. The following definition and theorem permit us bounding n_{sr} in terms of those runs.

D 3. *Given a Burrows-Wheeler transformed text $T^{bwt}[1,n]$, n_{bw} is the number of equal-letter runs in T^{bwt}, that is, n minus the number of positions j such that $T^{bwt}[j+1] = T^{bwt}[j]$.*

T 1. *The following relation between n_{sr} and n_{bw} holds: $n_{sr} \le n_{bw} \le n_{sr}+\sigma$, where σ is the alphabet size of T.*

P . Let $L = T^{bwt}$ be the last column in matrix $\mathcal{M}$ of the BWT. If $L[j] = L[j+1] = c$, then $T[\mathcal{A}[j]-1] = cX$, $T[\mathcal{A}[j+1]-1] = cY$, $T[\mathcal{A}[j]] = X$, and $T[\mathcal{A}[j+1]] = Y$. Let i be such that $j = \Psi(i)$ and i' such that $j+1 = \Psi(i')$, then $\mathcal{A}[j] = \mathcal{A}[i]+1$ and $\mathcal{A}[j+1] = \mathcal{A}[i']+1$. Hence $T[\mathcal{A}[i]] = cX$ and $T[\mathcal{A}[i']] = cY$. Since $X < Y$, it follows that $cX < cY$ and therefore $i < i'$. Moreover, there cannot be any suffix cZ such that $cX < cZ < cY$ because in this case $X < Z < Y$, and thus the pointer to suffix Z should be between j and $j+1$. Since there is no such suffix, it follows that $i' = i+1$, that is, $\Psi(i') - \Psi(i) = \Psi(i+1) - \Psi(i) = (j+1) - j = 1$, and therefore i and $i+1$ belong to the same maximal self-repetition.

Recall that n_{bw} is n minus the number of cases where $L[j] = L[j+1]$, and similarly n_{sr} is n minus the number of cases where $\Psi(i+1)-\Psi(i) = 1$. Since there is a bijection Ψ between i and j, and thus every j such that $L[j] = L[j+1]$ induces a different i such that $\Psi(i+1) - \Psi(i) = 1$, it follows immediately that $n_{sr} \le n_{bw}$. Actually, the inverse of the above argument is also true except for the possibility of a self-repetition spanning an area where the first character of the suffixes changes. As this happens at most σ times, we have $n_{sr} \le n_{bw} \le n_{sr}+\sigma$. □

We have established the relation between the runs in Ψ, the self-repetitions in $\mathcal{A}$, and the runs in T^{bwt}. We now prove that the number of equal-letter runs in T^{bwt} is $n_{bw} \le nH_k + \sigma^k$, for any $k \ge 0$.

D 4. *Let $rle(S)$ be the* run-length encoding *of string S, that is, sequence of pairs (s_i, ℓ_i) such that $s_i \ne s_{i+1}$ and $S = s_1^{\ell_1} s_2^{\ell_2} \ldots$, where $s_i^{\ell_i}$ denotes character s_i repeated ℓ_i times. The length $|rle(S)|$ of $rle(S)$ is the number of pairs in it.*

Hence, we want to bound $n_{bw} = |rle(T^{bwt})|$. An important observation for our development follows:

O 1. For any partition of S into consecutive substrings $S = S_1 S_2 \ldots S_p$, it holds $|rle(S)| \le |rle(S_1)| + |rle(S_2)| + \ldots + |rle(S_p)|$, as the runs are the same except in the frontiers between S_i and S_{i+1}, where a run in S can be split into two.

Recall string W_T as defined in Section 2.1 for a k-context W of string T. Note that we can apply any permutation to W_T so that Eq. (1) still holds. Now, characters in $T^{bwt} = L$ are sorted by the text suffix that follows them (that is, by their row in $\mathcal{M}$), and thus they are ordered by their k-context, for any k. This means that all the characters in W_T, for any $W \in \Sigma^k$, appear consecutively in T^{bwt}. Thus, T^{bwt} is precisely the concatenation of all the strings W_T for $W \in \Sigma^k$, if we take the order of characters inside each W_T according to how they appear in T^{bwt} [35]. As a consequence, we have that[7]

$$n_{bw} \;=\; |rle(T^{bwt})| \;\leq\; \sum_{W \in \Sigma^k} |rle(W_T)|, \tag{2}$$

where the permutation of each W_T is now fixed by T^{bwt}. In fact, Eq. (2) holds also if we fix the permutation of each W_T so that $|rle(W_T)|$ is maximized. This observation gives us a tool to upper bound $|rle(T^{bwt})|$ by the sum of code lengths when zero-order entropy encoding is applied to each W_T separately. We next show that $|rle(W_T)| \leq 1 + |W_T| H_0(W_T)$.

Let us call σ_S the alphabet size of S. First notice that if $\sigma_{W_T} = 1$ then $|rle(W_T)| = 1$ and $|W_T| H_0(W_T) = 0$, so our claim holds. Let us then assume that $\sigma_{W_T} = 2$. Let x and y ($x \leq y$) be the number of occurrences of the two letters, say a and b, in W_T, respectively. It is easy to see analytically that

$$H_0(W_T) \;=\; -(x/(x+y))\log(x/(x+y)) - (y/(x+y))\log(y/(x+y)) \;\geq\; 2x/(x+y). \tag{3}$$

The permutation of W_T that maximizes $|rle(W_T)|$ is such that there is no run of symbol a longer than 1. This makes the number of runs in $rle(W_T)$ to be $2x + 1$ in that case. By using Eq. (3) and since $|W_T| = x + y$ we have that

$$|rle(W_T)| \;\leq\; 2x + 1 \;=\; 1 + 2|W_T| x/(x+y) \;\leq\; 1 + |W_T| H_0(W_T). \tag{4}$$

We are left with the case $\sigma_{W_T} > 2$. This case splits into two sub-cases: (*i*) the most frequent symbol occurs at least $|W_T|/2$ times in W_T; (*ii*) all symbols occur less than $|W_T|/2$ times in W_T. Case (*i*) becomes analogous to case $\sigma_{W_T} = 2$ once x is redefined as the sum of occurrences of symbols other than the most frequent. In case (*ii*) $|rle(W_T)|$ can be $|W_T|$. On the other hand, $|W_T| H_0(W_T)$ must also be at least $|W_T|$, since it holds that $-\log(x/|W_T|) \geq 1$ for $x \leq |W_T|/2$, where x is the number of occurrences of any symbol in W_T. Therefore we can conclude that Eq. (4) holds for any W_T.

Combining Eqs. (1), (2) and (4) we get the following result:

T 2. *The length n_{bw} of the run-length encoded Burrows-Wheeler transformed text $T^{bwt}[1, n]$ is at most $nH_k(T) + \sigma^k$, for any $k \geq 0$. In particular, this is $nH_k(T) + o(n)$ for any $k \leq \alpha \log_\sigma n$, for any constant* $0 < \alpha < 1$.

[7] Note that, since the endmarker "\$" $\in \Sigma$, we are including in Σ^k contexts that include the endmarker. Those let us account for the last k characters of T: Instead of considering that those characters have a context shorter than k, we assume t_{n-i} belongs to context $t_{n-i+1} \ldots t_n \$^{k-i}$.

This theorem has two immediate applications to existing compressed indexes, all valid for any $k \leq \alpha \log_\sigma n$, for any constant $0 < \alpha < 1$. Note that of course $n_{br} \leq n$, so H_k actually stands for $\min(1, H_k)$.

(1) The size of the compact suffix array [28] is $O(nH_k \log n)$ bits. This is because the compact suffix array stores a constant number of pointers for each maximal self-repetition of the text, and there are $n_{sr} \leq n_{bw}$ self-repetitions. No previous useful analysis existed for this structure.

(2) A run-length compression of array Ψ permits storing the compressed suffix array (CSA) [45] in $nH_k(\log \sigma + \log\log n) + O(n)$ bits. It is still possible to search that CSA in $O(m \log n)$ time. This was already shown [33] using the ideas from this paper [29]. Yet, an extra constant 2 appeared in that case [33] due to our unnecessarily pessimistic previous analysis [29].

(3) A combination of the compact suffix array and the CSA, called CCSA for "compressed compact suffix array" [29] is a self index using $O(nH_k \log n)$ bits. Actually, this analysis was first presented in that paper to analyze the CCSA.

In the next section we use the result to design a new compressed index.

4. RLFM: A run-length-based FM-index

In Section 3, we have shown that the number of runs in the BWT transformed text is $nH_k + o(n)$ for $k \leq \alpha \log_\sigma n$, $0 < \alpha < 1$. We aim in this section at indexing only the runs of T^{bwt}, so as to obtain an index, called *run-length FM-index (RLFM)*, whose space is proportional to nH_k.

We exploit run-length compression to represent T^{bwt} as follows. An array S contains one character per run in T^{bwt}, while an array B contains n bits and marks the beginnings of the runs.

D 5. *Let string $T^{bwt} = c_1^{\ell_1} c_2^{\ell_2} \ldots c_{n_{bw}}^{\ell_{n_{bw}}}$ consist of n_{bw} runs, so that the ith run consists of ℓ_i repetitions of character c_i. Our representation of T^{bwt} consists of string $S = c_1 c_2 \ldots c_{n_{bw}}$ of length n_{bw}, and of bit array $B = 10^{\ell_1 - 1} 10^{\ell_2 - 1} \ldots 10^{\ell_{n_{bw}} - 1}$.*

It is clear that S and B contain enough information to reconstruct T^{bwt}: $T^{bwt}[i] = S[rank(B, i)]$. Since there is no useful entropy bound on B, we assume that *rank* is implemented in constant time using some succinct structure that requires $n + o(n)$ bits [22, 6, 37]. Hence, S and B give us a representation of T^{bwt} that permit us accessing any character in constant time.

The problem, however, is not only how to access T^{bwt}, but also how to compute $C[c] + Occ(T^{bwt}, c, i)$ for any c and i (recall Fig. 1). This is not immediate, because we want to add up all the run lengths corresponding to character c up to position i.

In the following we show that the above can be computed by means of a bit array B', obtained by reordering the runs of B in lexicographic order of the characters of each run. Runs of the same character are left in their original order. The use of B' will add other $n + o(n)$ bits to our scheme. We also use C_S, which plays the same role of C, but it refers to string S.

D 6. *Let* $S = c_1 c_2 \ldots c_{n_{bw}}$ *of length* n_{bw}*, and* $B = 10^{\ell_1 - 1} 10^{\ell_2 - 1} \ldots 10^{\ell_{n_{bw}} - 1}$. *Let* $d_1 d_2 \ldots d_{n_{bw}}$ *be the permutation of* $[1, n_{bw}]$ *such that, for all* $1 \le i < n_{bw}$*, either* $c_{d_i} < c_{d_{i+1}}$*, or* $c_{d_i} = c_{d_{i+1}}$ *and* $d_i < d_{i+1}$*. Then, bit array* B' *is defined as* $B' = 10^{\ell_{d_1} - 1} 10^{\ell_{d_2} - 1} \ldots 10^{\ell_{d_{n_{bw}}} - 1}$. *Let also* $C_S[c] = |\{i,\ c_i < c,\ 1 \le i \le n_{bw}\}|$.

We now prove our main results. We start with two general lemmas.

L 4. *Let* S *and* B' *be defined for a string* T^{bwt}*. Then, for any* $c \in \Sigma$ *it holds*

$$C[c] + 1 \quad = \quad select(B', C_S[c] + 1).$$

P . $C_S[c]$ is the number of runs in T^{bwt} that represent characters smaller than c. Since in B' the runs of T^{bwt} are sorted in lexicographic order, $select(B', C_S[c] + 1)$ indicates the position in B' of the first run belonging to character c, if any. Therefore, $select(B', C_S[c] + 1) - 1$ is the sum of the run lengths for all characters smaller than c. This is, in turn, the number of occurrences of characters smaller than c in T^{bwt}, $C[c]$. Hence $select(B', C_S[c] + 1) - 1 = C[c]$. □

L 5. *Let* S*,* B*, and* B' *be defined for a string* T^{bwt}*. Then, for any* $c \in \Sigma$ *and* $1 \le i \le n$*, such that* i *is the final position of a run in* B*, it holds*

$$C[c] + Occ(T^{bwt}, c, i) \quad = \quad select(B', C_S[c] + 1 + Occ(S, c, rank(B, i))) - 1.$$

P . Note that $rank(B, i)$ gives the position in S of the run that finishes at i. Therefore, $Occ(S, c, rank(B, i))$ is the number of runs in $T^{bwt}[1, i]$ that represent repetitions of character c. Hence it is clear that $C_S[c] < C_S[c] + 1 + Occ(S, c, rank(B, i)) \le C_S[c+1] + 1$, from which follows that $select(B', C_S[c] + 1 + Occ(S, c, rank(B, i)))$ points to an area in B' belonging to character c, or to the character just following that area. Inside this area, the runs are ordered as in B because the reordering in B' is stable. Hence $select(B', C_S[c] + 1 + Occ(S, c, rank(B, i)))$ is $select(B', C_S[c] + 1)$ plus the sum of the run lengths representing character c in $T^{bwt}[1, i]$. That sum of run lengths is $Occ(T^{bwt}, c, i)$. The argument holds also if $T^{bwt}[i] = c$, because i is the last position of its run and therefore counting the whole run $T^{bwt}[i]$ belongs to is correct. Hence $select(B', C_S[c] + 1 + Occ(S, c, rank(B, i))) = select(B', C_S[c]+1) + Occ(T^{bwt}, c, i)$, and then, by Lemma 4, $select(B', C_S[c] + 1 + Occ(S, c, rank(B, i))) - 1 = C[c] + Occ(T^{bwt}, c, i)$. □

We now prove our two fundamental lemmas that cover different cases in the computation of $C[c] + Occ(T^{bwt}, c, i)$.

L 6. *Let* S*,* B*, and* B' *be defined for a string* T^{bwt}*. Then, for any* $c \in \Sigma$ *and* $1 \le i \le n$*, such that* $T^{bwt}[i] \ne c$*, it holds*

$$C[c] + Occ(T^{bwt}, c, i) \quad = \quad select(B', C_S[c] + 1 + Occ(S, c, rank(B, i))) - 1.$$

P . Let i' be the last position of the run that precedes that of i. Since $T^{bwt}[i] \ne c$ in the run i belongs to, we have $Occ(T^{bwt}, c, i) = Occ(T^{bwt}, c, i')$ and also $Occ(S, c, rank(B, i)) = Occ(S, c, rank(B, i'))$. Then the lemma follows trivially by applying Lemma 5 to i'. □

L 7. *Let S, B, and B' be defined for a string T^{bwt}. Then, for any $c \in \Sigma$ and $1 \le i \le n$, such that $T^{bwt}[i] = c$, it holds*

$$C[c] + Occ(T^{bwt}, c, i) \;=\; select(B', C_S[c] + Occ(S, c, rank(B, i))) + i \;-\; select(B, rank(B, i)).$$

P . Let i' be the last position of the run that precedes that of i. By Lemma 5 we have $C[c]+Occ(T^{bwt}, c, i') = select(B', C_S[c]+1+Occ(S, c, rank(B, i')))-1$. Now, $rank(B, i') = rank(B, i) - 1$, and since $T^{bwt}[i] = c$, it follows that $S[rank(B, i)] = c$. Therefore, $Occ(S, c, rank(B, i')) = Occ(S, c, rank(B, i)-1) = Occ(S, c, rank(B, i))-1$. On the other hand, since $T^{bwt}[i''] = c$ for $i' < i'' \le i$, we have $Occ(T^{bwt}, c, i) = Occ(T^{bwt}, c, i') + (i - i')$. Thus, the outcome of Lemma 5 can now be rewritten as $C[c] + Occ(T^{bwt}, c, i) - (i - i') = select(B', C_S[c] + Occ(S, c, rank(B, i))) - 1$. The only remaining piece to prove the lemma is that $i - i' - 1 = i - select(B, rank(B, i))$, that is, $select(B, rank(B, i)) = i' + 1$. But this is clear, since the left term is the position of the first run i belongs to and i' is the last position of the run preceding that of i. □

Since functions *rank* and *select* can be computed in constant time, the only obstacle to complete the RLFM using Lemmas 6 and 7 is the computation of *Occ* over string S. We use the idea explained at the end of Section 2.5 [46]. Instead of representing S explicitly, we store one bitmap S_c per text character c, so that $S_c[i] = 1$ if and only if $S[i] = c$. Hence $Occ(S, c, i) = rank(S_c, i)$. It is still possible to determine in constant time whether $T^{bwt}[i] = c$ or not (so as to know whether to apply Lemma 6 or 7): $T^{bwt}[i] = c$ if and only if $S_c[rank(B, i)] = 1$. Thus the RLFM can answer counting queries in $O(m)$ time.

From the space analysis of the original article [46], we have that the bit arrays S_c can be represented in $|S|H_0(S)+O(|S|)$ bits. The length of sequence S is $|S| = n_{bw} \le nH_k + \sigma^k$, which is $nH_k + o(n)$ for $k \le \alpha \log_\sigma n$, for any constant $0 < \alpha < 1$. So the space for the S_c arrays is $(nH_k+o(n))(H_0(S)+O(1))$. Since run-length compression removes some redundancy, it is expected that $H_0(S) \ge H_0(T)$ (although this might not be the case). Yet, the only simple bound we know of is $H_0(S) \le \log\sigma$. Thus the space can be upper bounded by $(nH_k + o(n))(\log\sigma + O(1))$. Note that the $o(n)$ term comes from Theorem 2, and it is in fact $O(n^\alpha)$ for some $\alpha < 1$, so it is still $o(n)$ after multiplying it by $\log\sigma = O(\log n)$. Thus, the space requirement can be written as $nH_k(\log\sigma + O(1)) + o(n)$.

In addition to arrays S_c, the representation of our index needs the bit arrays B and B', plus the sublinear structures to perform *rank* and/or *select* over them, and finally the small array C_S. These add $2n + o(n)$ bits, for a grand total of $n(H_k(\log\sigma + O(1)) + 2) + o(n)$ bits. As H_k actually stands for $\min(1, H_k)$ (see the end of Section 3), we can simplify the space complexity to $nH_k \log\sigma + O(n)$ bits.

We recall that the solution we build on [46] works only for $\sigma = O(\text{polylog}(|S|))$. This is equivalent to the condition $\sigma = O(\text{polylog}(n))$ if, for example, $n_{bw} \ge n^\beta$ for some $0 < \beta < 1$. In the unlikely case that the text is so compressible that $n_{bw} = o(n^\beta)$ for any $0 < \beta < 1$, we can still introduce n^β artificial cuts in the runs so as to ensure that polylog(n) and polylog(n_{bw}) are of the same order.[8] This increases the space by a small $o(n)$ factor that does not affect the main term. In exchange, the RLFM index works for any $\sigma = O(\text{polylog}(n))$.

We have chosen to build over Sadakane's work [46] for historical reasons. However, we would like to remark that very recent developments [12] use multi-ary wavelet trees to obtain the same result with a somewhat simpler approach. That is, they can represent a sequence S over alphabet of size $\sigma = O(\text{polylog}(|S|))$ and answer *rank* and *select* queries over S in constant time.

Let us now consider the task of locating occurrences and displaying text. For this sake we can use the same marking strategy of the FM-index. Furthermore, to access $T^{bwt}[i]$ we can use the equivalence $T^{bwt}[i] = S[rank(B, i)]$, so the problem of accessing $T^{bwt}[i]$ becomes the problem of accessing $S[j]$. Just as in the FM-index, we can use the same *Occ* function to find out $S[j]$ in $O(\sigma)$ time, which yields the same FM-index complexities in all cases.

Note, however, that in this case we can afford the explicit representation of S in addition to the bit arrays S_c, at the cost of $nH_k \log\sigma + o(n)$ more bits of space. This gives us constant time access to T^{bwt} and thus completely removes σ from all RLFM index time complexities.

T 3. *The RLFM index, of size* $n \min(H_k, 1) \log\sigma + O(n) = nH_k \log\sigma + O(n)$ *bits for any* $k \leq \alpha \log_\sigma n$, *for any constant* $0 < \alpha < 1$, *can be built on a text* $T[1, n]$ *whose alphabet is of size* $\sigma = O(\text{polylog}(n))$, *so that the number of occurrences of any pattern* $P[1, m]$ *in* T *can be counted in time* $O(m)$ *and then each such occurrence can be located in time* $O(\sigma \log^{1+\varepsilon} n)$, *for any constant* $\varepsilon > 0$ *determined at index construction time. Also, any text substring of length* ℓ *can be displayed in time* $O(\sigma (\ell + \log^{1+\varepsilon} n))$. *By letting it use* $2n \min(H_k, 1) \log\sigma + O(n)$ *bits of space, the RLFM index can count in* $O(m)$ *time, locate occurrences in time* $O(\log^{1+\varepsilon} n)$, *and display text substrings of length* ℓ *in* $O(\ell + \log^{1+\varepsilon} n)$ *time.*

Binary alphabets. It is interesting to consider the case of a text T over a binary alphabet, say $\Sigma = \{a, b\}$. In this case, since the runs alternate, we have $S[2i] = a$ and $S[2i + 1] = b$ or vice versa depending on the first value $S[1]$. One has also to consider the only j for which $S[j] = \$$, as the even/odd rule may change after j. Thus, it should be clear that we do not need to represent S at all, and that, moreover, it is easy to answer $Occ(S, c, i)$ in constant time without any storage: For example, if $S[1] = a$ and $S[j + 1] = a$, then $Occ(S, a, i) = \lfloor (i + 1)/2 \rfloor$ for $i < j$ and $Occ(S, a, i) = \lfloor j/2 \rfloor + \lfloor (i - j + 1)/2 \rfloor$ for $i \geq j$. Similar rules can be derived for the other three cases of $S[1]$ and $S[j + 1]$. Therefore, on a binary alphabet the RLFM index requires only arrays B and B', which take $2n + o(n)$ bits of space, and maintains the complexities of Theorem 3. This result almost matches some recent developments [20], albeit for this case the FM-index [8] still obtains better results.

[8] This can be done because we never used the fact that consecutive runs must correspond to different characters.

5. Practical considerations

Up to now we have considered only theoretical issues. In this section we focus on practical considerations on the implementation of the RLFM index. Several theoretical developments in this area require considerable work in order to turn them into practical results [9, 45, 41, 17].

The most problematic aspect of our proposal (and of several others) is the heavy use of techniques to represent sparse bitmaps in a space proportional to its zero-order entropy [43, 46, 12]. Those techniques, albeit theoretically remarkable, are not so simple to implement. Yet, previous structures supporting *rank* in $n + o(n)$ bits [22, 37, 6] are considerably simpler.

The problem is that, if we used the simpler techniques for our sparse bitmaps S_c, we would need $n_{bw}(1 + o(1))$ bits for each of them, and would require $nH_k\sigma$ bits at least for the RLFM index, far away from the theoretical $nH_k \log\sigma$. This can be improved by using a wavelet tree (Section 2.7) built on the S string of the RLFM index (that is, the run heads), instead of the individual bitmaps S_c. The wavelet tree is simple to implement, and if it uses structures of $n+o(n)$ bits to represent its binary sequences, it requires overall $n_{bw} \log\sigma(1+o(1)) = nH_k \log\sigma(1+o(1))$ bits of space to represent S. This is essentially the same space used by the individual bit arrays (in a worst-case sense, as the real space complexity of Section 4 is $nH_kH_0(S)$).

With the wavelet tree, both the $O(1)$ time to compute $Occ(S, c, i) = rank_c(S, i)$ and the $O(\sigma)$ time to compute $S[i]$, become $O(\log\sigma)$. Therefore, a RLFM index implementation based on wavelet trees counts in $O(m \log\sigma)$ time, locates each occurrence in $O(\log\sigma \ \log^{1+\varepsilon} n)$ time, and displays any text substring of length ℓ in $O(\log\sigma\ (\ell + \log^{1+\varepsilon} n))$, for any constant $\varepsilon > 0$.

The same idea can also be applied on the structure that uses sparse bit arrays without run-length compression [46]. Let us call SSA (for "succinct suffix array") this implementation variant of the original structure [46]. Since in the SSA the bit arrays B_c are built over the whole T (not only over the heads of runs), the SSA index requires $n \log\sigma(1 + o(1))$ space, which is at least as large as the plain representation of T.

We propose now another simple wavelet tree variant that permits us representing the SSA using $n(H_0 + 1)(1 + o(1))$ bits of space, and obtains on average $O(H_0)$ rather than $O(\log\sigma)$ time for the queries on the wavelet tree.

Imagine that instead of a balanced binary tree, we use the Huffman tree of T to define the shape of the wavelet tree. Then, every character $c \in \Sigma$ will have its corresponding leaf at depth h_c, so that $\sum_{c\in\Sigma} h_c n_c \le n(H_0 + 1)$ is the number of bits of the Huffman compression of T (recall from Section 2.1 that n_c is the number of times character c occurs in T).

Let us now consider the size of the Huffman-shaped wavelet tree. Note that each text occurrence of each character $c \in \Sigma$ appears exactly in h_c bit arrays (those found from the root to the leaf that corresponds to c), and therefore it takes h_c bits spread over the different bit arrays. Summed over all the occurrences of all the characters we obtain the very same length of the Huffman-compressed text, $\sum_{c\in\Sigma} h_c n_c$. Hence the overall space is $n(H_0 + 1)(1 + o(1))$ bits.

Note that the time to retrieve $T^{bwt}[i]$ is proportional to the length of the Huffman code for $T^{bwt}[i]$, which is $O(H_0)$ if i is chosen at random. In the case of $Occ(T^{bwt}, c, i) = rank_c(T^{bwt}, i)$, the time corresponds again to $T^{bwt}[i]$ and is independent of c. Under reasonable assumptions, one can say that on average this version of the SSA counts in $O(H_0 m)$ time, locates an occurrence in $O(H_0 \log^{1+\varepsilon} n)$ time, and displays a text substring of length ℓ in $O(H_0(\ell + \log^{1+\varepsilon} n))$ time. It is possible (but not good in practice) to force the Huffman tree to have $O(\log \sigma)$ height and still have average depth limited by $H_0 + 2$, so we can ensure the same worst case factor $O(\log \sigma)$ instead of $O(H_0)$ [14].

Finally, we note that the Huffman-shaped wavelet tree can be used instead of the balanced version for the RLFM index. This lowers its space requirement again to $nH_k H_0(S)$, just like the theoretical version. It also reduces the average time to compute $rank_c(S, i)$ or $S[i]$ to $O(H_0(S))$, which is no worse than $O(\log \sigma)$.

6. Experiments

In this section we compare our SSA, CCSA[9] and RLFM implementations against other succinct index implementations we are aware of, as well as other more classical solutions. All these are listed below .

FM [8, 9]: The original *FM-index* (Section 2.4) implementation by the authors. The executables can be downloaded from `http://www.mfn.unipmn.it/~manzini/fmindex`.

FM-Nav [40]: An implementation of the *FM-index* by G. Navarro, downloadable from `http://www.dcc.uchile.cl/~gnavarro/software`. This implementation is faster than the original but uses more space, as it represent the Burrows-Wheeler transformed text as such.

CSA [45]: The *Compressed Suffix Array* (Section 2.5) implementation by the author. The code can be downloaded from `http://www.dcc.uchile.cl/~gnavarro/software`.

LZ [41]: The *Lempel-Ziv self-index* implementation by the author. The code can be downloaded from `http://www.dcc.uchile.cl/~gnavarro/software`. The implementation has been improved since the original publication.

CompactSA [28]: The *Compact Suffix Array* implementation by the author. This is not a self-index but a succinct index based on suffix array self-repetitions, useful to show which is the price of not having the text directly available. The code can be downloaded from `http://www.cs.helsinki.fi/u/vmakinen/software`.

SA [34]: The classical suffix array structure, using exactly $n\lceil \log n \rceil$ bits. We use it to test how much the above succinct structures lose in query times to the simple $O(m \log n)$ binary search algorithm.

[9] The CCSA belongs to the development that finished with the RLFM index [29]. We have excluded it from this paper because it is superseded by the RLFM index. Yet, it is interesting to show its relative performance.

BMH [21]: Our implementation of the classical sequential search algorithm, requiring only the plain text. This is interesting to ensure that there is some value in indexing versus sequentially scanning the text. Yet, note that just the plain text requires more space than several self-indexes.

The codes for the SSA, CCSA and RLFM index used in these experiments can be downloaded from `http://www.cs.helsinki.fi/u/vmakinen/software`. We made use of the practical considerations of Section 5. In particular, we use Huffman-shaped wavelet trees in both cases. Another complicated and time-critical part were the rank and select structures, as our indexes make heavier use of them compared to other implementations. Although the existing theoretical solutions [22, 37, 6] are reasonably efficient in practice, we needed to engineer them further to make them faster and less space consuming [13]. Those optimized variants were also used to improve other existing structures that used them, namely LZ and CCSA.

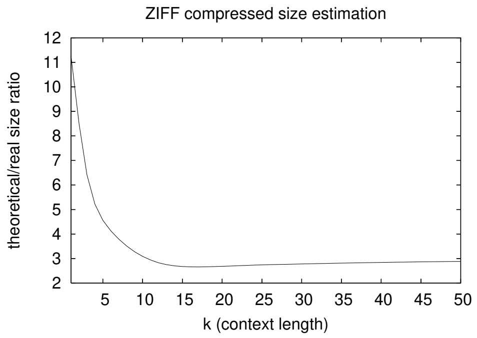
Our experiments were run over an 83 MB text collection obtained from the "ZIFF-2" disk of TREC-3 [19]. The exact size is of this T is $n = 87,415,922$ and its alphabet size is $\sigma = 96$. The number of runs in T^{bwt} is $n_{bw} = 29,014,589$, so $n_{bw}/n \approx 0.33$. The zero-order entropy of T is 5.04 bits/symbol, and that of the run heads, S, is very close, 5.03 bits/symbol. With this we can predict that our RLFM will take about $\frac{4}{3}(n_{bw}H_0(S) + 2n)/8 = 50.86$ megabytes, that is, 61% of the text size. The rationale is as follows: n_{bw} is the length of the text S we actually represent, in space close to $H_0(S)$ bits/symbol using the Huffman-shaped wavelet trees; arrays B and B' take n bits each; and we use rank/select structures over all that data, which in our implementation [13] takes 1/3 extra space overhead. As one can see in Table II, the estimation is quite accurate (the real size is 63% of the text size, without the structures for reporting).

Figure 3 displays some more statistical data on this text. We show the evolution of H_k and the number of different contexts as k grows. We also give a reality check of the theoretical bound $n_{bw} \leq nH_k + \sigma^k$ given in Theorem 2. For this sake, we divide $nH_k + c(k)$, being c the real number of contexts of order k, by n_{bw} (note that we use the empirical $c(k)$, which is much smaller than its brute bound σ^k). This ratio is minimized for $k = 17$, where the theoretical prediction is still 2.659 times the real n_{bw} value of this text. This explains why, as we see soon, our practical results on RLFM index size are far better than those we can prove in theory.

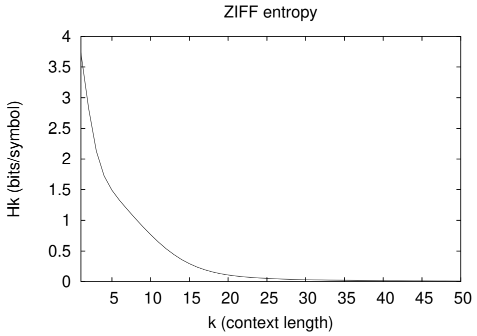
The tests ran on a Pentium IV processor at 2.6 GHz, 2 GB of RAM and 512 KB cache, running Red Hat Linux 3.2.2-5. We compiled the code with `gcc 3.2.2` using optimization option `-O3`. Times were averaged over 10,000 search patterns. As we work only in main memory, we only consider CPU times. The search patterns were obtained by pruning random text lines to their first m characters, but we avoided lines containing tags and non-visible characters.

We prepared the following test setups to compare different indexes.

Count settings: We tuned the indexes so that they only support counting queries. This usually means that they take the minimum possible space to operate.



k	H_k (bits/symb)	# contexts ($\leq \sigma^k$)	$\frac{nH_k+\sigma^k}{n_{bw}}$
0	5.036	1	15.171
1	3.736	97	11.256
2	2.824	7,215	8.510
3	2.128	105,441	6.416
4	1.725	579,007	5.216
5	1.493	1,841,918	4.561
6	1.324	4,059,661	4.130
7	1.176	7,250,505	3.793
8	1.034	11,492,049	3.511
9	0.896	16,703,680	3.276
10	0.766	22,588,340	3.088



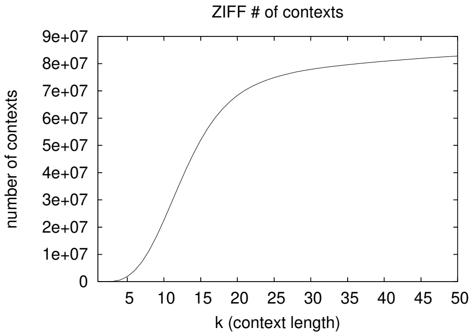


Fig. 3: Some statistics on our test text. In the theoretical prediction we replace σ^k by the real number of different contexts.

Same sample rate: For reporting queries most of the compared indexes use the same text tracking mechanism. It is thus interesting to see what happens when exactly the same number of suffixes are sampled (one out of 28 in our experiment).

Same size: We tuned all indexes to use about the same size (1.6 times the text size) by adjusting the space-time tradeoff for locating queries.

Control against other solutions: It is interesting to see how our compressed indexes behave against classical full-text indexes, plain sequential search, or non-self indexes.

Table II shows the index sizes under the different settings, as a fraction of the text size. Recall that these are self-indexes that replace the text. For consistency we have added the text size (that is, 1.00) to the options CompactSA, SA and BMH, as they need the text separately available.

Only the CSA has a space-time tradeoff on counting queries. For this reason we ran the counting experiment on several versions of it. These versions are denoted by CSAX in the table, where X is the tradeoff parameter (sample rate in Ψ array). For reporting queries (row labeled CSA), we used the default value $X = 128$. Other cells are missing because we could not make FM take that much space in the "same size" setting, or because alternative structures cannot take that little space, or because some structures have no concept of sampling rate.

T II: Sizes of the indexes tested under different settings.

index	count	same sample rate	same size
FM	0.36	0.41	—
FM-Nav	1.07	1.21	1.57
CSA	0.44	0.58	1.59
CSA10	1.16	—	—
CSA16	0.86	—	—
CSA32	0.61	—	—
CSA256	0.39	—	—
LZ	1.49	—	1.49
SSA	0.87	1.33	1.58
CCSA	1.65	—	1.65
RLFM	0.63	1.09	1.60
CompactSA	2.73	—	—
SA	4.37	—	—
BMH	1.00	—	—

Fig. 4 (left) shows the times to count pattern occurrences of length $m = 5$ to $m = 60$. To avoid cluttering the plot we omit CSA10 and CSA16, whose performace is very similar to CSA32. It can be seen that FM-Nav is the fastest alternative, but it is closely followed by our SSA, which needs 20% less space (0.87 times the text size). The next group is formed by our RLFM and the CSA, both needing around 0.6 times the text size. Actually RLFM is faster, and to reach its performance we need CSA10, which takes 1.16 times the text size. For long patterns CCSA becomes competitive in this group, yet it needs as much as 1.65 times the text size.

On the right of Fig. 4 we chose $m = 30$ and plot the times as a function of the space. This clearly shows which indexes represent an interesting space-time tradeoff for counting. SSA and RLFM indexes are among the relevant ones. Note for example that the SSA is the fastest counting index among those that take less space than the text. Also, the RLFM index counts faster than a CSA of the same size.

Fig. 5 shows the times to locate all the pattern occurrences. On the left we consider the same sampling rate for all applicable indexes. That is, all indexes make about the same number of steps to traverse the text until finding the occurrence position. It can be seen that FM-Nav is the best choice, albeit closely followed by SSA.

On the right of Fig. 5 we show the fairer comparison that gives about the same space to all structures. The space was chosen to be around 1.6 times the text size because this is close to the space requirement of LZ, an index that is relevant for this

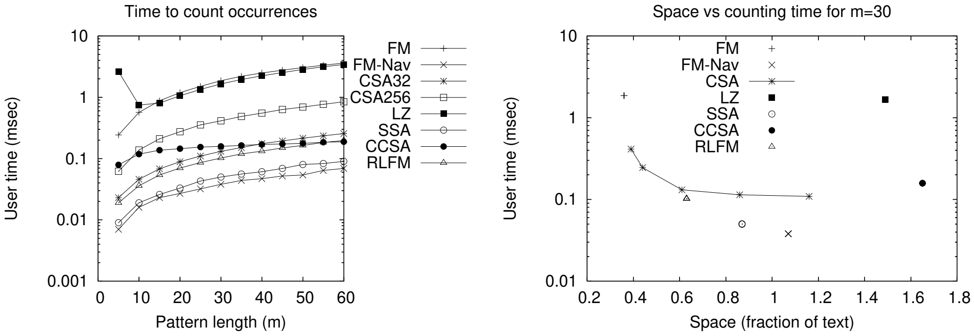


Fig. 4: Query times for counting the number of occurrences. On the left, time versus m. On the right, time versus space for $m = 30$.

task and whose size cannot be tuned. The other indexes were tuned by increasing their sampling rate.

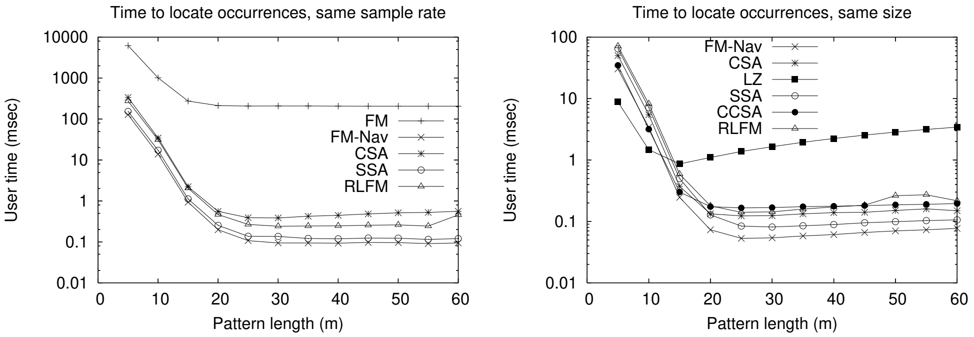


Fig. 5: Times for locating the pattern occurrences. On the left, under the same sample rate setting. On the right, all indexes using about the same space.

In this case LZ shows up as the fastest structure for locating, followed by FM-Nav, which takes over as soon as there are less occurrences to locate and the high counting times of LZ render it non-competitive. Our indexes perform reasonably well but are never the best for this task. Fig. 6 shows the space-time tradeoffs for locating times, illustrating our conclusions. Note that RLFM gives more interesting locating tradeoffs than SSA.

Our final experiment is to compare how our new structures compare against some alternative structures such as the original suffix array and the (succinct but not self-index) compact suffix array. It is also interesting to see how much slower or faster is sequential search compared to our indexes. The results are shown in Fig. 7.

It can be seen that the self-indexes are considerably fast for counting, especially for short patterns. For longer ones, their small space consumption is paid in a 10X slowdown for counting. Yet, this is orders of magnitude faster than a sequential search, which still needs more space as the text has to be in uncompressed form

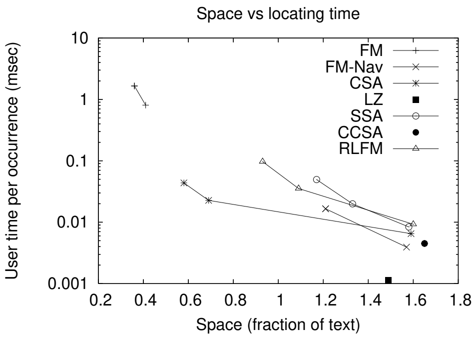


Fig. 6: Comparison of locating performance versus space requirement, for $m = 5$. We show the time per occurrence, not per pattern as in the rest of the experiments.

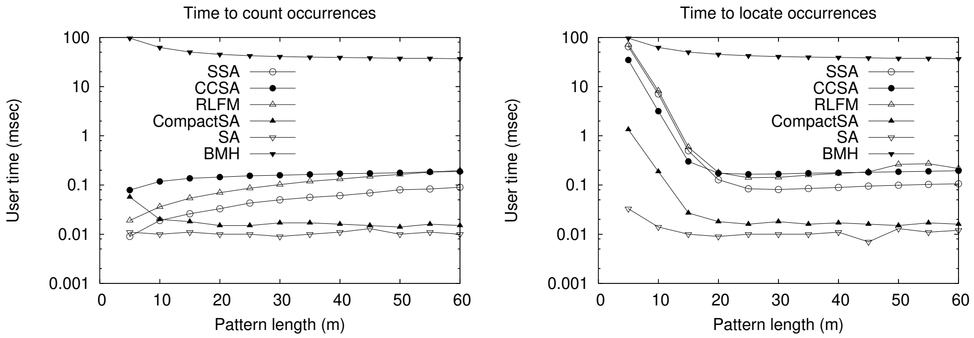


Fig. 7: Our self-index implementations against suffix array, compact suffix array and sequential search. We show counting times on the left and locating times on the right (with self indexes taking around 1.6 times the text size).

for reasonable performance. For locating, the slowdown is closer to 1000X and the times get closer to those of a sequential scan, albeit they are still much better in space and time. Some succinct indexes support output-sensitive locating queries [18, 10, 41, 20]. The technique, as described in [20], can be plugged into our indexes as well (or into any other index supporting efficient backward search mechanism). As a further experimental study, it would be interesting to see how this technique works in practice.

7. Conclusions

In this paper we have explored the interconnection between the empirical kth order entropy of a text and the regularities that appear in its suffix array, as well as in the Burrows-Wheeler transform of the text. We have shown how this connection lies at the heart of several existing compressed indexes for full-text retrieval.

Inspired by the relationship between the kth order empirical entropy of a text and the runs of equal letters in its Burrows-Wheeler transform, we have designed a new index, the RLFM index, that answers the mentioned counting queries in time linear in the pattern length for any alphabet whose size is polylogarithmic on the text length. The RLFM index was the first in achieving this.

We have also considered practical issues of implementing the RLFM index, obtaining an efficient implementation. We have in passing presented another index, the SSA, which is a practical implementation of an existing proposal [46]. The SSA is larger and faster than the RLFM index. We have compared both indexes against the existing implementations, showing that ours are competitive and obtain practical space-time tradeoffs that are not reached by any other implementation.

Acknowledgement

We wish to thank Rodrigo González for letting us use his codes for rank/select queries.

References

[1] A , M., O , E., K , S. 2002. Optimal exact string matching based on suffix arrays. In *Proc. 9th International Symposium on String Processing and Information Retrieval (SPIRE'02)*, LNCS 2476, 31–43.

[2] A , A. 1985. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series. Springer-Verlag, 85–96.

[3] B , T., C , J., W , I. 1990. *Text compression*. Prentice Hall.

[4] B , A., B , J., H , D., M C , R., E , A. 1987. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM 34*, 3, 578–595.

[5] B , M. W , D. 1994. A block sorting lossless data compression algorithm. Tech. Report 124, Digital Equipment Corporation.

[6] C , D. 1996. *Compact Pat Trees*. PhD thesis, University of Waterloo.

[7] C , D. M , I. 1996. Efficient suffix trees on secondary storage. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, 383–391.

[8] F , P. M , G. 2000. Opportunistic data structures with applications. In *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS'00)*, 390–398.

[9] F , P. M , G. 2001. An experimental study of an opportunistic index. In *Proc. 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'01)*, 269–278.

[10] F , P. M , G. 2002. On Compressing and Indexing Data. Tech. Report TR-02-01, Dipartamento di Informatica, Univ. of Pisa.

[11] F , P., M , G., M¨ , V., N , G. 2004. An alphabet-friendly FM-index. In *Proc. 11th International Symposium on String Processing and Information Retrieval (SPIRE'04)*, LNCS 3246, 150–160.

[12] F , P., M , G., M¨ , V., N , G. 2004. Succinct Representation of Sequences. Tech. Report TR/DCC-2004-5, Dept. of Computer Science, University of Chile.

[13] G ´ , R , G , S , M¨ , V , N , G . 2005. Practical Implementation of Rank and Select Queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05)*. CTI Press and Ellinika Grammata, Greece, 27–38.

[14] G , S ., M¨ , V., N , G. 2004. First Huffman, then Burrows-Wheeler: an alphabet-independent FM-index. In *Proc. 11th International Symposium on String Processing and Information Retrieval (SPIRE'04)*, LNCS 3246, 210–211.

[15] G , S ., M¨ , V., N , G., S , A. 2005. A Simple Alphabet-Independent FM-Index. In *Proc. 10th Prague Stringology Conference (PSC'05)*.

[16] G , R., G , A., V , J. 2003. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'03)*, 841–850.
[17] G , R., G , A., V , J. 2004. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*, 636–645.
[18] G , R. V , J.S. 2000. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd ACM Symposium on Theory of Computing (STOC'00)*, 397–406.
[19] H , D. 1995. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, 1–19.
[20] H , M., M , I., R , S. S . 2005. A categorization theorem on suffix arrays with applications to space efficient text indexes. In *Proc. 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'05)*, 23–32.
[21] H , R. N. 1980. Practical fast searching in strings. *Softw. Pract. Exp. 10*, 6, 501–506.
[22] J , G. 1989. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS'89)*, 549–554.
[23] K¨ ¨ , J. 1995. Suffix cactus: a cross between suffix tree and suffix array. In *Proc. 6th Annual Symposium on Combinatorial Pattern Matching (CPM'95)*, LNCS 937, 191–204.
[24] K¨ ¨ , J. S , E. 1998. Lempel-Ziv index for q-grams. *Algorithmica 21*, 1, 137–154.
[25] K¨ ¨ , J. U , E. 1996. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP'96)*. Carleton University Press, 141–155.
[26] K¨ ¨ , J. U , E. 1996. Sparse suffix trees. In *Proc. 2nd Annual International Conference on Computing and Combinatorics (COCOON'96)*, LNCS 1090, 219–230.
[27] K , S. 1998. Reducing the space requirements of suffix trees. Report 98-03, Technische Kakultät, Universität Bielefeld.
[28] M¨ , V. 2003. Compact suffix array — a space-efficient full-text index. *Fundamenta Informaticae 56*, 1–2, 191–210.
[29] M¨ , V. N , G. 2004. Compressed compact suffix arrays. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM'04)*, LNCS 3109, 420–433.
[30] M¨ , V. N , G. 2004. New Search Algorithms and Time/Space Tradeoffs for Succinct Suffix Arrays. Tech. Report C-2004-20, University of Helsinki, Finland.
[31] M¨ , V. N , G. 2004. Run-Length FM-index. In *Proc. DIMACS Workshop: "The Burrows-Wheeler Transform: Ten Years Later"*, 17–19.
[32] M¨ , V. N , G. 2005. Succinct Suffix Arrays based on Run-Length Encoding. In *Proc. 16th Annual Symposium on Combinatorial Pattern Matching (CPM'05)*, LNCS 3537, 45–56.
[33] M¨ , V., N , G., S , K. 2004. Advantages of Backward Searching — Efficient Secondary Memory and Distributed Implementation of Compressed Suffix Arrays. In *Proc. 15th Annual International Symposium on Algorithms and Computation (ISAAC'04)*, LNCS 3341, 681–692.
[34] M , U. M , G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing 22*, 5, 935–948.
[35] M , G. 2001. An analysis of the Burrows-Wheeler transform. *Journal of the ACM 48*, 3, 407–430.
[36] M C , E. M. 1976. A space-economical suffix tree construction algorithm. *Journal of the ACM 23*, 2, 262–272.
[37] M , I. 1996. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'96)*, LNCS 1180, 37–42.
[38] M , I. R , V. 1997. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. 38th IEEE Symposium on Foundations of Computer*

Science (FOCS'97), 118–126.
[39] M , I., R , V., R , S. 2001. Space efficient suffix trees. *Journal of Algorithms 39*, 2, 205–222.
[40] N , G. 2002. Indexing text using the Ziv-Lempel trie. Tech. Report TR/DCC-2002-2, Dept. of Computer Science, Univ. of Chile.
[41] N , G. 2004. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms 2*, 1, 87–114.
[42] P , R. 1999. Low redundancy in dictionaries with $O(1)$ worst case lookup time. In *Proc. 26th International Colloquium on Automata, Languages and Programming (ICALP'99)*, 595–604.
[43] R , R., R , V., R , S. S . 2002. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, 233–242.
[44] R , S. S . 2002. Time-space trade-offs for compressed suffix arrays. *Information Processing Letters 82*, 6, 307–311.
[45] S , K. 2000. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. 11th International Symposium on Algorithms and Computation (ISAAC'00)*, LNCS 1969, 410–421.
[46] S , K. 2002. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, 225–232.
[47] S , E. T , J. 1996. Filtration with q-Samples in Approximate String Matching. In *Proc. 7th Annual Symposium on Combinatorial Pattern Matching (CPM'96)*, LNCS 1075, 50–63.
[48] U , E. 1995. On-line construction of suffix trees. *Algorithmica 14*, 3, 249–260.
[49] W , P. 1973. Linear pattern matching algorithm. In *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*, 1–11.