FM-KZ: An Even Simpler Alphabet-Independent FM-Index

Rafał Przywarski[1], Szymon Grabowski[1], Gonzalo Navarro[2], and Alejandro Salinger[3]

[1] Computer Engineering Dept., Tech. Univ. of Łódź, Poland.
[2] Dept. of Computer Science, Univ. of Chile, Chile.
[3] David R. Cheriton School of Computer Science, University of Waterloo, Canada.

Abstract. In an earlier work [7] we presented a simple FM-index variant, based on the idea of Huffman-compressing the text and then applying the Burrows-Wheeler transform over it. The main drawback of using Huffman was its lack of synchronizing properties, forcing us to supply another bit stream indicating the Huffman codeword boundaries. In this way, the resulting index needed $O(n(H_0 + 1))$ bits of space but with the constant 2 (concerning the main term). There are several options aiming to mitigate the overhead in space, with various effects on the query handling speed. In this work we propose Kautz-Zeckendorf coding as a both simple and practical replacement for Huffman. We dub the new index FM-KZ. We also present an efficient implementation of the rank operation, which is the main building brick of the FM-KZ. Experimental results show that our index provides an attractive space/time tradeoff in comparison with existing succinct data structures, and in the DNA test it even wins both in search time and space use. An additional asset of our solution is its relative simplicity.

1 Introduction

A *full-text index* is a data structure that enables to determine the *occ* occurrences of a short pattern $P = p_1 p_2 \ldots p_m$ in a large text $T = t_1 t_2 \ldots t_n$ without a need of scanning over the whole text T. Text and pattern are sequences of characters over an alphabet Σ of size σ. The pattern may appear at any position in T, and its length is also arbitrary. 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*).

Classic full-text indexes, albeit powerful and versatile, need space several times greater than the text itself. Hence, a natural interest in *succinct* full-text indexes has been observed in recent years. A comprehensive survey of existing techniques in this very active research area can be found in [13].

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 kth 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 nicely bound both space and time complexities, preferably with no (or mild) dependence on the alphabet size.

The large alphabet dependence of the original FM-index 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 $5H_k n + O\left((\sigma \log \sigma + \log\log n)\frac{n}{\log n} + n^{\gamma}\sigma^{\sigma+1}\right)$ 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.

One of the proposals to eliminate an exponential dependence on the alphabet size was Huffman FM-index [7]: It was based on the *backward search* idea of [4] but the novelty was to Huffman encode the text (and the pattern) so as to reduce the alphabet to binary. As a result, any dependence on the alphabet size was removed. We showed 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.

Since the original presentation, its implementation has been optimized and also a variant with 4-ary Huffman has been checked [6]. Albeit not among the most succinct, the 4-ary Huffman FM-index appears to be among the fastest and thus practical indices.

In this paper we present an alternative to Huffman coding variants. Instead, we use Kautz-Zeckendorf coding [11, 17], capable of instant detection of codeword boundaries. To give the flavor of this idea, we note that in its basic variant, the Kautz-Zeckendorf code has no codeword with any two adjacent 1's.

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 a 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 ith row of $\mathcal{M}$ contains string $t_j t_{j+1} \cdots t_n \$ t_1 \cdots t_{j-1}$. Given the suffix array, the search for the occurrences of the pattern $P = p_1 p_2 \cdots p_m$ is trivial. The occurrences form an interval $[sp, ep]$ in $\mathcal{A}$ such that suffixes $t_{\mathcal{A}[i]} t_{\mathcal{A}[i]+1} \cdots t_n$, $sp \le i \le ep$, 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}:

1. 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).
2. 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 $X[1, i]$.
3. 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, 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.

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 $T[1 \dots n]$.

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 5.

3 Rank and Select Queries on Bit Arrays

A crucial building block we use is a data structure to perform *rank* operations over a bit array. Given a bit sequence $B[1 \dots n]$, $rank(B, i)$ 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 [10, 12, 2]. The solution, as well as its more practical implementation variants, are described in [5]; here we present a novel implementation, which seems to be fastest in practice.

For an input bit array B of size n and a given parameter bs we create a lookup table N with $\lceil n/2^{bs} \rceil$ entries. Namely, for each $k = 0 \dots \lfloor n/2^{bs} \rfloor - 1$ we compute: $N[k] = rank(B, (k+1) * 2^{bs})$. If $\lceil n/2^{bs} \rceil > \lfloor n/2^{bs} \rfloor$, then we also compute: $N[\lfloor n/2^{bs} \rfloor] = rank(B, n)$.

The above structure needs $32 * \lceil n/2^{bs} \rceil = O(n)$ bits, where the constant 32 is the number of bits per entry of N.

Now, we calculate $rank(B,i)$ as follows. If $i < 2^{bs}$, then $rank(B,i) = popcount(B, 0 \ldots i)$. Otherwise, $rank(B,i) = N[\lfloor i/2^{bs} \rfloor - 1] + popcount(B, (\lfloor i/2^{bs} \rfloor * 2^{bs}) \ldots i)$. The operation $popcount(B, a \ldots b)$ returns the number of set bits in the interval $B[a \ldots b]$, $a \leq b$, making use of a precomputed table. As long as the interval width is on the order of machine word, this is a constant time operation.

Sometimes we need to calculate the inverse function, $select(B,j)$, which gives the position of the j-th bit set in B. It can also be implemented in constant time using $o(n)$ additional space [10, 12, 2]. More practical implementations exist [5], but it is always significantly slower than $rank$, and also more rarely needed.

4 First Huffman, then Burrows-Wheeler

We focus now on our index representation, starting from the original variant. 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[4]. Let us call T' this sequence. Let us also 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, however, 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 $B = (T')^{bwt}$. The terminator character, "\$", is excluded from T' so as to have a binary alphabet.

More precisely, let $\mathcal{A}'[1 \ldots n']$ be the suffix array for text T', that is, a permutation of the set $1 \ldots n'$ such that $T'[A'[i] \ldots n'] < T'[A'[i+1] \ldots n']$ 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 $B[1 \ldots n']$, 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 $Bh[1 \ldots n']$, 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 $Occ(B, 0, i) = i - rank(B, i)$. 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 $n' - rank(B, n')$ should be precomputed in practice). 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

[4] 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.

code, we will ensure that this last bit is zero. Hence the 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. Converting P to P' takes $O(m)$ time. 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 the reason 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 depicts the search algorithm.

Algorithm Huff-FM_Search(P',B,Bh)
$i = m'$;
$sp = 1$; $ep = n'$;
while $((sp \le ep)$ **and** $(i \ge 1))$ **do**
  if $P'[i] = 0$ **then**
    $sp = (sp - 1) - rank(B, sp - 1) + 1 + 1 - [sp - 1 \ge p_\#]$;
    $ep = ep - rank(B, ep) + 1 - [ep \ge p_\#]$;
  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 $T'[1 \ldots n']$.

Therefore, the search complexity is $O(m(H_0 + 1))$, assuming that the zero-order distributions of P and T are similar. It is well-known that the longest Huffman codeword does not exceed $O(m \log n)$ bits. From this we immediately obtain the worst case search cost of $O(m \log n)$ for our index. This matches the worst case search time of the compressed suffix array (CSA) of Sadakane [16]. An exceptional situation 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.

Is it in fact possible to achieve $O(m \log n)$ search complexity also for the worst case, for the price of $2n$ extra bits. Basically, the idea is to use a length-limited Huffman coding variant but we omit the details and analysis due to lack of space. This idea, however, does not have much importance in practice because extremely skew symbol distributions almost never happen and thus optimizing the worst case is hardly worth any effort.

5 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 ε is paid in the nH_0/ε 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.

Our index can do better than the FM-index in this respect, 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 $O((H_0+1)(L+\frac{1}{\varepsilon}\log n))$. Those complexities hold for all the variants of our solution: based on the binary or higher arity Huffman, or on the Kautz-Zeckendorf coding. Still, the overall idea of reporting and displaying via sampling sorted suffixes at regular intervals was first presented in the seminal work on the FM-index, and is now widely used in the field. Details can be found e.g., in [6].

A related query type concerns displaying the text around each pattern occurence. More generally, we want to display a text substring $T[l \ldots r]$ of length $L = r - l + 1$. Again, we make use of a known technique, on the overall obtaining the following time complexities [6]: $O((H_0 + 1)(L + \frac{1}{\varepsilon}\log n))$ in the average case, and $O(L \log \sigma + (H_0 + 1)\frac{1}{\varepsilon}\log n)$ in the worst case.

6 *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. We call this generalization the k-ary FM-Huffman. 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 no longer can 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 $O(L \log \sigma + (H_0^{(k)} + 1)\frac{1}{\epsilon} \log n)$ in the worst case.

7 Kautz-Zeckendorf Coding

The condition for getting rid of the Bh array is to have a coding for which the bit stream enables instant synchronization at codeword boundaries. A solution could be based on the representation of integers, first advocated by Kautz [11] for its synchronization properties, which presents each number in a unique form as a sum of Fibonacci numbers. This technique is better known from a work by Zeckendorf [17], therefore we will call it Kautz-Zeckendorf coding.

Consider the Fibonacci sequence $f_1 = 1$, $f_2 = 2$, and $f_{i+2} = f_{i+1} + f_i$. The resulting sequence of *Fibonacci numbers* is 1, 2, 3, 5, 8, 13, ... It is easy to prove by induction that any integer number N can be uniquely decomposed into a sum of Fibonacci numbers, where each number is summed at most once and no two consecutive numbers are used in the decomposition. (If two consecutive numbers f_i and f_i and f_{i+1} appear in the decomposition we can use f_{i+2} instead.) Thus we can represent N as a bit vector, whose i-th bit is set iff the i-th Fibonacci number is used to represent N. No two consecutive bits can be set in this representation because this would mean that we used two consecutive numbers in the decomposition. This can be generalized to k consecutive ones [11]. The recurrence is now $f_i = i$ for $i \leq k$ and $f_{i+k} = f_{i+k-1} + f_{i+k-2} + \ldots + f_{i+1} + f_i$. In this representation we do not permit a sequence of k consecutive numbers in the decomposition, and thus no stream of k 1's appears in the bit vector.

We use this encoding as follows. We sort the source symbols by frequency and then assign the binary encoding of number N to the N-th most frequent symbol. In addition, all the encodings are prepended with a sequence of k 1's followed by one 0. Note that nowhere else in the encoding are there k adjacent 1's.

If, during the LF-mapping, we read a 0 and then k successive 1's from T', we know that we are at a codeword beginning. Thus, Bh is no longer needed. A practical side-effect is also that there is no need for *select* to find the successive matches: they all are in a contiguous range of the matrix rows. All the rest of the operatory remains unchanged.

Let us consider the performance of Kautz-Zeckendorf coding with the two most practical (at least for natural languages) parameters, $k = 2$ and $k = 3$. The regular expressions for all valid codewords in those cases are $110(0|10) * (\varepsilon|1)$ and $1110(0|10|110) * (\varepsilon|1|11)$, respectively. We calculated the average codeword length for the 80 MB English text used in Section 8. Note that all we needed to know for this estimation was the knowledge of zero-order symbol distribution in the text. For $k = 2$ and $k = 3$ the average lengths were 5.696 and 6.420 bits per symbol, respectively. The only component of the index, apart from the B array, is the *rank* structure for B. The fastest *rank* in the new implementation needs 25% of the text size. Taking this figure, we obtain approximately $0.89n$ and $1.00n$ overall space occupancy, respectively. Those results are better than of any other variant of our index, but the price is a longer search time. Note that even less space can be obtained with a *rank* implementation using 10% of the text size [5], for a relatively little slow-down. Other options can be better for other text types, e.g., for DNA using $k = 1$ (actually a unary code) is a better choice.

8 Experimental Results

We implemented our indexes, both the original, the k-ary and the KZ versions, making some practical considerations that differ from the theoretical ones. The main difference is the calculation of *rank* and *Occ*, where we used the solution described in [5], for the older index variants, or the new *rank* implementation described in Section 3. The new indexes will be called FM-KZ1 and FM-KZ2, corresponding to the parameters $k = 1$ and $k = 2$, respectively.

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 [15], Sadakane's CSA [16], the RLFM index [14], the SSA index [14] and the LZ index [15]. Other indexes whose implementations are available were not included because they are not comparable to the FM Huffman / FM-KZ index due either to their large space requirement or their high search times .

We considered three types of text for the experiments: 80 MB of English text obtained from the TREC-3 collection [5] (files `WSJ87-89`), 60 MB of DNA and 55 MB of protein sequences, both obtained from the BLAST database of the NCBI[6] (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 3.3.5` using optimization option `-O9`.

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

[5] Text Retrieval Conference, http://trec.nist.gov
[6] National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov

8.1 Space results

For the experiments we considered the binary, the 4-ary, and the KZ versions of our index. It is interesting to know how the space requirement of the Huffman-based index varies according to the parameter k. 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. It would be interesting to study the time performance to the versions of the index with $k = 8$ and $k = 16$. With $k = 8$ we do not expect an improvement on the query time since $\log k$ is not a power (reasons omitted) of 2 and therefore the computation of Occ is slower. The version with $k = 16$ could lead to a reduction in query time, but the access to 4 machine words for the calculation of Occ could negatively affect it. It is important to say that this 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	—	—

Structure	**FM-Huffman $k = 2$** Space [MB]			**FM-Huffman $k = 4$** 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 the left, 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 right, detailed comparison of $k = 2$ versus $k = 4$. We omit 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.

Now, let us take a look at the FM-KZ1 and FM-KZ2 space/time behavior. For DNA, the FM-KZ1 is a clear winner: among the fastest and definitely the most succinct, also it is hard to imagine a simpler full-text index (as the encoding is merely the unary code).

On the English text, FM-KZ2 is takes about $1.0n$ space, much less than other indexes from our family, but is also considerably slower, e.g. more than 1.5 times slower than FM Huffman with $k = 4$.

8.2 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. We obtained an average error of 2.6% with a confidence of 95%. Figure 3 (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 that the binary FM-Huffman index.

We also show 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 per character. For the CSA index we show a line to resemble the space-time tradeoff for counting queries. The time per character for each pattern length is the search time divided by the value of the length. The time per character shown on the plot is the average of these times for each length. Figure 3 (right) shows the search time per character for each index and for each type of text.

8.3 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. Our measures have a 2.2% error with 95% confidence. Figure 4 shows the times per occurrence reported for each index as a function of its size.

8.4 Displaying text

We measured the time that each index took to show the first character of a text context around the occurrences found. More precisely, this is the time of searching for a pattern, locating the position of an occurrence and showing one character of the text in the context area of the position located. Usually this character is the one at the position of the occurrence, but it can also be a different close one, depending on each index. We measured this time as a function of the size used by each index. We used the same 1000 patterns used for the reporting experiment, obtaining an average error of 1.6% with 95% confidence. Figure 5 (left) shows the time to display the first character as a function of the space requirement for each index and for each type of text.

In addition, 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. We subtracted from this time the time to display the first character and divided it by the amount of characters displayed. For this experiment, we obtained an average error of 6% with 95% confidence. Figure 5 (right) shows this time along with the minimum space required for each index for the counting functionality,

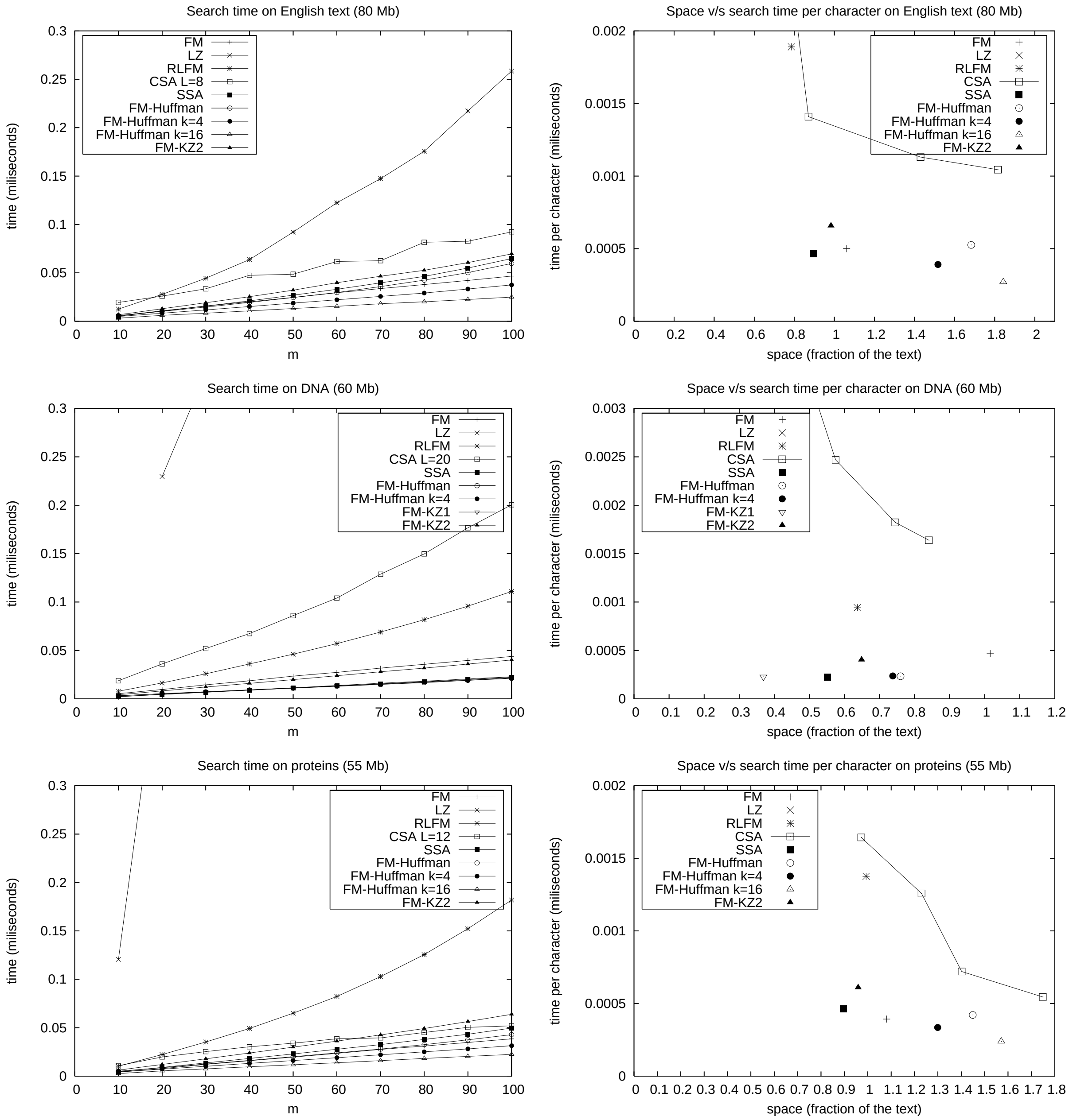

Figure 3. On the left, search time as a function of the pattern length over, English (80 MB), DNA (60 MB), and a proteins (55 MB). The times of the LZ index do not appear on the English text plot, as they range from 0.5 to 4.6 ms. In the DNA plot, the time of the LZ index for $m = 10$ is 2.6. The reason 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.

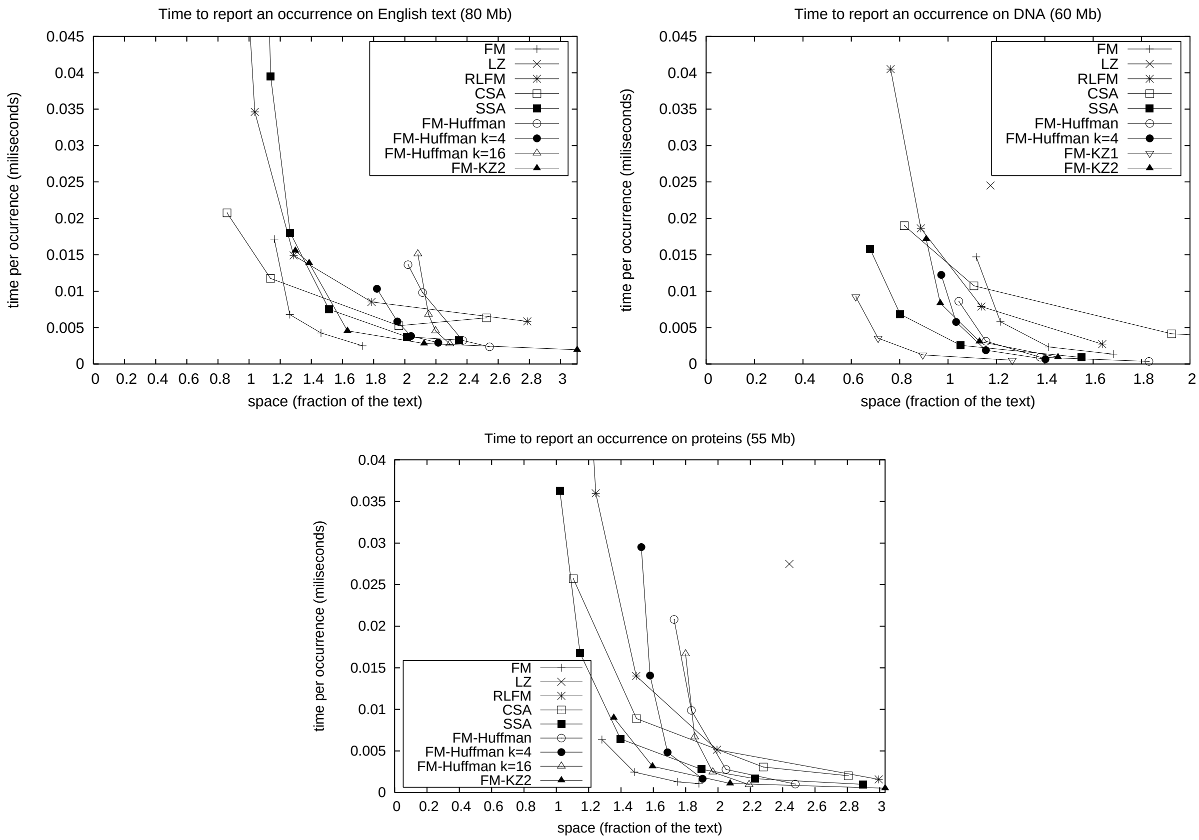

Figure 4. Time to report the positions of the occurrences as a function of the size of the index. We show the results of searching on 80 MB of English text, 60 MB of DNA and finally 55 MB of proteins.

since the display time per character does not depend on the size of the index. This is not true for the CSA index, whose time to display per character does depend on its size. For this index we show the time measured as a function of its size.

8.5 Analysis of Results

We can see that our FM-Hufman $k = 4$ and $k = 16$ indexes are among the fastest for counting queries for the three types of files. 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, all those versions are faster than CSA, RLFM and LZ, the latter not being competitive for counting queries. Regarding the space usage, the FM-index turns out to be a better tradeoff alternative for the English text and protein sequences, since it uses less space than our index and has low search times. For DNA, all the Huffman based versions of our index are good alternatives, considering their low space requirement and search time.

Still, the new player, FM-KZ index, is a particularly good choice for DNA. It is way ahead of the competition in the space use, while belonging to the fastest. At the same time its simplicity is striking.

Considering both speed and space use, for the English text and the proteins, the SSA index is the best choice, still, our variants come close, especially for proteins.

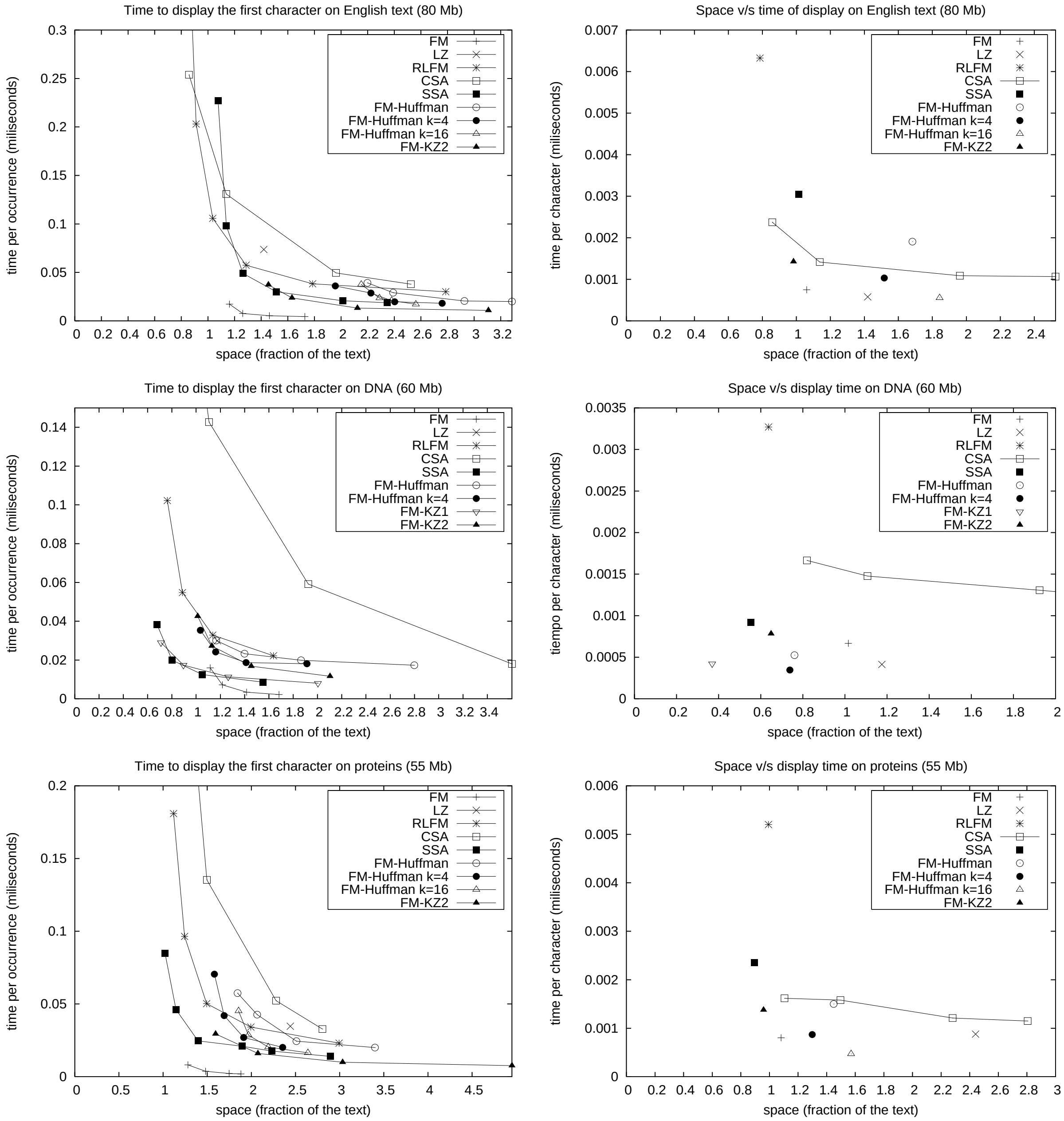

Figure 5. On the left, time to show the first character of a text context around the positions of the occurrences as a function of the size of the index. From top to bottom, we show the results of searching 80 MB of English text, 60 MB of DNA and 55 MB of proteins. In the plot of the DNA sequence, the point corresponding to the LZ index is covered. Its value is: space=1.18, time=0.03. On the right, time per character displayed around an occurrence and space for each 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 for large space usages. For DNA, however, our index, with the two versions, is better than the FM-index. This reduction in space is due to the low zero-order entropy of the DNA, which makes our index compact and fast.

Regarding the time for displaying the first character, the FM-index is faster than our index. Again, our index takes more space than the other indexes to get competitive time for English and proteins, and reduces its space for DNA. Regarding display time per character, our index with $k = 4$ is the fastest for DNA with a low space requirement, becoming an interesting alternative for this type of query.

The version of our index with $k = 4$ improved both the space and time with respect to the binary version and it became a very good alternative for counting and reporting queries, especially for DNA, due to the low zero-order entropy of this text.

9 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 σ. Our key idea is to encode the text with the Kautz-Zeckendorf coding, offering instant synchronization at codeword boundaries (a property missing in Huffman coding, thus implying a significant space penalty in FM indexes), at still being quite succinct. While not competitive to the best succinct indexes in theory, our solutions fare well in practice, and are simpler conceptually and easier to implement than the other structures.

Acknowledgements

This work was partially funded by Fondecyt Grant-1-050493, Chile (Gonzalo Navarro).

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. 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.
6. Sz. Grabowski, V. Mäkinen, G. Navarro, and A. Salinger. A Simple Alphabet-Independent FM-index. In *Proc. PSC'05*, pp. 230–244, 2005.
7. 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.
8. 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.
9. D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. TREC-3*, pages 1–19, 1995. NIST Special Publication 500-207.
10. G. Jacobson. *Succinct Static Data Structures.* PhD thesis, CMU-CS-89-112, Carnegie Mellon University, 1989.
11. W. H. Kautz. Fibonacci codes for synchronization control. *IEEE Trans. on Inf. Th.*, 11, pp. 284–292, 1965.

12. I. Munro. Tables. In *Proc. FSTTCS'96*, pp. 37–42, 1996.
13. G. Navarro and V. Mäkinen. Compressed Full-Text Indexes (2nd version). Technical Report TR/DCC-2006-6. Dept. of Computer Science, Univ. of Chile, April 2006. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/survcompr2.ps.gz`.
14. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic J. of Computing* 12(1):40–66, 2005.
15. G. Navarro. Indexing text using the Ziv-Lempel trie. *J. of Discrete Algorithms* 2(1):87–114, 2004.
16. 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.
17. E. Zeckendorf. Représentation des nombres naturels par une somme de nombres de Fibonacci ou de nombres Lucas. Bull. Soc. Roy. Sci. Liège, 41, pp. 179–182, 1972.