# A Simple Alphabet-Independent FM-Index

Szymon Grabowski[1], Veli Mäkinen[2] *,
Gonzalo Navarro[3] †, Alejandro Salinger[3]

[1] Computer Engineering Dept., Tech. Univ. of Łódź, Poland.
e-mail: sgrabow@zly.kis.p.lodz.pl

[2] Technische Fakultät, Bielefeld Universität, Germany
e-mail: veli@cebitec.uni-bielefeld.de

[3] Dept. of Computer Science, Univ. of Chile, Chile.
e-mail: {gnavarro,asalinge}@dcc.uchile.cl

**Abstract.** We design a succinct full-text index based on the idea of Huffman-compressing the text and then applying the Burrows-Wheeler transform over it. The resulting structure can be searched as an FM-index, with the benefit of removing the sharp dependence on the alphabet size, $\sigma$, present in that structure. On a text of length $n$ with zero-order entropy $H_0$, our index needs $O(n(H_0+1))$ bits of space, without any dependence on $\sigma$. The average search time for a pattern of length $m$ is $O(m(H_0+1))$, under reasonable assumptions. Each position of a text occurrence can be reported in worst case time $O((H_0+1)\log n)$, while any text substring of length $L$ can be retrieved in $O((H_0+1)L)$ average time in addition to the previous worst case time. Our index provides a relevant space/time tradeoff between existing succinct data structures, with the additional interest of being easy to implement. Our experimental results show that, although not among the most succinct, our index is faster than the others in many aspects, even letting them use significatively more space.

## 1 Introduction

A *full-text index* is a data structure that enables to determine the *occ* occurrences of a short pattern $P = p_1p_2\ldots p_m$ in a large text $T = t_1t_2\ldots t_n$ without a need of scanning over the whole text $T$. Text and pattern are sequences of characters over an alphabet $\Sigma$ of size $\sigma$. In practice one wants to know not only the value *occ*, i.e., how many times the pattern appears in the text (*counting query*) but also the text positions of those *occ* occurrences (*reporting query*), and usually also a text context around them (*display query*).

A classic example of a full-text index is the *suffix tree* [20] reaching $O(m + occ)$ time complexity for counting and reporting queries. Unfortunately, it takes $O(n \log n)$ bits,[1] and also the constant factor is large. A smaller space complexity factor is achieved by the *suffix array* [13], reaching $O(m \log n + occ)$ or $O(m + \log n + occ)$ in

*Funded by the Deutsche Forschungsgemeinschaft (BO 1910/1-3) within the Computer Science Action Program.

†Funded in part by Fondecyt Grant 1-050493 (Chile).

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

time (depending on a variant), but still the space usage may rule out this structure from some applications, e.g. in computational biology.

The large space requirement of traditional full-text indexes has raised a natural interest in *succinct* full-text indexes that achieve good tradeoffs between search time and space complexity [12, 3, 10, 19, 8, 15, 18, 16, 9]. A truly exciting perspective has been originated in the work of Ferragina and Manzini [3]; they showed a full-text index may discard the original text, as it contains enough information to recover the text. We denote a structure with such a property with the term *self-index*.

The FM-index of Ferragina and Manzini [3] was the first self-index with space complexity expressed in terms of $k$th order (empirical) entropy and pattern search time linear only in the pattern length. Its space complexity, however, contains an exponential dependence on the alphabet size; a weakness eliminated in a practical implementation [4] for the price of not achieving the optimal search time anymore. Therefore, it has been interesting both from the point of theory and practice to construct an index with nice bounds both in space and time complexities, preferably with no (or mild) dependence on the alphabet size.

In this paper we concentrate on improving the FM-index, in particular its large alphabet dependence. This dependence shows up not only in the space usage, but also in the time to show an occurrence position and display text substrings. The FM-index needs up to $5H_kn + O\left((\sigma \log \sigma + \log\log n)\frac{n}{\log n} + n^\gamma \sigma^{\sigma+1}\right)$ bits of space, where $0 < \gamma < 1$. The time to search for a pattern and obtain the number of its occurrences in the text is the optimal $O(m)$. The text position of each occurrence can be found in $O\left(\sigma \ \log^{1+\varepsilon} n\right)$ time, for some $\varepsilon > 0$ that appears in the sublinear terms of the space complexity. Finally, the time to display a text substring of length $L$ is $O\left(\sigma \ (L + \log^{1+\varepsilon} n)\right)$. The last operation is important not only to show a text context around each occurrence, but also because a self-index replaces the text and hence it must provide the functionality of retrieving any desired text substring.

The compressed suffix array (CSA) of Sadakane [19] can be seen as a tradeoff with larger search time but much milder dependence on the alphabet size. The CSA needs $(H_0/\varepsilon + O(\log\log\sigma))n$ bits of space. Its search time (finding the number of occurrences of a pattern) is $O(m \log n)$. Each occurrence can be reported in $O\left(\log^\varepsilon n\right)$ time, and a text substring of length $L$ can be displayed in $O\left(L + \log^\varepsilon n\right)$ time.

In this paper we present a simple structure based on the FM-index concept. We Huffman-compress the text and then apply the Burrows-Wheeler transform over it, as in the FM-index. The obtained structure can be regarded as an FM-index built over a binary sequence. As a result, we remove any dependence on the alphabet size. We show that our index can operate using $n(2H_0 + 3 + \varepsilon)(1 + o(1))$ bits, for any $\varepsilon > 0$. No alphabet dependence is hidden in the sublinear terms.

At search time, our index finds the number of occurrences of the pattern in $O(m(H_0 + 1))$ average time. The text position of each occurrence can be reported in worst case time $O\left(\frac{1}{\varepsilon}(H_0 + 1) \ \log n\right)$. Any text substring of length $L$ can be displayed in $O\left((H_0 + 1) \ L\right)$ average time, in addition to the mentioned worst case time to find a text position. In the worst case all the $H_0$ become $\log n$.

This index was first presented in a poster [5], where we only gave its rough idea. Now we present it in full detail and explore its empirical effectiveness in counting, reporting and displaying, for a broad scope of real-world data (English text, DNA and proteins). We also include a $k$-ary Huffman variant. We show that our index,

```
Algorithm FM_Search($P$,$T^{bwt}$)
$i = m$;
$sp = 1$; $ep = n$;
while (($sp \leq ep$) and ($i \geq 1$)) do
    $c = P[i]$;
    $sp = C[c] + Occ(T^{bwt}, c, sp-1)+1$;
    $ep = C[c] + Occ(T^{bwt}, c, ep)$;
    $i = i - 1$;
if ($ep < sp$) then return “not found” else return “found ($ep - sp + 1$) occs”.
```

Figure 1: Algorithm for counting the number of occurrences of $P[1 \dots m]$ in $T[1 \dots n]$.

albeit not among the most succinct indexes, is faster than the others in many cases, even if we give the other indexes much more space to work.

## 2 The FM-index Structure

The FM-index [3] is based on the *Burrows-Wheeler transform (BWT)* [1], which produces a permutation of the original text, denoted by $T^{bwt} = bwt(T)$. String $T^{bwt}$ is the result of the following *forward* transformation: (1) Append to the end of $T$ a special end marker \$, which is lexicographically smaller than any other character; (2) form a *conceptual* matrix $\mathcal{M}$ whose rows are the cyclic shifts of the string $T\$$, sorted in lexicographic order; (3) construct the transformed text $L$ by taking the last column of $\mathcal{M}$. The first column is denoted by $F$.

The *suffix array (SA)* $\mathcal{A}$ of text $T\$$ is essentially the matrix $\mathcal{M}$: $\mathcal{A}[i] = j$ iff the $i$th row of $\mathcal{M}$ contains string $t_j t_{j+1} \cdots t_n \$ t_1 \cdots t_{j-1}$. The occurrences of any pattern $P = p_1 p_2 \cdots p_m$ form an interval $[sp, ep]$ in $\mathcal{A}$, such that suffixes $t_{\mathcal{A}[i]} t_{\mathcal{A}[i]+1} \cdots t_n$, $sp \leq i \leq ep$, contain the pattern as a prefix. This interval can be searched for by using two binary searches in time $O(m \log n)$.

The suffix array of text $T$ is represented implicitly by $T^{bwt}$. The novel idea of the FM-index is to store $T^{bwt}$ in compressed form, and to simulate the search in the suffix array. To describe the search algorithm, we need to introduce the *backward* BWT that produces $T$ given $T^{bwt}$: ($i$) Compute the array $C[1 \dots \sigma]$ storing in $C[c]$ the number of occurrences of characters $\{\$, 1, \dots, c-1\}$ in the text $T$. Notice that $C[c] + 1$ is the position of the first occurrence of $c$ in $F$ (if any). ($ii$) Define the *LF-mapping* $LF[1 \dots n+1]$ as $LF[i] = C[L[i]] + Occ(L, L[i], i)$, where $Occ(X, c, i)$ equals the number of occurrences of character $c$ in the prefix $X[1, i]$. ($iii$) Reconstruct $T$ backwards as follows: set $s = 1$ and $T[n] = L[1]$ (because $\mathcal{M}[1] = \$T$); then, for each $n-1, \dots, 1$ do $s \leftarrow LF[s]$ and $T[i] \leftarrow L[s]$.

We are now ready to describe the search algorithm given in [3] (Fig. 1). It finds the interval of $\mathcal{A}$ containing the occurrences of the pattern $P$. It uses the array $C$ and function $Occ(X, c, i)$ defined above. Using the properties of the backward BWT, it is easy to see that the algorithm maintains the following invariant [3]: *At the ith phase, with i from m to 1, the variable sp points to the first row of $\mathcal{M}$ prefixed by $P[i, m]$ and the variable ep points to the last row of $\mathcal{M}$ prefixed by $P[i, m]$.* The correctness of the algorithm follows from this observation.

Ferragina and Manzini [3] describe an implementation of $Occ(T^{bwt}, c, i)$ that uses a compressed form of $T^{bwt}$. They show how to compute $Occ(T^{bwt}, c, i)$ for any $c$ and $i$ in constant time. However, to achieve this they need exponential space (in the size of the alphabet). In a practical implementation [4] this was avoided, but the constant time guarantee for answering $Occ(T^{bwt}, c, i)$ was no longer valid.

The FM-index can also show the text positions where $P$ occurs, and display any text substring. The details are deferred to Section 4.

## 3 First Huffman, then Burrows-Wheeler

We focus now on our index representation. Imagine that we compress our text $T\$$ using Huffman. The resulting bit stream will be of length $n' < (H_0 + 1)n$, since (binary) Huffman poses a maximum representation overhead of 1 bit per symbol[2]. We call $T'$ this sequence, and define a second bit array $Th$, of the same length of $T'$, such that $Th[i] = 1$ iff $i$ is the starting position of a Huffman codeword in $T'$. $Th$ is also of length $n'$. (We will not represent $T'$ nor $Th$ in our index.)

The idea is to search the binary text $T'$ instead of the original text $T$. Let us apply the Burrows-Wheeler transform over text $T'$, so as to obtain $B = (T')^{bwt}$. In order to have a binary alphabet, $T'$ will not have its own special terminator character "\$" (yet that of $T$ is encoded in binary at the end of $T'$).

More precisely, let $\mathcal{A}'[1 \dots n']$ be the suffix array for text $T'$, that is, a permutation of the set $1 \dots n'$ such that $T'[A'[i] \dots n'] < T'[A'[i+1] \dots n']$ in lexicographic order, for all $1 \leq i < n'$. In a lexicographic comparison, if a string $x$ is a prefix of $y$, assume $x < y$. Suffix array $\mathcal{A}'$ will not be explicitly represented. Rather, we represent bit array $B[1 \dots n']$, such that $B[i] = T'[\mathcal{A}'[i] - 1]$ (except that $B[i] = T[n']$ if $\mathcal{A}'[i] = 1$). We also represent another bit array $Bh[1 \dots n']$, such that $Bh[i] = Th[\mathcal{A}'[i]]$. This tells whether position $i$ in $\mathcal{A}'$ points to the beginning of a codeword.

Our goal is to search $B$ exactly like the FM-index. For this sake we need array $C$ and function $Occ$. Since the alphabet is binary, however, $Occ$ can be easily computed: $Occ(B, 1, i) = rank(B, i)$ and $Occ(B, 0, i) = i - rank(B, i)$, where $rank(B, i)$ is the number of 1's in $B[1 \dots i]$, $rank(B, 0) = 0$. This function can be computed in constant time using only $o(n)$ extra bits [11, 14, 2]. The solution, as well as its more practical implementation variants, are described in [7].

Also, array $C$ is so simple for the binary text that we can do without it: $C[0] = 0$ and $C[1] = n' - rank(B, n')$, that is, the number of zeros in $B$ (of course value $n' - rank(B, n')$ is precomputed). Therefore, $C[c] + Occ(T^{bwt}, c, i)$ is replaced in our index by $i - rank(B, i)$ if $c = 0$ and $n' - rank(B, n') + rank(B, i)$ if $c = 1$.

There is a small twist, however, due to the fact that we are not putting a terminator to our binary sequence $T'$ and hence no terminator appears in $B$. Let us call "#" the terminator of the binary sequence so that it is not confused with the terminator "\$" of $T\$$. In the position $p_{\#}$ such that $\mathcal{A}'[p_{\#}] = 1$, we should have $B[p_{\#}] = \#$. Instead, we are setting $B[p_{\#}]$ to the last bit of $T'$. This is the last bit of the Huffman codeword assigned to the terminator "\$" of $T\$$. Since we can freely switch left and right siblings in the Huffman code, we will ensure that this last bit is zero. Hence the

[2] Note that these $n$ and $H_0$ refer to $T\$$, not $T$. However, the difference between both is only $O(\log n)$, and will be absorbed by the $o(n)$ terms that will appear later.

```
Algorithm Huff-FM_Search($P'$,$B$,$Bh$)
    $i = m'$;
    $sp = 1$; $ep = n'$;
    while (($sp \leq ep$) and ($i \geq 1$)) do
        if $P'[i] = 0$ then
            $sp = (sp - 1) - rank(B, sp - 1) + 1 + [sp - 1 < p_{\#}]$;
            $ep = ep - rank(B, ep) + [ep < p_{\#}]$;
        else $sp = n' - rank(B, n') + rank(B, sp - 1) + 1$;
            $ep = n' - rank(B, n') + rank(B, ep)$;
        $i = i - 1$;
    if $ep < sp$ then $occ = 0$ else $occ = rank(Bh, ep) - rank(Bh, sp - 1)$;
    if $occ = 0$ then return "not found" else return "found ($occ$) occs".
```

Figure 2: Algorithm for counting the number of occurrences of $P'[1 \ldots m']$ in $T'[1 \ldots n']$.

correct $B$ sequence would be of length $n' + 1$, starting with 0 (which corresponds to $T'[n']$, the character preceding the occurrence of "#", since $\# < 0 < 1$), and it would have $B[p_{\#}] = \#$. To obtain the right mapping to our binary $B$, we must correct $C[0] + Occ(B, 0, i) = i - rank(B, i) + [i < p_{\#}]$, that is, add 1 to the original value when $i < p_{\#}$. The computation of $C[1] + Occ(B, 1, i)$ remains unchanged.

Therefore, by preprocessing $B$ to solve *rank* queries, we can search $B$ exactly as the FM-index. The extra space required by the *rank* structure is $o(H_0 n)$, without any dependence on the alphabet size. Overall, we have used at most $n(2H_0 + 2)(1 + o(1))$ bits for our representation. This will grow slightly in the next sections due to additional requirements.

Our search pattern is not the original $P$, but its binary coding $P'$ using the Huffman code we applied to $T$. If we assume that the characters in $P$ have the same distribution of $T$, then the length of $P'$ is $< m(H_0 + 1)$. This is the number of steps to search $B$ using the FM-index search algorithm.

The answer to that search, however, is different from that of the search of $T$ for $P$. The reason is that the search of $T'$ for $P'$ returns the number of suffixes of $T'$ that start with $P'$. Certainly these include the suffixes of $T$ that start with $P$, but also other superfluous occurrences may appear. These correspond to suffixes of $T'$ that do not start a Huffman codeword, yet they start with $P'$.

This is why we have marked the suffixes that start a Huffman codeword in $Bh$. In the range $[sp, ep]$ found by the search for $P'$ in $B$, every bit set in $Bh[sp \ldots ep]$ represents a true occurrence. Hence the true number of occurrences can be computed as $rank(Bh, ep) - rank(Bh, sp - 1)$. Figure 2 shows the search algorithm.

Therefore, the search complexity is $O(m(H_0 + 1))$, assuming that the zero-order distributions of $P$ and $T$ are similar. Next we show that the worst case search cost is $O(m \log n)$. This matches the worst case search cost of the original CSA (while our average case is better).

For the worst case, we must determine which is the maximum height of a Huffman tree with total frequency $n$. Consider the longest root-to-leaf path in the Huffman tree. The leaf symbol has frequency at least 1. Let us traverse the path upwards and consider the (sum of) frequencies encountered in the other branch at each node.

These numbers must be, at least: 1, 1, 2, 3, 5, …, that is, the Fibonacci sequence $F(i)$. Hence, a Huffman tree with depth $d$ needs that the text is of length at least $n \geq 1+\sum_{i=1}^{d} F(i) = F(d+2)$ [21, pp. 397]. Therefore, the maximum length of a code is $F^{-1}(n) - 2 = \log_{\phi}(n) - 2 + o(1)$, where $\phi = (1+\sqrt{5})/2$.

Therefore, the encoded pattern $P'$ cannot be longer than $O(m \log n)$ and this is also the worst case search cost, as promised. An exception to the above argument occurs when $P$ contains a character not present in $T$. This is easier, however, as we immediately know that $P$ does not occur in $T$.

Actually, it is possible to reduce the worst-case search time to $O(m \log \sigma)$, without altering the average search time nor the space usage, by forcing the Huffman tree to become balanced after level $(1+x)\log \sigma$. For details see [6].

# 4 Reporting Occurrences and Displaying the Text

Up to now we have focused on the search time, that is, the time to determine the suffix array interval containing all the occurrences. In practice, one needs also the text positions where they appear, as well as a text context. Since self-indexes replace the text, in general one needs to extract any text substring from the index.

Given the suffix array interval that contains the *occ* occurrences found, the FM-index reports each such position in $O(\sigma \log^{1+\varepsilon} n)$ time, for any $\varepsilon > 0$ (which appears in the sublinear space component). The CSA can report each in $O(\log^{\varepsilon} n)$ time, where $\varepsilon$ is paid in the $nH_0/\varepsilon$ space. Similarly, a text substring of length $L$ can be displayed in time $O(\sigma(L + \log^{1+\varepsilon} n))$ by the FM-index and $O(L + \log^{\varepsilon} n)$ by the CSA.

In this section we show that our index can do better than the FM-index, although not as well as the CSA. Using $(1+\varepsilon)n$ additional bits, we can report each occurrence position in $O(\frac{1}{\varepsilon}(H_0+1)\log n)$ time and display a text context in time $O(L \log \sigma + \log n)$ in addition to the time to find an occurrence position. On average, assuming that random text positions are involved, the overall complexity to display a text interval becomes $O((H_0+1)(L + \frac{1}{\varepsilon}\log n))$.

## 4.1 Reporting Occurrences

A first problem is how to extract, in $O(occ)$ time, the *occ* positions of the bits set in $Bh[sp \dots ep]$. This is easy using *select* function: *select*$(Bh, j)$, gives the position of the $j$-th bit set in $Bh$. This is the inverse of function *rank* and it can also be implemented in constant time using $o(n)$ additional space [11, 14, 2, 7]. Actually we need a simpler version, *selectnext*$(Bh, j)$, which gives the first 1 in $Bh[j, n]$.

Let $r = rank(Bh, sp{-}1)$. Then, the positions of the bits set in $Bh$ are $select(Bh, r+1)$, $select(Bh, r+2)$, …, $select(Bh, r+occ)$. We recall that $occ = rank(Bh, ep) - rank(Bh, sp{-}1)$. This can be expressed using *selectnext*: The positions $pos_1 \dots pos_{occ}$ can be found as $pos_1 = selectnext(Bh, sp)$, and $pos_{i+1} = selectnext(Bh, pos_i + 1)$. We focus now on how to find the text position of a valid occurrence.

We choose some $\varepsilon > 0$ and sample $\lfloor \frac{\varepsilon n}{2 \log n} \rfloor$ positions of $T'$ at regular intervals, with the restriction that only codeword beginnings can be chosen. For this sake, pick positions in $T'$ at regular intervals of length $\ell = \lceil \frac{2n'}{\varepsilon n} \log n \rceil$, and for each such position $1 + \ell(i-1)$, choose the beginning of the codeword being represented at $1 + \ell(i-1)$.

Recall from Section 3 that no Huffman codeword can be longer than $\log_{\phi} n - 2 + o(1)$

bits. Then, the distance between two chosen positions in $T'$, after the adjustment, cannot exceed

$$\ell + \log_\phi n - 2 + o(1) \quad \leq \quad \frac{2}{\varepsilon}(H_0+1)\log n + \log_\phi n - 1 + o(1) \quad = \quad O\left(\frac{1}{\varepsilon}(H_0+1)\log n\right)$$

Now, store an array $TS$ with the $\lfloor \frac{\varepsilon n}{2\log n} \rfloor$ positions of $\mathcal{A}'$ pointing to the chosen positions of $T'$, in increasing text position order. More precisely, $TS[i]$ refers to position $1+\ell(i-1)$ in $T'$ and hence $TS[i] = j$ such that $A'[j] = \mathit{select}(Th, \mathit{rank}(Th, 1+\ell(i-1)))$. Array $TS$ requires $\frac{\varepsilon n}{2}(1+o(1))$ bits, since each entry needs $\log n' \leq \log(n \log \min(n,\sigma)) = \log n + O(\log\log\min(n,\sigma))$ bits.

The same $\mathcal{A}'$ positions are now sorted and the corresponding $T$ positions (that is, $\mathit{rank}(Th, \mathcal{A}'[i])$) are stored in array $ST$, for other $\frac{\varepsilon n}{2}$ bits. Finally, we store an array $S$ of $n$ bits so that $S[i] = 1$ iff $\mathcal{A}'[\mathit{select}(Bh, i)]$ is in the sampled set. That is, $S[i]$ tells whether the $i$-th entry of $\mathcal{A}'$ pointing to beginning of codewords, points to a sampled text position. $S$ is further processed for *rank* queries.

Overall, we spend $(1+\varepsilon)n(1+o(1))$ bits for these three arrays, raising our final space requirement to $n(2H_0+3+\varepsilon)(1+o(1))$.

Let us focus first in how to determine the text position corresponding to an entry $\mathcal{A}'[i]$ for which $Bh[i] = 1$. Use bit array $S[\mathit{rank}(Bh, i)]$ to determine whether $\mathcal{A}'[i]$ points or not to a codeword beginning in $T'$ that has been sampled. If it does, then find the corresponding $T$ position in $ST[\mathit{rank}(S, \mathit{rank}(Bh, i))]$ and we are done. Otherwise, just as with the FM-index, determine position $i'$ whose value is $\mathcal{A}'[i'] = \mathcal{A}'[i]-1$. Repeat this process, which corresponds to moving backward bit by bit in $T'$, until a new codeword beginning is found, that is, $Bh[i'] = 1$. Now determine again whether $i'$ corresponds to a sampled character in $T$: Use $S[\mathit{rank}(Bh, i')]$ to determine whether $\mathcal{A}'[i']$ is present in $ST$. If it is, report text position $1 + ST[\mathit{rank}(S, \mathit{rank}(Bh, i'))]$ and finish. Otherwise, continue with $i''$ trying to report $2 + ST[\mathit{rank}(S, \mathit{rank}(Bh, i''))]$, and so on. The process must finish after $O\left(\frac{1}{\varepsilon}(H_0+1)\log n\right)$ backward steps in $T'$ because we are considering consecutive positions of $T'$ and that is the maximum distance among consecutive samples.

We have to specify how we determine $i'$ from $i$. In the FM-index, this is done via the LF-mapping, $i' = C[T^{bwt}[i]] + Occ(T^{bwt}, T^{bwt}[i], i)$. In our index, the LF-mapping over $\mathcal{A}'$ is implemented as $i' = i - \mathit{rank}(B, i)$ if $B[i] = 0$ and $i' = n' - \mathit{rank}(B, n') + \mathit{rank}(B, i)$ if $B[i] = 1$. This LF-mapping moves us from position $T'[\mathcal{A}'[i]]$ to $T'[\mathcal{A}'[i]-1]$.

Overall, an occurrence can be reported in worst case time $O(\frac{1}{\varepsilon}(H_0+1)\log n)$. Figure 3 gives the pseudocode.

## 4.2 Displaying Text

In order to display a text substring $T[l \ldots r]$ of length $L = r - l + 1$, we start by binary searching $TS$ for the smallest sampled text position larger than $r$. Given value $TS[j]$, we know that $S[\mathit{rank}(Bh, TS[j])] = 1$ as it is a sampled $\mathcal{A}'$ entry, and the corresponding $T$ position is simply $ST[\mathit{rank}(S, \mathit{rank}(Bh, TS[j]))]$. Once we find the first sampled text position that follows $r$, we have its corresponding position $i = TS[j]$ in $\mathcal{A}'$. From there on, we perform at most $O\left(\frac{1}{\varepsilon}(H_0+1)\log n\right)$ steps going backward in $T'$ (via the LF-mapping over $\mathcal{A}'$), position by position, until reaching

```
**Algorithm** Huff-FM_Position($i$,$B$,$Bh$,$ST$)
$d = 0$;
**while** $S[rank(Bh, i)] = 0$ **do**
    **do if** $B[i] = 0$ then $i = i - rank(B, i) + [i < p_{\#}]$;
        **else** $i = n' - rank(B, n') + rank(B, i)$;
    **while** $Bh[i] = 0$;
    $d = d + 1$;
**return** $d + ST[rank(S, rank(Bh, i))]$;
```

Figure 3: Algorithm for reporting the text position of the occurrence at $B[i]$. It is invoked for each $i = select(Bh, r + k)$, $1 \leq k \leq occ$, $r = rank(Bh, sp - 1)$.

```
**Algorithm** Huff-FM_Display($l$,$r$,$B$,$Bh$,$TS$)
$j = \min\{k,\ ST[rank(S, rank(Bh, TS[k]))] > r\}$; // binary search
$i = TS[j]$;
$p = ST[rank(S, rank(Bh, i))]$;
$L = \langle\ \rangle$;
**while** $p \geq l$ **do**
    **do** $L = B[i] \cdot L$;
        **if** $B[i] = 0$ **then** $i = i - rank(B, i) + [i < p_{\#}]$;
        **else** $i = n' - rank(B, n') + rank(B, i)$;
    **while** $Bh[i] = 0$;
    $p = p - 1$;
Huffman-decode the first $r - l + 1$ characters from list $L$;
```

Figure 4: Algorithm for extracting $T[l \ldots r]$.

the first bit of the codeword for $T[r + 1]$. Then, we obtain the $L$ preceding positions of $T$, by further traversing $T'$ backwards, collecting all its bits until reaching the first bit of the codeword for $T[l]$. The reversed bit stream collected is Huffman-decoded to obtain $T[l \ldots r]$.

Each of those $L$ characters costs us $O(H_0 + 1)$ on average because we obtain the codeword bits one by one. In the worst case they cost us $O(\log n)$. The overall time complexity is $O((H_0 + 1)(L + \frac{1}{\varepsilon} \log n))$ on average and $O(L \log n + (H_0 + 1)\frac{1}{\varepsilon} \log n)$ in the worst case. Figure 4 shows the pseudocode.

# 5 $K$-ary Huffman

The purpose of the idea of compressing the text before constructing the index is to remove the sharp dependence of the alphabet size of the FM index. This compression is done using a binary alphabet. In general, we can use Huffman over a coding alphabet of $k > 2$ symbols and use $\lceil \log k \rceil$ bits to represent each symbol. Varying the value of $k$ yields interesting time/space tradeoffs. We use only powers of 2 for $k$ values, so each symbol can be represented without wasting space.

The space usage varies in different aspects. Array $B$ increases its size since the compression ratio gets worse. $B$ has length $n' < (H_0^{(k)} + 1)n$ symbols, where $H_0^{(k)}$ is the zero order entropy of the text computed using base $k$ logarithm, that is, $H_0^{(k)} =$

$-\sum_{i=1}^{\sigma} \frac{n_i}{n} \log_k \left(\frac{n_i}{n}\right) = H_0/\log_2 k$. Therefore, the size of $B$ is bounded by $n' \log k = (H_0 + \log k)n$ bits. The size of $Bh$ is reduced since it needs one bit per symbol, and hence its size is $n'$. The total space used by these structures is then $n'(1 + \log k) < n(H_0^{(k)} + 1)(1 + \log k)$, which is not larger than the space requirement of the binary version, $2n(H_0 + 1)$, for $1 \leq \log k \leq H_0$.

The *rank* structures also change their size. The *rank* structures for $Bh$ are computed in the same way of the binary version, and therefore they reduce their size, using $o(H_0^{(k)} n)$ bits. For $B$, we can no longer use the *rank* function to simulate $Occ$. Instead, we need to calculate the occurrences of each of the $k$ symbols in $B$. For this sake, we precalculate sublinear structures for each of the symbols, including $k$ tables that count the occurrences of each symbol in a chunk of $b = \lceil \log_k(n)/2 \rceil$ symbols. Hence, we need $o(kH_0^{(k)} n)$ bits for this structures. In total, we need $n(H_0^{(k)} + 1)(1 + \log k) + o(H_0^{(k)} n(k + 1))$ bits.

Regarding the time complexities, the pattern has length $< m(H_0^{(k)} + 1)$ symbols, so this is the search complexity, which is reduced as we increase $k$. For reporting queries and displaying text, we need the same additional structures $TS$, $ST$ and $S$ that for the binary version. The $k$-ary version can report the position of an occurrence in $O\left(\frac{1}{\epsilon}(H_0^{(k)} + 1) \log n\right)$ time, which is the maximum distance between two sampled positions. Similarly, the time to display a substring of length $L$ becomes $O((H_0^{(k)} + 1)(L + \frac{1}{\epsilon} \log n))$ on average and $O(L \log n + (H_0^{(k)} + 1)\frac{1}{\epsilon} \log n)$ in the worst case.

# 6 Experimental Results

In this section we show experimental results on counting, reporting and displaying queries and compare the efficiency to existing indexes. The indexes used for the experiments were the FM-index implemented by Navarro [18], Sadakane's CSA [19], the RLFM index [17], the SSA index [17], and the LZ index [18]. Other indexes, like the Compressed Compact Suffix Array (CCSA) of Mäkinen and Navarro [16], the Compact SA of Mäkinen [15] and the implementation of Ferragina and Manzini of the FM-index were not included because they are not comparable to the FM Huffman index due either to their large space requirement (Compact SA) or their high search times (CCSA and original FM index).

We considered three types of text for the experiments: 80 MB of English text obtained from the TREC-3 collection [3] (files `WSJ87-89`), 60 MB of DNA and 55 MB of protein sequences, both obtained from the BLAST database of the NCBI[4] (files `month.est_others` and `swissprot` respectively).

Our experiments were run on an Intel(R) Xeon(TM) processor at 3.06 GHz, 2 GB of RAM and 512 KB cache, running Gentoo Linux 2.6.10. We compiled the code with `gcc 3.4.2` using optimization option `-O9`.

Now we show the results regarding the space used by our index and later the results of the experiments classified by query type.

[3] Text Retrieval Conference, `http://trec.nist.gov`

[4] National Center for Biotechnology Information, `http://www.ncbi.nlm.nih.gov`

## 6.1 Space results

Table 1 (left) shows the space that the index takes as a fraction of the text for different values of $k$ and the three types of files considered. These values do not include the space required to report positions and display text.

We can see that the space requirements are lowest for $k = 4$. For higher values this space increases, although staying reasonable until $k = 16$. With higher values the spaces are too high for these indexes to be comparable to the rest.

We did not consider the version of the index with $k = 8$ in the other experiments because we do not expect an improvement in the query time, since $\log k$ is not a power of 2 and then the computation of *Occ* is slower (reasons omitted for lack of space). The version with $k = 16$ can lead to a reduction in query time, but the access to 4 machine words for the calculation of *Occ* (reasons omitted for lack of space) could negatively affect it. It is important to say that these values are only relevant for the English text and proteins, since it does not make sense to use them for DNA.

It is also interesting to see how the space requirement of the index is divided among its different structures. Table 1 (right) shows the space used by each of the structures for the index with $k = 2$ and $k = 4$ for the three types of texts considered.

| $k$ | Fraction of text | | |
|---|---|---|---|
| | English | DNA | Proteins |
| 2 | 1,68 | 0,76 | 1,45 |
| 4 | 1,52 | 0,74 | 1,30 |
| 8 | 1,60 | 0,91 | 1,43 |
| 16 | 1,84 | — | 1,57 |
| 32 | 2,67 | — | 1,92 |
| 64 | 3,96 | — | — |

| | **FM-Huffman** $k = 2$ | | | **FM-Huffman** $k = 4$ | | |
|---|---|---|---|---|---|---|
| Structure | Space [MB] | | | Space [MB] | | |
| | English | DNA | Proteins | English | DNA | Proteins |
| $B$ | 48.98 | 16.59 | 29.27 | 49.81 | 18.17 | 29.60 |
| $Bh$ | 48.98 | 16.59 | 29.27 | 24.91 | 9.09 | 14,80 |
| $Rank(B)$ | 18,37 | 6,22 | 10,97 | 37,36 | 13,63 | 22,20 |
| $Rank(Bh)$ | 18,37 | 6,22 | 10,97 | 9,34 | 3,41 | 5,55 |
| Total | 134,69 | 45,61 | 80,48 | 121,41 | 44,30 | 72,15 |
| Text | 80,00 | 60,00 | 55,53 | 80,00 | 60,00 | 55,53 |
| Fraction | 1.68 | 0.76 | 1.45 | 1.52 | 0.74 | 1.30 |

Table 1: On top, space requirement of our index for different values of $k$. The value corresponding to the row $k = 8$ for DNA actually corresponds to $k = 5$, since this is the total number of symbols to code in this file. Similarly, the value of row $k = 32$ for the protein sequence corresponds to $k = 24$. On the bottom, detailed comparison of $k = 2$ versus $k = 4$. We omit the the spaces used by the Huffman table, the constant-size tables for *Rank*, and array $C$, since they are negligible.

For higher values of $k$ the space used by $B$ will increase since the use of more symbols for the Huffman codes increases the resulting space. On the other hand, the size of $Bh$ decreases at a rate of $\log k$ and so do its *rank* structures. However, the space of the *rank* structures of $B$ increases rapidly, as we need $k$ structures for an array that reduces its size at a rate of $\log k$, which is the reason of the large space requirement for high values of $k$.

## 6.2 Counting queries

For the three files, we show the search time as a function of the pattern length, varying from 10 to 100, with a step of 10. For each length we used 1000 patterns taken from random positions of each text. Each search was repeated 1000 times. Figure 5 (left) shows the time for counting the occurrences for each index and for the three files considered. As the CSA index needs a parameter to determine its space for this type of queries, we adjusted it so that it would use approximately the same space of the binary FM-Huffman index.

We show in Figure 5 (right) the average search time per character along with the minimum space requirement of each index to count occurrences. Unlike the CSA, the other indexes do not need a parameter to specify their size for counting queries. Therefore, we show a point as the value of the space used by the index and its search time. For the CSA index we show a line to resemble the space-time tradeoff for counting queries.

## 6.3 Reporting queries

We measured the time that each index took to search for a pattern and report the positions of the occurrences found. From the English text and the DNA sequence we took 1000 random patterns of length 10. From the protein sequence we used patterns of length 5. We measured the time per occurrence reported varying the space requirement for every index except the LZ, which has a fixed size. For the CSA we set the two parameters, namely the size of the structures to report and the structures to count, to the same value, since this turns out to be optimal. Figure 6 (left) shows the times per occurrence reported for each index as a function of its size.

## 6.4 Displaying text

We measured the time to display a context per character displayed. That is, we searched for the 1000 patterns and displayed 100 characters around each of the positions of the occurrences found. Figure 6 (right) shows this time along with the minimum space required for each index for the counting functionality, since the display time per character does not depend on the size of the index. This is not true for the CSA index, whose display time does depend on its size. For this index we show the time measured as a function of its size.

## 6.5 Analysis of Results

We can see that our FM-Huffman $k = 16$ index is the fastest for counting queries for English and proteins and that the version with $k = 4$ is, together with the SSA,

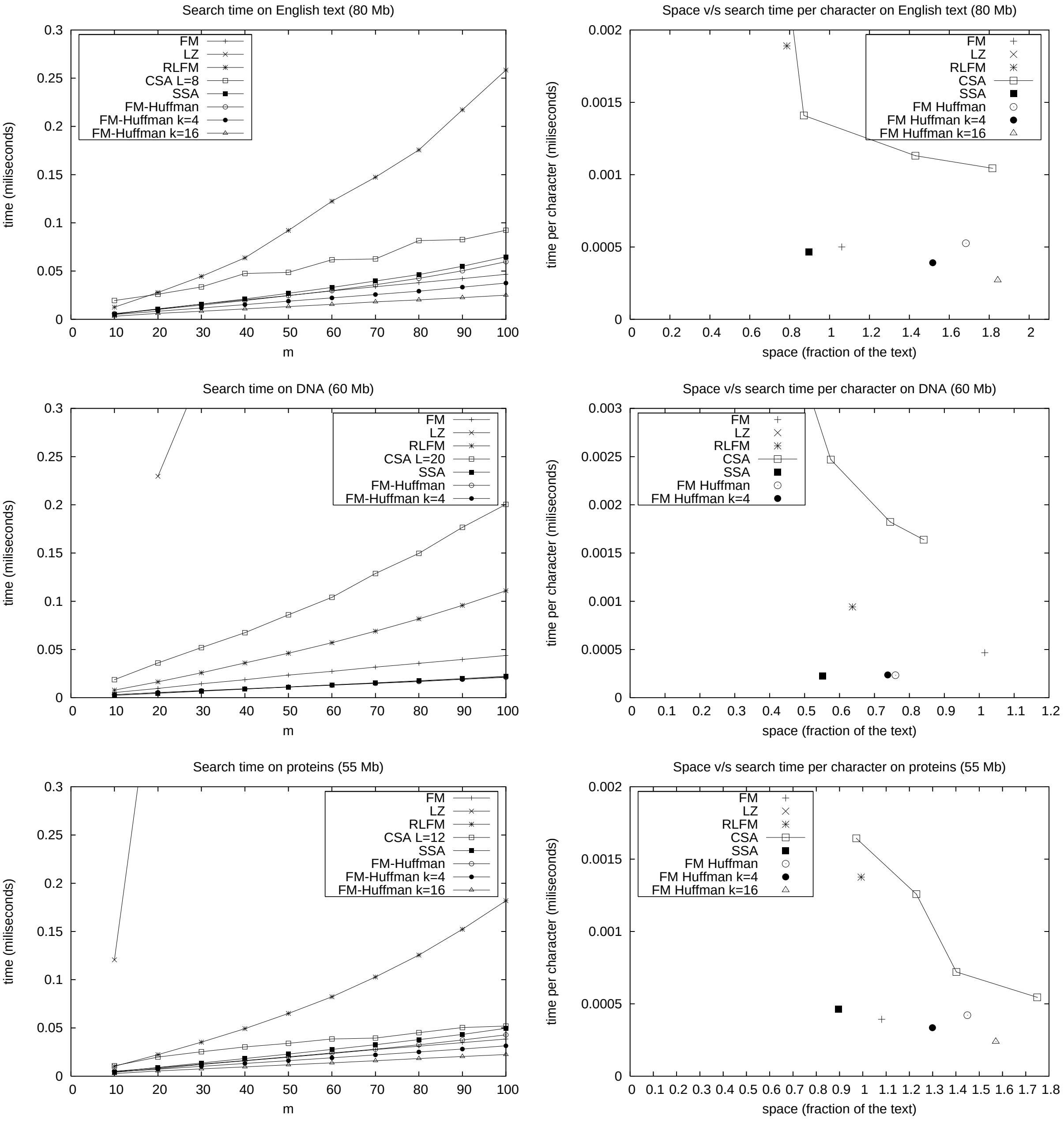

Figure 5: On the left, search time as a function of the pattern length over, English (80 MB), DNA (60 MB), and a proteins (55 MB). The times of the LZ index do not appear on the English text plot, as they range from 0.5 to 4.6 ms. In the DNA plot, the time of the LZ index for $m = 10$ is 0.26. The reason of this increase is the large number of occurrences of these patterns, which influences the counting time for this index. On the right, average search time per character as a function of the size of the index.

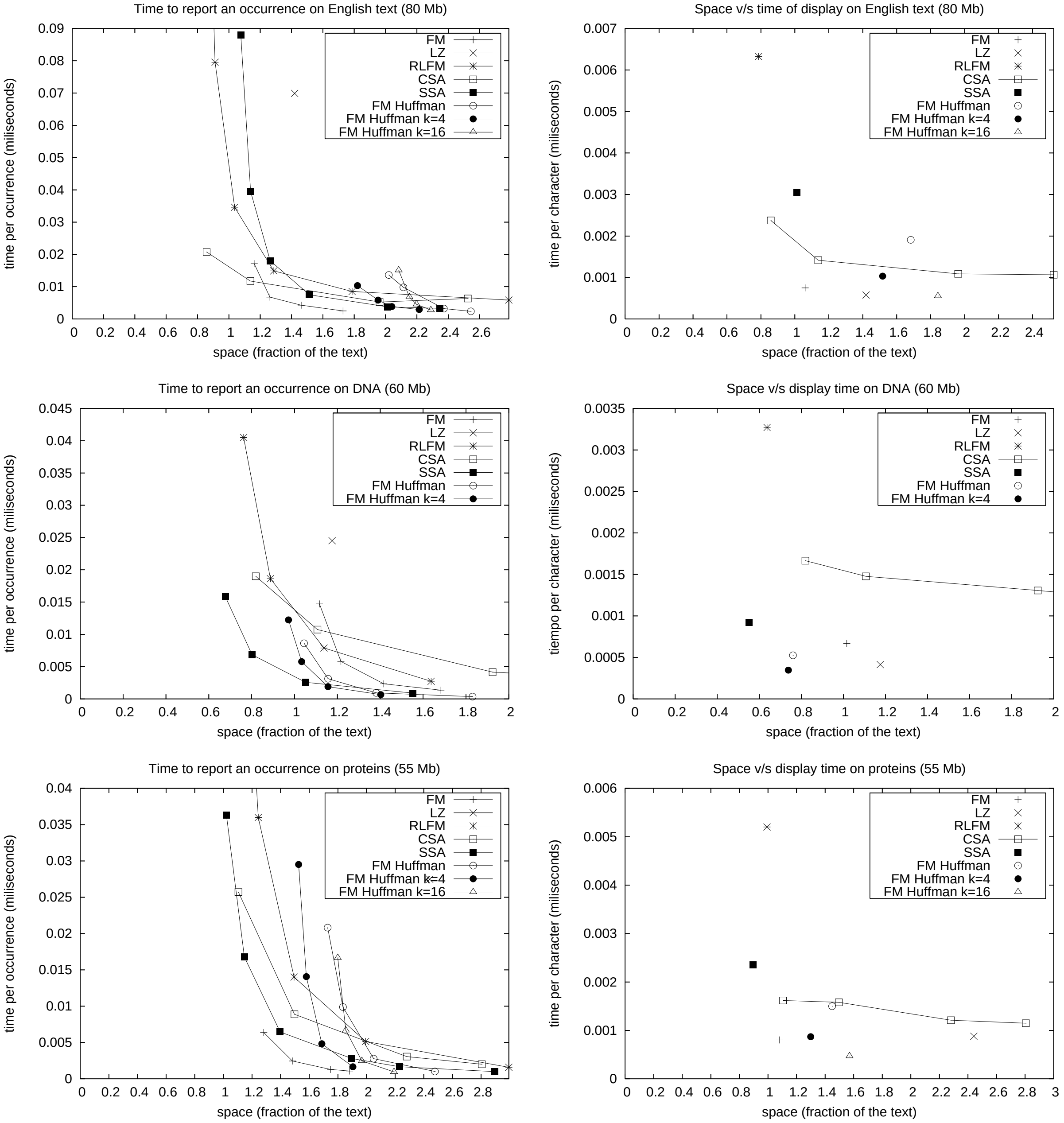

Figure 6: On the left, time to report the positions of the occurrences as a function of the size of the index. On the right, time per character to display text passages. We show the results of searching on 80 MB of English text, 60 MB of DNA and finally 55 MB of proteins.

the fastest for DNA. The binary FM-Huffman index takes the same time that $k = 4$ version for DNA and it is a little bit slower that the FM-index for the other two files. As expected, the three versions are faster than CSA, RLFM and LZ, the latter not being competitive for counting queries. Regarding the space usage, the SSA is an attractive tradeoff alternative for the three files, since it uses less space than our index and has low search times (although not as good as our index except on DNA). The same happens with the FM-index, although not for DNA, where it uses more space and time than our index.

For reporting queries, our index loses to the FM-index for English and proteins, mainly because of its large space requirement. Also, it only surpasses the RLFM and CSA, and barely the SSA, for large space usages. For DNA, however, our index, with $k = 2$ and $k = 4$, is better than the FM-index, although it loses to the SSA for low space usage. This reduction in space in our index is due to the low zero-order entropy of the DNA, which makes our index compact and fast.

Regarding display time, our index variants are again the fastest. On English text, however, the LZ is equally fast and smaller (version $k = 16$ is the relevant one here). On DNA, the $k = 4$ version is faster than any other, requiring also little space. Those taking (at best 20%) less space are about 3 times slower. Finally, on proteins, the version $k = 16$ is clearly the fastest. The best competitor, the FM-index, uses 30% less space but it is twice as slow.

The versions of our index with $k = 4$ improved the space and time of the binary version. The version with $k = 16$ increased the space usage, but resulted in the fastest of the three for counting and display queries. In general, our index is not the smallest but it is the fastest among those using the same space.

# 7 Conclusions

We have focused in this paper on a practical data structure inspired by the FM-index [3], which removes its sharp dependence on the alphabet size $\sigma$. Our key idea is to Huffman-compress the text before applying the Burrows-Wheeler transform over it. Over a text of $n$ characters, our structure needs $O(n(H_0+1))$ bits, being $H_0$ the zero-order entropy of the text. It can search for a pattern of length $m$ in $O(m(H_0 + 1))$ average time. Our structure has the advantage over the FM-index of not depending at all on the alphabet size, and of having better complexities to report text occurrences and displaying text substrings. In comparison to the CSA [19], it has the advantage of having better search time.

Furthermore, our structure is simple and easy to implement. Our experimental results show that our index is competitive in practice against other implemented alternatives. In most cases it is not the most succinct, but it is the fastest, even if we let the other structures use significatively more space.

# References

[1] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. *DEC SRC Research Report 124*, 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] Sz. Grabowski, V. Mäkinen, and G. Navarro. First Huffman, then Burrows-Wheeler: an alphabet-independent FM-index. In *Proc. SPIRE'04*, pp. 210–211, 2004. Poster.

[6] Sz. Grabowski, V. Mäkinen, and G. Navarro. First Huffman, then Burrows-Wheeler: an alphabet-independent FM-index. Technical Report TR/DCC-2004-4. Dept. of Computer Science, Univ. of Chile, July 2004. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/huffbwt.ps.gz`.

[7] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proc. WEA'05*, pp. 27–38, 2005.

[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*, 2004.

[10] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. STOC'00*, pp. 397–406, 2000.

[11] G. Jacobson. *Succinct Static Data Structures.* PhD thesis, CMU-CS-89-112, Carnegie Mellon University, 1989.

[12] J. Kärkkäinen. *Repetition-Based Text Indexes*, PhD Thesis, Report A-1999-4, Department of Computer Science, University of Helsinki, Finland, 1999.

[13] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22, pp. 935–948, 1993.

[14] I. Munro. Tables. In *Proc. FSTTCS'96*, pp. 37–42, 1996.

[15] V. Mäkinen. Compact Suffix Array — A space-efficient full-text index. *Fundamenta Informaticae* 56(1-2), pp. 191–210, 2003.

[16] V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In *Proc. CPM'04*, pp. 420–433. LNCS 3109, 2004.

[17] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. In *Proc. CPM'05*, pp. 45–56. LNCS 3537, 2005.

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

[19] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. ISAAC'00*, LNCS 1969, pp. 410–421, 2000.

[20] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14, pp. 249–260, 1995.

[21] I. Witten, A. Moffat and T. Bell. *Managing Gigabytes.* Morgan Kaufmann Publishers, New York, 1999. Second edition.