Succinct Suffix Arrays based on Run-Length Encoding

Veli Mäkinen[1] * and Gonzalo Navarro[2] **

[1] AG Genominformatik, Technische Fakultät
Universität Bielefeld, Germany.
veli@cebitec.uni-bielefeld.de
[2] Center for Web Research
Dept. of Computer Science, University of Chile.
gnavarro@dcc.uchile.cl

Abstract. A succinct full-text self-index is a data structure built on a text $T = t_1 t_2 \ldots t_n$, which takes little space (ideally close to that of the compressed text), permits efficient search for the occurrences of a pattern $P = p_1 p_2 \ldots p_m$ 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. They usually take $O(nH_0)$ or $O(nH_k)$ bits, being H_k the kth order empirical entropy of T. The time to count how many times does P occur in T ranges from $O(m)$ to $O(m \log n)$.
We present a new self-index, called run-length FM-index (RLFM index), that counts the occurrences of P in T in $O(m)$ time when the alphabet size is $\sigma = O(\text{polylog}(n))$. The index requires $nH_k \log_2 \sigma + O(n)$ bits of space for small k. We then show how to implement the RLFM index in practice, and obtain in passing another implementation with different space-time tradeoffs. We empirically compare ours against the best existing implementations of other indexes and show that ours are fastest among indexes taking less space than the text.

1 Introduction

The classical problem in string matching is to determine the *occ* occurrences of a short pattern $P = p_1 p_2 \ldots p_m$ in a large text $T = t_1 t_2 \ldots t_n$. 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*). When the same text is queried several times with different patterns, the text can be preprocessed to build an *index* structure that speeds up searches.

* Funded by the Deutsche Forschungsgemeinschaft (BO 1910/1-3) within the Computer Science Action Program.
** Funded by Millennium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile.

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 $t_i t_{i+1} \ldots t_n$). These kind of indexes are called *full-text indexes*. The *suffix tree* [24] is the best-known full-text index, requiring $O(m)$ time for counting and $O(occ)$ for locating.

The suffix tree, unfortunately, takes $O(n \log n)$ bits of space, 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* [16]. Yet, the space complexity is still $O(n \log n)$ bits.

Many efforts to reduce these space requirements have been pursued. This has evolved into the concept of a *self-index*, a succinct index that contains enough information to reproduce any text substring. Hence a self-index *replaces* the text. Several self-indexes that require space proportional to the *compressed* text have been proposed recently [10, 3, 22, 19, 8, 5].

Some structures [8, 5] need only $nH_k + o(n)$ bits of space, which is currently the lowest asymptotic space requirement that has been achieved. Here H_k stands for the *kth order entropy* of T [17]. Using a recent sequence representation technique [6], the structure in [5] supports counting queries in $O(m)$ time on alphabets of size $O(\text{polylog}(n))$. Other self-indexes require either more time for counting [22, 8, 19, 5], or $O(nH_0)$ bits of space [23].

In this paper we describe how run-length encoding of Burrows-Wheeler transformed text [1], together with the backward search idea [3], can be used to obtain a self-index, called RLFM index for "run-length FM-index", requiring $nH_k \log \sigma + O(n)$ bits of space, which answers counting queries in $O(m)$ time on alphabets of size $O(\text{polylog}(n))$. Just as for other indexes [8, 5], the space formula is valid for any $k \leq \alpha \log_\sigma n$, for any constant $0 < \alpha < 1$. The index is interesting in practice as well. We give several considerations to implement the RLFM index, which are of interest by themselves and also yield an implementation for a simpler index, which we call SSA for "succinct suffix array". We show experimentally on an English text collection that the RLFM index and the SSA take less space than the text, and among such indexes, they are the fastest in counting queries.

The RLFM index motivated the work in [5, 6]. The latter supersedes our original idea [15] in theory, yet in practice the RLFM index is still appealing.

2 Basic Concepts

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 "\$" 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. For clarity, we assume that P and T are drawn over alphabet $\Sigma = \{\$, 1, \ldots, \sigma\}$.

Empirical *k*th Order Entropy. Let n_c denote the number of occurrences in T of character $c \in \Sigma$. The zero-order empirical entropy of string T is $H_0(T) =$

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

$-\sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n_c}{n}$, where $0 \log 0 = 0$. If we use a fixed codeword for each character 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 characters that precede 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 [17] as

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

where W_T is the concatenation of all characters t_j such that Wt_j is a substring of T. String W is the *k-context* of each such t_j. We use H_0 and H_k as shorthands for $H_0(T)$ and $H_k(T)$.

The Burrows-Wheeler Transform (BWT). The BWT [1] 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 $t_i t_{i+1} \ldots t_n t_1 t_2 \ldots t_{i-1}$ 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 main step to reverse the BWT is to compute the *LF mapping*, so that $LF(i)$ is the position of character $L[i]$ in F. This is computed as $LF(i) = C[L[i]] + Occ(L, L[i], i)$, where $C[c]$ is the number of occurrences of characters $\{\$, 1, \ldots, c-1\}$ in T, and $Occ(L, c, i)$ is the number of occurrences of character c in the prefix $L[1, i]$. Then T can be obtained backwards from T^{bwt} by successive applications of LF.

We note that matrix $\mathcal{M}$ is essentially the *suffix array* $\mathcal{A}[1, n]$ of T, 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 $\mathcal{M}[i] = t_j t_{j+1} \ldots t_{n-1} \$ t_1 \ldots t_{j-1}$.

The FM-Index. The FM-index [3, 4] 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 before. Figure 1 shows the counting algorithm. It maintains the invariant that, at the ith phase, variables *sp* and *ep* point, respectively, to the first and last row of $\mathcal{M}$ prefixed by $P[i, m]$.

Note that while array C can be explicitly stored in little space, implementing $Occ(T^{bwt}, c, i)$ is problematic. The first solution [3] 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 [4] this exponential dependence on σ was avoided, but the constant time guarantee for answering $Occ(T^{bwt}, c, i)$ was no longer valid.

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 occurrences of P in T.

The method to locate pattern occurrences or to show contexts around them also employs the $Occ()$ function, but we omit the details here.

Succinct Data Structures for Binary Sequences. 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 results [12, 18, 2] 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 [20, 21] answer those queries in constant time using only $nH_0(B) + o(n)$ bits of space.

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* [8]. This tree retrieves any s_i in $O(\log \sigma)$ time. Within the same time bounds, it also answers generalized *rank* and *select* queries.

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 B_v at the node v 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 single alphabet characters and require no space.

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 $rank_c(S,i)$. The character s_i is obtained similarly, this time going left or right depending on whether $B_v[i] = 0$ or 1 at each level, and finding out which leaf we arrived at. Query $select_c(S,i)$ is answered by traversing the tree bottom-up.

If every bitmap in the wavelet tree is represented using a data structure that takes space proportional to its zero-order entropy, then it can be shown that the whole wavelet tree requires $nH_0(S) + o(n \log \sigma)$ bits of space [8]. 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 [6].

3 RLFM: A Run-Length-based FM-Index

We studied in [14] the relationship between the runs in the Burrows-Wheeler transformed text and the kth order entropy. We summarize the main result in the following.

Theorem 1. *The length n_{bw} of the run-length encoded Burrows-Wheeler transformed text $T^{bwt}[1,n]$ is at most $n \min(H_k(T), 1) + \sigma^k$, 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 $0 < \alpha < 1$.*[2]

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.

Definition 1. *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 the string $S = c_1 c_2 \ldots c_{n_{bw}}$ of length n_{bw}, and of the 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 [2, 18]. 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 Figure 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.

[2] The original analysis [14] has constant 2 multiplying $nH_k(T)$. We later noticed that the analysis can be tightened to give constant 1. This comes from showing that $-(x/(x+y)) \log(x/(x+y)) - (y/(x+y)) \log(y/(x+y)) \geq 2x/(x+y)$ for any $x, y \geq 0$, while in Eq. (4) of [14] the expression at the right of the inequality was $x/(x+y)$.

Definition 2. *Let* $S = c_1c_2\ldots c_{n_{bw}}$ *of length* n_{bw}*, and* $B = 10^{\ell_1-1}10^{\ell_2-1}\ldots 10^{\ell_{n_{bw}}-1}$*. Let* $d_1d_2\ldots d_{n_{bw}}$ *be the permutation of* $[1, n_{bw}]$ *such that, for all* $1 \leq i < n_{bw}$*, 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* $C_S[c] = |\{i,\ c_i < c,\ 1 \leq i \leq n_{bw}\}|$*.*

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

Lemma 1. *Let* S *and* B' *be defined for a string* T^{bwt}*. Then, for any* $c \in \Sigma$ *it holds*

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

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

Lemma 2. *Let* S*,* B*, and* B' *be defined for a string* T^{bwt}*. Then, for any* $c \in \Sigma$ *and* $1 \leq i \leq n$*, such that* i *is the final position of a run in* B*, it holds*

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

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

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

Lemma 3. *Let* S*,* B*, and* B' *be defined for a string* T^{bwt}*. Then, for any* $c \in \Sigma$ *and* $1 \leq i \leq n$*, such that* $T^{bwt}[i] \neq c$*, it holds*

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

Proof. Let i' be the last position of the run that precedes that of i. Since $T^{bwt}[i] \neq c$ 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 2 to i'.

Lemma 4. *Let S, B, and B' be defined for a string T^{bwt}. Then, for any $c \in \Sigma$ and $1 \leq i \leq n$, such that $T^{bwt}[i] = c$, it holds*

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

Proof. Let i' be the last position of the run that precedes that of i. Then, by Lemma 2, $C[c] + Occ(T^{bwt}, c, i') = select(B', C_S[c] + 1 + Occ(S, c, rank(B, i')))$ -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 $i' < i'' \leq i$, we have $Occ(T^{bwt}, c, i) = Occ(T^{bwt}, c, i') + (i - i')$. Thus, the outcome of Lemma 2 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 3 and 4 is the computation of Occ over string S. This can be done in constant time using a new sequence representation technique [6], when the alphabet size is $O(\text{polylog}(n))$. This needs a structure of size $|S|H_0(S) + o(|S|)$. Using Theorem 1, this is no more than $nH_k H_0(S) + o(n)$ for $k \leq \alpha \log_\sigma n$, for constant $0 < \alpha < 1$.[3]

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 $n(H_k(H_0(S) + o(1)) + 2) + o(n)$ bits. As H_k actually stands for $\min(1, H_k)$,and $H_0(S) \leq \log \sigma$, we can simplify the space complexity to $nH_k \log \sigma + O(n)$ bits.

Theorem 2. *The RLFM index, of size $n \min(H_k, 1) \log \sigma + 2n + o(n) = nH_k \log \sigma + O(n)$ bits for any $k \leq \alpha \log_\sigma n$, for any constant $0 < \alpha < 1$, can be built on a text $T[1, n]$ with alphabet size $\sigma = O(polylog(n))$, so that the occurrences of any pattern $P[1, m]$ in T can be counted in time $O(m)$.*

The RLFM index can easily be extended to support reporting and context queries. We defer the details to the journal version of this paper.

[3] This can also be solved in $nH_k H_0(S) + O(n)$ space, with the same restrictions, using older techniques [23]. The final complexity changes only in small details.

4 Practical Considerations

The most problematic aspect to implement our proposal is the use of a technique to represent sequences in a space proportional to its zero-order entropy [6]. This technique has not yet been implemented, and this will require considerable additional effort. The same is true with the alternative technique that could be used [23] to obtain similar time and space complexity.

Yet, previous structures supporting *rank* on binary sequences in $n + o(n)$ bits [18, 2] are very simple to implement. So an alternative is to use a wavelet tree built on the S string of the RLFM index (that is, the run heads). 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 $|S| \log \sigma(1 + o(1)) = nH_k \log \sigma(1 + o(1))$ bits of space to represent S. This is essentially the same space we achieved using the theoretical approach.

With the wavelet tree, the $O(1)$ time to compute $Occ(S, c, i) = rank_c(S, i)$, becomes $O(\log \sigma)$. Therefore, a RLFM index implementation based on wavelet trees counts in $O(m \log \sigma)$ time.

The same idea can also be applied *without* run-length encoding. Let us call SSA (for "succinct suffix array") this implementation. We notice that the SSA can be considered as a practical implementation of a previous proposal [23]. It has also been explicitly mentioned as a simplified version in previous work [5], with the same $O(m \log \sigma)$ time complexity.

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. Instead of a balanced binary tree, we use the *Huffman tree* of T to define the shape of the wavelet tree. Then, every character $c \in \Sigma$, of frequency n_c, will have its corresponding leaf at depth h_c, so that $\sum_{c \in \Sigma} h_c n_c \leq n(H_0 + 1)$ is the number of bits of the Huffman compression of T.

Consider the size of this 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 thus 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.

Finally, we note that the Huffman-shaped wavelet tree can be used 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 $O(\log \sigma)$.

5 Experiments

We compare our SSA and RLFM implementations against others. We used an 87 MB text file (ZIFF collection from TREC-3) and randomly chose 10,000 patterns of each length from it. We compared counting times against the following indexes/implementations: FM [4] (0.36), Navarro's implementation of FM-index, FM-Nav [19] (1.07), CSA [22] (0.39-1.16), LZ [19] (1.49), CompactSA [13] (2.73), CCSA [14] (1.65), our $n\lceil\log n\rceil$-bits implementation of the suffix array, SA [16] (4.37), and our implementation of a sequential search algorithm, BMH [11] (1.0). The last three, not being self-indexes, are included to test the value of compressed indexing. The values in parantheses tell the space usage of each index as a fraction of the text size. Our indexes take SSA (0.87) and RLFM (0.67). We applied the ideas of Section 4, and also optimized rank/select implementations [7] for all the indexes. The codes for FM-Nav, CSA, and LZ indexes are available at `http://www.dcc.uchile.cl/~gnavarro/software` and the codes for the other indexes at `http://www.cs.helsinki.fi/u/vmakinen/software`.

Only the CSA has a tradeoff in counting time and space usage. We denote by CSAX the tradeoffs (X is the sampling rate for absolute Ψ-values). The sizes of CSA10, CSA16, CSA32, and CSA256, are 1.16, 0.86, 0.61, and 0.39, respectively. Figure 2 shows the times to count pattern occurrences of length $m = 5$ to $m = 60$. We omit CSA10 and CSA16, whose performace is very similar to CSA32. It can be seen that FM-Nav is the fastest self-index, 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 space around 0.6. Actually RLFM is faster, and to reach its performance we need CSA10, which takes space 1.16. For long patterns CCSA becomes competitive in this group, yet it needs as much space as 1.65 times the text size. Compared to non-self-indexes (that take much more space), we see that 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 for reasonable performance. Figure 3 illustrates the space/time tradeoff, for $m = 30$.

Finally, we notice that SSA gives a good estimate for the efficiency obtainable with the more succinct index in [5]. That index has potential to be significantly more space-efficient, but needs a more careful wavelet tree implementation (in terms of constant terms in space usage) than what we have currently in SSA. This is needed in order to gain advantage of the compression boosting mechanism. Also, implementing the sequence representation developed in [6] will probably improve in practice the performance of the index in [5], as well as that of the SSA and RLFM index.

6 Conclusions

Inspired by the relationship between the kth order empirical entropy of a text and the runs of equal characters in its Burrows-Wheeler transform, we have

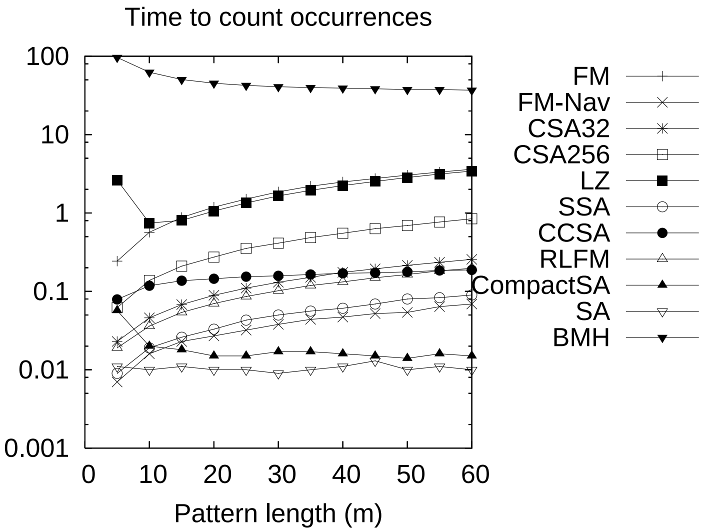

Fig. 2. Query times (msec) for counting the number of occurrences.

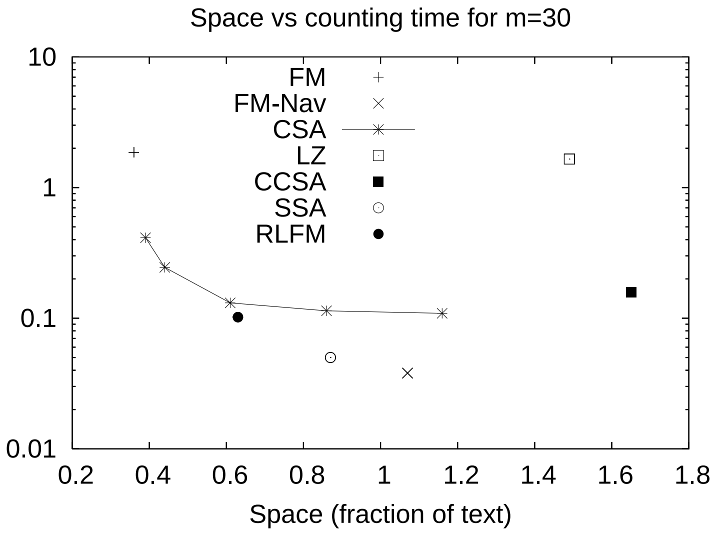

Fig. 3. Space/time tradeoff for counting the number of occurrences.

designed a new index, the RLFM index, that answers 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, that 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.

References

1. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
2. D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.
3. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS'00*, pp. 390–398, 2000.
4. P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. SODA'01*, pp. 269–278, 2001.
5. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proc. SPIRE'04*, pp. 150–160, 2004.
6. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Succinct representation of sequences. Technical Report TR/DCC-2004-5, Dept. of CS, Univ. Chile, Aug. 2004.
7. R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. To appear in *Proc. WEA'05* (poster).
8. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA'03*, pp. 841–850, 2003.
9. R. Grossi, A. Gupta, and J. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proc. SODA'04*, pp. 636–645, 2004.
10. R. Grossi and J.S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. STOC'00*, pp. 397–406, 2000.
11. R. N. Horspool. Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980.
12. G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS'89*, pp. 549–554, 1989.
13. V. Mäkinen. Compact suffix array — a space-efficient full-text index. *Fundamenta Informaticae*, 56(1–2):191–210, 2003.
14. V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In *Proc. CPM'04*, pp. 420–433, 2004.
15. V. Mäkinen and G. Navarro. Run-length FM-index. In *Proc. DIMACS Workshop: "The Burrows-Wheeler Transform: Ten Years Later"*, pp. 17–19, Aug. 2004. Also in *New Search Algorithms and Time/Space Tradeoffs for Succinct Suffix Arrays*, Tech. Report. C-2004-20, Univ. Helsinki, Apr. 2004.
16. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

17. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
18. I. Munro. Tables. In *Proc. FSTTCS'96*, pp. 37–42, 1996.
19. G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms*, 2(1):87–114, 2004.
20. R. Pagh. Low redundancy in dictionaries with $O(1)$ worst case lookup time. In *Proc. ICALP'99*, pp. 595–604, 1999.
21. R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA'02*, pp. 233–242, 2002.
22. K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. ISAAC'00*, pp. 410–421, 2000.
23. K. Sadakane. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In *Proc. SODA'02*, pp. 225–232, 2002.
24. P. Weiner. Linear pattern matching algorithm. In *Proc. IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.