International Journal of Foundations of Computer Science

A SIMPLE ALPHABET-INDEPENDENT FM-INDEX *

SZYMON GRABOWSKI
Computer Engineering Department, Technical University of Łódź
sgrabow@zly.kis.p.lodz.pl
Al. Politechniki 11
90-924 Łódź, Poland

GONZALO NAVARRO
Department of Computer Science, University of Chile
gnavarro@dcc.uchile.cl
Blanco Encalada 2120, 3er piso,
Santiago, Chile

RAFAŁ PRZYWARSKI
Computer Engineering Department, Technical University of Łódź
rafal.przywarski@svensson.com.pl
Al. Politechniki 11
90-924 Łódź, Poland

ALEJANDRO SALINGER
David R. Cheriton School of Computer Science, University of Waterloo
ajsalinger@cs.uwaterloo.ca
200 University Avenue West,
Waterloo, Ontario, Canada N2L 3G1

VELI MÄKINEN
Department of Computer Science, University of Helsinki
vmakinen@cs.helsinki.fi
P. O. Box 68 (Gustaf Hällströmin katu 2b),
FIN-00014 Helsinki, Finland

Received (received date)
Revised (revised date)
Communicated by Editor's name

*Earlier partial versions of this work appeared in [7, 9, 21].

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, σ, 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 significant dependence on σ. 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 located 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. We also explore other coding variants alternative to Huffman and exploit their synchronization properties. Our experimental results on various types of texts show that our indexes are highly competitive in the space/time tradeoff map.

1. Introduction

A *full-text index* is a data structure that enables one 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 the need of scanning the whole text T. Text and pattern are sequences of characters over an alphabet Σ of size σ. In practice one wants to know not only the value *occ*, i.e., how many times the pattern appears in the text (a *counting query*) but also the text positions of those *occ* occurrences (a *locating query*), and usually also a text context around them (a *displaying query*).

A classic example of a full-text index is the *suffix tree* [24], which achieves $O(m)$ and $O(m + occ)$ time complexities for counting and locating queries, respectively. Unfortunately, a suffix tree requires $O(n \log n)$ bits of space[a], and also the constant factor is large. A smaller space complexity factor is achieved by the *suffix array* [15], where term m in the time complexities becomes $m \log n$ or $m + \log n$ depending on the variant. Still the space usage is high and may rule out the structure from some applications, for example 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 [3, 5, 10, 11, 12, 13, 16, 18, 20, 23]. A truly exciting perspective originated in the work of Ferragina and Manzini [3]: They showed that a full-text index may allow discarding the original text, as it contains enough information to recover the text and even access any arbitrary substring of it. We denote a structure with such a property a *self-index*.

The FM-index of Ferragina and Manzini [3], in addition, had a space complexity proportional to H_k, the kth order (empirical) entropy of T. The space complexity, however, contains an exponential dependence on the alphabet size σ. A dependence on σ also appears in the time used to solve a locating or displaying query. Such weaknesses make the original FM-index appealing only for texts with very small alphabets, such as DNA.

More precisely, the FM-index needs up to $5H_k n + O\left((\sigma \log \sigma + \log\log n)\frac{n}{\log n} + \right.$

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

$n^\gamma \sigma^{\sigma+1}\big)$ bits of space, where $0 < \gamma < 1$. The time needed to solve a counting query is just $O(m)$. The text position of each occurrence can be located in $O\left(\sigma \log^{1+\varepsilon} n\right)$ time, for some $0 < \varepsilon < 1$ that shows up in the sublinear terms of the space complexity. Finally, the time needed 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.

This alphabet dependence is eliminated in a practical implementation of the FM-index [4], at the price of not achieving the optimal search time anymore. Further developments [5] achieve $nH_k + o(n \log \sigma)$ bits of space for any $k \leq \alpha \log_\sigma n$ and constant $\alpha < 1$. The counting time complexity now raises to $O(m \log \sigma)$, yet the σ terms multiplying the locating and displaying complexities of the FM-index become now $\log \sigma$.

The compressed suffix array (CSA) of Sadakane [23] offers another tradeoff related to the dependence on σ. The CSA needs $(H_0/\varepsilon + O(\log \log \sigma))n$ bits of space. Its counting time is $O(m \log n)$. Each occurrence can be located in $O\left(\log^\varepsilon n\right)$ time, and a text substring of length L can be displayed in time $O\left(L + \log^\varepsilon n\right)$. Other later developments in the line of the CSA [10, 11] achieve results similar to those in [5].

In this paper we present an alternative approach to removing the large space dependence of the FM-index. We Huffman-compress the text and then, as in the FM-index, apply the Burrows-Wheeler transform over it. The resulting structure can be regarded as an FM-index built over a binary sequence. As a result, we remove any significant dependence on the alphabet size.

Our index needs $n(2H_0 + 3 + \varepsilon)(1 + o(1))$ bits of space, for any $0 < \varepsilon < 1$. It solves counting queries in $O(m(H_0 + 1))$ average time. The text position of each occurrence can be located 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 required to locate a text position. In the worst case all the terms $(H_0 + 1)$ in the time complexities become $\log n$. It is possible to convert this $\log n$ into $\log \sigma$ without affecting the average complexities [8], but we refrain from this idea in a real implementation.

We also study several variants of the original index that reduce the term 2 in front of the space complexity, such as K-ary Huffman and Kautz-Zeckendorf coding. Our experimental results on English and proteins show that, although not among the most succinct, our index is faster than the others in many aspects, even letting the others use significantly more space. On the other hand, on DNA our index is both the fastest and smallest compared to previous work. Furthermore, our index is attractive for its simplicity.

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 ith 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 two definitions that will be useful later as well.

Definition 1 *Given a text T over an ordered alphabet $\Sigma = \{c_1, \ldots, c_\sigma\}$, $C[c_1, c_\sigma]$ stores in $C[c_i]$ the number of occurrences of characters $\{c_1, \ldots, c_{i-1}\}$ in T.*

Definition 2 *Let X be a sequence, then $Occ(X, c, i)$ is the number of occurrences of character c in the prefix $X[1, i]$.*

With these definitions we can introduce the *backward* BWT that produces T given T^{bwt}.

1. Compute the array C for T. Notice that $C[c] + 1$ is the position of the first occurrence of c in F (if any).
2. Define the *LF-mapping* $LF[1, n+1]$ as $LF[i] = C[L[i]] + Occ(L, L[i], i)$.
3. Reconstruct T backwards as follows: set $s = 1$ (because $\mathcal{M}[1] = \$T$) and then, for each $i \in n, \ldots, 1$ 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, and returns the number of occurrences. The algorithm 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]: *After phase i, 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).

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

Algorithm FM_Count(P,T^{bwt})
(1) $i = m$;
(2) $sp = 1$; $ep = n$;
(3) **while** (($sp \leq ep$) **and** ($i \geq 1$)) **do**
(4) $\quad c = P[i]$;
(5) $\quad sp = C[c] + Occ(T^{bwt}, c, sp-1)+1$;
(6) $\quad ep = C[c] + Occ(T^{bwt}, c, ep)$;
(7) $\quad i = i - 1$;
(8) **if** ($ep < sp$) **then return** 0 **else return** $ep - sp + 1$.

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

3. First Huffman, then Burrows-Wheeler

We now introduce our new index. From now on assume T already contains the terminator \$ at the end[b]. To begin, this text T will be Huffman-compressed into a binary stream T' and the codeword beginnings marked in Th (the final index will not store T' nor Th). The idea is that, instead of searching T for P, we can Huffman-encode P into P' and search the binary text T' for P'. Yet we have to ensure that the occurrences of P' are codeword-aligned.

Definition 3 *Let $T'[1,n']$ be the binary stream resulting from Huffman-compressing T, where $n' < (H_0+1)\,n$ since (binary) Huffman poses a maximum representation overhead of 1 bit per symbol. Let $Th[1,n']$ be a second binary stream such that $Th[i] = 1$ iff i is the starting position of a Huffman codeword in T'. In the Huffman code, we ensure that the last bit assigned to the end marker "\$" is zero.*

The reason for the final condition will be clear later. Note that this can always be done, by making the node corresponding to "\$" a left child of its parent in the Huffman tree.

3.1. Structure

We apply the Burrows-Wheeler transform over text T', so as to obtain $B = (T')^{bwt}$. Yet, in order to have a binary alphabet, T' will not have its own special terminator character "\$" (note that the end marker of T is encoded in binary at the end of T', just as any other character of T). To formally define B we resort to the suffix array $\mathcal{A}'$ of T', yet the final index will not store $\mathcal{A}'$.

Definition 4 *Let $\mathcal{A}'[1,n']$ be the suffix array for text T', that is, a permutation of $[1,n']$ such that $T'[A'[i],n'] < T'[A'[i+1],n']$ in lexicographic order, for all $1 \leq i < n'$. In these lexicographic comparisons, if a string x is a prefix of y, we assume $x < y$.*

Our index will represent $\mathcal{A}'$ in succinct form, via array B and another array Bh used to track the codeword beginnings in $(T')^{bwt}$.

[b] Thus the term nH_0 will refer to this new text with terminator included. The difference with the term nH_0 corresponding to the text without the terminator is only $O(\log n)$, and will be absorbed by the $o(n)$ terms that will appear later in the space complexity.

Definition 5 *Let $B[1,n']$ be a binary stream such that $B[i] = T'[\mathcal{A}'[i]-1]$ (except that $B[i] = T[n']$ if $\mathcal{A}'[i] = 1$). Let $Bh[1,n']$ be another binary stream such that $Bh[i] = Th[\mathcal{A}'[i]]$. This tells whether position i in $\mathcal{A}'$ points to the beginning of a codeword.*

3.2. Searching

Our goal is to search B exactly like the FM-index. For this sake we need array C and function Occ of Definitions 1 and 2, now applied to T' and B. As we are dealing with binary sequences, C and Occ are easy to compute using the well-known function $rank$.

Definition 6 *Given a binary sequence X, $rank(X,i)$ is the number of 1's in $X[1,i]$. In particular $rank(X,0) = 0$. The inverse function, $select(X,j)$, tells the occurrence of the jth 1 in X.*

Functions $rank$ and $select$ can be computed in constant time using only $o(n)$ extra bits on top of the original sequence of n bits [19, 2]. An optimized practical variant is described in [6].

Note that our C array has only two entries, which are easily precomputed. Similarly, Occ can be expressed in terms of $rank$.

$$\begin{array}{lcl|lcl} C[0] & = & 0 & Occ(B,0,i) & = & i - rank(B,i) \\ C[1] & = & n - rank(B,n') & Occ(B,1,i) & = & rank(B,i) \end{array}$$

Therefore, formulas $C[c] + Occ(T^{bwt}, c, i)$ in the search algorithm of Figure 1 are solved in our index by using $rank$ on B.

There is a small twist, however, because we are not putting a terminator to our binary sequence T' and hence no terminator appears in B. Let us call "#" (# < 0 < 1) the terminator that should appear in T', 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, and it is zero according to Definition 3. 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 "#"), and it would have $B[p_\#] = \#$. To obtain the right mapping to our binary B, we must add 1 to $C[0] + Occ(B,0,i)$ when $i < p_\#$. The computation of $C[1] + Occ(B,1,i)$ remains unchanged. Overall, formula $C[c] + Occ(T^{bwt}, c, i)$ is computed as follows

$$C[c] + Occ(T^{bwt}, c, i) \quad = \quad \begin{cases} i - rank(B,i) + [i < p_\#], & \text{if } c = 0 \\ n - rank(B,n') + rank(B,i), & \text{if } c = 1 \end{cases} \qquad (1)$$

where $p_\# = (\mathcal{A}')^{-1}[1]$.

Therefore, by preprocessing B to solve $rank$ queries, we can search B exactly as in the FM-index. Our search pattern is not the original P, but its binary encoding $P'[1,m']$ using the Huffman code we applied to T.

```
Algorithm Huff-FM_Count(P',B,Bh)
(1)  i = m';
(2)  sp = 1; ep = n';
(3)  while ((sp ≤ ep) and (i ≥ 1)) do
(4)       if P'[i] = 0 then
               sp = (sp − 1) − rank(B, sp − 1) + [sp − 1 < p#] + 1;
               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);
(7)       i = i − 1;
(8)  if ep < sp then return 0 else return rank(Bh, ep) − rank(Bh, sp − 1);
```

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

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 suffixes of T' that do not start a Huffman codeword, yet start with P'.

Array Bh now comes into play to filter out those spurious occurrences. In the range $[sp, ep]$ found by the search of B' for P', every bit set in $Bh[sp, ep]$ 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 final search algorithm.

3.3. Analysis

The index stores B and Bh, each of $n' < (H_0 + 1)\, n$ bits. The extra space required by the *rank* structures is $o(n') = o((H_0 + 1)\, n)$. The only dependence on σ is that we must store the Huffman code, for which $\sigma \log n$ bits is sufficient (say, using a canonical Huffman tree). Thus our index requires at most $2n(H_0 + 1)(1 + o(1)) + \sigma \log n$ bits. The latter term is $o(n)$ even for very large alphabets, $\sigma = o(n/\log n)$. Note that alternative indexes achieving kth order compression [5, 10, 11, 18] require $\sigma = O(n^{1/k})$. The space of our index will grow slightly in the next sections due to additional requirements for locating and displaying queries.

Let us now consider the time for counting queries. If we assume that the characters in P have the same distribution of T (which holds in particular if P is randomly chosen from T, or generated by the same statistical source), then the length of P' is $m' < m(H_0 + 1)$. This is the number of steps to search B using the algorithm of Figure 2, so the search complexity is $O(m(H_0 + 1))$. Since $H_0 \leq \log \sigma$, our time is better than the $O(m \log \sigma)$ complexity of several indexes [5, 10, 11][c].

We now analyze our worst-case search cost, which depends on 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,

[c] In practice, those indexes can also achieve $O(m(H_0 + 1))$ average time using Huffman-shaped wavelet trees.

the Fibonacci sequence $F(i)$. Hence, a Huffman tree with depth d needs that the text is of length at least $n \geq 1 + \sum_{i=1}^{d} F(i) = F(d+2)$ [25, pp. 397]. Therefore, the maximum length of a codeword is $F^{-1}(n) - 2 = \log_\phi(n) - 2 + o(1)$, where $\phi = (1+\sqrt{5})/2$.

Thus, the encoded pattern P' cannot be longer than $O(m \log n)$ and this is also the worst case search cost. This matches the worst-case search cost of the original CSA, while our average case is better. It is actually possible to reduce our worst-case 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 some suitable constant $x > 0$. For details see [8].

4. Locating Occurrences and Displaying the Text

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

Given the suffix array interval that contains the *occ* occurrences found, the FM-index locates each such position in $O(\sigma \log^{1+\varepsilon} n)$ time, for any $0 < \varepsilon < 1$ (which affects the sublinear space component). The CSA can locate each occurrence in $O(\log^{\varepsilon} n)$ time, where ε is paid in the space, nH_0/ε. 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 locate each occurrence in time $O(\frac{1}{\varepsilon}(H_0+1)\log n)$ and display a text context in time $O(L \log\sigma + \log n)$ in addition to locating time. On average, if random text positions are involved, the overall complexity to display a text interval is $O((H_0+1)(L + \frac{1}{\varepsilon}\log n))$.

A first problem is how to extract, in $O(occ)$ time, the *occ* positions of the bits set in $Bh[sp, ep]$. This is easy using *select* function of Definition 6. Actually we need a simpler version, $selectnext(Bh, j)$, which returns the position of 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 $pos_1 \ldots pos_{occ}$ can be found as

$$\begin{aligned} pos_1 &= selectnext(Bh, sp), \\ pos_{i+1} &= selectnext(Bh, pos_i + 1). \end{aligned}$$

To complete the locating and displaying processes, we need additional structures.

4.1. Structure

We sample T' at approximately regular intervals, so that only codeword begin-

nings can be sampled. A sampling parameter $0 < \varepsilon < 1$ will control the density of the sampling and the corresponding space/time tradeoff.

Definition 7 *Given $0 < \varepsilon < 1$, let $\ell = \lceil \frac{2n'}{\varepsilon n} \log n \rceil$ be the* sampling step. *Our sampling of T' is a sequence $\mathcal{S}[1, \lfloor \frac{\varepsilon n}{2 \log n} \rfloor]$, so that $\mathcal{S}[i]$ is the first position of the codeword that covers position $1+\ell(i-1)$ in T', that is, $\mathcal{S}[i] = select(Th, rank(Th, 1+\ell(i-1)))$.*

Our index will include three additional structures called ST, TS, and S. TS is an array storing the positions of $\mathcal{A}'$ that point to the sampled positions in T', in increasing text position order.

Definition 8 *$TS[1, \lfloor \frac{\varepsilon n}{2 \log n} \rfloor]$ is an array such that $TS[i] = j$ iff $\mathcal{A}'[j] = \mathcal{S}[i]$.*

Array ST is formed using the same positions of $\mathcal{A}'$, now sorted by position in $\mathcal{A}'$ and storing their position in T.

Definition 9 *$ST[1, \lfloor \frac{\varepsilon n}{2 \log n} \rfloor]$ is an array such that $ST[i] = rank(Th, \mathcal{A}'[j])$, where j is the i-th position in $\mathcal{A}'$ that points to a position present in $\mathcal{S}$.*

Finally, $S[i]$ tells whether the i-th entry of $\mathcal{A}'$ that points a codeword beginning, points to sampled a text position. S will be further processed for *rank* queries.

Definition 10 *$S[1, n]$ is a bit array such that $S[i] = 1$ iff $\mathcal{A}'[select(Bh, i)]$ is in $\mathcal{S}$.*

4.2. Locating

We have to determine the text position corresponding to an entry $\mathcal{A}'[i]$ for which $Bh[i] = 1$, that is, a valid occurrence. We use bit array $S[rank(Bh, i)]$ to determine whether $\mathcal{A}'[i]$ points or not to a codeword beginning in position in $ST[rank(S, rank(Bh, i))]$. If it does, we are done. Otherwise, just as with the FM-index, we determine position i' whose value is $\mathcal{A}'[i'] = \mathcal{A}'[i] - 1$. This process is repeated until a new codeword beginning is found, that is, $Bh[i'] = 1$ (this corresponds to moving backward bit by bit in T'). We then check again whether this position is sampled, and so on until finding a sampled codeword beginning. If we finally obtain position *pos* after d repetitions, the answer is $pos + d$ as we have moved backward d positions in T.

It is left to specify how to 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 using Eq. (1). Figure 3 gives the pseudocode.

4.3. Displaying

In order to display a text substring $T[l, r]$ of length $L = r - l + 1$, we start by binary searching TS for the smallest sampled text position larger than r. Let j be the index found in TS. Given value $i = TS[j]$, we know that $S[rank(Bh, i)] = 1$ as i is a sampled entry in $\mathcal{A}'$. The corresponding position in T is $ST[rank(S, rank(Bh, i))]$.

Once we find the first sampled text position that follows r, we know its corresponding position i in $\mathcal{A}'$. From there on, we move backwards in T' (via the

```
Algorithm Huff-FM_Locate(i,B,Bh,S,ST)
(1)  d = 0;
(2)  while S[rank(Bh, i)] = 0 do
(3)       do if B[i] = 0 then i = i − rank(B, i) + [i < p_#];
               else i = n′ − rank(B, n′) + rank(B, i);
(4)       while Bh[i] = 0;
(5)       d = d + 1;
(6)  return d + ST[rank(S, rank(Bh, i))];
```

Figure 3: Algorithm for locating 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,S,ST,TS)
(1)  j = min{k, ST[rank(S, rank(Bh, TS[k]))] > r}; // binary search
(2)  i = TS[j];
(3)  p = ST[rank(S, rank(Bh, i))];
(4)  L = ⟨ ⟩;
(5)  while p ≥ l do
(6)       do L = B[i] · L;
(7)            if B[i] = 0 then i = i − rank(B, i) + [i < p_#];
               else i = n′ − rank(B, n′) + rank(B, i);
(8)       while Bh[i] = 0;
(9)       p = p − 1;
(10) Huffman-decode the first r − l + 1 characters from list L;
```

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

LF-mapping over $\mathcal{A}'$), position by position, until reaching the first bit of the codeword for $T[r + 1]$. Then, we obtain the L preceding characters of T, by further traversing T' backwards, now collecting all its bits until reaching the first bit of the codeword for $T[l]$. The collected bit stream is reversed and Huffman-decoded to obtain $T[l, r]$. Figure 4 shows the pseudocode.

4.4. Analysis

Array TS requires $\frac{\varepsilon n}{2}(1 + o(1))$ bits, since there are n'/ℓ entries and each entry needs $\log n' \leq \log n + O(\log \log n)$ bits. Array ST requires other $\frac{\varepsilon n}{2}$ bits, as its entries require $\log n$ bits. Finally, array S preprocessed for $rank$ queries requires $n(1 + o(1))$ bits. Overall, we spend $(1 + \varepsilon)n(1 + o(1))$ additional bits of space for locating and displaying queries. This raises our final space requirement to $n(2H_0 + 3 + \varepsilon)(1 + o(1)) + \sigma \log n$ bits.

Let us now consider the time for locating. This corresponds to the maximum distance between two consecutive samples in T', as we traverse it backwards until finding a sampled position. Recall from Section 3 that no Huffman codeword can be longer than $\log_\phi n - 2 + o(1)$ bits. Then, the distance between two consecutive

samples in T', after the adjustment to codeword beginnings, cannot exceed

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

which is therefore the worst-case locating complexity.

For the displaying time, each of the L characters obtained costs us $O(H_0+1)$ on average because we obtain the codeword bits one by one. In the worst case they cost us $O(\log n)$. Note that we might have to traverse some additional characters from the next sampled position until reaching the text area of interest. Finally, we must consider the $O(\log n)$ time for the binary search of TS. overall, the 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.

Theorem 1 *Given a text $T[1,n]$ over an alphabet σ and with zero-order entropy H_0, the FM-Huffman index requires $n(2H_0+3+\varepsilon)(1+o(1))+\sigma\log n$ bits of space, for any constant $0<\varepsilon<1$ fixed at construction time. It can count the occurrences of $P[1,m]$ in T in average time $O(m(H_0+1))$ and worst-case time $O(m\log n)$. Each such occurrence can be located in worst-case time $O(\frac{1}{\varepsilon}(H_0+1)\log n)$. Any text substring of length L can be displayed in time $O((H_0+1)(L+\frac{1}{\varepsilon}\log n))$ on average and $O((L+(H_0+1)\frac{1}{\varepsilon})\log n)$ in the worst case.*

5. K-ary Huffman

While storing B seems necessary as we are using zero-order compression of T, doubling the space requirement to store Bh seems a waste of space. In this section we explore a way to reduce the size of Bh. Instead of using Huffman over a binary coding alphabet, we can use a coding alphabet of $k>2$ symbols, so that each symbol needs $\lceil\log k\rceil$ bits. Varying the value of k yields interesting time/space tradeoffs. We use only powers of 2 for k values, so that each symbol can be represented without wasting space.

The space usage varies in different aspects. The size of B increases since Huffman's compression ratio degrades as k grows. 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, $H_0^{(k)}=H_0/\log_2 k$. Therefore, the size of B is bounded by $n'\log k=(H_0+\log k)n$ bits. The size of Bh, on the other hand, is reduced since it needs one bit per symbol, that is n' bits.

The total space used by B and Bh 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 $1\le\log k\le H_0$. In particular, if we choose $\log k=\alpha H_0$, then the space is upper bounded by $n((1+\alpha)H_0+1+1/\alpha)$, which is optimized at $\alpha=1/\sqrt{H_0}$ (that is, $\log k=\sqrt{H_0}$). Using this optimal α value, the overall space required by B and Bh is $n(\sqrt{H_0}+1)^2<n(H_0+1)(1+2/\sqrt{H_0})$. The original overhead factor of 2 over pure Huffman compression has been reduced to $1+O(1/\sqrt{H_0})$.

The space for the *rank* structures changes as well. The *rank* structure for Bh is computed in the same way of the binary version, and therefore its size is reduced to

$o(H_0^{(k)} n)$ bits. To solve $Occ(B, c, i)$ queries, we must build the sublinear-size *rank* structures over σ virtual binary sequences $B_c[1, n']$, so that $B_c[i] = 1$ iff $B[i] = c$. Therefore, $Occ(B, c, i) = rank(B_c, i)$ can be computed in constant time. The size of those *rank* structures adds up $o(kH_0^{(k)} n)$ bits. (The solution for *rank* requires accessing the bit vectors B_c, but one can use B itself instead.)

If we use the optimum k derived above, the space for the *rank* structures is $o(n2^{\sqrt{H_0}}/\sqrt{H_0})$ extra space, which turns out to be still $o(n)$ (more precisely, $O(n/\log\log n)$) for $H_0 \leq (\log\log n)^2$. This value is reasonably large in practice.

Regarding the time complexities, the pattern has average length $< m(H_0^{(k)} + 1)$ symbols. This is the counting complexity, which is reduced as we increase k. Using the value $k = 2^{\sqrt{H_0}}$ that optimizes the space complexity, the counting time is $O(m\sqrt{H_0})$. On the other hand, the average counting time can be made $O(m)$ by using a constant α. For locating queries and displaying text, we need the same additional structures TS, ST and S as for the binary version. The k-ary version can locate the position of an occurrence in $O\Big(\frac{1}{\epsilon}(H_0^{(k)} + 1)\log n\Big)$ time, which is the maximum distance between two sampled positions. Similarly, the time used 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. Again, with the optimum k, $H_0^{(k)}$ is $\sqrt{H_0}$, and it can be made $O(1)$ by using a constant α.

6. Kautz-Zeckendorf Coding

The previous section aims at reducing the size of Bh in exchange for increasing the size of other structures. In this section we aim at completely getting rid of the Bh array, by replacing Huffman coding with another for which the bit stream itself enables synchronization at codeword boundaries. Our solution is based on a representation of integers first advocated by Kautz [14] for its synchronization properties, that presents each number in a unique form as a sum of Fibonacci numbers. This technique is better known from a work by Zeckendorf [26], therefore we will call it Kautz-Zeckendorf coding.

Consider the (slightly displaced) Fibonacci sequence 1, 2, 3, 5, 8, 13, ..., that is, $f_1 = 1$, $f_2 = 2$, and $f_{i+2} = f_{i+1} + f_i$. It is easy to prove by induction that any natural number N can be uniquely decomposed into a sum of Fibonacci numbers, where each number is summed at most once and no two consecutive elements of the sequence are used in the decomposition. (If two consecutive elements 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 [14]. The recurrence is now $f_i^{(k)} = i$ for $i \leq k$ and $f_{i+k}^{(k)} = f_{i+k-1}^{(k)} + f_{i+k-2}^{(k)} + \ldots + f_{i+1}^{(k)} + f_i^{(k)}$. In this representation we do not permit a sequence of k consecutive elements of the sequence in the decomposition, and thus no stream of k 1's appears in the bit vector.

The binary encoding we use for symbols differs slightly from the above descrip-

tion. The reason is that, for example, 0, 00, 000, ... are all different codewords, albeit all of them represent $N = 0$. Operationally, our codes of a given length are obtained by generating all the binary sequences of that length and then removing those having k consecutive 1's. We also require the codeword to finish with a 0, for reasons to be made clear soon. We then generate the codes by increasing length, assigning the i-th code to the i-th most frequent source symbol. In addition, all the codewords are prepended with a sequence of k 1's followed by one 0.

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. This is expected to outweight the fact that the encoding is not optimal as Huffman. An important side-effect is also that there is no need for *select* nor *selectnext* to find the successive matches: they all are in a contiguous range in $\mathcal{A}'$. All the rest of the operatory remains unchanged.

There is another consequence of the way we generate the codewords. Because the codewords are zero-terminated, the longest runs of 1's are precisely the codeword headers, of k 1's. Those are the lexicographically largest suffixes of T', and thus the characters preceding them occupy the n largest positions in B. As all those preceding characters are 0, we can remove the last n bits from B knowing that they will be zero. This saves one additional bit per symbol in T. Letting codewords finishing with up to $k - 1$ 1's does not save that much space.

7. Experimental Results

In this section we present experimental results on counting, locating and displaying queries, and compare the efficiency to existing indexes. The indexes used for the experiments were the FM-index implemented by Navarro [20], Sadakane's CSA [23], the RLFM index [18], the SSA index using balanced wavelet trees [18], and the LZ index [20]. Other indexes, like the Compressed Compact Suffix Array (CCSA) of Mäkinen and Navarro [17], the Compact SA of Mäkinen [16] 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[d] (files `WSJ87-89`), 60 MB of DNA and 55 MB of protein sequences, both obtained from the BLAST database of the NCBI[e] (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`.

We first give the results regarding the space used by our index and then the results of the experiments classified by query type.

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

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

7.1. Space Consumption

Table 1 (top) shows the space that the k-ary Huffman index takes as a fraction of the text for different values of k and for the three types of text considered. These values do not include the space required to locate positions and display text.

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

It is also interesting to see how the space requirement of the index is divided among its different structures. Table 1 (bottom) shows the space used by each of the structures for the index with $k = 2$ and $k = 4$, considering the three types of text. For higher values of k the space used by the *rank* tables will increase too fast compared to the reduction in Bh.

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 k-ary Huffman index for different values of k. The value corresponding to 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 spaces used by the Huffman table, the constant-size tables for *rank*, and array C, as they are all negligible.

A similar study is carried out on Kautz-Zeckendorf coding in Table 2, although in this case there is no array Bh. The space is not the result of a tension between B and Bh, but between the length of the header and the number of different codewords of each length.

k	Fraction of text		
	English	DNA	Proteins
1	2.04	0.41	1.39
2	0.91	0.54	0.88
3	1.04	0.71	1.02
4	1.20	0.89	1.19
5	1.37	1.06	1.36

Table 2: Space requirement of our FM-KZ index with parameter k, for different values of k.

7.2. Counting Queries

For the three files, we show the counting 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 at random positions from 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 has a space/time tuning parameter space for this type of queries, we adjusted it to 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 space requirement to count occurrences. Only the CSA permits a space/time tradeoff for counting queries, so the it appears as a line while the other indexes are represented by points.

7.3. Locating and Displaying

We measured the time each index took to search for a pattern and locate 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.

Figure 6 (left) shows the time per occurrence located for each index as a function of its size. Most indexes (except LZ) permit a space/time tradeoff for locating, so they appear as lines in the plots. The CSA has two such parameters now, and we show the optimal combination that achieves each space occupancy.

Figure 6 (right) shows the time to display a text character as a function of the index size. For the same searched patterns above, we displayed 100 characters around each of their occurrences. As for counting, only the CSA permits a space/time tradeoff for this operation.

7.4. Analysis of Results

We can see that our FM-Huffman index with $k = 16$ is the fastest for counting queries for English and proteins. The version with $k = 4$ also gives relevant space/time tradeoffs. On the other hand, FM-KZ is the clear winner on DNA, as it takes by far the least space and its counting time is the best, together with SSA

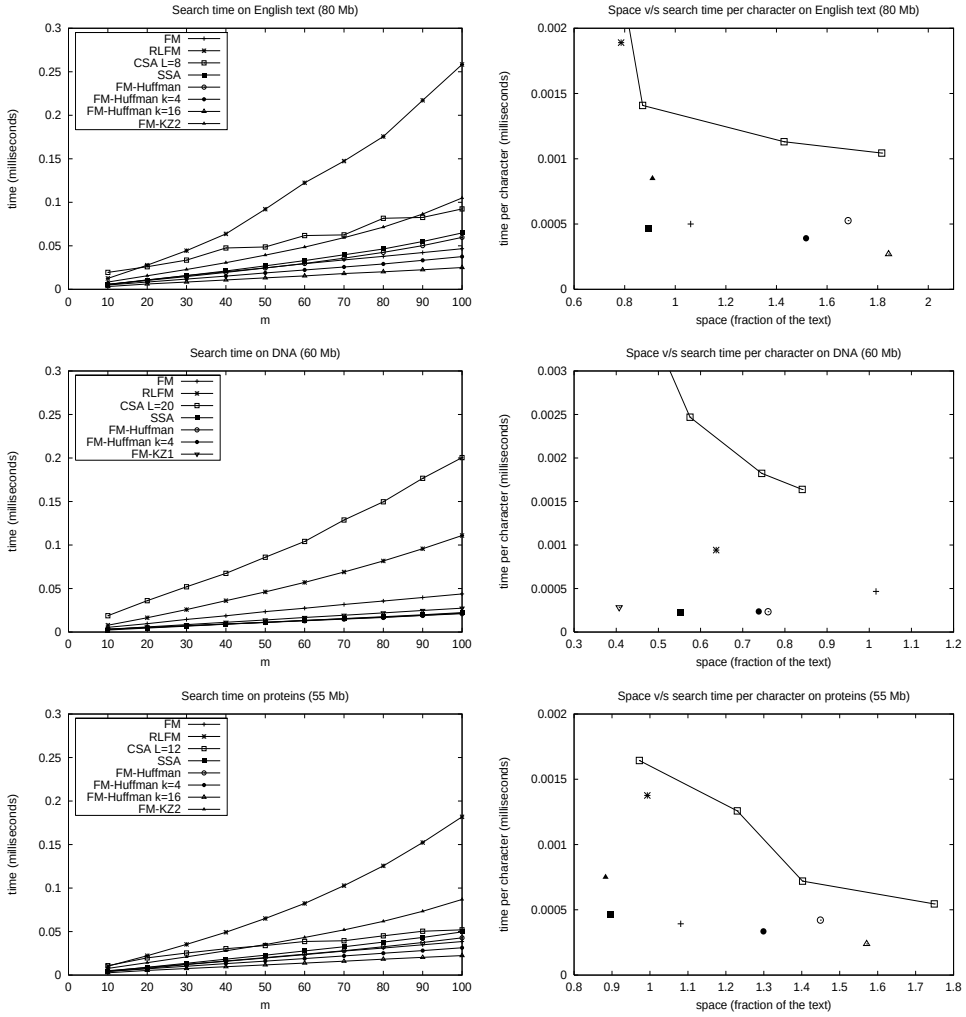

Figure 5: On the left, counting time as a function of the pattern length over English (80 MB), DNA (60 MB), and a proteins (55 MB). On the right, average search time per character as a function of the index size. The times of the LZ index are not competitive in this experiment.

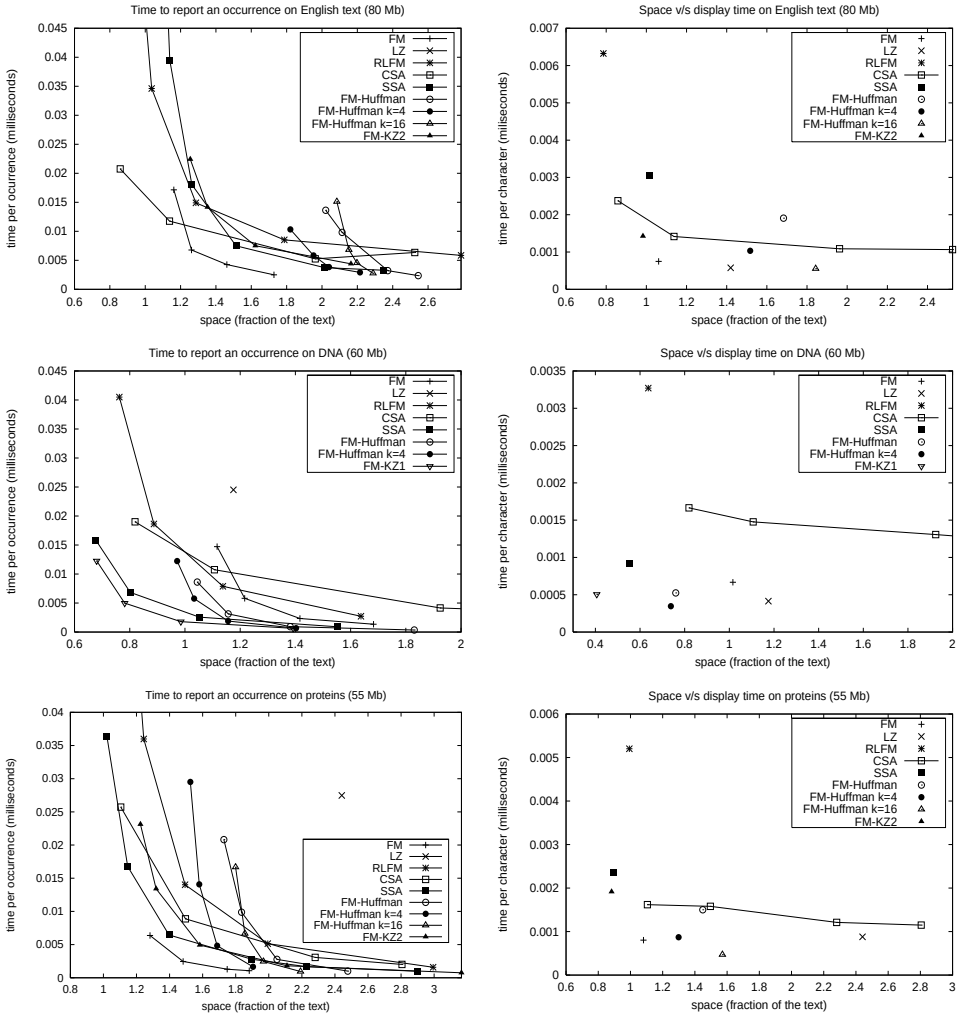

Figure 6: On the left, time to locate the positions of the occurrences as a function of the size of the index. On the right, time per character to display text passages. We show the results of searching on 80 MB of English text, 60 MB of DNA and finally 55 MB of proteins. The reporting time of LZ on English is 0.07 milliseconds.

and FM-Huffman. The other outstanding index is SSA, as it offers an attractive space/time tradeoff on English and proteins, being second-best on DNA. As expected, all the FM-Huffman and FM-KZ versions are faster than CSA, RLFM and LZ, the latter not being competitive for counting queries.

For locating queries, our indexes do not give competitive space/time tradeoffs on English nor proteins, where the FM-index, CSA and SSA dominate in all the spectrum. When all the indexes use much space, our FM-index variants can be faster than RLFM, CSA, LZ, and barely the SSA. For DNA, however, our FM-KZ index gives the best tradeoff in all the spectrum. The next relevant indexes are the SSA and the FM-Huffman variants.

Regarding display time, our FM-Huffman index variants are again the fastest. On English text, however, the LZ is equally fast and much smaller ($k = 16$ is the relevant FM-Huffman version here). The FM-index, FM-KZ, and CSA also give relevant space/time tradeoffs. On DNA, the FM-Huffman version with $k = 4$ is the fastest, requiring also little space. The only other interesting tradeoff is given by FM-KZ, which takes by far the least space and competitive time. Finally, on proteins, FM-Huffman version $k = 16$ is clearly the fastest. The best competitor, the FM-index, uses 30% less space but it is twice as slow. The other relevant space/time tradeoff is given by FM-KZ.

In general we can see that the FM-Huffman index is in many cases the fastest, albeit it cannot operate on very little space as other indexes. On DNA, on the other hand, FM-KZ is in most cases the smallest and fastest index.

8. Conclusions

We have presented 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 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 of (almost) not depending on the alphabet size, and of having better complexities than other indexes for some operations. We also discussed and tested alternative variants of our index, where the binary Huffman was replaced with other encodings with stronger synchronization properties.

Our structures are simple and easy to implement. Our experimental results show that our indexes are competitive in practice against other implemented alternatives. In some cases they are not the most succinct, but they are the fastest, even if we let the other structures use significantly more space. In other cases, our indexes are both the smallest and fastest among the compared alternatives.

After several years of mainly theoretical development, the field of compressed full-text self-indexing is moving fast to practical considerations. Our work can be seen as one of the first practice-oriented developments [7]. Recently, new indexes and variants have been implemented and a site devoted to practical implementations and testbeds is being developed (`http://pizzachili.dcc.uchile.cl` and

http://pizzachili.di.unipi.it). New implementations are being constantly added to this site. Our immediate future work is to adapt the most promising variants of our indexes to the common interface of this site, so as to permit a uniform comparison among the most up-to-date implementations. We also plan to continue the research on coding variants whose properties can be used to reduce the size of the index.

Acknowledgments

We thank the anonymous referees for suggesting improvements to the manuscript. This work was partially funded by Fondecyt Grant 1-050493 (Gonzalo Navarro).

References

1. M. Burrows and D. 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. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proc. SPIRE'04*, pp. 150–160, 2004. LNCS 3246.
6. 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.
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. 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.
9. Sz. Grabowski, V. Mäkinen, G. Navarro, and A. Salinger. A simple alphabet-independent FM-index. In *Proc. PSC'05*, pp. 230–244, 2005.
10. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA'03*, pp. 841–850, 2003.
11. R. Grossi, A. Gupta, and J. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proc. SODA'04*, 2004.
12. 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.
13. J. Kärkkäinen. *Repetition-Based Text Indexes*, PhD Thesis, Report A-1999-4, Department of Computer Science, University of Helsinki, Finland, 1999.
14. W. Kautz. Fibonacci codes for synchronization control. *IEEE Trans. on Inf. Th.*, 11, pp. 284–292, 1965.
15. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22, pp. 935–948, 1993.
16. V. Mäkinen. Compact Suffix Array — A space-efficient full-text index. *Fundamenta Informaticae* 56(1-2), pp. 191–210, 2003.

17. V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In *Proc. CPM'04*, pp. 420–433. LNCS 3109, 2004.
18. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic J. of Computing* 12(1):40–66, 2005.
19. I. Munro. Tables. In *Proc. FSTTCS'96*, pp. 37–42, 1996.
20. G. Navarro. Indexing text using the Ziv-Lempel trie. *J. of Discrete Algorithms* 2(1):87–114, 2004.
21. R. Przywarski, Sz. Grabowski, G. Navarro, and A. Salinger. FM-KZ: An even simpler alphabet-independent FM-index. In *Proc. PSC'06*, 2006. To appear.
22. R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th ACM-SIAM SODA*, pp. 233–242, 2002.
23. 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.
24. P. Weiner. Linear pattern matching algorithm. *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory* pp. 1–11, 1973.
25. I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, New York, 1999. Second edition.
26. 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.